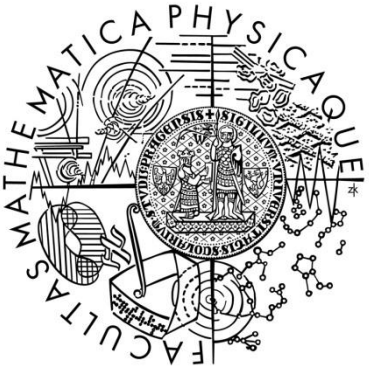
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Michal Brabec

Analýza paralelizovatelnosti programů na základě jejich bytecode

Department of Software Engineering

Supervisor of the master thesis: RNDr. David Bednárek, Ph.D.

Study programme: Informatics
Specialization: Software Systems

Prague 2013

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne ....

Název práce: Analýza paralelizovatelnosti programů na základě jejich bytecode

Autor: Michal Brabec

Katedra / Ústav: Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. David Bednárek, Ph.D., Katedra softwarového inženýrství

Abstrakt: Práce se zabývá analýzou možností aplikace algoritmů pro automatickou paralelizaci na programy, u kterých máme k dispozici jejich bytecode, nebo podobný mezikód. Nejdůležitějším vstupem těchto algoritmů je identifikace částí kódu, které by mohly být spuštěny zároveň, tyto části se nazývají nezávislé a právě testování závislostí v kódu je nejtěžší problém automatické paralelizace. Tento problém je v úplně obecném případě algoritmicky neřešitelný a práce se snaží zjistit, jestli je možné najít nezávislosti v bytecode alespoň v nějakém omezeném případě. Prvním krokem analýzy kódu funkce je integrace volaných funkcí, které umožní analyzovat výsledný kód najednou a získat tak přesnější informace. Dále je třeba identifikovat podmíněné skoky a cykly, až pak je teprve možné hledat nezávislosti v kódu a ty potom použít při aplikace paralelizačních algoritmů. Součástí práce je implementace integrace funkcí a analýzy kódu pro platformu Microsoft .NET Framework.

Klíčová slova: paralelizmus, testování závislostí, automatická paralelizace, inlinování funkcí, rozpoznávání konstruktů

Title: Analysis of automatic program parallelization based on bytecode

Author: Michal Brabec

Department / Institute: Department of Software Engineering

Supervisor: RNDr. David Bednárek, Ph.D., Department of Software Engineering

Abstract: There are many algorithms for automatic parallelization and this work explores the possible application of these algorithms to programs based on their bytecode or similar intermediate code. All these algorithms require the identification of independent code segments, because if two parts of code do not interfere with one another then they can be run in parallel without any danger of data corruption. Dependence testing is an extremely complicated problem and in general application, it is not algorithmically solvable. However, independences can be discovered in special cases and then they can be used as a basis for application of automatic parallelization, like the use of vector instructions. The first step is function inlining that allows the compiler to analyze the code more precisely, without unnecessary dependences caused by unknown functions. Next, it is necessary to identify all control flow constructs, like loops, and after that the compiler can attempt to locate dependences between the statements or instructions. Parallelization can be achieved only if the analysis discovered some independent parts in the code. This work is accompanied by an implementation of function inlining and code analysis for the .NET framework.

Keywords: parallelism, dependence testing, automatic parallelization, function inlining, construct recognition

Table of Contents

1 Introduction

1.1 Modern technologies and their challenges

1.1.1 Computer design

Computer design and construction experienced significant changes in the past decade and one of the most important is the use of multicore processors. This seems to be the easiest way to increase computational power of modern computers, since increasing clock rate proved to be impractical, because it causes problems like overheating and increased power consumption. Even though multicore processors are nothing new; in the past ten years, personal computers and even mobile phones started to use such processors massively and practically every contemporary computer contains multiple processor cores or even multiple processors. This presents a new problem for modern applications, which did not have to use parallel programming in the past, unlike scientific applications that have used parallel machines since the 1960th. Today, it is necessary to make even commercial applications run in parallel, to fully utilize the power of modern computers and that requires new techniques of programming and compiler design.

1.1.2 Parallelization strategies

There are basically two ways an application can run its code in parallel, first way is to design it manually to take advantage of multiple threads or vector instructions and the second possibility is to use a compiler that is able to recognize parts of code that can be executed simultaneously and compile the application to utilize parallelism.

Writing parallel application by hand requires more complicated architecture and their designers and programmers must solve complicated problems, like data sharing, resource management and user input. However, even difficulties encountered during design and creation of these applications can be overshadowed by problems that become apparent when the application must be test, verified, ported to a different platform or even executed on newer version of CPU.

The best way to use multicore processors is to use a compiler that can transform the application automatically with minimal or no help from programmers. Such a compiler must analyze the application and locate its parts that can run at the same time

without interfering with one another. This task is extremely difficult in general situation, but it can be done for specific applications or code fragments. For example, there are many compilers capable of using vector instructions to optimize numerical applications written in FORTRAN, which is one of the reasons why is it used for high performance computing.

1.1.3 Automatic parallelization

Automatic parallelization has been studied for many years and it is well implemented for FORTRAN and many optimizations are developed for C, but it is not commonly implemented in languages with reference semantics, like Java or C#. One of the possible reasons can be the fact that those languages are designed for business or web applications and these applications do not take advantage of parallelization techniques developed for older languages, since they are mostly designed for high-performance computing.

FORTRAN compilers can achieve very impressive increase in execution speed thanks to the simplicity of the language, which simplifies its analysis and allows the compiler to make more drastic transformations safely. The situation is much worse when the same techniques are used on other, more complex languages, like C++ or other languages inspired by C, like Java or C#, because their structure is much more complicated and the use of pointers or references can make it even more difficult to find any parts that could be executed simultaneously. Using FORTRAN may seem like a logical solution, but modern languages are much better suited for team development and common programmers are familiar with them, unlike FORTRAN.

Automatic parallelization is very complicated optimization but it could mean a great improvement for many applications and it would be even better, if it was possible to transform applications without recompilation which can be achieved when the transformation is performed on their bytecode[1].

1.2 Motivations

This work is motivated by the fact that automatic parallelization is not commonly used among the languages with reference semantics and its application could improve

[1] Intermediate code generated by a compiler that is later executed by a virtual execution engine or a virtual machine; a technique commonly used by modern interpreted languages, like C# or Java.

their performance or it could allow these languages to be used for more specialized applications.

First motivation for this work is to allow the development of specialized applications in modern languages because that would make it possible for common programmers to implement these applications with familiar language and development environment. Specialized applications, like complex numerical calculation, are traditionally developed in old structured languages, FORTRAN most prominently, and many programmers do not know these languages. The reason languages like FORTRAN are used for numerical applications is that it can be very effectively parallelized and the implementation of similar features for languages like C# can make it usable for similar purposes.

Another possibility is to use the C# as a frontend and transform its parallelized CIL code to another specialized code or language. This is a good way to provide a familiar programming environment to some complex system, which requires very specific parallel code, without forcing programmers to use some obscure language.

There is one great opportunity that any application created for the .NET Framework could be optimized by parallelizing its CIL code and it could be done without new compilation. This way, it would be possible to optimize every application written in an appropriate language, even without the knowledge of its source code and multiple languages could be supported when it is applied on platforms such as the Microsoft .NET framework[2]. This is not a goal of this work, but it may be a great motivation for the future work.

1.3 Goals

The main goal of this work is to find out if it is possible to apply existing parallelization algorithms on languages with reference semantics, like Java or C#. The actual parallelization is not the goal here, because it is very difficult and this work is meant mostly as a proof of concept. This work is based on the idea, that if the code can be transformed to a structure supported by the existing algorithms then the application could be parallelized using these efficient and well tested algorithms and the actual parallelization would be very similar to other languages, like C.

[2] The .NET Framework is a software framework developed by Microsoft that runs primarily on Microsoft Windows.

The goal is to show, at least theoretically, that it is feasible to perform automatic parallelization on the selected bytecode at least in some special cases. Used transformations will be presented on a couple of selected, well-known problems, that offer good opportunity for parallel execution and they are commonly used in many application. This approach may not be very ambitious, but the amount of work necessary to perform even the most basic parallelization is immense. This work is meant as a proof of concept; it tries to find out, if it is possible to apply existing parallelization algorithms to languages with reference semantics. The actual parallelization is discussed only theoretically, because this concentrates mostly on the steps necessary to the parallelization. The problems studied in this work are specified in section 1.5.2.

1.4 Types of parallelism

There are three main types of parallel execution: fine-grained parallelism, coarse grained parallelism and instruction-level parallelism. Each type requires different structure of code and each type uses a different part of the processor.

1.4.1 Fine-grained parallelism

Fine-grained parallelism is based on vector instructions and it produces the best results when the code contains arrays processed in loops. Vector instructions apply a single operation on vector of values and they are usually called SIMD[3], which means single operation on multiple data. Their most important characteristic is that each instruction performs a single operation on a vector of values that has to be as long as possible for the instruction to be efficient and it is not possible to use different transformation on parts of the vector. That is the reason, why loops are the best opportunity for this type of parallelism.

Loops must have specific structure to allow vectorization to be used. The statements in the loop must not depend on any other statements including themselves, because then it is not possible to reduce the loop to a vector instruction, one possible problem is shown in the following example.

[3] SIMD is term that refers to instructions that apply a single operation on multiple data, usually a vector.

```
    for (int k = 0; k < A[0].Length; k++)
    {
S1:     C[i][j] += A[i][k] * B[k][j];
    }
```

Figure 1 - Internal loop from matrix multiplication algorithm.

This example shows the internal loop of the most basic algorithm implementing matrix multiplication. The problem here is that the statement S1 cannot be optimized using vector instructions, because it always changes the same element `C[i][j]` in the loop, and therefore there is a loop carried dependence from S1 to itself. This type of parallelism optimizes the internal loops because they contain the actual statements that can be vectorized.

The main advantage of fine-grained parallelism is the fact that it is very stable and its application produces predictable increase of speed. Its stability relies on the known structure of vector instructions and their execution that does not require any scheduling or load-balancing.

1.4.2 Coarse-grained parallelism

Coarse-grained parallelism relies on threads and processes which are more flexible because they can execute any code concurrently, not just a single operation on vectors, but they are much more difficult to use. Threads can consume significant system resources and their use must be well justified. Coarse-grained parallelism produces best results, when the code contains long complicated code segments, which do not depend on one another and this is most common when there is a complex loop and its body is independent, because then it is possible to run the body in multiple threads. The type of parallelism optimizes outer loops, unlike vectorization.

```
    for (int j = 0; j < B[0].Length; j++)
S1:     for (int k = 0; k < A[0].Length; k++)
        {
S2:         C[i][j] += A[i][k] * B[k][j];
        }
```

Figure 2 - Matrix multiplication for threads.

This example shows the same matrix multiplication as Figure 1, but this time, it does not matter that the statement S2 depends on itself in the internal loop S1, because it is independent in the outer loop (`C[i][j]` is different in every iteration of the outer loop because the variable `j` changes). Therefore, the loop S1 does not depend on any other code and it can be run in parallel threads without conflicts.

Coarse-grained parallelism is more difficult to use, because it can be very unpredictable and it can even slow down the optimized application. The unpredictable results are caused by the unknown behavior of the tasks run in separate threads since the parallelism can be efficient only when the compiler can keep all the threads occupied. The application can be slowed when the initialization of the threads is longer that the time saved by parallel execution, which is usually caused by short tasks.

1.4.3 Instruction-level parallelism

Instruction-level parallelism is based on the fact that several instructions can be run at the same time by the CPU which strongly depends on the architecture of the actual platform. This parallelism is not discussed in this work, because it must be performed by the execution environment and this work concentrates on bytecode that cannot contain any platform dependent code.

1.5 Preparation for parallelization

This work mostly concentrates on the steps necessary to parallelization, because their successful completion decides if it is possible to apply any parallelization algorithms. The most important analysis is dependence testing, because it identifies independent code segments that can be run concurrently. Dependence testing is a very complicated process and the most important complication, it must face, is the analysis of methods called in the optimized method. Calling an unknown method can cause almost anything and it can ruin the entire analysis. There are two possible solutions: inter-procedural analysis and method inlining. Method inlining has been selected because it is easier and more precise, even though it has certain limits that make it less suitable for general application, but this project is only theoretical proof of concept and this simplification does not interfere with specified goals.

1.5.1 Function inlining

Inlining is a process where the body of a called function is integrated into the body of the function that called it which allows more precise analysis of the entire code. This transformation eliminates the necessity to perform inter-procedural analysis but it has some limitations that may be problematic in general application. The main problem is that inlining cannot be used on recursive methods, since it would try to inline the method in itself and it would do it infinitely. Another problem is the fact that the analyzed method might call many methods and they in turn can call other methods and

the final method can become too big and it can even run out of local variables since their number is limited.

Goals for this step are to analyze the possibility and difficulty of inlining in the bytecode and implement it, at least for a restricted version of the programming language. Restrictions should forbid unnecessary or dangerous features of the language, like unsafe code[4] in C#. The inlining is not perfect and it is possible that it might not be able to handle long methods or complex series of calls. To overcome some of these limitations, this work proposes the use of special method attributes that identify methods that should be parallelized (this process is discussed in Appendix B). Methods should not be inlined by default, but the programmer should be able to identify the methods for inlining.

1.5.2 Dependence testing

This is the most difficult part of the entire process, because the optimizer must understand the behavior of an application and it must recognize what variables occupy the same memory locations since changing the value of such variables does influence the content of others. The problem is that the analysis is very difficult in many situations which are very common. For example, if a function has two reference parameters of the same type then it is generally impossible to decide, during static analysis[5], if these parameters represent the same object or not, because the optimizer knows nothing about them.

```
public double[][] Multiply(double[][] A, double[][] B) { }
```

This short example shows a simple interface for matrix multiplication function and the problem here is that the compiler is unable to decide if the parameters reference different objects or not. It depends on the way the function is called, but it can be called multiple times with different parameters, some that are different and some that are the same and even the inter-procedural analysis might not be able to separate the parameters.

This work does not make it its goal to solve dependence testing in general situation, because it is an algorithmically unsolvable problem. Even proving independence between two statements modifying an array indexed by variables can be

[4] Code that contains pointer types and pointer arithmetic, it is considered unverifiable because the unmanaged pointers can cause memory corruption, or security breach.
[5] Static analysis is performed during the compilation or optimization. The opposite if runtime analysis, performed by the virtual machine when the application is running.

problematic. The goal is to examine possible solutions for the most common situations in the context of the selected language. Examined situations should include the following:

1) Simple array accessed using single loop induction variable[6] (vector addition).
2) Multidimensional array accessed using an induction variable (matrix multiplication).
3) Array accessed using an induction variable increased by a constant.
4) Analysis of relations between local variables initialized by a new object and method parameters.
5) Analyze relationship between local variables based on the way they were initialized.

These examples are very common in many applications and solving them, at least theoretically, would prove that it is possible to parallelize some applications with the knowledge of their bytecode.

1.6 Java and C#

This work is aimed at the most common languages with reference semantics, Java and C#. However, the actual analysis and implementation is done only for C# as the main language of the .NET Framework, because its intermediate language is compatible with the international standard Ecma [1] and because the author is more familiar with C#. Important fact is that both languages have a similar intermediate code, even though Java bytecode is not the same as C# CIL, but they share many common features and principles. Their similarities are the reason why it could be possible to apply most of the presented transformations on both languages without drastic changes. Even though both platforms are similar, it is difficult to implement all algorithms for both platforms, because there are many technical details that have to be solved before the implementation can be completed. More detailed comparison is presented in section 2.4. Another reason is that the parallelization of .NET CIL code is less explored and its analysis may open new possibilities for the platform.

This work is meant as a theoretical proof of concept and its goal is to analyze possible parallelization of languages with reference semantics, while the implementation

[6] Induction variable is a variable used control the number of iterations of a loop; it is updated in every iteration. The loop runs until the variable reaches certain value.

for both platforms would most likely provide no additional understanding of the parallelism.

1.7 C# restrictions

It is very complicated to implement inlining and dependence testing for a general application, but the original language can be reasonably restricted to simplify the entire process. Restrictions can be used, because this work is meant as a proof of concept and not as a general implementation.

Restrictions used in this work include the following:

1) Unsafe code cannot be used, it is not necessary for programming in C# and it significantly complicates the dependence testing since it introduces pointers.
2) Exception management and protected blocks are supported, but their use should be limited.
3) Switch construct is not supported, because it can be replaced by a few if/else branches and even the compiler does it in simple cases.
4) Generic types are not supported, because their use leads to a great number of generated types and methods and their analysis can be very difficult

1.8 Basic Assumptions

The first assumption taken by this project is that the optimizer works with a correct code produced by Visual Studio 2010 .NET compiler. This is very important, because any incorrect manipulation with the code prior to the optimization may lead to a corrupted program. The project also expects the program to be written in restricted C#, because many operations depend on standard program structure, like the identification of essential C# control flow constructs. The optimizer can be used on assemblies compiled from other languages, but their structure may prevent the optimization to locate any possibilities for parallelization and using very different languages, like F#, may lead to incorrect code.

There is an important assumption that the stack is empty before and after every statement in the original C# code. This is true, since the stack is used to store temporary values and results and every statement ends when all the values are written to variables and fields. This is true for the modified code as well, with the only exception of inlined methods. The statements of the inlined method may not satisfy this condition, because

there may be something on the stack that has been left there before the method has been called (and inlined). The problem is caused by the fact that the call itself is a part of a statement and the stack is not empty, because that statement has yet to be completed. This is a problematic situation and it is discussed thoroughly the section 4.1.1.3.

1.9 Structure of the thesis

This thesis is divided into multiple chapters that present studied problems and their solutions and the following list describes their content.

- **Related work** – The second chapter contains additional information about studied topics and its main purpose is to introduce the most important terms and facts used in the following chapters. There are presented other works that focus on similar topics.
- **Architecture** – The third chapter discusses the proposed architecture of the optimizer created as part of this work. This work is mostly theoretical and it is possible that some parts of the optimizer will not be implemented, but the architecture specifies their potential function.
- **Code preparation** – The fourth chapter discusses the implementation of method inlining in CIL code.
- **Preliminary code analysis** – The fifth chapter is aimed at the recognition of control flow constructs and variables. This information is crucial for dependence testing and it is important to analyze the code thoroughly.
- **Dependence testing** – The sixth chapter presents many transformations that can be used to prepare the code so it can be analyzed by existing algorithms for dependence testing. This approach is more efficient then creating new algorithms, because the existing algorithms are well tested and documented.
- **Results** – The seventh chapter sums the achieved results of the entire work.
- **Conclusions** – The eighth chapter presents the collected information and it compares the goals of this work and its results.
- **Appendix** – The appendix contains the information about the actual implementation and there is the table of contents of the DVD distributed with this work.

2 Related work

2.1 Automatic parallelization

Automatic parallelization is a process that has been studied for many years and it has been successfully implemented for many programming languages and computer systems. There are many implementations for domain specific languages, which are widely used for specific tasks, like high performance computing and scientific calculations. Many books and papers about general compiler design study dependences for general optimizations and some of them cover even automatic parallelization or vectorization.

There are three types of automatic parallelism, fine-grained parallelism, coarse-grained parallelism and instruction-level parallelism. This work concentrates on the first two types, because they can be used in a bytecode, while the instruction-level parallelism can be exploited only by the execution environment, since it is heavily platform dependent. All the types of parallel execution are described in more detail in section 1.4.

2.1.1 Basic definitions

This section contains definition of the most common terms used in this work and more detailed description can be found in [2].

Definition	Dependence between statements S1 and S2 can exist only if both statements access the same memory location and there is a feasible execution path from S1 to S2.

Definition	Loop-carried dependence between statements S1 and S2 in a loop body can exist only if there exist indices `i` and `j`, such that `i <= j`, and S1 accesses the same memory in iteration `i` as statement S2 in iteration `j`.

Definition	Induction variables value in a loop is defined solely by the number of the actual iteration and the initial value of the variable assigned before the loop.

Definition	Array subscript is an expression used to identify the accessed element in an array. For example in `A[i+1] = 5;` the expression `i+1` is a subscript.

Definition	Aliasing is an effect encountered when a single memory location can be accessed via multiple separate symbols, usually pointers or references.

2.1.2 Problems of automatic parallelization

Automatic parallelization is very difficult to implement and it must be supported by the architecture of the compiler, because it requires a lot of additional information about the optimized application. The compiler must be able to analyze the structure of the application and it must locate dependences between statements before any parallelization can be used. The general structure of compilers is well documented and many books about compiler design introduce basic parallelization.

The most important step before automatic parallelization is dependence testing, responsible for the analysis of relationship between statements, which is thoroughly discussed in [3]. The dependence analysis can be used to perform other advanced optimization, like scheduling and loop optimizations, and that is the main reason why it is studied in [3], because the book presents only very simple low level parallelization.

Vectorization is most commonly used type of automatic parallelism, because it offers very good improvement of speed for many specialized applications and it is more stable than coarse-grained parallelism. Vectorization requires many special transformations that prepare the optimized code for vector instructions and these transformations strongly influence the final efficiency of the compiler. Many transformations enhancing fine-grained parallelism are discussed in [4] in chapter 11 and the book describes most important transformations, like loop exchange or index splitting. Loop exchange swaps loops in a loop nest to remove loop-carried dependences from the internal loop, which can be later parallelized, and index splitting divides a single loop in two that contain less dependences.

While vectorization is an optimization used in scientific for decades, coarse-grained parallelism has been considered to be too expensive and unpredictable. There are some papers that explore coarse-grained parallelism and its application in modern environment. One possible approach is to extract long running threads from pipeline parallelism, which means decomposing pipeline parallelism to multiple threads and executing these threads in parallel. This technique is presented in [5] along with

implemented compiler. Another way is to use speculative parallelization based on transactions [6].

Important book about automatic parallelization is [2], because it concentrates only on parallelization and it presents many transformations necessary for real application of vector instructions or parallel tasks. The book contains detailed analysis of dependences including dependence testing in loops and conditional branches and much attention is devoted to analysis of array subscripts[7], because arrays along with loops represent the best opportunities for parallelization and it is necessary to know what elements are accessed. This thesis is mostly based on this book and the optimization steps presented in section 3.2 are inspired by the transformations discussed there, but this thesis adapts the concepts to the environment of the .NET Framework.

All these books are very thorough and they are usually based on successfully implemented compilers, but their main problem is that they concentrate on old structured language and they do not address challenges presented by OOP[8]. The book [3] is based mostly on FORTRAN and [4] is based on C, while [2] discusses both. Both FORTRAN and C are languages with strong support for automatic parallelization and they are traditionally used for domain specific tasks, like high performance computing. This work tries to find out if it is possible to use similar algorithms and transformations to optimize applications written in C# or Java.

2.2 Parallelization of Java applications

FORTRAN and C are traditional languages for automatic parallelization, but there are many projects trying to implement similar optimizations for Java and similar languages. Java is one of the most common programming languages in the world, but it is more complex than FORTRAN and that is one of the reasons, why is the automatic parallelization usually unavailable.

Java is very effective for common applications, but it is not well suited for high performance computing and multimedia applications [7] and one way to improve that is to use vector instructions available on most modern platforms. Vectorization is a prime example of fine-grained parallelism and it is has been studied for many years. Automatic vectorization is not the only way to use vector instructions in Java applications because it is possible to provide the user with a library containing vector operations, which can be

[7] Expression used to access array elements, be it a single integral variable or a calculated value.
[8] Object oriented programming is based on classes that implement separate parts of application.

translated to vector instructions if they are available. Both approaches are discussed in [7]. Vectorization usually requires many loop transformations, because the best opportunities for vector instructions is array processing in a loop. There are many loop transformations described for languages like FORTRAN, but Java and other reference languages may require a different approach [8].

Main problem of vectorization is that it is strongly platform dependent and Java applications should be portable, but this can be solved with the help of the runtime environment. One solution is to generate SIMD instructions with a retargetable compiler, based on tree-pattern matching [9].

Very interesting work is [10], because it studies automatic parallelization based on the analysis of Java bytecode and that is very close to the techniques presented in this work. It presents tools that can parse and analyze bytecode and optimize the applications even without their source code, but the main presented parallelization technique is method parallelization and the work does not specialize on coarse-grained parallelization.

Parallelization of Java code or bytecode is not the only option. It is possible to implement virtual machine that can execute sequential bytecode and transform it at runtime to take advantage of vector instructions or other forms of parallelism. Even though the most common implementation of Java runtime, the Oracle JRE[9], supports basic automatic parallelization, there are systems that are able to perform more advanced parallelization. One example is system JAPS presented in [11], which is a system able to compile Java based applications and execute them concurrently on a NOW[10] distributed system. The environment has been improved in second version JAPS-II, which is able to exploit even data parallelism, unlike its predecessor, which was able to use only function parallelism. Another runtime environment supporting automatic parallelization is JRPM [12] which uses thread level speculation to parallelize sequential Java programs.

Even though vectorization is most common way to optimize many applications, Java is much better suited for coarse-grained parallelism thanks to its native support of threads and multithreaded execution is completely portable. Threads represent more difficult challenge, because they are more expensive to execute and they can be very unpredictable in their execution time, which introduces a complicated load-balancing.

[9] Java runtime environment is virtual execution machine used to run Java applications.
[10] Network of workstations is a basic distributed system based on simple workstations, usually common PC.

Possible solution is to use method parallelism based on inter-procedural analysis, which produces an instrumented code used by a specialized environment [13].

2.3 Parallelization of C# applications

Microsoft implementation of .NET Framework supports basic automatic parallelization thanks to its TJIT[11], similar to Java JRE and it shares most of the problems encountered in Java applications, because both environments are very similar. Problem of the parallelization implemented by the CLR is the fact that it is done at runtime according to the trace information collected during current execution and it does not perform any advanced dependence testing [14]. There is simply no time for extensive analysis of the code, because the code must be executed as quickly as possible and dependence testing is a complicated process.

It is necessary to transform the C# code or the CIL code to apply automatic parallelization during compilation, but the problem is that the CLR does not support any explicit parallelization constructs. One possible solution is to use meta-data annotations to identify independent code ready for parallelization [15]. This technique had the problem that the execution of an annotated program would require a specific implementation of CLR and that can be difficult to implement. However, this idea can be slightly modified and the attributes can be used by a special source-to-source compiler to automatically generate code that explicitly uses threads [16]. The generated code can be simply compiled with Visual studio and executed by standard .NET.

There is one very interesting option available in .NET, because it supports functional programming and lazy calls. Lazy call is a special type of method call, which is executed only when its result is needed. Lazy calls can be used to generate infinite data structures, because the actual values are generated when they are needed and the application can use safely use finite parts of the infinite data structure. Lazy evaluation is an evaluation strategy commonly used by functional languages. Pure functional programs can be parallelized very simply, because their functions must depend only on their parameters and they cannot modify any global state. Therefore, functional program contains only independent functions that can be run in parallel without any additional transformations. Problem is that the compiler must recognize lazy calls and it must be

[11] Tracing just-in-time compiler is a JIT compiler with the additional tracing that collects the information about executed code, which is used for optimization including parallelization.

able to verify that the called function is really independent, but that can be accomplished according to [17].

The automatic parallelization is not the only way to develop parallel applications in modern C#, because .NET offers a wide support for parallelization, aside from basic threads. There are many libraries for explicit parallel programming, including parallel loops, library for task parallelism and parallel LINQ[12] [18].

Other possibilities are described in section 2.6.

2.4 Java and .NET comparison

The intention of this work is to examine the possible application of automatic parallelization on applications written using modern languages with reference semantics, Java and C# most prominently, but it proved too difficult to implement the discussed algorithms for both platforms. Therefore, only one language has been chosen for deeper analysis. C# and the .NET platform have been selected, because the structure of CLI[13] is based on an international standard Ecma [1] and the author is much more familiar with the .NET platform then with Java. There are many useful tools available for analysis and verification of .NET applications that proved to be absolutely necessary for low-level tasks such as inlining.

Even though only .NET have been selected, it may be possible to modify proposed solutions and algorithms to work on the other platform, because both platforms are based on similar concepts and most of the work may be used for both languages and their bytecode.

2.4.1 Java and C# virtual machine

This section is dedicated to the exploration of similarities between Java bytecode and .NET CIL[14], these similarities are very important, because the solutions are presented for .NET and their potential application on Java would require that both platforms share at least some basic concepts.

[12] Language-Integrated Query is a set of methods that implement query similar to SQL; the library contains special syntax for the queries.
[13] Common language infrastructure is a software platform specified by an Ecma standard [1] and its most used implementation is Microsoft .NET framework.
[14] Common intermediate language is an intermediate language used in platforms compatible with the standard Ecma-335 [1].

Both JVM[15] and .NET CLR[16] are stack-based virtual machines designed to execute applications compiled to appropriate intermediate code, bytecode for Java and MSIL[17]/CIL for .NET. The execution stack is used to pass parameters and return values from one instruction to another and completed statements usually leave the stack empty because the final value is stored in local variable or it is passed as a parameter to a method. Aside from the execution stack, both platforms use a call stack to handle method calls and exception management. Methods have access to two kinds of memory, the execution stack and local variables, which can be accessed only by the method. .NET methods have parameters as a separate type of memory, but they are used almost the same way as local variables and they can be considered to be a special type of local variables.

2.4.2 Intermediate code comparison

The instruction sets of JVM and CLR are very similar as well, since both use load/store instructions to move values between the memory and the stack and all the instructions, that perform some actual computation, use the execution stack to obtain their parameters and they return results back on the stack. The next example shows a simple method that is later compiled to both bytecode and MSIL to present similarities in the instructions and code structure.

```
static int factorial(int n)
{
    int res;
    for (res = 1; n > 0; n--) res = res * n;
    return res;
}
```

Figure 3 - Sample code used to produce bytecode and CIL

The source code is the same for both java and C# and it has been taken from [19] as well as the Java bytecode presented next.

[15] Java virtual machine is an execution environment able to run programs written in Java and many other languages compiled to bytecode.
[16] Common language runtime is en execution engine able to run applications compiled to an intermediate language compatible with Ecma CIL.
[17] Microsoft intermediate language is a company specific implementation of Ecma CIL.

```
method static int factorial(int), 2 registers, 2 stack slots
        0: iconst_1      // push the integer constant 1
        1: istore_1      // store it in register 1 (the res variable)
        2: iload_0       // push register 0 (the n parameter)
        3: ifle 14       // if negative or null, go to PC 14
        6: iload_1       // push register 1 (res)
        7: iload_0       // push register 0 (n)
        8: imul          // multiply the two integers at top of stack
        9: istore_1      // pop result and store it in register 1
       10: iinc 0, -1    // decrement register 0 (n) by 1
       11: goto 2        // go to PC 2
       14: iload_1       // load register 1 (res)
       15: ireturn       // return its value to caller
```

Figure 4 - Example bytecode generated from the code shown in Figure 3

First, it is necessary to note that local variables are called registers in the Java bytecode. The bytecode shows the use of stack to perform multiplication (blue colored instructions), first both operands are loaded, then the result is calculated and the result is stored in a local variable (register). The code contains a conditional jump (`3: ifle 14`) and an unconditional jump (`11: goto 2`) to implement the for-loop and the code is closed by the `ireturn` instruction that terminates the method and leaves its result on the stack.

```
.method private hidebysig static int32  factorial(int32 n) cil managed
{
  // Code size       19 (0x13)
  .maxstack  2
  .locals init ([0] int32 res)
  IL_0000:  ldc.i4.1              // load constant 1 on the stack
  IL_0001:  stloc.0               // store constant in local variable
  IL_0002:  br.s       IL_000d    // jump at the start of loop condition
  IL_0004:  ldloc.0               // load local variable
  IL_0005:  ldarg.0               // load parameter value
  IL_0006:  mul                   // multiply argument with loop index
  IL_0007:  stloc.0               // store the result of multiplication
  IL_0008:  ldarg.0               // load parameter
  IL_0009:  ldc.i4.1              // load constant 1
  IL_000a:  sub                   // subtract the (parameter - 1)
  IL_000b:  starg.s    n          // store the result in the parameter
  IL_000d:  ldarg.0               // load the parameter
  IL_000e:  ldc.i4.0              // load constant 0
  IL_000f:  bgt.s      IL_0004    // jump back if (argument > 0)
  IL_0011:  ldloc.0               // load result on the stack
  IL_0012:  ret                   // return the control to the caller
} // end of method Program::factorial
```

Figure 5 - MSIL code generated from the code shown in Figure 3

This is the MSIL code generated from the exact same code as the byte code in the Figure 4; it was compiled by Visual studio 2010 for .NET 4.0 and decompiled by the ILDasm utility. The code bears many similarities to the bytecode shown in Figure 4, for example, the multiplication is identical (blue colored instructions). The code shows how

the code works with method parameters (red colored instructions); they are used as local variables, which is similar to Java that accesses parameters as local variables. Even MSIL leaves the return value on the stack prior to calling the `ret` instructions. Both code samples contain almost the same number of instructions.

2.4.3 Summary

The example codes presented in Figure 4 and Figure 5 show that the structure of both intermediate codes is very similar, they both use stack to pass values between instructions and even the instructions are very similar. Detailed description of JVM instruction set can found in [20] and the CIL instructions are specified by Ecma standard [1]. This similarity implies that most of the algorithms designed for the MSIL/CIL code can be ported for the Java bytecode without major changes and the paper is mostly valid for JVM, even if it is presented written the .NET platform. Chapters and sections that discuss topics valid only for the .NET platform are clearly marked and they can be skipped by readers interested only in Java.

2.5 .NET execution environment

The .NET virtual machine called CLR is a stack-based execution environment that uses the execution stack to pass values between consecutive instructions. It is able to execute applications compiled to an intermediate code compatible with CIL, like the MSIL used by applications written in C#. It is very similar to the Load/store architectures used by many RISC[18] processors, like the MIPS[19] processor family, because only load/store instruction can access local variables or method parameters and all other instructions work only with the values pushed on the stack. Most common structure is that load instructions prepare values on the stack, then some arithmetic instruction processes these values and its result is stored in memory or used as input for another calculation. The stack remains empty after a statement is completed, because all the values are stored in memory.

The code is divided into classes and their methods that contain all the actual instructions and the application starts by executing the static main function. Classes are stored in modules called assemblies and one application can contain many assemblies

[18] Reduced instruction set computing is a CPU design strategy based on the insight that simplified instructions can provide higher performance.
[19] MIPS is a reduced instruction set computer (RISC) instruction set architecture (ISA) developed by MIPS Technologies.

where one assembly can use classes defined in another as along as it is able to locate the actual assembly file. Assemblies are stored as a specific version of DLL[20] with the exception of the main file, which is an executable file.

Detailed description of the infrastructure and principles of CLI compatible platforms, which include .NET framework, can be found in Partition I in [1].

2.5.1 Basic instructions

This section describes some basic instructions to allow better understanding of the following examples, because the MSIL code is used to present many concepts and operations discussed in the following text. Most instructions exist in many versions that share the same basic mnemonic name and the version is specified by it suffix, but to understand the code, it is necessary to know only the base name. The instructions use simple mnemonics to identify the type of object they work with, the following list shows the most common:

- `loc` – local variables
- `arg` – method parameters
- `fld`, `sfld` – class fields and static fields
- `elem` – array elements
- `c`, `str` – numeric or string constant

The next list describes basic instruction groups used in the examples presented in this paper.

- `ldloc`, `ldarg`, `ldc`, `ldstr`, `ldfld`, `ldelem` – loading instructions used to read memory available to the analyzed method; this is the only way to read memory, excluding unsafe code that is not allowed in this paper. Values are pushed on the execution stack and later used by other instructions.
- `ldloca`, `ldarga`, `ldflda`, `ldelema` – instructions used to access the memory address of specified variables, which can be later used for indirect writing or reading, which is very common in the code produce by Visual studio.
- `stloc`, `starg`, `stfld`, `stelem` – storing instructions used to modify local variables, parameters, fields and arrays.

[20] A dynamic-link library is an executable file that acts as a shared library of functions.

- `br`, `bgt`, `ble` … – conditional and unconditional jumps, `br` is unconditional and the others are conditional and they change jump, if the values on the stack satisfy their condition.
- `call`, `calli`, `callvirt` – instructions for calling different types of methods, `callvirt` is used to call virtual methods
- `mul`, `add`, `sub` … - standard arithmetic instructions

This list contains most common instructions used in examples in the following chapters and their detailed description can be found in Partition III in [1].

2.6 Parallel execution in .NET Framework

The actual implementation of automatic parallelism is problematic in .NET applications, because there are no vector instructions in CIL and the introduction of threads to sequential code requires drastic changes to the structure of the code, because the parallel tasks must be exported as new methods and then they can be executed in separate threads.

Possible solution is to use code instrumentation to specify which code can be parallelized and the execution environment then decides what will be parallelized and how [15]. This is a good approach but it requires a special execution environment, because standard .NET CLR does not support any parallelization meta-data information, and that can be problematic because implementation of an entire CLR is very difficult and the new implementation would have to be ported to all required platforms. One positive fact is that meta-data can be ignored by CLR and the program can be executed even under standard implementation of .NET, even though it would run without any parallelism.

Another possibility is to use a special library that provides vector operations as simple functions. If the actual platform supports then these functions would use them and if the platform offers no support for vector parallelism then they would implement the operation using standard CIL instructions. These functions can be easily inlined to increase execution speed. This solution would most likely require new or modified CLR, because the CLI instruction set does not support vector instructions, and that makes it impractical for general use.

One of the best solutions is to use method parallelism along with a thread pool, because it does not require major restructuring of the code and it does not need a special implementation of CLR. The thread pool and its management can be implemented in C#

and its CIL code can be inserted in the optimized application without changing its existing code, with the exception of some initialization that must be added to the main function. The actual parallelization would require only the analysis of the dependences between separate methods and independent methods can be executed in parallel threads obtained from the pool. Main problem of this method is that is can parallelize only entire methods and it is not able to parallelize loops that do not call methods. Another problem is that it would require runtime load-balancing, since it is not possible to determine method run time during compilation.

The actual parallelization technique is not discussed in much detail in this work because its main motivation is to use C# as a front-end for some parallel system. The CIL produced by the compiler / optimizer would be transformed to a specific language that supports parallelism natively and this problem would not be encountered at all.

2.7 Intermediate code

The optimizations and transformations cannot be performed on the raw bytecode, because it is designed for quick execution and it is not well suited for transformation. Therefore, it is necessary to parse the actual byte code to an intermediate code that can be easier transformed and analyzed. This work uses the Cecil library to parse the code and the internal format produced by the library is used as an intermediate code. This approach has been chosen, because Cecil is a well-known library and its format is well suited for transformations. Another benefit is the fact that it is much simpler for other developers to extend the project without the necessity to learn some obscure intermediate language. The only problem of the Cecil library is the lack of detailed documentation, which caused some problems during the implementation.

3 Architecture

This chapter describes the general structure of the optimizer designed as part of this work. It is not a thorough documentation of software architecture, because this is a theoretical work and the optimizer is designed as a theoretical concept. The architecture of the actual implementation is documented in Enterprise architect and it can be found in the Architecture module of the project, the exact location is specified in Appendix A.

3.1 Structure of the optimizer

The structure of the optimizer closely follows the series of transformations and analyses used to prepare the code for parallelization. Following sections discuss every step and its problems. These steps prepare information for automatic parallelization and their order is very important, because every step depends on the previous steps, with the exception of transformations that are applied to simplify next steps.

Each step provides input for the next step and every step is implemented in a separate module, because modules can be later replaced by better implementation. The construction is similar to a network stack, because each module provides input for the next and it can communicate only with its neighbors. The optimization steps and their order are as follows:

1) **Code preparation**
2) Preliminary transformations
3) **Preliminary code analysis**
4) **Dependence testing**
5) Transformations enhancing parallelism
6) Automatic parallelization

This work concentrates on the steps that are absolutely necessary for parallelization of .NET CIL because the other steps are used to simplify the analysis and their application is not a necessity for simple applications. The necessary steps are Code preparation, Preliminary code analysis and Dependence testing. The other steps are discussed in some detail in this chapter, unlike the main steps which are discussed in separate chapters, and they are described here only briefly. These steps are described, because they are part of the architecture and they would be necessary for a complete implementation to be efficient.

The main steps are Code preparation, Preliminary analysis and Dependence testing. Automatic parallelization is very important as well, but this work tries to prove that it is possible to gather required information and the actual application is discussed mainly in section 2.6. This work solves problems specific for languages with reference semantics and the parallelization process might be similar to any other language, once the dependences are identified.

Aside from the main modules, the optimizer must contain other modules that manage communication with users or configuration. They provide a general framework for the optimizer and they are not discussed in this chapter, because they do not implement any optimizations. These modules are described in Appendix B.

3.2 Optimization steps

The optimization steps presented in the previous section have been inspired by the transformations presented in [2] and their order is defined by the information require in each step. The order can be reconstructed from the last step according to the input of each step. Parallelization requires the knowledge of dependences, because they identify the code that can be executed concurrently. Dependence testing must understand the structure of the code and it must be able to identify separate memory locations and that information is provided by the analyses in the second step. The entire analysis must know as much as possible about the code which is the main motivation for inlining. These steps represent only one possible solution, but they should provide the information necessary for automatic parallelization.

Algorithms for automatic parallelization are not too complicated, but they require the knowledge of dependences between statements or instructions and dependence testing, on the other hand, is an extremely complicated problem that needs to know a lot about the code, before it can be even attempted. Because of that, there are many steps, which must be completed, before performing the dependence testing and ultimately, the automatic parallelization. Not all these steps are absolutely necessary but they must be performed in the specified order since the next steps might not work otherwise.

3.2.1 Code preparation

Code preparation module provides two operations that should be done before the code can be analyzed. First is a transformation called function inlining[21] and second is called code verification (this is not the verification of CIL code performed by the execution environment). Inlining means that the body of a called function is copied into the body of the function that called it, this allows for more precise analysis and understanding of the code. Code verification simply checks if the code does not contain any instructions or constructs, which are not supported by the optimizer.

3.2.2 Preliminary transformations

Preliminary transformations are basic optimization performed before dependence testing, because they simplify the code and the analysis can be done faster. These transformations are very general and they may be even performed by the compiler, which created the analyzed bytecode. They include transformations, like dead code removal (deletion of unreachable code) or constant propagation (removal of variables with constant values). The book [2] contains many examples of useful transformations in the chapter 4.

Next sections discuss two important transformations, which should be implemented by a complete optimizer, but they are not implemented in this project, because they are not necessary. All of them are explained in more detail in [2], in Chapter 4, and in [3]. Dead code analysis and elimination is explained very thoroughly in [4], in Chapter 8. These optimizations are very common and they are well documented in many scientific books and papers, but this section discusses their possible implementation in the .NET environment.

3.2.2.1 Constant propagation

Dependence testing examines the relationship between separate memory accesses and it is more difficult when there are many variables mixed together and constant propagation is a process that can be used to reduce the number of variables, mainly those introduced by inlining. Constant propagation locates instructions that read a variable, which has a known constant value, and it replaces those instructions with instructions that load the constant directly, thus instead of reading the value from a variable, it is

[21] Function inlining is a process where the body of a function is included in the body of the function that called it and the call instruction is removed. The resulting code produces the same results, but it may be more efficient and I can be analyzed more precisely.

loaded directly and the variable is not accessed in the statement. This transformation can remove many variable accesses and thus it can remove a lot of dependences, because unlike variable, constant is stored in the program code and then it is loaded to the stack (at least in .NET).

Sadly, this optimization is not implemented in the Visual studio compiler for C#. One of the possible reasons it the fact that instructions loading constants can have more than 1 byte and the constant have to be moved from the instruction on the stack, which is usually implemented in registers, while instructions reading local variables are 1 byte long and the variables are usually allocated in a register. The optimization is discussed thoroughly in Chapter 4 in [2].

3.2.2.2 Dead code elimination

Definition	Dead code is a code that does not contribute to the final value of any output variable or it does not produce any visible output (like writing to a console).

Basically, dead code is a code fragment that can be removed from the application without changing its results or behavior. Removing a useless code can help dependence testing, because it does not have to test code that does not do anything or it will never be executed

This transformation can be very complicated and it is strongly platform dependent, but luckily, the Visual studio C# compiler contains a fair implementation, that can remove unreachable code as well as code that does not contribute to any result. The example shows how the compiler transforms the following method, which contains three useless statements.

```
    static int deadCode()
    {
        var y = 10;
S1:     int x = y * 2;
        return y;
S2:     y = x * x;
S3:     return x + y;
    }
```

Statements S1, S2 and S3 are useless, because S1 does not change the returned value and S2, S3 will be never executed, because there is a return statement just before them. The final code generated by the C# compiler is shown in the next example.

```
IL_0000:  ldc.i4.s   10
IL_0002:  stloc.0
IL_0003:  ldloc.0
IL_0004:  ret
```

The final code contains only the initialization of a single local variable and its value is then returned as the result of the method. All three useless statements have been removed.

3.2.2.3 Summary

This step is not implemented, because it is not necessary for parallelization, but it would be important for the efficiency of the final optimizer and that is the reason why it is discussed in this section.

3.2.3 Preliminary code analysis

Preliminary analysis is responsible for recognition of basic code constructs and variables and its results are later used in dependence testing. It is necessary to locate all control flow constructs and mainly loops, because they offer the best opportunities for parallelization and the execution order strongly influence relations between parts of the code. On the other hand, variable recognition must locate all the variables used in the code and categorize them according to their type and usage.

3.2.4 Dependence testing

Dependence testing is a process that located dependences between statements (or instructions) that can be used to locate code which can be executed in parallel. This is the most difficult part of the entire parallelization process, because the optimizer must understand the behavior of the application and it must recognize what variables occupy separate memory locations since changing the value of such variables does influence the value of the others. The problem is that this is impossible to do in many cases and that these cases are very common, for example, if a function has two reference parameters of the same type then it is generally impossible to decide during static analysis[22] if these parameters represent the same object or not, because the optimizer knows nothing about them. One possible solution is an inter-procedural analysis, but the method can be called many times and its caller can use its own parameters and that adds another method that

[22] Static analysis is performed during the compilation or optimization. The opposite is runtime analysis, performed by the virtual machine when the application is running.

must be analyzed. Inter-procedural analysis can solve some situations, but it is not a universal solution.

3.2.5 Transformations enhancing parallelism

This step is very similar to preliminary transformations (section 3.2.2), but the transformations used here require very deep knowledge of the code and they can significantly improve the code for later optimization. The transformations are usually used as a response to specific problem and the quality of the compiler is strongly influenced by the number of transformations it can perform, because every transformation allows it to optimize different type of application.

This project tries to solve only simple cases and it does not have to implement any of these transformations do that, but the final optimizer would have to implement a lot of these transformations to be efficient and some transformations are presented in the next chapter to elucidate the function of this step.

3.2.5.1 Transformations

It is necessary to select what type of parallelism would be used to optimize the application before any transformations can be used because all these transformations are designed for specific type of parallelism.

Fine-grained parallelism is mostly improved by eliminating loop carried dependences from internal loops in a nest, because the internal loop can be transformed to use vector instructions. Examples of these transformations are loop interchange, scalar expansion or loop skewing. Loop interchange swaps loops in a loop nest to remove loop-carried dependences from the internal loop, which can be vectorized, while the dependence is carried by some outer loop in the nest. Scalar expansion transforms a scalar variable, used in a loop, to a vector that can be processed by vector instruction and the vector is transformed back to a scalar after the loop ends. Loop skewing is an advanced transformation that modifies the loop induction variables to change the iterations space, which can be divided into parallel iterations. All these transformations are explained in detail in chapter 5 in [2].

Transformations that improve coarse-grained parallelism are more complicated, because it is more difficult to use and it can be very unpredictable. Simple transformations are loop distribution and loop interchange (different version then in the case of fine-grained parallelism). Loop distribution splits a complex loop into multiple simpler loops that can be run in separate thread. Loop interchange used for parallel

threads is similar to the interchange used for vectorization, but in this case it removes the dependences carried by the outer loops, because the internal loops can be run in separate threads regardless of the loop-carried dependences. Many transformations for coarse-grained parallelism are discussed in chapter 6 in [2].

3.2.6 Automatic parallelization

There are basically two types of parallel execution discussed in this work, fine-grained parallelism (vectorization) and coarse-grained parallelism (threads or processes). Contemporary computers are usually able to combine both these approaches together using multiple cores for multiple threads and SIMD instructions for vectorization. More precise analysis of available parallelization techniques is in section 1.4 and in section 2.6.

This is the last step and it is the ultimate goal of this work, but it is discussed only theoretically because the main goal is to find out if it is even possible to perform some automatic parallelization and the actual implementation is very complicated.

4 Code preparation

This step includes method inlining and code verification. Both are very low level operations that must be performed, before the code is analyzed, to make sure the actual analysis is as simple as possible. The following steps can work with the resulting code safely and it allows the following analyses to be much more precise and detailed.

4.1 Inlining

The main idea of method inlining is to remove unnecessary dependencies caused by calling an unknown method that can change almost anything. The called method is included in the calling method and the resulting code can be analyzed more precisely, since it is now clear how the parameters are used and what static data is changed, if any. Even though the inlined method can be analyzed independently, the flow of data can only be accurately analyzed after it is inlined, since then it is clear where are the parameters obtained and where are they used and how.

```
    for (int i = 0; i < arr.Length; i++)
    {
S1:     func(arr, i);
    }

    void func(int[] arr, int i)
    {
S2:     arr[i] = arr[i] * 2;
    }
```

Figure 6 - Simple inlining example.

Inlining the function `func` in the loop would replace the statement S1 with the statement S2 and then the loop can be easily optimized using threads or vector instructions, since the loop simply multiplies each element of an array by 2 and it does no modify any element more than once. This kind of analysis requires either inter-procedural analysis or inlining. Inlining has been chosen, because it is easier to implement and it is sufficient, with respect to the motivation.

Second important effect is that the inlined code can be optimized along with the code of the calling method and that can improve performance or it can even make certain optimizations applicable.

```
for (int i = 0; i < arr.Length; i++)
{
    func(arr, i);
}

private void func(int[] arr, int i)
{
    for (int j = 0; j < i; j++)
    {
        arr[j] = arr[j] * 2;
    }
}
```

Figure 7 - More complex inlining example.

This is a more complicated version of the previous code and there is no parallelization available before the function `func` is inlined, because the calling method contains a function call and we do not know what it can do. The `func` cannot be optimized because we do not know anything about the arguments and one is used to control the main loop. The situation becomes simpler after inlining.

```
    for (int i = 0; i < arr.Length; i++)
    {
        for (int j = 0; j < i; j++)
        {
S1:         arr[j] = arr[j] * 2;
        }
    }
```

Figure 8 - Inlined loop nest.

After inlining it can be proven that there is no dependence carried by the internal loop involving statement S1 and the internal loop can be parallelized.

One positive side effect of code inlining is the analysis of the used types and methods. All the following transformations can work only with the body of the calling method and they can rely only on its instructions and data. The inlining process makes sure all the used types, methods and fields are imported to the optimized module and they can be used safely, this is necessary for the CLR to work correctly.

Problem is that not all the methods are available for inlining, like library functions written in native code, like Windows API functions. These functions or methods are left in the code and they work like dependence barriers, since everything depends on them and they in turn depend on everything because their code cannot be analyzed. This problem is not important for this work, because the optimization of libraries is not its goal. This work concentrates on parallelization of calculations.

4.1.1 Inlining in depth

Following sections describe the inlining process in more detail. It is important to point out that all the operations are necessary and must be executed in exact order, because they depend closely on one another. Most of them are not .NET dependent, but there are transformations that may not be necessary on other platforms, they are clearly marked. The examples present the transformations on the MSIL code produced by C# compiler that is part of Visual studio 2010.

To inline a method completely and safely, it is necessary to perform following steps in this exact order:

1) Jump target backup
2) Parameter and call elimination
3) Protected block patch
4) Local variable reference patch
5) Parameter reference patch
6) Code inlining
7) Entry and exit point patch
8) Jump patch
9) Protected block registration
10) Maximal stack size calculation

4.1.1.1 Jump target backup

The very first step is to save important information about jump instructions. The target instruction must be detected for all jumps in both calling and called method and the jump distance is then stored as the number of instructions from the jump to its target (negative number means backward jump). This is necessary, because the inlining changes the instruction stream and some jumps can become invalid. The eighth step restores invalid jumps according to the information collected in this step; the process is explained in the section 4.1.1.8.

This problem is more complicated in the MSIL by the fact that all the jump instructions identify their target distance by the number of bytes; they do not use the number of instructions because they vary in size and the necessary calculation would hurt performance. Luckily, this problem is solved by almost every reflection library available for .NET of Mono platform.

There is the same problem with the protected blocks of the calling method that has to be addressed as well, because the protected blocks that span over the call

instruction has to be extended to correctly include all the instructions added during the inlining process. This problem can be solved the same way as the jumps; all the dimensions of all the protected blocks are stored before any code is changed and they are updated after the code is complete. The update is described in the section 4.1.1.8.

4.1.1.2 Parameter and call elimination

Next step of the inlining process is parameter elimination. It means that the parameters, stored on the stack prior to the call, must be stored to local variables, this way the inlined code can access them the same way as local variables and it can work with them as if they were parameters. This approach is a bit crude, but the parameters can be placed on the stack in many ways, they can be a result of an inlined method, and it is not possible to simply replace parameter load with the instruction that placed it on the stack before the call, at least not always. Some optimization may be possible with stack simulator because it can identify the point where a specific parameter is placed on the stack and thus the specific instruction can directly replace parameter load without the need to create a new local variable, but this instruction must be a simple load of either local variable or a parameter.

```
var = Called(2.4f, "hello", fltVar, 2.4f, 2.4f, s);
```

This statement is translated by the .NET compiler to the following code.

```
IL_0041:  ldarg.0
IL_0042:  ldc.r4   2.4000001
IL_0047:  ldstr    "hello"
IL_004c:  ldarg.0
IL_004d:  ldfld    float32 ParallaXExample1.InlinerTest::fltVar
IL_005c:  ldloc.1
IL_005d:  call     instance int32 InlinerTest::Called(
                                                      float32,
                                                      string,
                                                      float32,
                                                      valuetype ParallaXExample1.str)
IL_0062:  stloc.0
```

Figure 9 - Method call in CIL.

The instructions responsible for parameter preparation are colored cyan.

```
IL_0096:  ldarg.0
IL_0097:  ldc.r4      2.4000001
IL_009c:  ldstr       "hello"
IL_00a1:  ldarg.0
IL_00a2:  ldfld       float32 ParallaXExample1.InlinerTest::fltVar
IL_00b1:  ldloc.1
IL_00b2:  stloc.s     V_25
IL_00b4:  stloc.s     V_26
IL_00b6:  stloc.s     V_27
IL_00b8:  stloc.s     V_28
IL_00ba:  stloc.s     V_29
```

Figure 10 - Parameters stored to local variables before inlining.

The green colored instructions store the parameter values from the stack to local variables. This process can be wasteful in cases like this, but most methods have just a few parameters. Not to mention that the actual call instruction stores the parameter values as well.

The call instruction is removed afterwards, because it is no longer needed. Finally, two lists are created for simple removal of parameter references - load replacements, store replacements. These lists are used to eliminate parameter references, system simply calculates the index of the referenced parameter and then it takes replacement instruction from the list. The lists contain load / store instructions and they reference the appropriate local variable so the elimination process does not have to know which variable contains which parameter and it allows the parameter elimination process to optimize and load the parameter value from some other place by providing a different replacement. This process is discussed in the section 4.1.1.5.

```
// Load replacements
  ldloc.s     V_25
  ldloc.s     V_26
  ldloc.s     V_27
  ldloc.s     V_28
  ldloc.s     V_29
// Store replacements
  stloc.s     V_25
  stloc.s     V_26
  stloc.s     V_27
  stloc.s     V_28
  stloc.s     V_29
```

Figure 11 - Lists of replacements for access to parameters.

These are the two lists that would be created for the previous example after this step is completed. When a parameter zero is read by the instruction Ldarg.0 in the inlined method, this instruction can be simply replaced by the first instruction in the load replacement list, based on the parameter index.

4.1.1.3 Protected block patch (.NET specific transformation)

After the call instruction is eliminated and all parameters are stored in local variables, it is finally possible to find out if there is something remaining on the stack. This can be a problem, if the inlined method contains protected blocks, because the program can enter protected blocks with non-empty stack and that is forbidden. Therefore these remaining values must be stored in temporary local variables before the call and then they must be restored after the call. These values are restored to the stack after the inlined method is completed; the only problem that remains is to return these values under return value of the inlined method. The return value is stored a local variable and then it is placed on the top of the stack, once it is restored to the state it has been just before the call. A stack simulator is necessary to analyze the stack and it must be used to find out if there is something remaining on the stack. The situation when there is a non-empty stack prior to a call may be very problematic and the following example shows that such a situation is very common and it must be addressed.

```
var = 1 + ExceptionCall(var, 5.6f);
```

In this example, an integral number will remain on the stack when the inlined code of the method `ExceptionCall` is invoked. The reason is that the compiler loads the constant and then it calls the method, which must provide the other value for the `add` instruction called last. Using a method call as a parameter has the same effect as shown in the following example.

```
var = ExceptionCall(ExceptionCall(var, 8.4f), 5.6f);
```

However, this step has to be performed only when the called method contains at least one protected block, because that is the only time the stack can be non-empty prior to a protected block.

Observation If we assume that the stack is consistent in the main method as well as in the inlined method then the stack can remain non-empty before entering a protected block is in the code of an inlined method.

Discussion All the protected blocks in the calling method are correct based on the assumption. Called method is not changed; therefore, there cannot be error in it. Only place where a protected block is added to a function is inlining and it is the only place where can the stack state create an error. At the same time, if the

inlined method does not contain any protected blocks then there cannot be any errors caused by the stack.

According to the observation, the inlined method is the only place where the stack must be cleared and it is necessary only when there are any protected blocks in the inlined method. Using the local variables is a correct solution for this problem, it may look wasteful, but it is necessary to empty the stack and this is the only option, because local variables are the only private memory available to a method.

4.1.1.4 Local variable reference patch

Next, it is necessary to add local variables of the called method to the calling method and redirect all the references to local variables in the inlined method code to these new variables. The code of the inlined method must be cloned at this point, since the original code must not be affected by any changes made during the inlining process. The redirection is applied to the cloned code, which will be later copied directly to code of the calling method.

```
// Original variables of the calling method
 .locals init ([0] int32 var
  )
// Original code of the inlined method
  IL_0000:  ldc.i4.0
  IL_0001:  stloc.0
  IL_0002:  ldloc.0
  IL_0003:  ret
```

Figure 12 - Local variables accessed in an inlined method.

This example shows local variables of the calling method and the code of the inlined method and the next shows the code after the patch.

```
// Modified variables of the calling method
 .locals init ([0] int32 var,
               [1] int32 newVar
  )
// Patched code of the inlined method
  IL_0000:  ldc.i4.0
  IL_0001:  stloc.s newVar
  IL_0003:  ldloc.s newVar
  IL_0005:  ret
```

Figure 13 - Local variables after patch.

The basic store and load instructions have been replaced with more general version and they target the new variable created in the calling method. The general

version is used because the shortest instructions (i.e. `stloc.0`) can target only the first four variables which are usually used by the main method (this example presents a very short method). The implementation is able to use both versions of these instructions (short and long) and it generates the short version if it is able to access the variable.

4.1.1.5 Parameter reference patch

The inlined code still contains references to the parameters of the inlined method and they must be redirected to the local variables created to store the parameter values in the step two (section 4.1.1.2). This can only be done after all the local variable references have been patched, since the previous transformation would not know which references must be patched.

All parameter references are replaced by references to the local variables created in the step two, this replacement maintains the reference form (by-value or by-reference). The replacement does not work directly with local variables or their indices, instead it gets list of replacement instructions and uses them, so it is possible to pass it any instructions that can obtain given parameter. This way, it is possible to provide different replacements to optimize the inlining process for special cases. The replacement lists are created in the second step and their construction and function is explained in the section 4.1.1.2.

```
// All three lists are passed as parameters created in the previous steps
List LoadReplacements;
List AddressLoadReplacements;
List StoreReplacements;
Foreach(instruction in ClonedInstructions)
{
   Int index = 0;
   If(IsParameterReference(intruction))
      index = GetParameterIndex(instruction);  // must work even for Ldarg.0 etc.

   If(IsParameterLoad(instruction))
   {
      ClonedInstructions .ReplaceInstruction(instruction, LoadReplacements[index]);
   }
  Else  If(IsParameterAddressLoad(instruction))
   {
      ClonedInstructions .ReplaceInstruction (instruction, AddressLoadReplacements [index]);
   }
   Else If(IsParameterStore(instruction))
   {
      ClonedInstructions .ReplaceInstruction (instruction, StoreReplacements [index]);
   }
}
```

Figure 14 – Algorithm used to patch parameter references

4.1.1.6 *Code inlining*

The code of called method is now completely modified and it does not reference any private data of the called method, it references only local variables of the calling method and static data, because all the parameter and local variable references have been patched. The code is simply copied in the calling method, in the place where the call instruction used to be and result of this operation is an interval that indicates where the code has been inserted (index of the first and the last inserted instruction). This example shows inlined code colored green.

```
// Code before inlining
  IL_0003:  brtrue.s   IL_000d
  IL_0005:  ldarg.0
  IL_0006:  ldloca.s   var
  IL_0008:  call       instance void InlinerTest::RefCall(int32&)
  IL_000d:  ldc.i4.2

// Code after inlining
  IL_0003:  brtrue     IL_001c
  IL_0008:  ldarg.0
  IL_0009:  ldloca.s   V_0
  IL_000b:  stloc.s    V_9
  IL_000d:  stloc.s    V_10
  IL_000f:  ldloc.s    V_9
  IL_0011:  dup
  IL_0012:  ldind.i4
  IL_0013:  ldc.i4.s   10
  IL_0015:  add
  IL_0016:  stind.i4
  IL_0017:  br         IL_001c
  IL_001c:  ldc.i4.2
```

Figure 15 - Code inlined in a method.

4.1.1.7 *Entry and exit point patch*

Entry point to the inlined method is the place where it has been inlined and it does not need any special attention. The exit point is any RET instruction included in the inlined method, all the RET instruction are therefore replaced with jump instructions that transfer the control to the instruction that is immediately after the inlined code.

4.1.1.8 *Jump patch*

This is a very important step, because without it the code would be completely broken, since most of the jump would target invalid instructions and their execution would cause a fatal error. This step relies on the information collected in the first step, discussed in the section 4.1.1.1 and it finishes all the code transformations, since after

this is done, the code should be valid and equivalent to the original. Only thing that remains are the protected block discussed in the next section.

Restoring jumps in the inlined code is simple, because it just requires to find the instruction on the position indicated by the stored offset and register it as the target of the jump, this is a correct operation since no instructions has been added to the inserted code, some may have been replaced. Therefore, the jumps distance is still valid it just has to be added to the new position of the jump in the code and the new target can be registered.

Restore the main method jumps can be a little trickier, because the inlined method code has been inserted in it. Therefore, all the jumps that jump over the inlined code have extended offset according to the number of added instruction. Inserted code includes instruction of the inlined method, storage of the parameters and management of the stack required by the protected block patch. After the offset is corrected, then the target can be recovered and the jump can be patched and everything is correct again.

```
IL_0000:  ldc.i4.0
IL_0001:  stloc.0
IL_0002:  ldloc.0
IL_0003:  brtrue.s   IL_000d // Jump distance is 4 instructions
IL_0005:  ldarg.0
IL_0006:  ldloca.s   var
IL_0008:  call       instance void InlinerTest::RefCall(int32&)
IL_000d:  ldc.i4.2
```

Figure 16 - Jump before patch.

The example shows a jump instruction with a distance calculated in the first step as four instructions forward. The next example shows the same code fragment after the `RefCall` method is inlined and the jump distance is patched.

```
IL_0000:  ldc.i4.0
IL_0001:  stloc.0
IL_0002:  ldloc.0
IL_0003:  brtrue     IL_001c // Jump distance is 12
IL_0008:  ldarg.0
IL_0009:  ldloca.s   V_0
IL_000b:  stloc.s    V_9
IL_000d:  stloc.s    V_10
IL_000f:  ldloc.s    V_9
IL_0011:  dup
IL_0012:  ldind.i4
IL_0013:  ldc.i4.s   10
IL_0015:  add
IL_0016:  stind.i4
IL_0017:  br         IL_001c
IL_001c:  ldc.i4.2
```

Figure 17 - Patched jump.

The jump distance has been updated according to the number of inlined instructions (colored red) the ne jump distance is calculated according to the following formula, where the `JmpOffset` represents the number of instructions added to the calling instruction.

```
int JmpOffset = (InlineInterval.last - InlineInterval.first) +  // Number of inlined instructions
                (InlinedMethod.Parameters.Count – 1) + // Instructions added to store parameters
                 (!InlinedMethod.IsStatic ? 1 : 0) +          // Instruction for storing this argument
                (BlockPatchInstrCount + (swap ? 2 : 0));   // Instructions used to patch protected block
                                                           // swap indicated if the result had to be swapped
int newJumpDistance = OldJumpDistance +
                      Math.Sign(OldJumpDistance)  * JmpOffset;
                      // Signum function makes sure that even backward jump are updated correctly
```

Figure 18 - Formula for computing new jump distance after inlining is complete.

The same update must be done for the protected blocks of the calling method since they may have been changed by adding instruction to the method. The process is very similar to the restoration of jump instructions. When a block contains the call instruction that have been inlined then it must be extended to contain all the newly added instructions. This means, that the block end must be moved according to the formula presented in the Figure 18, only the new end is equal to the following.

```
int NewBlockEnd = OldJumpDistance +  JmpOffset;
```

Figure 19 - Update of the protected blocks after inlining.

Where the `JmpOffset` is the same value as in the Figure 18 and it represents the total number of instructions inserted during the inlining process.

4.1.1.9 Protected block registration

One of the last required transformations is to register all the protected block of the inlined method to the main method. This is necessary, since the code would not be correct otherwise, not to speak of instructions like Leave or `Endfinally` that would remain in the code unattended. The solution is very simple, all the protected blocks are added to the main method and their start and end is updated according to the interval returned by the Code inlining operation, discussed in section 4.1.1.6, the interval start is simply added to the start and the end of each block.

4.1.1.10 Maximal stack size calculation (.NET specific transformation)

The very last step is to calculate the maximal size the stack could have when the new method is called. This is important only in the case when there is an inlined method that requires deeper stack then the main method or when there is a nonempty stack when an inlined method is called, in which case the maximal size would be: stack size before call + maximal stack size of the inlined method. The stack simulator is used to compute the maximal size and the result is the maximal size of the stack in the simulator that is encountered during the simulation.

Observation The only reason why the new method would require a deeper stack is the inlined code and if the depth is changed then the new maximal required depth is equal to: *stack size before the call + maximal stack size of the inlined method.*

Discussion The required depth cannot be changed anywhere else then in the inlined code, since no other code has been changed. If the depth is changed by the inlining then it is due to the inlined code and depth is stack size before call + maximal stack size of the inlined method, because the code of the inlined method is the only reason the depth can change and the state of the stack before must be included as well. The depth cannot be influenced by any other instructions added during the inlining process, because parameter storage just removes the parameters from the stack and they used to be there even in the original code. The instructions used to manage inlined protected blocks cannot influence the depth, because they remove everything from the stack prior to the call and then they return it to the stack along with the return value, but all the values has to be there even in the original code. If the blocks have been patched then the new size is only: *maximal stack size of the inlined method* (if the inlined method requires a deeper stack).

4.1.2 Data types and inlining

This section is .NET specific because the type system used Ecma-335 compatible environment is very strict and it requires careful manipulation.

It is very important to note that all the new local variables and any temporary variables must have a correct data type and that type must be imported to the module containing the calling method, otherwise the resulting code would be invalid and it could not be executed. This means, that whenever a new local variable is added, it must have the same type as the value it stored in it and that type must added to the module via an import function provided by the used reflection library. This is done by every transformation that adds any instructions to the code, like the section 4.1.1.4 for example.

There may be some complications present, when determining the data type of the value that should be stored in a new local variable because the value may have been added on the stack by another method or a complicated calculation. The most important and complicated case is present during the protected block patch presented in the section 4.1.1.3, because the values, present on the stack prior to the inlined call, must be stored in temporary, local variables and these variable must have a correct data type. The best way to find the data type of these values is to use a stack simulator and find out what instruction produced the value and what type it may have.

```
Type GetDataType(Instruction instr)
{
        // ConstantType method determines the type of constant loaded by instruction
        If(IsConstantLoad(instr))
            return ConstantType(instr);

        // Operand references the object accessed by the instruction.
        // Its type is determined accordint to what it is, based on the instruction.
        Else If(IsParameterAccess(instr))
            return instr.Operand.ParameterType;

        Else If(IsLocalVarAccess(instr))
            return instr.Operand.VariableType;

        Else If(IsFieldAccess(instr))
            return instr.Operand.FieldType;

        // Recursion used on the parameters passed to the instruction
        // this solves the problem with data type of instructions like add, mul
        Else return GetDataType(GetParameters(instr));
}
```

Figure 20- Algorithm used to determine data type returned by an instruction.

4.1.3 Summary

The calling method should be now a complete valid method and its execution should yield the same results as the original. The following analysis and transformations can work with the code easily and safely, because all the data types have been imported to the methods module and the code can be analyzed in much greater depth, because the called method has been inlined and its code has been added to the calling method. This can help remove some dependencies which would not have been possible without the knowledge of the code inside the called method, since the optimizer cannot assume anything about a called method and it must make it dependent on everything as well as the method has to depend on everything else.

The inlining process is very complicated in the ECMA CIL compatible environment, because there are many things that must be taken care of for the code to work correctly. Most important is to keep the jump instructions valid and to use correct types for temporary local variables. Many problems arise thanks to the fact that it is very difficult to manually control the produced code or to verify it automatically, ILDasm and PEVerify proved to be essential tools for this task and even then it was not easy at all to resolve all the problems. The implemented inlining process that is part of this project works with all the constructs of the C# language considered safe code; it is not prepared to handle unsafe code, because it has been designed to optimize numerical computations written in standard C#.

One disadvantage of inlining is the inability to handle virtual methods, because the virtual methods are selected at runtime according to the used class and the specific method is not known during the compilation. Virtual methods can be partially handled by inter-procedural analysis, but even then, this remains an algorithmically unsolvable problem very similar to aliasing discussed in section 6.5.

4.2 Code verification

Code verification is a process designed to check if the code follows the restrictions discussed in section 1.7 and it is not a verification of the code validity. It is performed after inlining and it must make sure that the final code does not contain any forbidden instructions or constructs. Verifying that there are no forbidden instructions is very easy and it is solved by the following algorithm, where the filter is hash table containing all forbidden instructions.

```
foreach (Instruction instr in method.Instructions)
{
    if (filter.Contains(instr.OpCode.Code))
        return false;
}
return true;
```

Figure 21 - Instruction verification

The verification can check for things other than instructions; it can check the presence of protected blocks or calls to certain methods, which have not been inlined. There are many other things that may not be supported by the transformations performed after this step and the verifier must make sure the code will not cause any failure due to some unsupported features.

The verification does not have to check if the inlined code is valid, because it assumes, that the inlining was done correctly and valid code inlined correctly must be valid. It would be possible to verify the resulting code using the PEVerify tool that could be run separately and its results would indicate if the code is valid or not.

The implemented verifier checks if the analyzed method is not part of a module that may contain unsafe code. A module containing an unsafe code has to be compiled with the `/unsafe` option and the resulting assembly has the `UnverifiableCodeAttribute` attribute indicating, that there might be unsafe code and the verifier checks the presence of that attribute. It is not a perfect method, but according to the assumptions in Chapter 1.7, the analyzed code must have been produced by the Visual studio compiler and therefore it must have the custom attribute.

4.3 Summary

Both parts of code preparation have been successfully implemented, but the inlining proved to be much more complicated the expected and its implementation extremely slowed the entire project. There were many problems that were not initially anticipated and they were very difficult to solve, because corrupted CIL applications does not report any specific errors, it just crashes. Finally, the problems were solved using tools like PEVerify and ILDasm, which proved invaluable for analyzing CIL code.

5 Preliminary code analysis

The code has been has been optimized and verified and it is almost ready for dependence testing but first, it is necessary to gather information about the control-flow constructs and then it the optimizer must create a list of all variables used in the method. Both types of information are later used during dependence detection, since dependence can be based on data or control-flow, and both types can be transformed to data dependences. This is explained later in the chapter about dependence testing. This step does not contain any transformations or optimizations and the code is not modified here.

5.1 Construct recognition

Definition	A control-flow construct is a code segment that changes the standard execution order. The constructs are divided in two groups: conditional and unconditional. Unconditional constructs work always the same way, unlike the conditional constructs, whose behavior changes based on a condition.

This analysis must recognize all control-flow constructs present in the analyzed method and because this project assumes, that the method was written in C#, there can be up to five different types of construct: loops, `if/else` branches, `switch` statements, protected blocks and `return` statements. This analysis is performed first because it gives an important insight into the behavior of the optimized method and its aptitude for parallelization. The information gathered here is important for two main reasons:

1) If the method does not contain loops then it is probable that it would not be feasible to use any parallelism to increase its performance, because it is not be possible to use SIMD instructions since they require a single operation applied on various data and most common construct compatible with that pattern is an array transformed inside a loop. Threads are a little more flexible but more expensive and it is not efficient to launch separate thread just to perform a few simple instructions concurrently and without loops there is usually not enough instructions to justify the use of threads. There is one rare exception, if the method is very long and there are big sets of mutually independent instruction then each set can be run in parallel, but such a situation is very rare.
2) Another very important effect is the analysis of control flow and data transfer, this is very important for dependence testing, because if there is no path from one

instruction to another then there cannot possibly be dependence between them (only one of them can be executed in a single call to the method). The situation can be illustrated on the following example, where the statements S1 and S2 can never be executed together when the method is called and there cannot be dependence between them.

```
    int IfElse()
    {
        int var = 1;
        if (var > 0)
S1:         val += 1;
        else
S2:         val -= 1;
        return var;
    }
```

Figure 22 - Execution path modified by control-flow.

5.1.1 Construct recognition in depth

Assuming that the optimized method was written in C# and compiled by MS .NET compiler, there are three groups of constructs that can influence the order of execution of the method:

1) **Loops** – including `while`, `for` and `foreach` loop, where the `foreach` loop is transformed to a `while` loop, when working with a collection (`IEnumerable<T>`), or to `for` loop, when working with arrays.
2) **Branches** – conditional and unconditional jumps including the conditional branches `if`, `if/else` and `switch` and the unconditional `return` statements. The method calls are not considered here since they are either inlined or they are executed as a single instruction in the context of the method (call jumps somewhere, does something and then jumps back, but it does not change the instructions of the method).
3) **Exception management** – including `try`, `catch` and `finally` blocks, along with the instructions that can jump to and from those blocks, like `throw`, `leave` and `finally`. All the protected blocks can change execution order and this change can be very unpredictable. There are many problems with exceptions, since the exception handling can depend on the order of execution inside the protected block and this dependence may be impossible to recognize.

```
    try
    {
S1:     File a = File.Open("test1.txt");
S2:     File b = File.Open("test2.txt");
    }
    catch (System.Exception ex)
    {
        File.Close(a);
    }
```

Figure 23 - Protected block and dependence testing.

The example shows the problems with protected blocks. The protected block is designed to close the first file, if the second throws an exception, but that would fail in case these files were being opened in parallel and the second was opened first. If may be necessary to execute everything in a protected block without any parallelism; therefore, using any protected blocks and exceptions is strongly discouraged unless it is completely necessary. Best way to handle this is to create a protected block, then call the optimized method in it and handle all the exceptions there, not in the method.

5.1.1.1 Possible solutions

Most precise and universal way to identify all control-flow constructs is to construct a graph of all the basic blocks and then study the relationship between them via connections based on possible execution order. Loops are recognized as cycles, conditional jumps transfer the control to one of two following blocks and protected blocks would require a special treatment. But since this project assumes that the method has been written in C# and compiled by Visual studio compiler, there is a simpler way to identify the constructs.

Recognition method presented in this work is based on categorization of all jumps according to their direction and target. The constructs are detected in a specific order, and the order is very important and it must be followed or the recognition might fail. Protected blocks do not have to be identified since they are already recorded in the method metadata and any jumps related to them are easy to find.

5.1.1.2 Jump recognition

All jumps are gathered to a list at the beginning of the analysis to allow the operations to quickly recognize jumps and remove them to simplify following operations. Only jumps related to exception management are omitted, because they cannot be part of any construct other than a protected block and they are located in the last step.

5.1.1.3 *Loop recognition*

There are two ways to compile loops and one is better than the other. The worse option is to place the termination condition at the start and jump away, once the condition is not satisfied. The solution is shown in the Figure 24.

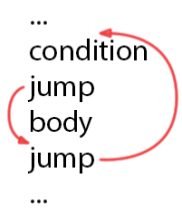


Figure 24 - Less effective implementation of loops.

The second version is to place the termination condition at the end and jump to it from the beginning. The loop ends when the condition is not satisfied and the backward jump is not executed. This solution is better, because it saves one jump instruction per iteration. This is the solution used by the Visual studio compiler and Figure 25 shows its schema.

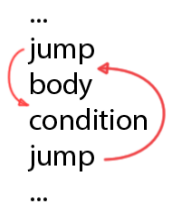


Figure 25 - Better implementation of loops.

In MSIL, loops consist of two jumps, a forward jump that goes from the beginning to the termination condition (placed at the end) and a backward jump executed when the termination condition is not satisfied. Loops are the only source of backward jumps and that is the way to find them, the forward jump is just before the target of the backward jump. Both jumps are then excluded from further recognition so they are not mistaken for conditional jumps.

The difference between `while-loop` and `for`-loop is the fact that the `for`-loop should contain an induction variable. This Variable is updated at the end of its body and a new value is stored just before the start of the termination condition. Generally, the recognition of the induction variables is an unsolvable problem and the Figure 26 shows a loop that does not have a single obvious induction variable.

```
int j = 0;
for (int i = 0; j < 100 && i < j; i--, j++)
{
    i = func(j) + i;
    arr[j] = 2 * arr[i];
}
```

Figure 26 - Complicated for loop.

It is impossible to decide what is the induction variable in this loop and what exactly does it do, since the `func` function is unknown. There is a solution, that does not rely on the structure of the `for`-loop and it is discussed in the section 6.4. For now, it is just necessary to locate all loops even if their type cannot be identified.

5.1.1.4 Conditional jump recognition

Now, all the remaining conditional jumps belong to either `if` or `if/else`, because all the jumps belonging to the loops have been eliminated. The analysis of the jumps must start at the beginning of the method and continue to the end and the structure of the jumps can be used to separate simple `if` and more complex `if/else` constructs. Simple `if` contains a simple jump that goes just behind the block (jump is taken when condition is not satisfied), shown in next example.

Figure 27 - Basic if statement.

`if/else` is constructed from two jumps, conditional jump that targets the start of the `else`-block and a basic jump that goes from end of the `if`-block behind the `else`-block. The second jump is just before the target of the first. Following example shows the structure of the `if/else` construct, where the red arrow represents the conditional jump and the green arrow represents the unconditional jump.

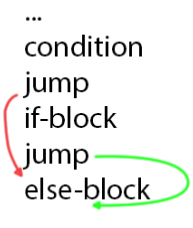


Figure 28 - if/else construct.

Both construct can be recognized by the fact that they contain a forward conditional jump, and there is no other construct that contains conditional jumps, since all the jumps belonging to loops have been removed from the jumps register. The `switch` construct uses a special instruction of the same name and it is easily recognized.

All the jumps, that belong to some `if` or `if/else` construct, are removed from the jump register at the end of this step.

5.1.1.5 Switch recognition

`switch` is very easy to find since it has its own instruction of the same name, but it is not supported by this project because it is not that common and it can be replaced by a few `if/else` constructs chained together.

5.1.1.6 Protected block recognition

Protected blocks themselves are already recorded in the methods metadata, but it is necessary to find all the special instructions that work with exceptions and can change execution order. These instructions include `throw`, `leave` and `endfinally` and finding them is very simple. The algorithm just goes through the code and it notes all the instances of those instructions. It may be a good idea to perform a simple check if all the special instructions are used in appropriate places, for example `endfinally` cannot be used outside of a `finally` handler block.

5.1.1.7 Return statement recognition

Locating the return statements is about locating all the `ret` instructions, because there may have been return statements introduced by inlined methods; even though, they have been converted to unconditional jumps. The jump register can be used to locate the additional return statements. All the jumps that remain in the register now are simple jumps that replaced `ret` instructions present in inlined methods and they can be easily added to the `ret` instructions.

5.1.2 Summary

At this point it is known if the method contains any loops and what kind. All control flow constructs are now recorded and only two problems can remain. Recognition of induction variable is extremely difficult and it may not be done at all, but that is solved by dependence testing. And all the problems of exception handling remain but with a little luck, the programmer is wise enough not to use them here (with the

possible exception of `throw` instruction that can be handled as `ret` if it is not in a protected block).

5.2 Variable recognition

This step is extremely important because all the variables considered for dependence testing must be identified now, because the dependence testing must know all the variables so it can explore their relationship. It is not simple to decide what is a variable since a variable can be an object as a whole or its fields can be considered separate variables, the same can be said about arrays but it is essential to consider all elements as independent variables since arrays are the main source of opportunities for parallelization.

There are also special temporary variables created on the stack as a result of some operation and they are later consumed by some other instruction. These variables can be recognized by a stack simulator and they represent the relationship between instructions that constitute separate commands, because a command usually contains three phases: reading parameters, performing some operation and storing the result. The stack is empty prior to a command and it should be empty after the command is completed, with a few exceptions like exception management (the exception object is on the stack when entering the handler block).

The fields and array elements are considered separate variables even though they have a special relationship to the object they belong to and that must be considered, but this way, it is possible to find more opportunities for parallelism, because there are more variables that can be independent.

It is not enough to just recognize the variables, it is very important to identify all the places where the variable is accessed - written or read. This is pretty simple to find out for temporary variables, the stack simulator reports when a value is added and removed. It is a little trickier for local variables because they can be accessed by value or by reference. The same can be said about fields and array elements, but there is another problem, because when a field is accessed, it is necessary to find the object it belongs to as well.

The next section discusses how different types of variables can be identified and locate.

5.2.1 Variable recognition in depth

This step requires two important decisions, what is a variable and how to find and recognize it? This section tries to find answer to the second question. There are two types of variables that must be recognized, temporary stack variables, local variables, parameters, object fields, array elements and static fields.

5.2.1.1 Stack variables

Stack variables can be located using the stack simulator, a variable is created when a value is added to the stack and that variable ceases to exist when that value is removed from the stack. These variables are defined as return values of some operation and the best way to identify them is by this operation (instruction). Therefore, stack variable must know the instruction that created it and that is at the same time the only place this variable is changed, and it must know the instruction that reads it, which is the only place it is read. After a temporary variable is read, it is removed from the stack and thus it is destroyed.

5.2.1.2 Local variables and parameters

Local variables and parameters can be processed together, because they are used the same way. Static variables could be counted to this category as well since they exist the entire time the method is executed and they can be accessed more than once, but they are resolved separately, because they need to know the class they belong to. Locating these variables is very simple, all the local variables must be listed along with all the parameters in the metadata of the analyzed method and they just have to be collected. As simple as it is to find and identify these variables, it is more complicated to find all the places they are read or changed. The code must be analyzed instruction by instruction and when any access to a variable is found, than that place is recorded and added to the variable. As a result, the local variables are identified by a reference to the parameter / variable they represent and then they contain a list of instructions that read or change them.

5.2.1.3 Fields

Field variables represent field objects accessed in the method and they can be easily located by spotting `ldfld` and `stfld` instructions. More important problem is to find out the object this field belongs to and that can be done by analyzing the stack just before the instruction is executed. The object in question must be on the top of the stack and a stack variable can be used to locate the variable that references the object. The

Figure 29 shows variables used to find the object that is later used to identify the field along with its name that is part of the `ldfld` instruction. The instruction consumes the object reference placed on the stack.

```
// reading parameter dat, writing to a stack variable
IL_000a:  ldarga.s   dat
// reading the stack variable that contains the object
IL_000c:  ldfld      int32 ParallaXExample1.str::var // name of field
// stack variable is used to find parameter dat containing the object
```

Figure 29 - Stack and instructions used to access field variables.

Following algorithm is used to locate the object variable using the stack variable that represents the object on the stack. It is important to note, that only one variable can be read by a single instruction (with the exception of stack variables), because the MSIL has a load/store architecture and all.

```
public var FindVariable(int fieldAccess)
{
    // find stack variable, based on the instruction where it is read
    var stackVar = stackVariable.FindByReadIndex(fieldAccess);
    // find where it was created
    int objectIndex = stackVar.FirstWriteIndex;
    // find the object variable
    // only one variable can be read by one instruction
    var objectVar = varables.FindByReadIndex(objectIndex);
    return objectVar;
}
```

Figure 30 - Algorithm used to locate object containing a field.

5.2.1.4 Elements

Array elements are very similar to fields, but they are located by finding the `ldelem` or `stelem` instructions and they are identified by the array they belong to and by the stack variable used as index to the array, both these variables can be found using the stack variables and the algorithm shown in the Figure 30.

5.2.1.5 Static fields

Static fields are accessed by `ldsfld` and `stsfld` instructions and they can be identified by their name and class, both of which are part of the instruction used to access them (it is an operand).

5.2.1.6 Reference variables

The reference variables are special variable type used to manage the situation, when a variable is accessed by a reference. This variable is created when an address is read using instructions like `ldloca` and it is identified by the variable whose address it

contains. Locating and managing these variables is similar to field variables, because they are tied to a variable as well. These variables are located by finding instructions that read address and they are identified by the variable whose address was been read. However, they are not treated as normal variables, instead all accesses are registered in the variable they reference, and this way the reference just represents the variable without creating an actual new variable and all accesses are still controlled.

5.2.2 Summary

Locating variables is not that difficult, unlike dependence testing, which must decide what variables are independent and it must be done precisely or the resulting code may be incorrect. The only tricky part is recognition of accessed fields, since they are accessed through a stack variable created by reading a local variable and it is necessary to record that relationship. The same problem is presented by the array elements, but they are even more problematic, because the index used to access them can be even impossible to analyze.

6 Dependence testing

6.1 Introduction

Dependence testing is the most difficult part of this project and there are many problems that must be solved before the actual dependence testing can be performed. Since dependence testing is an algorithmically unsolvable problem and its approximation can be arbitrarily complex, this work follows a different path. It presents algorithms that transform the CIL code to a structure that can be analyzed by algorithms described in literature, because there are algorithms for similar languages like C that can be used on the transformed code.

This work is meant as proof of concept and it focuses on simple applications to analyze potential parallelization of specific .NET applications. This chapter presents transformations necessary to analyze the algorithms discussed in section 1.5.2: matrix multiplication and vector addition. The section 6.9 shows the application of these transformations to prepare the code for dependence testing and the actual testing is taken from [2]. The last two problems described in section 1.5.2, relationship between local variables and parameters, are discussed in section 6.5, which offers several solutions available in .NET applications. This chapter does not present any algorithms for dependence testing; instead it focuses on transformations that prepare the code so it can be analyzed by algorithms presented in other books, namely [2].

The construction of CIL/MSIL and CLR engine can help with parts of this task, but there are some problems that make the dependence testing more difficult than it is in FORTRAN or C. The most prominent difficulty is the array access analysis, because the index subscripts are computed using the stack which makes their analysis complicated. Not to mention the fact that the identification of loop induction variables very difficult in the CIL.

There are two important facts that help dependence testing in CIL code. First, there are no pointers allowed and there are no arbitrary addresses, because everything must represent valid, allocated objects. Second, local variables are completely private and they cannot be modified anywhere outside the method, with the only exception of reference parameters and it is possible to check if a local variable have been passed by reference or not.

The first part of this chapter contains general discussion about aliasing and dependences in CIL applications, and then follows the description of techniques used in this work to locate dependences in studied algorithms.

6.2 Definitions

This section contains the definition of basic terms used in this chapter and the definition of general terms used in the entire work can be found in the section 2.1.1. This work refers to the method parameters as either arguments of parameters and both terms mean the same here.

Definition	Variables based on a value type contain a value directly and they do not reference a memory. These variables cannot cause any aliasing, because they do not represent a memory location.

Definition	Variables based on reference types represent an address of a memory location that contains an object and they may lead to aliasing. These variables can reference only a valid object or `null`, because C# is language with reference semantics and the reference cannot represent an arbitrary memory location.

Definition	Reference parameter is a parameter that can be modified by the method and the modification is apparent in the calling method. These parameters are technically a reference to a reference.

Following definitions specify different types of variables used in this chapter, because the terminology used here is specific.

Definition	Variable is a term used in this chapter to identify any memory location; it does not have to be a local variable or other variable type used in C#

Definition	Local variables are typical C# local variables that belong to a method.

Definition	Parameter variables represent the parameters of a method and their usage is very similar to local variables.

Definition	Stack variables represent the values added on the execution stack during the computation. These values represent inputs and outputs of instructions.

Definition	Field variables represent the fields of objects and these objects can be represented by any type of variable, even another field variable for composited objects.

Definition	Static field variables represent the static fields of classes.

Definition	Array variables represent the elements of arrays and they are very important for automatic parallelization. The array can be represented by any variable type, similar to the field variables.

6.3 Variable dependence and aliasing

This section contains general discussion about aliasing and dependences between different types of variables encountered in CIL code. This section ignores variables based on value types, because they cannot cause aliasing.

- **Parameters and local variables** – parameters and local variables both represent independent memory locations that can be accessed only by the method itself. The only exceptions are reference parameters of methods called by the optimized method, but it is possible to spot any place where a local variable or argument is passed by a reference, since the address is read using `Ldloca` or `Ldarga` instructions. Therefore, all reads and writes to different local variables or arguments are independent operations that do not collide with each other, because local variable is just an address of an object and assignment simply rewrites the address. However, there may be collisions when a field is accessed using two local variables referring to a single object. Using the reads and writes (explained in section 5.2), it may be possible to keep track of some variables that refer to the same object, but the aliasing is an algorithmically unsolvable problem.

- **Stack variables** – stack variables represent values added and removed from the stack and every variable is written and read just once, since value cannot remain on the stack after it has been used. The value on the stack is independent of its source since it is placed in the memory assigned to the stack. Every stack variable simply represents a single true dependence with a source in the instruction that created the variable and the sink is in the instruction that consumes it. It does not matter if the stack variable is a reference or a value, because the CLR treats all values on the stack as 32 bit or 64 bit numbers [1].
- **Field variables** – fields bring many difficulties, because they depend on the actual object that is used to access them, and it is necessary to keep track of the local variable writes since it is important to know what variables refer the same object. Two field variables can access the same memory, only when they access the same field in the same object, otherwise they are independent. There is a problem, when an object is passed as an argument to some called method because its fields may be changed by the called method, even though it is a normal argument (not a reference argument). To prove independence between fields, it is necessary to keep track of the object they belong to and all possible dependences must be considered when this object cannot be properly monitored.
- **Static fields** – static fields introduce many complex relations between object that can significantly complicate the architecture of the project and their usage is discouraged by many C# textbooks. The use of static fields in in parallel applications should be limited, because they introduce many dependences which are very difficult to eliminate. Therefore, any static field is considered to depend on everything and everything depends on it.
- **Array elements** - the most important type of variable is array element, because arrays represent the best opportunity for parallelization, but their analysis is the most difficult. Two elements represent the same memory location only if they are in the same array and if they are accessed with the same index. Techniques used to handle field variables can be used here to compare two arrays, but it may difficult to identify the difference between used indices. Another problem is that the indices are calculated using the execution stack which makes their analysis even more difficult. First step towards solution is to use stack variables to locate all the instructions used to calculate the actual subscripts, this may be a simple variable reading or a complex computation and both can be problematic. The first concern must be correctness and

the implementation must simply assume that all the indices may be the same except those that can be proved different with an absolute certainty. The subscript analysis is a very complex problem and there are many techniques presented in [2], in chapter 3, that can be used in CIL applications after special transformations presented in section 6.7.

6.4 Induction variables

Induction variables are defined by loop iterations and they are essential to understand the behavior of a loop. There can be many induction variables used in a single loop since every variable that depends on its iterations can be considered an induction variable.

This analysis is simplified in FORTRAN, because its loops contain the definition of a variable used to manage loops iterations and that is usually the main induction variable. This variable is very good for analysis, because it has a known behavior since the loop defines its initial and maximal value and both values can be usually calculated during compilation. There may be other induction variables in the loop, but the main variable is usually used to access arrays since its step and boundaries can be defined according to the processed array.

The situation is more complicated in languages similar to C, because they have more complex structure of `for`-loops and there may be no main induction variable and even if the loop is simple, it may not be possible to find the variable in the analyzed bytecode. It may be possible to analyze the loop structure in more detail, but it is more reliable to analyze the behavior of the variables regardless of their presence in the loop definition. This analysis can be very complicated, but the simple implementation presented in the following figure can be sufficient for many common applications.

```
InductionVariables(int loopStart, int loopEnd)
{
    // list of all arrays accessed in the loop body, written or read
    arrays = ArrayAccessedIn(loopStart, loopEnd);
    // analyze array subscripts and extract variables
    List InductionVariables;
    foreach(array in arrays)
    {
        // reconstruct the statement used to access the array
        tree = ReconstructSubscriptTree(array);
        // if the subscript is simple then extract the induction variable
        if( (tree.root == loadVariable) ||
            (tree.root == operator &&
                tree.child == IsVariableLoad() &&
                tree.otherChild == IsConstLoad()) )
        {
            InductionVariables.Add(loadedVariable);
        }
    }
    // analyze the behavior of the variables used to access arrays
    foreach (var in InductionVariables)
    {
        // if the variable is not modified just once, remove it
        if(GetAllWrites(var, loopStart, loopEnd).Count() != 1)
            InductionVariables.Remove(var);
        // analyze the only change performed in the body of the loop
        access = GetAllWrites(var, loopStart, loopEnd);
        // reconstruct the value assigned to the variable
        tree = ReconstructTree(access);
        // if the assigned value is not a modification of the previous
        // value then the variable is removed from the list
        if( tree.root != operator ||
            tree.child != IsVariableLoad (var) ||
            tree.otherChild != IsConstLoad() )
        {
            InductionVariables.Remove(var);
        }
    }
    // return all induction variables used to access arrays.
    return InductionVariables;
}
```

Figure 31 - Induction variable analysis.

The algorithm presented in the Figure 31 is able to locate only simple induction variables, but it can be safely used to analyze common vector and matrix calculation. The algorithm analyzes the array subscripts and if the subscript contains a variable then the variable is tested as an induction variable. The subscript can contain only a single variable or a variable modified by a constant, for example `arr[i]` or `arr[i+1]`. The selected variables are analyzed according to the value that is assigned to them in the loop. A variable is considered to be an induction variable, when it is changed just once in every iteration and the new value is the old value modified by a constant, for example `i++` or `i=i*2`. All the variables selected by this algorithm are induction variables and they are easy to analyze, because their behavior is very simple. Other induction variables

are ignored because they are not used to access arrays, or because their behavior is too complicated.

Theorem	A variable that is modified once in every iteration of a loop is an induction variable, if the modification is a simple addition of a constant, or a multiplication by a constant value.
Proof	Let `i` be the number of iteration and `x` is a variable modified once in iteration. If `x` is changed according to the formula `x=x+const` then the value of `x` is equivalent to `initial+i*const` and that value depends only on the initial value of x and on the number of the actual iteration. If `x=x*const` then its value is equivalent to $initial+(const)^i$ and that is based on the iterations as well.

The following example shows a basic implementation of matrix multiplication that can be easily analyzed by the algorithm presented in this section, because it would find all the induction variables.

```
for (int i = 0; i < A.Length; i++) // M
    for (int j = 0; j < B[0].Length; j++) // N
        for (int k = 0; k < A[0].Length; k++) // K
        {
            C[i][j] += A[i][k] * B[k][j];
        }
```

Figure 32 - Simple matrix multiplication.

6.5 Aliasing

Aliasing is an effect present only in languages that support pointers or references, because both these constructs represent a memory address that is usually unknown during the compilation and it may be impossible to determine which pointers address the same memory. The aliasing is caused by the fact that multiple symbols can represent the same memory location. This effect makes dependence testing very difficult, because the dependences are caused by modifying the same memory location, which can be identified by different symbols. If the compiler is not able to determine what pointers reference the same memory then it must consider that they can reference the same memory, since it is a possibility. The aliasing is caused by reference types and any variables with value type cannot cause it and they can be ignored in this entire analysis.

Aliasing in .NET is simplified by two important facts. There are no pointers and the references are controlled by the type system which forbids certain references to address the same object. Another important fact is that the reference must always address

a valid object; it cannot be assigned some random address. Still, aliasing is an algorithmically unsolvable problem, because there are many situations that cannot be analyzed due to the lack of information. The most common example is the unknown relationship between parameters of a method, because the parameters can reference the same value objects and there is no way of knowing what value they may contain. The following sections present algorithms that can solve aliasing in simple applications which are the main focus of this work, like matrix multiplication.

The aliasing between parameters and static variables can be partially solved by inter-procedural analysis, but even this analysis is not able to solve similar problems completely. Inlining used in this work is more limited, but it can solve parameter aliasing for the inlined methods, because the values passed as their parameters are known after the method is inlined.

6.5.1 Parameter aliasing

It is possible to solve the aliasing between some parameters without inter-procedural analysis, but it is expensive and it should be used only when it can significantly improve the dependence analysis. The parameters can be safely separated using code duplication. The original code of a method can be duplicated and both versions can be separated by a conditional jump. The following figure shows the general structure of the method after the parameter separation.

```
void Method(Parameter param1, Parameter param2)
{
    if(param1 != param2)
    {
        Code // copy of the original body
    }
    else
    {
        Body // the original body of the method
    }
    // the first branch can consider the parameters
    // to represent separate memory locations
}
```

Figure 33 - Parameter aliasing solution.

This algorithm duplicates the entire body of the method and the new body is constructed from the original body and its duplicate and a conditional jump is used to decide what code will be executed. The original code is executed when the parameters reference the same memory and the cloned code is executed when the parameters are different. The cloned code can be optimized because it is certain that the parameters reference different memory. This transformation can be used for more than two

parameters, but it can quickly create a huge method and it should be only used when its application can help significantly with parallelization, like in the following example, where the separation of parameters allows the method to be completely parallel.

```
void CopyVector(double[] A, double[] B)
{
    int copyCount = Math.Min(A.Length, B.Length);
    for (int i = 0; i < copyCount; i++)
    {
        B[i] = A[i];
    }
}

// method after parameter separation
void CopyVector(double[] A, double[] B)
{
    if (A != B)
    {
        // the code can be parallelized in this branch
        int copyCount = Math.Min(A.Length, B.Length);
        for (int i = 0; i < copyCount; i++)
        {
            B[i] = A[i];
        }
    }
    else
    {
        int copyCount = Math.Min(A.Length, B.Length);
        for (int i = 0; i < copyCount; i++)
        {
            B[i] = A[i];
        }
    }
}
```

Figure 34 - Copy vector function before and after parameter separation.

This transformation is not necessary when the parameters are based on a value type, because those types cannot cause aliasing. This algorithm does not help much when the parameters are changed in the method and it is reasonable to omit its application in such case.

6.5.2 Variable aliasing

Code duplication can solve the aliasing between parameters at the beginning of the method, but it is possible that the aliasing can occur again since it is possible to assign a reference to a new object to a parameter. The same is true for all local variables and their fields or elements. However, there is one aspect that can help the analysis of aliasing: many languages, that use garbage collection, allocate results in the method that produced theme, because it is safe and efficient. This programming technique may help, since the result is not passed as a parameter, which could be difficult to analyze.

Observation	Aliasing is possible only between variables with a reference or pointer type. All variables with value type are unaffected by aliasing, because they do not address a memory. Stack variables can be excluded as well, because they are never modified. Stack variables are treated as a number and they are destroyed when they are read.

Aliasing between local variables can happen only when a variable is assigned a value, because they are uninitialized at the beginning of the method. It is possible to trace the assignments to construct the dependence sets based on the assigned values. The following example presents a skeleton implementation of an algorithm that works with basic blocks.

```
void BlockVariableAliasing(BasicBlock, PreviousVariables)
{
    // update variable dependences according to the previous block
    BasicBlock.Variables.Merge(PreviousVariables);
    // the aliasing can change in every instruction and it is necessary
    // to store the dependence sets for every instruction in the block
    for (instr in BasicBlock.Instruction)
    {
        // find out if a variable has been modified
        // only one variable can be modified by a single isntruction
        var = BasicBlock.Variables.ModifiedVariable(instr);
        if(var != null)
        {
            // find the value assigned to the variable
            source = GetAssignedValue(instr);
            // if it is the value of another variable
            // then the sets are merged and assigned to both variables
            if(source.IsVariable())
                var.DependenceSet =
                     source.DependenceSet.Join(var.DependenceSet);
            else
            {
                // new set is created if the assigned value is
                // a new object or the result of some calculation
S1:             var.DependenceSet = new Set(var);
            }
        }
        // current state of variables is assigned to the instruction
        CloneAndAssign(instr, BasicBlock.Variables);
    }

    // follow recursion only if the block has changed in this pass
    // this is necessary to prevent infinite recursion caused by loops
    if(BasicBlock.HasChanged())
        // update the variables for the following blocks
        foreach (block in BasicBlock.Children)
        {
            BlockVariableAliasing(block, BasicBlock.Variables);
        }
}
```

Figure 35 - Aliasing analysis algorithm.

This algorithm uses recursion to go through all the basic blocks and a set of dependences is calculated for every instruction in the method. The recursion can process some blocks multiple times, because the conditional jumps may create different dependences and the recursion must merge all these possibilities. The most important step is the statement S1, because the aliasing is solved when a variable is assigned a new object created in the method, which cannot be referenced by anything else, until the variable is assigned to another variable.

The algorithm presented in Figure 35 is very complicated and it may need a lot of time to analyze a complex method, while it can produce only very conservative results. But many numerical applications written in C# can have a very specific structure similar to the method shown in Figure 41, where the result is allocated in the method and then it is filled with the calculated values. This situation can be easily analyzed by the following algorithm.

```
    void VariableAliasing()
    {
        foreach (var in Variables)
        {
            // the variable is not aliased unless proved otherwise
            var.NoAliasing = true;
            // if the variable has value type then it is not aliased
            if(var.HasReferenceType() || var.IsStackVaraible())
                continue;
            // only local variables or their elements are allowed
            if(var.IsLocalVaraible() || var.IsLocalArrayElement())
                // analyzing all the assigned values
                foreach (write in var.GetAllWrites())
                {
                    // get the value actually assigned to the variable
                    value = GetWrittenValue(write);
                    // if the assigned value is another variable
                    // then the variable can be aliased
S1:                 if(value.IsVariable() || value.IsMethodResult())
                        var.NoAliasing = false;
                }
                // analyzing all reads
                foreach (read in var.GetAllReads())
                {
                    // get the instruction that consumed the variable
                    instruction = GetConsumerOfValue(read);
                    // the aliasing is possible if the consumer writes to
                    // another variable
S2:                 if(Variables.GetWittenVariable(instruction) != null)
                        var.NoAliasing = false;
                }

            else var.NoAliasing = false;
        }
    }
```

Figure 36 - Simple algorithm for aliasing analysis.

This algorithm is very simple and it is very conservative, because it solves aliasing only for two types of variables: variables with value type and variables initialized by a newly allocated objects. Value types and stack variables are never subject to aliasing and their elimination is always correct. Variables that have been assigned only new objects created in the method cannot be affected by aliasing either, because each new object is placed in a newly allocated memory. It is also important to analyze all the places where variables are read, because their content can be stored in other variables which can cause aliasing. The modified variables can be located by the writes, explained in the section 5.2, according to the instruction that consumed the value placed on the stack when the analyzed variable was read.

Problem is that the fields of newly created objects can be aliased, because their constructor can access static variables and parameters, but this is not a problem for arrays, since they do not use constructor. New arrays must be initialized by hand and the initialization can be easily analyzed by the same algorithm because the elements are not aliased, when they are initialized by a new object or array. This algorithm does not have problem with conditional branches, because it relies on the fact that the analyzed code is correct and the selected variables are never assigned anything else then new objects, regardless of control-flow.

6.5.3 Summary

Aliasing is an algorithmically unsolvable problem in most modern languages which support either reference or pointers and presented solutions are designed to work only for simple applications studied in this work. This work is meant as a proof of concept and provided solutions suggest that it is possible to analyze at least simple applications. Even though the presented solutions are very conservative, they could be sufficient for specific applications and they can be further improved.

6.6 Stack dependence

Stack dependences are caused by the fact that the instructions communicate through the stack and this type of dependence usually represents the dependence between the instructions that implement a single C# statement. These dependences can be solved very easily using the stack variables; their identification is explained in section 5.2.1.1.

Each stack variable represents a value pushed on the stack by an instruction that is later removed by another instruction. Important is that each stack variable is created

when it is first modified and it is destroyed when it is first read. Each stack variable represents a single true dependence with the source in the instruction that created the variable and the sink is in the instruction that consumes the variable. The stack semantics can be treated this way as any other dependences and it simplifies the entire analysis, because there is no special analysis required.

6.7 Array subscript analysis

Array subscripts can be very complicated and their analysis has to address many special situations, but it is not necessary to understand every subscript to optimize special applications. Following sections present an algorithm that can reconstruct subscripts from the CIL code and then there is a discussion about allowed subscripts and their structure.

6.7.1 Subscript reconstruction

Array access can be recognized by used instructions, like `Ldelem` or `Stelem`, and these instructions require the address of the accessed array and an index. Both these parameters are stored in the array variable identified during the code analysis (section 5.2.1.4) and in this step it is necessary to reconstruct the entire subscript based on the stack variable stored as the index. Subscript is technically an expression that leaves one integral number on the stack and that number is represented by the stack variable. The reconstruction process can use stack variables to rebuild the entire calculation tree containing all the instructions used to calculate the actual index. The following figure shows the algorithm.

```
public Tree ReconstructStatement(StackVariable var)
{
    // create new tree that represents the statement
    // that produced the stack variable
    Tree statement = new Tree();
    // get the index of the instruction that created the
    // variable; each stack variable can be changed only once
    int sourceIndex = var.GetAllWrites().First();
    // locate the instruction among the method instructions
    Instruction source = body.instructions[sourceIndex];
    // set the instruction as root to this sub-tree
    statement.Root = source;
    // analyze all other stack instructions to find
    // potential operands of the source instruction
    foreach (var in StackVariables)
        // if the variable is read by the source instruction
        if(var.GetAllReads().First() == sourceIndex)
        {
            // it is analyzed as the root of a sub-statement
            statement.AddChild(ReconstructStatement(var));
        }
}
```

Figure 37 - Index reconstruction algorithm.

This recursive algorithm uses stack variables to track the instruction that produced the index. If the instruction has any parameters, then there must be stack variables that represent these operands and they are used to reconstruct the statements that produced them. The entire tree contains all the instructions that were used to calculate the actual index and its leaves contain instructions without operands, which usually load variables or constants. The following code shows an example reconstruction.

```
// colored subscript will be reconstructed
C[i] = A[i+i*2] + B[i];

// colored instructions represent the index calculation
IL_0006:  ldarg.1
IL_0007:  ldloc.0          // 0 operands
IL_0008:  ldloc.0          // 0 operands
IL_0009:  ldc.i4.2         // 0 operands
IL_000a:  mul              // 2 operands
IL_000b:  add              // 2 operands
IL_000c:  ldelem.r8
```

Figure 38 - Array subscript.

The array access is show in the CIL code and the index calculation is colored red. The instructions follow closely the calculation and there are five stack variables created during the calculation of the index. Three stack variables are created by the load instructions; two of them are consumed by the multiplication and third is consumed by the addition, along with the variable created by the multiplication. Fifth variable is

produced by the addition and that is the actual index used to access the array. The reconstructed tree is shown in Figure 1 and the red arrows represent the stack variables produced by the instructions.

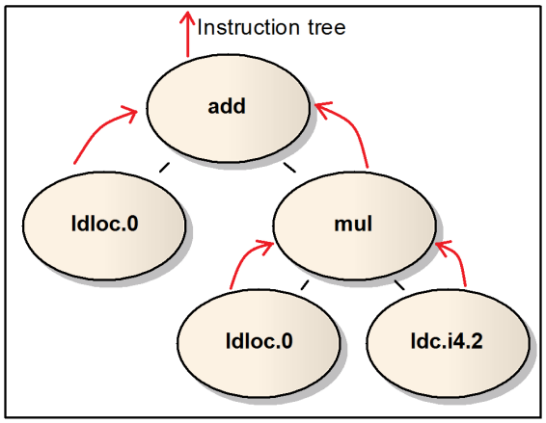


Figure 39 - Array subscript calculation tree.

6.7.2 Multidimensional arrays

Multidimensional arrays in .NET applications are problematic, because they are represented as an array of arrays, rather than as a multidimensional array used in FORTRAN. It is necessary to analyze all array accesses for possible multidimensional arrays, because it is necessary to analyze them as a single array access. It is necessary, because the algorithms taken from [2] expect multidimensional arrays to be analyzed as a single access to a single data structure; the algorithms do not work with consequent accesses to a series of arrays. This can be solved by the following algorithm.

```
    public List MultidimensionalArrays(ArrayVariable arrayVar)
    {
        // list of arrays composing the multidimensional array
        List dimensions = new List();
        // the studied array is added to the list as the last dimension
        dimensions.Add(arrayVar);
        // variable that contains the array accessed by the statement
        Variable array = arrayVar.sourceArray;
        // decide if the accessed array is an element in another array
S1:     while (array.IsArrayVariable())
        {
            // when the array is in another array then the source is
            // extracted from the variable
            array = array.sourceArray;
            // the the array is added at the beginning of the list
            dimensions.AddFirst(array);
        }
        // the list is returned with the outside array first
        return dimensions;
    }
```

Figure 40 - Multidimensional array identification algorithm.

This algorithm simply follows the accessed array and it looks if the array is an element in another array and if that array is an element and so forth. The critical statement in the algorithm is S1, because that is the point where is determined if the array is inside another array or not. It is important to note that the array variable does not represent the actual array but rather the accessed element (the definition is in section 6.2). The algorithm stops once the array accessed by the last array variable is not an array variable itself and that means that it is not an array element.

Another possibility is to use multidimensional arrays supported by the .NET Framework, but they are not well implemented and they are not used as standard arrays, even though they look similar in C# code. The multidimensional arrays are accessed using methods that extract the required elements, unlike the standard arrays that use special instructions. This is problematic, because method calls can ruin dependence analysis and it is not easy to separate normal methods from those that access these arrays, not to mention the fact that method call is slower than simple instruction.

6.7.3 Subscript analysis

The algorithm presented in the previous section can reconstruct any subscript encountered in the CIL code, but many subscripts can be too complex for further analysis. The subscript can contain a method call or a very complex calculation and such a subscript is usually impossible to analyze, since they can contain unknown code. There are many techniques that can be used to compare subscripts and they are discussed very thoroughly in [2] in chapter 3. It is possible to use most of them, because the reconstruction algorithm rebuilt the subscripts to a tree that can be analyzed in a similar way as the FORTRAN code used in [2] and simple C# calculations are very similar to FORTRAN.

It is not necessary to copy the algorithms and they are not needed to optimize the applications analyzed in this work. This work focuses on applications that use very simple subscripts and they can be recognized based on their tree constructed in the previous section. This work recognizes only subscripts that contain a single variable load or a variable modified by a constant. Examples can be `A[x]` or `A[x+5]`. These subscripts can be easily analyzed and their analysis is strong enough to optimize many applications that work with matrices or vectors. Two array accesses are analyzed together only when they are both in the same loop and if they both contain the same variable.

6.7.4 Summary

Now that all the subscripts have been reconstructed and the multidimensional arrays can be addressed as a single data structure, it is possible to use the algorithms presented in [2]. This work focuses on simple applications which require only the simplest algorithms to optimize them and the section 6.9 shows the use of these algorithms to prepare the code for dependence testing.

6.8 Dependence testing

The dependence testing can use the algorithms presented in [2], because the loops and their induction variables have been identified and array subscripts have been reconstructed, along with multidimensional arrays. The analysis of aliasing should provide some help for the testing and all the variables which have not been separated may be treated as a single variable for the purposes of this analysis. It is not necessary to create new algorithms for dependence testing, because the code have been analyzed and transformed so the existing algorithms can be used. The following section shows the application of all presented techniques to analyze the applications studied by this work.

6.9 Example analysis

This section presents the application of the presented transformations on simple applications that were selected as the main focus of this work. They are used to illustrate the fact that it may possible to parallelize special C# application in a similar way as C or FORTRAN applications.

6.9.1 Matrix multiplication

Matric multiplication is a traditional problem for automatic parallelization and any parallelizing compiler must be able to optimize some basic implementation of matrix multiplication. The following listing shows a simple implementation of matrix multiplication that omits safety check and it relies on the fact that the parameters are correct, because the safety checks would pollute the code and they are not important for parallelization.

```
    public double[][] Multiply(double[][] A, double[][] B)
    {
        // allocate new output matrix, this helps eliminate aliasing
        double[][] C = new double[A.Length][];
        for (int i = 0; i < A.Length; i++)
            C[i] = new double[B[0].Length];

        // Simple multiplication using three nested loops
        for (int i = 0; i < A.Length; i++) // M
        {
            for (int j = 0; j < B[0].Length; j++) // N
            {
                for (int k = 0; k < A[0].Length; k++) // K
                {
S1:                 C[i][j] += A[i][k] * B[k][j];
                }
            }
        }
        return C;
    }
```

Figure 41 - Simple matrix multiplication.

Frist step to parallelization is to locate the induction variables used in all the loops and analyze their behavior. It is possible to use the algorithm presented in the section 6.4, because all the induction variables are assigned only once in every iteration and they are always increased by one. The following example shows the CIL code of the loop used to initialize the output matrix.

```
IL_000b:  br.s       IL_001e
IL_000d:  ldloc.0
IL_000e:  ldloc.1
IL_000f:  ldarg.2
IL_0010:  ldc.i4.0
IL_0011:  ldelem.ref
IL_0012:  ldlen
IL_0013:  conv.i4
IL_0014:  newarr     [mscorlib]System.Double
IL_0019:  stelem.ref
IL_001a:  ldloc.1
IL_001b:  ldc.i4.1
IL_001c:  add
IL_001d:  stloc.1
IL_001e:  ldloc.1
IL_001f:  ldarg.1
IL_0020:  ldlen
IL_0021:  conv.i4
IL_0022:  blt.s      IL_000d
```

Figure 42 - Initialization loop.

This is the entire loop and the induction variable is changed only once and it is increased by one. This is performed by the red colored instructions and the blue instructions implement the termination condition that reads the variable. The other loops

are very similar and their induction variables are located the same way. All the induction variables are located after the analysis is completed and there are three: `i, j, k`.

Next step is the aliasing analysis and the simple algorithm, presented in Figure 36, can be used to separate the result matrix from both parameters, because the array is assigned only once and the assigned value is a new array, which can be seen in the Figure 42. There are no other assignments to the output matrix with the exception of the matrix elements, but they have a value type, namely `double`. The algorithm is not able to separate the parameters, but they are only read in this algorithm and only writes can cause dependences.

Next step is subscript reconstruction, which is very simple, because all the subscripts contain only a simple variable loads, and all the subscripts of a single array can be analyzed because they all contain the same variable. More complex is the analysis of multidimensional arrays, because all the matrices are basically two-dimensional. The algorithm, presented in section 6.7.2, is able to locate all multidimensional arrays in the code, because they are accessed one dimension after another.

```
IL_0031:  ldloc.0
IL_0032:  ldloc.2
IL_0033:  ldelem.ref
IL_0034:  ldloc.3
IL_0035:  ldelema    [mscorlib]System.Double
IL_003a:  dup
IL_003b:  ldobj      [mscorlib]System.Double
IL_0040:  ldarg.1
IL_0041:  ldloc.2
IL_0042:  ldelem.ref
IL_0043:  ldloc.s    k
IL_0045:  ldelem.r8
IL_0046:  ldarg.2          // first dimension is in an argument
IL_0047:  ldloc.s    k
IL_0049:  ldelem.ref       // second dimension
IL_004a:  ldloc.3
IL_004b:  ldelem.r8        // the actual element
IL_004c:  mul
IL_004d:  add
IL_004e:  stobj      [mscorlib]System.Double
```

Figure 43 - Main statement in matrix multiplication.

This example shows the CIL code of the statement S1 in the Figure 41, which is the main statement that performs the actual multiplication. The red colored instructions are used to read a value from the matrix B. When the instruction 4b is analyzed then the instruction 49 is located as its source array and the instruction 46 is located as the next source. The last array is in parameter B and that is the main array. All the red instructions

are recognized as a single array access to multidimensional array with the main array stored in the parameter B.

The dependence testing can use algorithms from the [2] to analyze the main loop nest, because all the arrays and their indices have been analyzed. When the dependence testing is applied on the main loop nest, then there is only a single dependence from S1 to S1 with the direction vector `[=,=,*]` (explained in chapter 2 in [2]), which means that there is a loop-carried dependence from S1 to itself in the internal loop, but it is independent in both outer loops. This information can be directly used to parallelize this method, because every iteration of the two outer loops can be run in parallel which leads to the coarse grained parallelism – if the matrices are big enough then every iteration of the outer loop can be run a separate thread. Other possibility is to use loop interchange (chapter 5 in [2]) to move the internal loop outside, because then the dependence would change to `[*,=,=]` and the internal loop can be parallelized using vector instructions.

This is a simple algorithm, but it is used by many applications and the fact that it can be automatically parallelized in CIL code suggests that C# can be potentially used for high performance computing. The entire CIL code of the multiplication algorithm is on the DVD and Appendix A provides the table of content.

6.9.2 Vector addition

The vector addition is simpler than the matrix multiplication, because it does not use multidimensional arrays, but the implementation shown in Figure 44 passes all the vectors as parameters and that make any parallelization impossible without inter-procedural analysis, since the parameters are the main source of aliasing.

```
public void Add(double[] A, double[] B, double[] C)
{
    for (int i = 0; i < C.Length; i++)
        C[i] = A[i] + B[i];
}
```

Figure 44 - Vector addition.

This algorithm contains only one simple loop and it is not difficult to locate its induction variable, because it is increase by one once in every iteration. This is the same situation as in the matrix multiplication presented in the previous section.

Aliasing is a much bigger problem in this algorithm, because all the vectors are passed as parameters and there is no indication if they can be aliased or not. But since the method is short it is possible to use the algorithm presented in section 6.5.1 to separate

the output vector C from the others. This transformation uses code duplication and the following example shows the method after the code has been duplicated.

```
    public void AddNew(double[] A, double[] B, double[] C)
    {
        if (C != A && C != B)
        {
            // C represents an array different from A and B
            for (int i = 0; i < C.Length; i++)
S1:             C[i] = A[i] + B[i];
        }
        else
        {
            for (int i = 0; i < C.Length; i++)
                C[i] = A[i] + B[i];
        }
    }
```

Figure 45 - Vector addition after parameter separation.

The code in the first branch can be optimized, because C must be a different array then A and B. This optimization can be wasteful, but it can help tremendously in this case, because the method (or at least its first part) can be completely parallelized afterwards, using vector instructions or threads.

The following steps are very similar to the matrix multiplication and the only exception is that this algorithm does not contain multidimensional arrays. After dependence testing is completed, there is a single dependence in the first part of the method and it is from the statement S1 to itself with the direction vector `[=]`, which means that the statement depends on itself in the same iteration. This basically means that the statement is independent and the first part can be completely parallelized with vector instructions, which are ideal for this task, because they usually provide a single instruction for vector addition (at least for vectors of limited size).

6.10 Summary

The studied applications can be successfully analyzed using the transformations, presented in this chapter, for code preparation and existing algorithms for dependence testing. This suggests that it might be possible to parallelize .NET applications based on their CIL code, even though there are many problems that can complicate similar optimizations for general applications. It might be possible to parallelize special .NET applications and that may allow programmers to develop these types of applications in C#. For example, high performance applications might even be optimized in .NET with a similar efficiency as in C or FORTRAN, at least when they are written without some advanced features that can ruin the dependence testing, like unsafe code.

It is unfortunate that it was not possible to implement the dependence testing, but there are many technical details that have to be solved, before it is possible to use the testing algorithms presented in literature. The main problem is that it is not possible to omit anything, because then it would not be possible to prove any independence.

7 Conclusions

Automatic parallelization discussed in this work is a very difficult optimization, but most of the encountered problems were successfully solved and many presented solutions were implemented. The first steps necessary to parallelization have been successfully implemented and dependence testing has been analyzed in much detail, but the implementation of dependence tester and the actual parallelization proved to be too difficult, because it would be necessary to solve many technical details to use the algorithms presented in the literature. Another problem is that even the simplest parallelization requires most of the dependence testing to be implemented, because otherwise, it is impossible to prove any independence and the parallelization cannot be used at all.

Following recapitulation sums the achieved results in relation to separate steps presented in section 3.2.

7.1 Inlining

Method inlining has been successfully analyzed and implemented, but it proved to be very complicated even for simple C# programs that do not contain advanced features, like unsafe code or lambda functions. The final implementation is able to successfully inline more than fifteen calls into a single method and it can handle even very complicated code, for example: inline a method called in an inlined method, inline methods containing protected blocks (exception handling), inline method with parameters passed by reference or inline method that changes the maximal size of the execution stack. But it was mainly the complications connected with the implementation of inlining, that significantly slowed the entire project and that may have caused the fact that the dependence testing have not been implemented.

The optimizer, that is part of this work, is able to inline methods in an application, based on custom attributes assigned by programmers and the optimization is controlled by a special file that is used as a parameter for the optimizer. Further information can be found in Appendix B.

7.2 Preliminary code analysis

This step contains recognition of variables and control flow construct and both have been analyzed and implemented successfully. This analysis can be run using the unit tests (distributed with the project) and it is not used in the optimizer, because the optimizer does not need it since it does not implement dependence testing. Further information can be found in Appendix B or in the technical documentation of the project located on the DVD.

7.3 Dependence testing

Dependence testing is the most difficult step of the entire parallelization process. The problem with the dependence testing is that it must be implemented very extensively or it is not able to analyze even very simple programs, like matrix multiplication. The potential implementation must recognize if the result matrix does not depend on the parameters (if it is not a parameter). Then it must be able to analyze the behavior of induction variables in loops and it must be able to analyze array accesses with multiple subscripts, just to analyze a simple matrix multiplication and that is not all. The analysis of matrix multiplication is not possible without either one of the presented analyses. It is not possible to use incremental development for dependence testing, because it is a complex that requires most of the presented algorithms to be complete before it can be used.

The analysis of dependence testing concentrates on the specific features of CIL and the work presents many transformations that prepare the code so it can be analyzed by algorithms described in literature. The transformations were designed for special applications and they may not be efficient for general applications, but their analysis suggests, that is might be possible to parallelize C# applications (at least some specific applications).

The algorithms described in literature were not implemented, because there are many technical details that have to be addressed, before these algorithms can be ported for .NET Framework, but the transformations presented in chapter 6 are able to prepare the CIL code so the algorithms can be used without significant modifications.

7.4 Status of the work

The final implementation is divided in two parts, first part is a framework designed to control the optimizations and the second part is the implementation of the transformations that prepare the code for parallelization.

The optimization framework is based on special attributes that identify optimized methods and the optimization is controlled by a special XML file. Its usage is similar to a makefile. The framework is able to inline selected methods and it is described in the Appendix B.

The second part contains the implementation the algorithms presented in this work and the correctness of these algorithms is checked by unit tests. The implementation contains method inlining and then there is the detection of control-flow constructs and variables. The dependence testing was not implemented, because there are many technical details that must be solved, before it is possible to use the algorithms presented in literature. The algorithms for dependence testing are usually designed for older languages, like FORTRAN, and their application on languages with reference semantics requires small modifications, because they must be able to work with the code including many special constructs introduced to the .NET by more complicated languages, like F#. The optimizer cannot be used for practical applications, because there are many technical details that must be solved before the dependence testing can be completed and parallelization is impossible without the dependence analysis.

The functionality and correctness of the implemented algorithms is controlled by a series of unit tests designed to check all algorithms and even their separate parts. These tests are not based on some well-known applications, like benchmarks, but they were developed along with the project, because the implementation has not been completed, so it could be tested on general applications. The tests are distributed along with the project and they can be run directly from the Visual studio. Detailed information about tests can be found in the readme file on the DVD (Appendix A described the DVD contents).

Even though this work is aimed mostly on the .NET platform, it should be possible to apply most of the presented solutions on Java as well, thanks to the similarity of their bytecode and runtime architecture.

7.5 Further work

The main step that has to be completed is the dependence testing, because it is the necessary step before the actual parallelization. The analysis of dependences has been performed, but the implementation proved to be too complicated, because there are many technical details that have to be solved before the parallelization algorithms can be used. The algorithms described in literature must be modified for the .NET platform and these modifications, while small, would require a lot of work.

The parallelization was the ultimate goal of this project, even though it was not expected that there would be a working implementation, because the main purpose of this work was only to explore the possible approaches to automatic parallelization of .NET applications. The project did never make it its goal to actual implement the parallelization, but we were hoping it would be possible, at least for some simple program. It was not possible to implement any actual parallelization algorithm because the dependence analysis proved to be too difficult to implement even though it has been thoroughly analyzed. Even though the implementation was not possible, the analysis of the dependence testing showed that it should be possible to parallelize at least specific applications based on loops and arrays, both of which represent the most common opportunity for parallelization. Another reason why the parallelization was not implemented is the fact that it is very difficult to parallelize a .NET application, because CIL does not contain any vector instructions and threads are difficult to use.

The results are not at all negative, even though they did not meet our initial expectations. There are many things that must be completed, before the presented implementation would be able to actually parallelize an application, but the results of the analysis of dependences suggests that it should be possible. The most important part that has to be implemented is dependence testing along with some transformations enhancing the final parallelism. Basic dependence testing is complicated, but it is possible to implement and then it has to be continually improved to optimize wider array of applications. The improvements usually entail implementing more enhancing transformations and it is necessary to improve the analysis of array subscripts and loops.

The overall study of parallelism and other complex transformations and programing technique shows that there will most likely always be a place for parallel programming performed by a human programmer, because even the best compilers are able to find independences only between short code fragments and even that is extremely difficult. Humans however can design application to run many different tasks

concurrently or even distribute the application across multiple computers, which is most likely impossible to do automatically, at least in a general case.

Bibliography

1. Ecma-335. [Online] 2012. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-335.pdf.
2. **Allen, Randy and Kennedy, Ken.** *Optimizing Compilers for Modern Architectures.* San Francisco : Morgan Kaufmann Publishers, 2001.
3. **Muchnick, Steven S.** *Advanced Compiler Design Implementation.* San Francisco : Morgan Kaufmann Publishers, 1997.
4. **Aho, Alfred V. and Lam, Monica S.** *Compilers: principles, techniques, & tools.* Boston : Pearson/Addison Wesley, 2006.
5. **Ottoni, Guilherme, et al.** Automatic thread extraction with decoupled software pipelining. *In Proceedings of the 38th IEEE/ACM International Symposium on Microarchitecture.* s.l. : IEEE Computer Society, 2005, pp. 105--118.
6. **Raman, Arun, et al.** Speculative Parallelization Using Software Multi-threaded Transactions. [Online] 2010. [Cited: 3 25, 2013.] http://liberty.princeton.edu/Publications/asplos15_smtx.pdf.
7. **Nie, Jiutao, et al.** Vectorization for Java. *Network and Parallel Computing.* Berlin : Springer Berlin Heidelberg, 2010, pp. 3-17.
8. **Artigas, Pedro V., et al.** Automatic loop transformations and parallelization for Java. *Proceedings of the 14th international conference on Supercomputing.* New York : s.n., 2000, pp. 1-10.
9. **El-Shobaky, Sara, El-Mahdy, Ahmed and El-Nahas, Ahmed.** Automatic vectorization using dynamic compilation and tree pattern matching technique in Jikes RVM. *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems.* New York, NY : s.n., 2009, pp. 63-69.
10. **Felber, Pascal A.** Semi-automatic Parallelization of Java Applications. *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE.* Berlin : Springer Berlin Heidelberg, 2003, pp. 1369-1383.
11. *JAPS: an automatic parallelizing system based on JAVA.* **Du, Jiancheng, Chen, Daoxu and Xie, Li.** 4, 1999, Science in China Series E: Technological Sciences, Vol. 42, pp. 396-406.
12. **Chen, M.K. and Olukotun, K.** The Jrpm system for dynamically parallelizing Java programs. *Computer Architecture, 2003. Proceedings. 30th Annual International*

Symposium on}, title={The Jrpm system for dynamically parallelizing Java programs. s.l. : IEEE Computer Society, 2003, pp. 434-445.

13. *Run-Time Support for the Automatic Parallelization of Java Programs.* **Chan, Bryan and Abdelrahman, Tarek S.** 1, 2004, The Journal of Supercomputing, Vol. 28, pp. 91-117.
14. **Schulte, Wolfram, et al.** *Automatic Parallelization in a Tracing Just-in-Time Compiler System. 20110265067* United States, 4 21, 2010.
15. **Dittamo, Cristian, Cisternino, Antonio and Danelutto, Marco.** Parallelization of C# Programs Through Annotations. *Computational Science – ICCS 2007.* Berlin : Springer Berlin Heidelberg, 2007, pp. 585-592.
16. *Freely Annotating C#.* **Cazzola, Walter, Cisternino, Antonio and Colombo, Diego.** 10, 2005, Journal of Object Technology, Vol. 4, pp. 31-48.
17. **Chatterjee, Soumya S. and Gururaj, R.** Lazy-Parallel Function Calls for Automatic Parallelization. *Computational Intelligence and Information Technology.* Berlin : Springer Berlin Heidelberg, 2011, pp. 811-816.
18. **Freeman, Adam.** *Pro .NET 4.0 Parallel Programming in C#.* New York : apress, 2010.
19. **Leroy, Xavier.** Java bytecode verification: algorithms and formalizations. [Online] http://gallium.inria.fr/~xleroy/publi/bytecode-verification-JAR.pdf.
20. **Lindholm, Tim and Yellin, Frank.** The JavaTM Virtual Machine Specification. [Online] 1999. http://docs.oracle.com/javase/specs/jvms/se5.0/html/VMSpecTOC.doc.html.

Table of figures

List of Abbreviations

- .NET Framework – A development platform implemented by Microsoft for the Windows operating system. It is compatible with CLI.
- CIL – Common intermediate language is an intermediate language similar to Java bytecode and it is executed by a CLR virtual machine
- CLI – Common language infrastructure is the general architecture common language platform compatible with the [1]. The standard contains a detailed description of the architecture in Chapter I.
- CLR – Common language runtime is the virtual machine responsible for execution of applications compiled to a bytecode compatible with CIL.
- Ecma International – An industry association founded in 1961, dedicated to the standardization of information and communication systems.
- Ecma-335 – An international standard that specifies the architecture of CLI and its parts.
- Java – A modern object-oriented language which is usually compiled to bytecode and then executed by a virtual machine called JVM. The bytecode has a structure and function similar to the CIL.
- JRE – Java runtime environment is a platform capable of execution of Java applications. It is more than just a JVM, because it must contain an implementation of Java standard libraries and other features.
- JVM – Java virtual machine is a machine able to execute applications compiled to the Java bytecode.
- MSIL – Microsoft intermediate language is a specific intermediate language compatible with the CIL as it is defined by the [1]. The MSIL is produced by the standard C# compiler available in the Visual studio development environment.
- NOW – network of workstations is a distributed system composed of many common computers (usually PCs) connected by a network.
- SIMD – instructions that can perform a single operation on multiple data, otherwise called vector instructions.

Appendix A - DVD content

The DVD contains the implementation of the Parallax project accompanied by its generated documentation and other documents related to the work. The following structure explains the folder structure, along with its contents. Very important document is readme file that describes the deployment, compilation and execution of the ParallaX project or its parts.

- **ParallaX** – contains the ParallaX project and all its source files
  - **ParallaX** – the main project that contains all modules and tests
    - **Architecture** – diagrams describing projects architecture
    - **TestResults** – results of the unit tests
    - other – other folders contain modules of the project
  - **ParallaXExample1** – sample project used to test ParallaX project
  - **ParallaXHelp.shfbproj** – project for the sandcastle tool that is used to generate the documentation from XML comments in source files
- **Documentation**
  - **Documentation.chm** – generated documentation for ParallaX
  - **ECMA-335.pdf** – actual version of the standard ECMA-335
  - **MatrixMultiplicationCIL.txt** – CIL code for the matrix multiplication analyzed in section 6.9.1
  - **Readme.docx, Readme.pdf** – instructions for compilation and execution of the ParallaX project
  - **ParallaX.docx, Parallax.pdf** – electronic version of this text

Appendix B - ParallaX optimizer

The ParallaX project has been designed as a stand-alone optimizer which would be used after compilation to parallelize .NET applications. The actual parallelization has not been implemented, but the project contains a framework that would allow programmers to decide the way their methods should be optimized or inlined and this framework is fully functional. Then there is the implementation of method inlining and code verification, which prepares the code for further analysis. Finally there is the code analysis, which is able to recognize control-flow constructs and variables used in the code. The project was implemented in C# under the Visual studio 2010 and the following diagram shows its general structure.

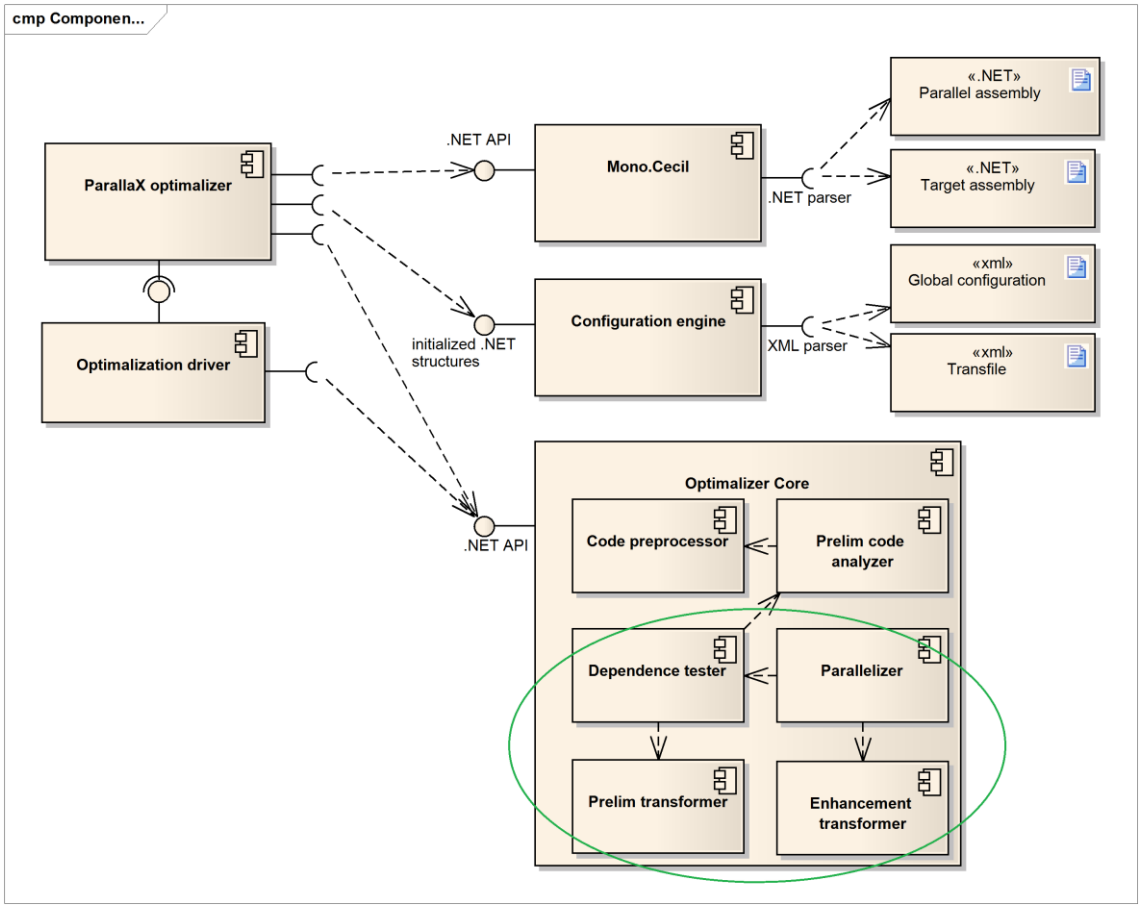


Figure 46 - Architecture of ParallaX modules.

The diagram shows all the modules included in the project and all of them are implemented using the algorithms presented in this work, with the exception of the modules indicated by the green ellipse, which were analyzed only theoretically because their implementation proved to be too difficult.

The architecture of the project is documented in the Enterprise architect and the documentation is in the Architecture module of the project.

Optimization framework

The project has been designed as a stand-alone optimizer for existing .NET applications and the actual implementation contains the configuration that can be used to select the application and its methods that should be optimized, but the actual optimization has not been implemented.

The optimizer uses a transfile that is similar to makefile, because it is used to locate the optimized application and it can contain other parameters that specify the desired optimization. The transfile is a XML file that is loaded and stored using the XML serialization provided by the standard .NET library, while its structure must correspond to the structure of the class `ParallaX.Common.Configuration.Transfile`. The general behavior of the optimizer is configured by global configuration file that is loaded by the XML serialization and its structure must be compatible with the class `ParallaX.Common.Configuration. GlobalConfiguration.`

The optimized application must use special attributes provided in the module `ParallaX.Interface` to specify which methods should be optimized and how.

- The attribute `Parallelize` is used to identify methods which should be parallelized by the optimizer.
- The `Inline` attribute is used to specify how should be this method inlined if it is called by an optimized method. Methods are not inlined by default, since it may be impossible for some library API, but programmers can specify certain methods that can be inlined. Methods are inlined only in methods selected for parallelization attribute and nowhere else.
- The `Dependence` attribute can be used to help the optimizer to analyze dependences caused by the specified method and it should be used for methods that cannot be inlined.

Framework capabilities

The actual implementation of the optimizer is able to use the transfile to find target assembly and it is able to use the attributes to locate methods for optimization, but it can only inline selected methods since the parallelization was not implemented. The framework provides special functions to generate empty transfile or global configuration,

which can be used to simplify the optimization process, because the empty files contain all the required properties set to default values. The inlined methods should belong to the same module as the method that called them, because the different modules may cause problems during inlining.

The optimizer can be run with parameter –h that prints a help text containing all the available parameters and their function.

Optimizer modules

The optimizer modules closely follow separate steps of optimization presented in chapter 3. Each step is implemented in a single module and the module uses the algorithms presented in the appropriate chapter of this thesis. The most important modules are `ParallaX.Core.CodePreprocessor` and `ParallaX.Core.PrelimCodeAnalyzer`, because the dependence testing has not been implemented. `CodePreprocessor` contains method inlining and code verification as it is presented in chapter 4 and `PrelimCodeAnalyzer` implements the analyses discussed in chapter 5.

The module `ParallaX.Common` contains general interfaces and functions used by the entire project and it is used by every other module. The most important part of this module is a set of extension methods for the Cecil library, which implement certain functionality originally unavailable in the library, like code cloning.

Very important module is `UnitTest` which contains all the unit tests used to verify the algorithms implemented in the other modules. These tests must be used to run all the algorithms implemented in this project, because the actual optimizer was not completed. The compiled optimizer is able to inline selected methods into the optimized methods according the attributes presented in the previous sections, but the test can execute almost every method in the optimizer and verify its results. These tests use the test project `ParallaXExample1` distributed along with the ParallaX project.

Used libraries and tools

This implementation relies heavily on the Cecil library for CIL parsing and reconstruction. The Cecil library is used to decode CIL assemblies to objects and their methods which is similar to standard reflection, but Cecil is able to decode method body to a stream of instructions, unlike the standard reflection library. The library is used as the basis for further analysis and it is was a necessary tool for this project.

There are other tools which were used to test, verify and analyze the code produced by the optimizer. The most important are PEVerify and ILDasm; both are distributed as part of the Microsoft SDK which contains tools and libraries for the development of windows applications. PEVerify has been used to verify inlined code and it has been invaluable for locating errors in the produced CIL. ILDasm has served as a general tool for the disassembly of the analyzed code.