

# TABLE OF CONTENTS

<b>1. Introduction.....</b>	<b>3</b>
<b>2. Literature Survey.....</b>	<b>4</b>
2.1 Introduction to Mono Framework.....	4
2.2 But if .NET already existed, why Mono?.....	4
2.3 Technical advantages of using Mono.....	5
2.4 Languages supported by Mono.....	5
<b>3. Improvement Opportunity: Define Phase.....</b>	<b>6</b>
3.1 JIT Engine.....	6
3.2 Building a Code Optimizer.....	7
3.3 But why loop unrolling?.....	7
3.4 Inferences about loop unrolling.....	11
<b>4. Methodology.....</b>	<b>12</b>
4.1 How does loop look like in the CIL.....	12
4.2 Methodology.....	12
<b>5. Implementation.....</b>	<b>16</b>
5.1 The program we set out to unroll.....	16
5.2 Program that brought the unrolling.....	18
5.3 The CIL Instructions.....	23

5.4 Enhanced GMCS.....	27
<b>6. Performance Measure and Analysis of Enhanced GMCS.....</b>	<b>30</b>
6.1 Program that was unrolled.....	30
6.2 The unrolled program.....	30
<b>7. Extension to the Project.....</b>	<b>34</b>
7.1 Benefits of loop fusion.....	34
7.2 Program on which loop fusion was done.....	34
7.3 Program containing the loop fusion.....	36
7.4 Program that brought above the loop fusion.....	37
7.5 Performance analysis.....	42
<b>8. Conclusion.....</b>	<b>44</b>
<b>9. Future Work.....</b>	<b>45</b>
9.1 What do we have at the end of our project.....	45
9.2 Suggestions for further work.....	45
<b>10. References.....</b>	<b>46</b>

# 1. INTRODUCTION

Mono is a .NET equivalent framework which is meant for Linux environments. The specialty of these frameworks is that they can execute a variety of programming languages on different hardware platforms.

The compilation engine of Mono comprises of the Just in time compiler, partial and full Ahead of time compilation. **Our project focuses on easing the work of the Just in time compiler by building a code optimizer phase just before the just in time compilation phase.**

The problem statement was framed around the month of September and we have spent the next six months trying to incorporate a loop optimization in the code optimizer.

CIL- common intermediate language is the instruction set to which these programming languages are converted, which is then executed by Mono. Today many compilers that target the CIL are known to generate fairly straightforward CIL code leaving most of the work to the JIT engine. An optimizer that could perform some tasks before the JIT compiler hits the code would be useful. Although Mono code generator is fairly good, and getting better, the most effective optimizations are never turned on for JIT-use as they are time consuming.

We call this code optimizer the ‘Enhanced GMCS’, which does a loop optimization-loop unrolling, before handing over the CIL to the JIT compiler. The word ‘GMCS’ is used because; we are targeting CSharp programs, whose front end compiler happens to be GMCS. In order to do this, we have used the ‘Cecil’ library which allows us to read, browse and modify the assembly before writing it back as modified CIL.

We intended to see how the performance of the new framework which consisted of the code optimizer varied from the performance without it. The goal was to see if the speedup gained by loop unrolling and adding an additional phase of optimization to the compilation engine of Mono, exceeded that of the JIT compilation, without the code optimizer.

## 2. LITERATURE SURVEY

### ***2.1 Introduction to Mono framework***

Mono is a .NET equivalent framework which is meant for Linux environments. It is like a two mouthed funnel which enables a variety of programming languages to execute on a variety of hardware platforms.

**CIL-Common Intermediate Language** is the instruction set to which these programming languages are converted, which is then executed by these frameworks.

CIL is the lowest level human readable programming language defined by the CLI specification, and is used by the .NET framework and mono. It is an **object oriented, stack based** assembly language.

Mono comprises of the **library** which provides thousand of built in classes to improve productivity and the **Common language runtime**, which is an application virtual machine that provides features like memory management, exception handling and security.

### ***2.2 But if .NET already existed, why Mono?***

1. .NET is meant only for windows platform
2. Its closed source, while Mono is open source, and is dual licensed under free software and proprietary software.
3. As told by the Mono forum, .NET is a nebulous initiative, a part of which is the development framework, whereas Mono is restricted to the implementation of the development framework alone.

## ***2.3 Technical advantages of using Mono***

1. Built on the success of .NET, developers have experience building C# applications
2. It is cross platform, runs on Linux, Windows, MAC OS X, BSD, Sun Solaris, Nintendo Wii, Sony Playstation 3, Apple iPhone. And on architectures like x86, x86-64, ARM, PowerPC, s390, s390x, IA63, Alpha.
3. Common Language Runtime: You can choose to write code in a variety of programming languages to be executed by Mono. You can write code in one CLR language and interoperate it with any other CLR language
4. Provides high level language programming constructs, like automatic memory management, reflection, generics, threading, which allow the programmers to concentrate on the application development rather than on writing system infrastructure code
5. The base class library, which provides user interface, data access, database connectivity, cryptography, web application development, numeric algorithms and network connectivity.

## ***2.4 Languages supported by Mono***

The mono project supports C#, Basic and Ilasm compilers, but there are open source and commercial compilers that can be used with Mono. The languages should compile to the CIL which can execute on Mono. Microsoft's managed C++ for instance, does not run on mono. The languages should have the potential to be truly platform independent.

The languages which have the potential to be converted to a CIL and have compilers prepared for the same, have been listed in the following link: [Languages supported by mono](#)

## 3. IMPROVEMENT OPPORTUNITY: DEFINE PHASE

### 3.1 *JIT Engine*

The compilation engine of Mono comprises of the Just in time compiler, partial and full Ahead of time compilation. We focus on the just in time compilation. It converts the CIL into the 3 address bytecode, where the complex opcodes are broken down. This is the first pass of the JIT compiler, the second pass includes decomposing large instructions, and the third includes some optimizations, like copy and constant propagation, dead code elimination, simple branch optimizations.

Today many compilers that target the CIL are known to generate fairly straightforward CIL code leaving most of the work to the JIT engine. An optimizer that could perform some tasks before the JIT compiler hits the code would be useful. Although mono's code generator is fairly good, and getting better, the most effective optimizations are never turned on for JIT-use as they are time consuming.

The code optimizer would convert a CIL stream into a different CIL stream that is compatible but has applied some optimizations before the JIT sees them. These optimizations could be a lot more complex than what the JIT compiler is capable of doing like loop unrolling, or processing a file to automatically multithread some pieces of program

The traditional optimization techniques generally used by compilers are listed below:

- Method inlining
- Constant propagation
- Dead code elimination
- Loop invariant hoisting and loop unrolling

- Common sub expression elimination
- Global value numbering
- Null check and Range check elimination

The optimizations that JIT currently does

- Dead code elimination
- Constant and copy propagation
- Many Array bound checks and null checks are eliminated

The above optimizations are performed on the Medium intermediate representation (MIR), which the JIT compiler converts the CIL into. This MIR is a linear IR, a 3 Address code.

### ***3.2 Building a code optimizer***

We call this code optimizer the ‘enhanced gmcs’, which does loop unrolling before handing over the CIL to the JIT compiler. In order to do this, we have to use a library called the **Cecil library**, which allows us to read, browse and modify the assembly before writing it back as modified CIL.

We first tried to include a print statement by reading a CIL and adding a print statement to it, before writing it back.

### ***3.3 But why loop unrolling? Does it really have significant benefits with respect to performance?***

#### **Experiment 1:**

##### Without unroll

```
for(i=0;i<a.length();i+=1)
```

```
a[i]=i;
```

### After unroll

```
for(i=0;i<a.length();i+=2)
    a[i]=i;
    a[i+1]=i+1;
```

### Setup

We called the above two methods separately 1 billion times and considered the average of the time of billion executions.

### Observation

- If the number of iterations is below 10,000 there is performance improvement of loop unrolled program over otherwise. As the number of iterations increase, the performance difference decreases.
- If we loop unroll more number of times, there does not seem to be improved performance gain over unrolling twice. The above two observations have been tabulated in **Table 1**.
- Another point to note is that, if we change the iteration bound parameter to a constant or a parameter having the value of the count like 'n', instead of a.length(). When we did this, there was no performance gain of loop unrolled code over the former code. This was uninterpretable. This point has been tabulated in **Table 2**.



**Table 1. Performance without and with loop unrolling with variable iterations**

Array length	Performance without loop unrolling(ns)=t <sub>1</sub>	Performance with loop unrolling(ns)=t <sub>2</sub>	Performance gain of loop unrolling (t <sub>1</sub> /t <sub>2</sub> )
12	21.9	15.95	1.373
10000	11753	9733	1.207
10000000	2571063	2627573	0.9784

**Table 2. Performance without and with loop unrolling with fixed iterations**

iteration bound parameter used	Performance without loop unrolling(ns)=t <sub>1</sub>	Performance with loop unrolling(ns)=t <sub>2</sub>	Performance gain(t <sub>1</sub> /t <sub>2</sub> )
n=12	15.79	16.37	0.964
array.length()	21.9	15.95	1.373

### Analysis

- The loop unrolled program shows a performance gain since the two instructions within the loop can be scheduled simultaneously during each iteration. Loop unrolling exploits instruction level parallelism for machines with multiple functional units.
- Loops with large number of iterations when unrolled, tend to increase the overall size of the program. Loop unrolling is about space to performance trade off. Perhaps that is why when we increased the iteration bound to a huge number, loop unrolling did not show significant gain in performance.
- But increasing the unroll factor did not make significant difference. The reason could be that there are only two ‘add’ functional units. Hence at most two add instructions can be scheduled simultaneously. More unrolling will only make the unrolled instructions wait.

## Experiment 2:

### Without unroll

```
for (i=0; i<10; i+=1)
{
    sum+=a[i];
}
```

### After unroll

```
for (i=0; i<10; i+=2)
{
    sum1+=a[i];
    sum2+=a[i+1];
}
```

### Setup:

Same as Experiment 1

### Observation:

There is very little variation in the performance of the former and latter codes. Both take 24ns.

### Analysis

Although loop unrolling could exploit instruction level parallelism as shown in Experiment 1, Possible increased register usage in a single iteration to store temporary variables may have reduced performance, although much will depend on the possible optimizations.

In the above program four registers are being used, one for sum1, one for sum2 and the other for array, the last for the increment variable, as against the loop in Experiment 1, where 2 registers are being used.

### ***3.4 Inferences about loop unrolling***

1. The overhead in "tight" loops often consists of instructions to increment a pointer or index to the next element in an array (pointer arithmetic), as well as "end of loop" tests. If an optimizing compiler or assembler is able to pre-calculate offsets to each individually referenced array variable, these can be built into the machine code instructions directly, therefore requiring no additional arithmetic operations at run time.
2. Significant gains can be realized if the reduction in executed instructions more than compensates for any performance reduction caused by any increase in the size of the program.
3. If the statements in the loop are independent of each other (i.e. where statements that occur earlier in the loop do not affect statements that follow them), the statements can potentially be executed in parallel.

**Considering the results of the above experiments and the inferences drawn, we decided to unroll Execution time simple ‘for’ loops.**

## **4. METHODOLOGY**

### ***4.1 How does a loop look like in the CIL***

Loop unrolling, requires changes in three places: the increment variable, the condition variable, and the replication of the body. In order to do this we wrote the CIL into a file as each instruction was being read. This enabled us to see which part of the CIL, the increment instructions were, which part, the condition checks were and which part, the body of the loop was.

The CIL is a set of pop push operations, performing the tasks. A loop consists of a branch instruction to the check condition, which involves pushing the number to be checked with, on the stack and stores the increment parameter in a variable. The conditional branch instruction compares them to decide if the body needs to be executed or not.

After the body gets executed, the increment variable is pushed on the stack, the amount to be incremented is also pushed on the stack and the add instruction is performed. These instructions have been shown in detail in section 5.3 The CIL INSTRUCTIONS.

### ***4.2 Methodology***

#### **Process of loop unrolling**

We wrote a CSharp program which does the following.

We read the instruction which pushes the amount to be incremented and replaced this with an increment of 2.

The body within the loop is found between the branch instruction mentioned above and the check condition. We stored the position of both these instructions and inserted before the check condition the new loop body.

The full code can be viewed in **Code 3** in Implementation section.

### Process of integrating it into mono

Mono has the capacity to be embedded in applications. We write a C program, which is the application in question and then, the following steps are involved:

- Initialize mono runtime

```
MonoDomain domain = mono_jit_init (domain_name);
```

- Load the required assembly

```
MonoAssembly assembly = mono_domain_assembly_open (domain, "file.exe");
```

File.exe will be loaded into the domain

- Run it on jit

```
retval = mono_jit_exec (domain, assembly, argc - 1, argv + 1);
```

- Shutting down the runtime

```
mono_jit_cleanup (domain);
```

- Compiling the C program

```
$ gcc sample.c `pkg-config --cflags --libs mono-2`
```

This links the c program which we call 'sample.c' with the JIT runtime libraries

When we run the above command, effectively our assembly gets executed. Here our assembly will be the CSharp program that does the loop unrolling on a given csharp program containing a simple 'for' loop, and produces the modified/loop unrolled program.

The full code can be viewed in **Code 5** in the section Implementation.

## Enhanced GMCS

To bring in the picture of enhanced gmcs back, we have written a script, which calls the gmcs on the source code, then calls the C Program, which together produce the loop unrolled assembly. Then the user can execute mono on the loop unrolled assembly. So the commands in question will be:

1. `exec script- includes 'gmcs helploopunroll.cs; gcc sample.c'` which produces `assembly.exe`
2. `mono assembly.exe`

The full code can be viewed in **Code 6** in the section Implementation.

**This enhanced GMCS is the required code optimizer.**

## **Assumptions**

1. Detection of loop: we have assumed that every backward jump signifies a loop, in an assembly level program.
2. We have considered only single block execution time 'for' loops only, especially the ones with arrays in them.

**Supporting Assumption 1:** the kind of control flow constructs present are:

1. If-else statements which usually have forward conditional jumps
2. Goto statements are unconditional backward or forward jumps
3. Function calls are of type 'call' and not branch
4. Loops have conditional backward jumps

## 5. IMPLEMENTATION

### ***5.1 The program that we set out to unroll***

#### **Program that was unrolled**

```
using System;
using System.Diagnostics;

class Program1
{
    const int _max = 1000000000;
    static void Main()
    {
        int[] array = new int[12];
        for (int i = 0; i < _max; i++)
        {
            Method1(array);
        }
    }

    static void Method1(int[] array)
    {
        // Initialize each element in for-loop.
        for (int i = 0; i < array.Length; i++)
        {
            array[i] = i;
        }
    }
}
```

#### **Code 1**



## The unrolled program

```
using System;
using System.Diagnostics;

class Program1
{
    const int _max = 1000000000;

    static void Main()
    {
        int[] array = new int[12];
        for (int i = 0; i < _max; i++)
        {
            Method1(array);
        }
    }

    static void Method1(int[] array)
    {
        // Initialize each element in for-loop.
        for (int i = 0; i < array.Length; i++)
        {
            array[i] = i;
            array[i+1] = i+1;
        }
    }
}
```

## Code 2

## 5.2 Program that brought about the above unrolling

All the classes referred below are found in Cecil library

```
using System;
using System.Collections;
using System.Diagnostics;
using System.Linq;
using System.Collections.Generic;
using System.Text.RegularExpressions;
using System.Text;
using System.Reflection;
using System.IO;
using Mono.Cecil;
using Mono.Cecil.Cil;
using Mono.Collections.Generic;

public class ceciltry1{
    const int _max = 1;
    public static void Main(string[] args)
    {
        int i=0,k=0;

        //Gets all types which are declared in the Main Module of
        "MyLibrary.dll"
        //Here types refers to classes

        AssemblyDefinition myLibrary = AssemblyDefinition.ReadAssembly
(args[0]);
        foreach (TypeDefinition type in myLibrary.MainModule.Types)
        {
```

```

//Get all methods in the class
foreach( MethodDefinition method in type.Methods)
{
    if(method.FullName=="System.Void
        Program1::Method1(System.Int32[])")
    {
        i=0;
        k=0;
        int[] j=new int[5];    //j is used to hold the index
                                //of the method instruction
                                //before add
        int[] branch=new int[5]; //to store the indices of
                                //branch      instructions      in
                                // eachloop of the method
        int[] ldloc=new int[5]; //to store the indices of
                                //the instruction till which the
                                //body needs to be replicated

//Gets the CilWorker of the method for working with CIL
//instructions

        ILProcessor worker = method.Body.GetILProcessor();
        bool usearray=false;

        foreach( Instruction instruct in method.Body.Instructions)
        {
            //take the instruction number of branch

            if(instruct.OpCode.FlowControl==FlowControl.Branch) {
                branch[k]=i;
            }
        }
    }
}

```

```

        if (instruct.OpCode.FlowControl == FlowControl.Cond_Branch)
        {
            j[k] = i - 7;
            ldloc[k] = i - 8;
            k++;
        }

        i++;

//checking if there is an array instruction within the loop only
//then we loop unroll

        if (instruct.OpCode.Code == Code.Stelem_I4)
        {
            usearray = true;
        }
    }

    if (usearray == true)
    {
        //changing the increment variables of all loops

        for (i = 0; i < k; i++) {
            Instruction ins = method.Body.Instructions[j[i]];

            //Change the increment variable to 2
            Instruction changeIncrVarSentence;
            changeIncrVarSentence =
                worker.Create(OpCodes.Ldc_I4_2);

            worker.Replace(ins, changeIncrVarSentence);
        }
    }

```

```

//for each loop
for(i=0;i<k;i++)
{
    //we unroll only method1
    if(i==0&&method.FullName=="System.Void
Program1::Method1(System.Int32[])"){
        Instruction ins1,ins2,ins3,ins4,ins5,ins6,ins7,ins8;

        ins1= worker.Create(OpCodes.Ldarg_0);
                //load the array

        ins2= worker.Create(OpCodes.Ldloc_0);
                //load the current array index

        ins3= worker.Create(OpCodes.Ldc_I4_1);
                //load 1 onto stack

        ins4= worker.Create(OpCodes.Add);
                //add 1+arrayindex

        ins5= worker.Create(OpCodes.Ldloc_0);
                //load value onto stack

        ins6= worker.Create(OpCodes.Ldc_I4_1);
                //load 1 onto stack

        ins7= worker.Create(OpCodes.Add);
                //i+1

        ins8= worker.Create(OpCodes.Stelem_I4);
                // array[arrayindex+1]=i+1;

```

```

worker.InsertBefore(method.Body.Instructions[ldloc[i]], ins1);
    ldloc[i]++;
worker.InsertBefore(method.Body.Instructions[ldloc[i]], ins2);
    ldloc[i]++;
worker.InsertBefore(method.Body.Instructions[ldloc[i]], ins3);
    ldloc[i]++;
worker.InsertBefore(method.Body.Instructions[ldloc[i]], ins4);
    ldloc[i]++;
worker.InsertBefore(method.Body.Instructions[ldloc[i]], ins5);
    ldloc[i]++;
worker.InsertBefore(method.Body.Instructions[ldloc[i]], ins6);
    ldloc[i]++;
worker.InsertBefore(method.Body.Instructions[ldloc[i]], ins7);
    ldloc[i]++;
worker.InsertBefore(method.Body.Instructions[ldloc[i]], ins8);
    ldloc[i]++;

    }
    }//end of for loop
} // end of if loop

} //end of for each method
} //end of for each type
myLibrary.Write("try3.exe");

```

### **Code 3**

The CIL instructions pertaining to the loop in the below program-Code 4, is mentioned in section 5.3 so as to give an idea of how the CIL instructions look like

```
int i;
int[] a;
int sum=0;
int sum1=0,sum2=0;
Stopwatch sw = new Stopwatch();
a=new int[10];

sw.Start();
for(i=0;i<10;i+=2)
{
    a[i]=i;
    a[i+1]=i+1;
}
for(i=0;i<10;i+=2)
{
    sum+=a[i];
    sum1+=a[i+1];

}
sum=sum+sum1;
```

#### **Code 4**

### ***5.3 The CIL INSTRUCTIONS pertaining to the above unrolled loop is shown below***

IL_0000: ldc.i4.0	//push 0 onto stack
IL_0001: <u>stloc</u> .2	//pop stack value into local var2(sum).

IL_0002: <u>newobj</u> System.Void System.	//..declaring the stop watch object and
Diagnostics.Stopwatch:: <u>ctor</u> ()	storing it on top of stack
IL_0007: <u>stloc</u> .3	//pop stack value into local var3(stopwatch)
IL_0008: ldc.i4 10	//push 10 on top of stack
IL_000d: <u>newarr</u> System.Int32	//declaring a new array object
IL_0012: <u>stloc</u> .1	//storing array obj in local var1(array)
IL_0013: ldc.i4.0	//push 0 ontop of stack
IL_0014: <u>stloc</u> .0	//store 0 in local var0 (i)

/\*\*\*\*\*\*begin loop 1 of initialization\*\*\*\*\*\*/

IL_0015: <u>br</u> IL_002a	//branch instruction
IL_001a: <u>ldloc</u> .1	//load var1(array) ontop of stack
IL_001b: <u>ldloc</u> .0	//load var0(i) as index ontop of stack
IL_001c: <u>ldloc</u> .0	//load var0(i) as value ontop of stack
IL_001d: stelem.i4	//array[index i]=value i
IL_001e: <u>ldloc</u> .1	//load var1(array) ontop of stack
IL_001f: <u>ldloc</u> .0	//load var0(i) ontop of stack
IL_0020: ldc.i4.1	//load 1 ontop of stack
IL_0021: add	//index=i+1
IL_0022: <u>ldloc</u> .0	//load var0(i) ontop of stack
IL_0023: ldc.i4.1	//load 1 ontop of stack
IL_0024: add	//value=i+1
IL_0025: stelem.i4	//array[index]=value

----loop condition testing begin

IL_0026: <u>ldloc</u> .0	//load var0(i) ontop of stack
IL_0027: ldc.i4.2	//load 2 ontop of stack
IL_0028: add	//i+2
IL_0029: <u>stloc</u> .0	//i=i+2
IL_002a: <u>ldloc</u> .0	//load var0(i)ontop of stack



```

IL_002b: ldc.i4 10                //load 10(threshold of loop) ontop stack
IL_0030: blt IL_001a              //compare top of stack and second top...
                                   //...if i<10 branch back

```

----loop condition testing end

```

/*****end loop 1 *****/

```

```

IL_0035: ldloc.3                  //the next few statements refer to
                                   //intermediate statements

```

```

IL_0036: callvirt System.Void System.
        Diagnostics.Stopwatch::Start()

```

```

IL_003b: ldc.i4.0                //load 0 ontop of stack
IL_003c: stloc.0                 //store 0 to var0(i) for next loop
IL_003d: ldc.i4.0                //load 0 ontop of stack
IL_003e: stloc V_5               //store 0 in localvar(sum1)at index 5
                                   //all other local var<index 4 are used

```

```

/*****begin loop 2 of initialization*****/

```

```

IL_0042: br IL_005f              //second loop
IL_0047: ldloc.2                  //load var2(sum) ontop of stack
IL_0048: ldloc.1                  //load array object on top of stack
IL_0049: ldloc.0                  //load i ontop of stack
IL_004a: ldelem.i4                //load array[index i] ontop of stack
IL_004b: add                      //sum+array[index i]
IL_004c: stloc.2                  //store above sum in sum
IL_004d: ldloc V_5               //load var5(sum1)
IL_0051: ldloc.1                  //load array obj ontop of stack
IL_0052: ldloc.0                  //load i ontop of stack

```

IL_0053: ldc.i4.1	//load 1 ontop of stack
IL_0054: add	//i+1
IL_0055: ldlem.i4	//load array[index i+1] ontop of stack
IL_0056: add	//sum1+array[index i+1]
IL_0057: <u>stloc</u> V_5	//store above sum in sum1

----loop condition testing begin

IL_005b: <u>ldloc</u> .0	//load var0 ontop of stack
IL_005c: ldc.i4.2	//load 2 ontop of stack
IL_005d: add	//i+2
IL_005e: <u>stloc</u> .0	//i=i+2
IL_005f: <u>ldloc</u> .0	//rest of instructions are same as above
	//loop till blt instruction

IL\_0060: ldc.i4 10  
 0IL\_0065: blt IL\_0047  
 IL\_0065: blt IL\_0047

----loop condition testing end

/\*\*\*\*\*\*end loop 2 \*\*\*\*\*/

IL\_006a: ldloc.2  
 IL\_006b: ldloc V\_2  
 IL\_006f: add  
 IL\_0070: stloc.2  
 IL\_0071: ldloc.2  
 IL\_0072: call System.Void System.Console::WriteLine(System.Int32)

## 5.4 Enhanced GMCS

### Embedding Mono

```
#include "mono/mini/jit.h"
#include "mono/metadata/environment.h"
#include <stdlib.h>
/*
 * Very simple mono embedding example.
 * Compile with:
 * gcc -o teste teste.c `pkg-config --cflags --libs mono-2`-lm
 */

static void main_function (MonoDomain *domain, const char *file,
int argc, char** argv)
{
    MonoAssembly *assembly,*assembly1;

    MonoDomain *domain1;
    assembly = mono_domain_assembly_open (domain, file);
    if (!assembly)
        exit (2);
    /*
     * mono_jit_exec() will run the Main() method in the
assembly.
     * The return value needs to be looked up from
     * System.Environment.ExitCode.
     */
    //program that does the loop unrolling will run
    //the assembly does loop unrolling on argv[2]
```

```

    mono_jit_exec (domain, assembly, argc, argv);

    //and after it runs it creates an executable with the
modified CIL
    //which has the name as passed in argv[2] by the user
}

int main(int argc, char* argv[]) {
    MonoDomain *domain;
    const char *file,*file1;
    int retval;

    if (argc < 2){
        fprintf (stderr, "Please provide an assembly to load\n");
        return 1;
    }
    file = argv [1]; //file that does the loop unrolling
    //also note argv[2] is passed the csharp program which needs
to be unrolled

    /*
    * Load the default Mono configuration file, this is needed
    * if you are planning on using the dllmaps defined on the
    * system configuration
    */
    mono_config_parse (NULL);
    /*
    * mono_jit_init() creates a domain: the above assembly-file is
    * loaded and run in a MonoDomain.
    */
    domain = mono_jit_init (file);

```

```

main_function (domain, file, argc - 1, argv + 1);
retval = mono_environment_exitcode_get ();
mono_jit_cleanup (domain);
return retval;

}

```

### **Code 5**

#### **Enhanced GMCS script**

```

1. gmcs $1 -r:Mono.Cecil.dll;
2. gcc teste.c -o teste `pkg-config --cflags --libs mono-2`
3. ./teste ceciltry.exe `echo $1|sed 's/.cs/.exe/'` $2;
4. mono $2

```

### **Code 6**

- Argument 1 should be the program that you want unrolled.
- Line1 compiles it by linking it to the cecil library if needed.
- Line 2 compiles our embedded mono c program-teste.c, given in **code 5**
- Line 3 runs it by passing the assembly formed by command in Line1
- Ceciltry.exe is the CSharp file which does the loop unrolling.
- Argument 2 is the output file that you wish to write the modified CIL of argument 1 into, say loopUnrolled.exe

## 6. PERFORMANCE MEASURE AND ANALYSIS OF ENHANCED GMCS

### 6.1 Program that was unrolled

```
using System;
using System.Diagnostics;

class Program1
{
    const int _max = 1000000000;
    static void Main()
    {
        int[] array = new int[12];

        for (int i = 0; i < _max; i++)
        {
            Method1(array);
        }
    }
    static void Method1(int[] array)
    {
        // Initialize each element in for-loop.
        for (int i = 0; i < array.Length; i++)
        {
            array[i] = i;
        }
    }
}
```

#### Code 7

## 6.2 The unrolled program

```
using System;
using System.Diagnostics;

class Program1
{
    const int _max = 1000000000;
    static void Main()
    {
        int[] array = new int[12];

        for (int i = 0; i < _max; i++)
        {
            Method1(array);
        }
    }

    static void Method1(int[] array)
    {
        // Initialize each element in for-loop.
        for (int i = 0; i < array.Length; i++)
        {
            array[i] = i;
            array[i+1] = i+1;
        }
    }
}
```

### Code 8

**Below are the scripts whose performance we compared to see the result of integration of loop unrolling into the Mono framework.**

## Script 1

```
1. before="$(date +%s%N)";
2. gmcs $1 -r:Mono.Cecil.dll; //compile program to be
                               //unrolled
3. mono $2;                    //execute the above prog
4. after="$(date +%s%N)";
5. elapsedsec="$(expr $after - $before)";
6. echo $before;
7. echo $after;
8. echo $elapsedsec;
```

- \$1: code 7 to be compiled
- \$2: Executable produced by the compilation
- before and after record time in nanoseconds

## Script 2- Enhanced GMCS

```
1. before="$(date +%s%N)";
2. gmcs $1 -r:Mono.Cecil.dll;
3. mono ceciltry.exe program1.exe loopunrolled.exe;
   //run the program which unrolls the loop in the program
   //in $1 and writes the modified program in
   //loopunrolled.exe
4. mono loopunrolled.exe;
5. after="$(date +%s%N)";
6. elapsedsec="$(expr $after - $before)";
7. echo $before;
8. echo $after;
9. echo $elapsedsec;
```

Script 1 is essentially the execution of the code which has no loop optimization performed on it.



**Table 3 Comparing performance of script1 and script2**

_max	Performance script1(ns)=t1	Performance of script2(ns)=t2	performance gain(t1/t2)
10,000,000	672967157	879724338	0.764
100,000,000	4076261625	4133325161	0.987
1,000,000,000	40110794125	36399792991	1.101

The above table shows that the overhead caused due to loop unrolling( $\Delta x$ ) is patched up by the performance gain due to loop unrolling( $\Delta g$ ) when \_max exceeds  $10^9$ .

## 7. Extension to the Project

We experimented with another loop optimization-loop fusion.

### ***7.1 Benefits of Loop fusion***

When two loops use the same data, it is beneficial to reduce the amount of data accessed between the loops. Cache lines fetched into the cache by the first loop may then still be in the cache when the second loop is run, so that it doesn't have to fetch them. The less data touched between the two loops, the better. Ultimately, if it is possible to completely merge the loop bodies, then all potential misses from the second loop will disappear.

It is not always possible to move the operations between loops. Special care must be taken not to move any data accesses past a write accesses to the same address, since that will change the meaning of the program.

Some optimizations like this don't always improve the run-time performance. This is due to architectures that provide better performance if there are two loops rather than one, for example due to increased data locality within each loop.

### ***7.2 Program on which loop fusion was done***

```
using System;
using System.Diagnostics;

class Program1
{
    const int _max = 10000000;
```

```

static void Main()
{
    int[] array = new int[12];
    int[] array1 = new int[12];

    var s1 = Stopwatch.StartNew();
    for (int i = 0; i < _max; i++)
    {
        Method1(array,array1);
    }
    s1.Stop();
}

static void Method1(int[] array, int[] array1)
{
    // Initialize each element in for-loop.
    for (int i = 0; i < 12; i++)
    {
        array[i] = i;
    }
    for (int i = 0; i < 12; i++)
    {
        array1[i] = i;
    }
}
}

```

### **Code 9**

### **7.3 Program containing the loop fusion**

```
using System;
using System.Diagnostics;

class Program1
{
    const int _max = 100000000;
    static void Main()
    {
        int[] array = new int[12];
        int[] array1 = new int[12];
        var s1 = Stopwatch.StartNew();
        for (int i = 0; i < _max; i++)
        {
            Method1(array, array1);
        }
        s1.Stop();

    }
    static void Method1(int[] array, int[] array1)
    {
        // Initialize each element in for-loop.
        for (int i = 0; i < 12; i++)
        {
            array[i] = i;
            array1[i] = i;
        }
    }
}
```

**Code 10**

## ***7.4 Program that brought about the above loop fusion***

```
using System;
using System.Collections;
using System.Diagnostics;
using System.Linq;
using System.Collections.Generic;
using System.Text.RegularExpressions;
using System.Text;
using System.Reflection;
using System.IO;
using Mono.Cecil;
using Mono.Cecil.Cil;

using Mono.Collections.Generic;

{

public class ceciltry1{
public static void Main(string[] args)
{
int i=0,k=0;

AssemblyDefinition myLibrary = AssemblyDefinition.ReadAssembly
(args[0]);

//Gets all types which are declared in the Main Module of
"MyLibrary.dll"

//Here types refers to classes
foreach (TypeDefinition type in myLibrary.MainModule.Types) {
```

```

//Get all methods in the class
foreach( MethodDefinition method in type.Methods){

    if(method.FullName=="System.Void
        Program1::Method1(System.Int32[],System.Int32[])")
    {

        i=0;
        k=0;
        int[] j=new int[5]; //j is used to hold the index
                                //of the method instruction
                                //before add

        int[] branch=new int[5]; //to store the indices of
                                //branch instructions in
                                //eachloop of the method

        int[] ldloc=new int[5]; //to store the indices of the
                                //instruction till which the
                                //body needs to be replicated

        int[] cond=new int[5]; //to store the indices of the
                                //instruction containing
                                //backward conditional jump

        //Gets the CilWorker of the method for working with
        //CIL instructions

        ILProcessor worker = method.Body.GetILProcessor();
    }
}

```

```

    foreach( Instruction instruct in
                method.Body.Instructions)
    {
        //take the instruction number of branch

        if(instruct.OpCode.FlowControl==FlowControl.Branch)
        {
            branch[k]=i;
        }

        if(instruct.OpCode.FlowControl==FlowControl.Cond_Branch) {

            saveBltInst=i;
            j[k]=i-5;
            ldloc[k]=i-6;
            cond[k]=i-1;
            k++;
        }

        i++;
    }

```

```

FieldInfo[] myFields =
method.Body.Instructions[j[0]].GetType().GetFields(BindingFlags.NonPublic | BindingFlags.Instance
    | BindingFlags.Public);    //contains different
    fields of the instructions

```

```

if(myFields[1].GetValue(method.Body.Instructions[j[0]]).Equals(myFields[1].GetValue(method.Body.Instructions[j[1]]))

```

```

&&(myFields[2].GetValue(method.Body.Instructions[cond[0]]).
Equals(myFields[2].GetValue(method.Body.Instructions[cond[1
]))))
)// myFields[1] checks if the increment in both the loops
//is the same and myFields[2] checks if the number of
//iterations of both loops is the same
{
    //insert the body of the 2nd loop into the
    //body of the first loop

    int beginBody=branch[1]+1;
    int endBody=ldloc[1]-1;
    numberOfInstructionsToReplicate=(endBody-
beginBody)+1;

    int
    numberOfInstructionsToEliminate=numberOfInstructi
onsToReplicate+1+1+6;//1 for branch instr,1 for
ldloc instr,6 for the
    //remaining loop testing instructions

    //replicating the body of loop2 within loop1

    i=1;

    for(int l=1;
    l<=numberOfInstructionsToReplicate; l++)
    {
        Instruction          ins          =
        method.Body.Instructions[branch[i]+1];

worker.InsertBefore(method.Body.Instructions[ldloc[0]],

```



```

        ins);
        ldloc[0]++;
        branch[i]++;
        ldloc[1]++;
    }

    //remove the branch instruction of loop2

worker.Remove(method.Body.Instructions[branch[1]]);

    for(i=0;i<7;i++){
        //remove instructions which belong
        //to the loop testing condition in
        //loop2

worker.Remove(method.Body.Instructions[ldloc[1]-1]);
    }
    //Problem arises if the loop body is
    //removed.
    //it eliminates the loop body of loop 2
    //which has been inserted in loop1.
    //therefore we did not remove the loop
    //body
    }
}
}

myLibrary.Write(args[1]);

}
}

```

### **Code 11**

## 7.5 Performance analysis

Below are the scripts whose performance we compared to see the result of integration of loop fusion into the Mono framework.

### Script run without loop optimization

#### Script 3

```
before="$(date +%s%N)";  
gmcs $1 -r:Mono.Cecil.dll;  
mono $2;  
after="$(date +%s%N)";  
elapsedsec="$(expr $after - $before)";  
echo $before;  
echo $after;  
echo $elapsedsec;
```

### Enhanced GMCS: Script run with loop optimization

```
before="$(date +%s%N)";  
gmcs $1 -r:Mono.Cecil.dll;  
mono cecilfusiontry.exe fusion.exe loopunrolled.exe;  
mono loopunrolled.exe;  
after="$(date +%s%N)";  
elapsedsec="$(expr $after - $before)";  
echo $before;  
echo $after;  
echo $elapsedsec;
```

The explanation to the above scripts is similar to the scripts in Section 6.2. \$1 stands for code 9.

**Table 4 Comparing performance of script3 and Enhanced script**

_max	Performance script1(ns)=t1	Performance of script2(ns)=t2	performance gain(t1/t2)
10,000,000	928526112	1096921133	0.846
100,000,000	6689564996	5674842597	1.17
1,000,000,000	63264124735	58812566270	1.07

The above table shows that the overhead caused due to loop fusion( $\Delta x$ ) done manually is patched up by the performance gain due to loop fusion( $\Delta g$ ) when \_max exceeds  $10^8$ .

## 8. CONCLUSION

Mono is a brilliant framework, supporting various languages to run on different operating systems and hardware architectures. We learnt the intricacies of the working of a compiler through this project.

Our aim was to improve the execution time by Mono. We focused on the JIT compiler. Essentially JIT compilers have the same information than a batch compiler has today. For a JIT compiler, the problem comes down to striking a balance between the quality of the generated code and the time it takes to generate the code.

JIT compilers tend to go for fast compile times over quality of the generated code. Mono allows users to configure this threshold by allowing users to pick the optimization level defaults and even lets them pick LLVM to perform the heavy duty optimizations on the code. But this could result in slow execution.

Therefore we experimented with the idea of introducing a code optimizer phase just before JIT compilation which could ease the task of JIT, without hindering the speed of its execution.

We have implemented two loop optimizations: loop unrolling and loop fusion into the mono framework. Both show performance gain as shown in the section 8 and section 9.

However the task of loop detection and the kind of loops included in our optimization, can be bettered. We just experimented with the idea of introducing an additional phase of compilation in the mono framework and the results prove to be encouraging.

We therefore wish either we carry on the task of improving this idea, or any of the students of the next batch to carry on the work forward. This document intends to help them begin with the idea with a firm footing.

## 9. FUTURE WORK

### ***9.1 What do we have at the end of our project?***

A modified Mono framework on x86, which includes a code optimizer, which does loop unrolling, on simple loops. The performance result is documented.

### ***9.2 Suggestions for further work***

- Our work has brought in a middle box into mono, which is the code optimizer. So, further optimizations can be added into this box.
- Loop detection may be done more accurately
- More kind of loops may be added to be optimized
- More loop optimizations may be included
- This entire frame comprising of ‘gmcs-code optimizer-mono’ can be made to run on the power PC architecture or any other architecture., as loop optimizations are architecture independent optimizations.

## 10. REFERENCES

- [1] A new JIT compiler for the Mono Project, Miguel de Icaza, Paolo Molaro
- [2] The current and future optimizations performed by the Java HotSpot Compiler, Huib van den Brink
- [3] An Aggressive approach to Loop Unrolling, Jack W Davidson, Sanjay Jinturkar
- [4] Embedding Mono Runtime from the mono forums
- [5] Mono Project-IRC
- [6] List of CIL Instructions-Wikipedia
- [7] Can JITs be faster?
- [8] Reader Q&A: When will better JITs save managed code? By Herb Sutter
- [9] How does .NET/Mono Performance compare to JVM?