Student: **Ciprian Khlud**
Coordinator: **PhD. F.L.Tiplea**

UNIVERSITATEA
ALEXANDRU IOAN CUZA IAŞI
FACULTATEA DE
INFORMATICĂ

LUCRARE DE LICENŢĂ

Code Refractor

Compilation and optimization of

Stack Virtual Machines

Proposed by
Khlud Ciprian

Sesiunea: iunie, 2014

Scientific coordinator
Professor, PhD Ferucio Laurentiu Țiplea

Student: **Ciprian Khlud**
Coordinator: **PhD. F.L.Tiplea**

This page is intentionally left blank

Claim about originality and respecting intelectual property and copyright

With present paper I submit that the Barchelor's paper, “Code Refractor - Compilation and optimization of Stack Virtual Machines” is written entirely by me and it was never presented to any other faculty or other institution for higher level of education in Romania or any other country. I also claim, that all claims, even the ones taken from internet, are clearly described in this work, with respecting rules of avoiding the plagiarism:

- all the fragments of text are exactly reproduced, even if they are translated from another language, they are written in quotes and they keep an exact reference to the source

- paraphrasing in my own worlds of written texts to another authors keep an exact reference to the source;

- source code, source images, etc. taken from open-source projects or another sources are used in accord with the copyright laws and it keeps direct references

- the summary of ideas of other authors keeps exact reference to original text

Iassy, <today>

Graduating student Khlud Ciprian

_____

(signature)

Declaration of acceptance

I claim that I agree that the Bachelor's paper with title: "Code Refractor: Compilation and optimization of stack-based virtual machines", the source code of the programs and the other third party content (graphs, multimedia, input data, etc.) that come with this paper to be used by the Faculty of Informatics.

Also, I agree that Faculty of Informatics from University "Alexandru Ioan Cuza" Iassy to use, change, copy and distribute under noncommercial ways the programs, executables and sources, written by me as part of this Bachelor's paper.

Iassy, <today>

Graduating student Khlud Ciprian

_____

(signature)

Agreement about rights of ownership

Faculty of Informatics agrees that the copyright of computer programs, executable or source code to belong to the author of the present thesis, Khlud Ciprian.

This agreement is important for the following reasons:

- the Code Refractor compiler front-end, initial middle-end design and the backend (code generator) is under GPL version 2+, a license which is owned by the Free Software Foundation. As author of CodeRefractor, I (Khlud Ciprian) allow the version on the time of submitting of the Bachelor's paper to be submitted under a license that fits the University's needs, but the Code Refractor as is remains Free Software under Free Software Definition of GPL2 or any newer versions;

- Code Refractor's runtime library is under MIT/X11 license, a liberal license which allows public modification. Because Code Refractor's is public domain (even I'm the author of it), the public version at the time of publishing the Bachelor's paper can be used for University's needs, but Code Refractor's runtime library cannot be bind under any proprietary license that does not allow public domain changing, including but not limited to, commercial needs.

Iași, <today>

Dean,

Khlud Ciprian

_____

_____

Contents

0. Introduction

0.1. Abstract

Virtual machines implement various techniques like interpretation, Just-in-time (JIT) compilation or Ahead-Of-Time (AOT) compilation. This project proposes a method of AOT compilation using the host CIL virtual machine (also known as .Net ™ virtual machine) and using a standard C++ compiler (C++ 11) to give the same semantics and in the meantime to also improve performance offering a full compilation step, and in the very same time to remove the dependency to the host virtual machine.

A similar technique could be implemented in a JVM (Java Virtual Machine), but CIL is in more ways advanced, because the CIL opcodes do not specify types, and the CIL Virtual Machine (VM) will have to infer based on context.

0.2. Introduction

An advanced (optimizing) compiler is split into following steps:

- scanning (sometimes named tokenization);

- parsing (which generates an AST tree);

- semantic analysis;

- transformation of AST tree to intermediate representation (IR) which defines all operations of the source program into a small steps, that can be easily understood what they do;

- optimization steps which consist into various visitors of the IR and they rewrite the IR into an equivalent, but a bit more efficient more form of the defined operations;

- IR is visited and written into a “low level representation” like Assembly language, or binary form. A critical part of this part, is to find a good way to use the minimum resources (mostly CPU registers).

This project will show a way to write an optimizing compiler against virtual machines, and focusing for correctness, performance and simplicity of understanding of the code. Also, in this introduction, the reader is informed how a virtual machine works, and how to map most operations into a low level implementation. Based on simplicity and practicality the low level backend implementation is C++, but the output code is “very C-like” and it uses C++ just to not reimplement some parts like smart-pointers. This note is to make clear that Code Refractor is not a “code translator” but a full optimizing compiler, as it has even a section dedicated to optimizations.

A virtual machine like Java (JVM = Java Virtual Machine) or .Net executes “on demand” an intermediate form instructions (look earlier for IR), named “bytecode”. This bytecode describes the original semantic of the original Java written language (in Java world) or C# (Vb.Net, Boo, F#, etc. in .Net technologies).

If any user wants to get a better performing application, would likely want to get an efficient compiler to evaluate its program. Historically there are many approaches in literature:

- Java’s HotSpot: is a JIT compiler which will compile “hot” parts of the program. This is great for some programs which benefit from this dynamic evaluation, but sometimes the startup time is a direct consequence of this approach, as some parts of the code are interpreted and profiled before knowing which parts are hot

- another approach is done by Excelsior Jet[1], which (even is not fully documented how it works, being a proprietary product), but it looks that it compiles every bytecodes into an intermediate representation, and based on your feature choice based on the bought edition, you will have enabled more or less optimizations. It also looks that their static compiler is written in Java (the controls look like written in Swing, and at least once in the evaluation version, I've got Java exception in compilation)

- RoboVM[2] and GCJ[3] both read the Java bytecodes and rewrite it directly into an optimizing compiler backend (RoboVM into LLVM bytecodes, GCJ into GCC's GIMPLE bytecodes)

- Mono[4] has two Ahead-of-time compilers, one is their made LinearIR[5] (their "optimizing framework") which is fast, and another one slow, which writes into LLVM bytecodes

For practical reasons, if a person wants to implement a full virtual machine specification, will have to implement tens to hundreds of bytecode instructions that are defined by the virtual machine, to make possible to execute average or a bit bigger than "Hello world" but also it shouldn't implement all (but as not all instructions are implemented it will make it a non-conformant virtual machine).

This thesis describes the implementation of a nonconforming Ahead-of-time compiler for CIL instruction set virtual machine of instruction set, and will try to use practical approaches to make it easy to be studied, extended and to make possible to give both a good performance of the final code and a fairly decent compatibility, regardless of virtual machine versions.

[1] http://www.excelsior-usa.com/jet.html
[2] http://www.robovm.org/
[3] http://gcc.gnu.org/java/
[4] http://www.mono-project.com/Main_Page
[5] http://mono-project.com/Linear_IR

1. Virtual Machines description and evaluation

There are many parts in implementing a virtual machine, but before starting this, is better to describe what a virtual machine does in between the bytecode and the final executing code.

A virtual machine does separate the operate system and processor architecture by offering own instruction set, and guaranteeing an implementation for the supported platform.

So, a developer will program in any language that the virtual machine supports, and at the end will be able to run this program on any computer where this virtual machine is running with no regard on the CPU architecture, file system, etc.

In practice because computers/computing devices (as phones or embedded boards) have various specifications, the virtual machines offer a partial implementation and even in cases where the VM runs the application, the behavior of it differs in subtle ways.

For example, Java's trigonometric implementation is known that corrects in software rounding errors on x86 CPUs from Java 1.4 (so Java 1.3 or older would have a bit faster trigonometric operations but with a small error). .Net does not specify precise floating system, so the host CPU rounding for float operations can make the programs to give a bit different values in floating point math.

Even more important, is that virtual machines have runtimes addressing different use-cases, like Java Client, Server, .Net's Desktop and Server runtime, Android Dalvik, etc. which makes important to describe the utility of Code Refractor by having a precompiled, offline generated binary. This executable code will run closer to what the original developer intends as the final executable will not have unexpected behavior from what it runs from developer machine.

A virtual machine implementation that compile the original instruction set to a final binary is named Ahead of time (or shortly AOT) and is widely used in both Java and .Net world. .Net offers as part of its distribution the **ngen** (Native Image Generator) tool which allows compilation of .Net assemblies into "native" dlls.

A virtual machine is great but in some cases, a static compiler that executes the same semantics as the virtual machine and the advantages can be:

- Fast execution (if your program you benchmark it and it finished let's say the problem in 1 second, you know that tomorrow the program will still run in 1 second, even the user forgot to update the VM to have the same version as you have)
- Fast startup (virtual machines before executing have to evaluate bytecodes, a static compiler did this before running, so no startup overhead)
- Lower memory: virtual machine data like the executed functions and bytecodes are using the same memory as the process you're running. Compiling in advance makes your program to store only the data your program needs. This is less important on most desktops, but is important both on servers and on the phones

1.1. Stack based Virtual Machines Overview

1.1.1. Execution model

A Java and .Net VMs would do the following:

- would start reading your entry method;

- It will read the bytecode as operations;

- Depending of the design, will start executing that method operations up-to when will hit another method,

- If this method was never compiled before, this method will be read too and executed recursively

- If the method is compiled before, the method is executed normally, and the virtual machine will put guards around areas that can give exceptions or execute another method (named trampolines)

1.1.2. Memory model

Virtual machines do implement a garbage collector, which means, through other things that memory is cleared (collected) periodically if is not reachable from other pointers.

Modern garbage collectors are generational, meaning that memory heap is split into three heaps:

- a small area named young generation (in .Net is named Gen0)
- a medium sized named "aging generation" (in .Net is named Gen1)
- the bigger heap named "old generation" (Gen2 is in .Net implementation)

Every new object is allocated in Gen0. When Gen0 is full, the GC algorithm will look which of the Gen0 objects are still in use and move them to Gen1. After some time Gen1 gets filled and Gen1 is moved to Gen2. When Gen2 is filled, a "full collection" is made.

Java has a bit different "aging generation", and the old objects are simply objects that survived more than a specified number of Gen0 fills, but are very important to understand that:

- Gen0 collection and allocation is very fast, around 10x faster than a C/C++ malloc strategy

- There is no cost for deallocating objects; the cost of GC is related with how many objects are reachable and to be visited/copied

- Old generation GC is slow, for big applications can take literally seconds, as it implies visiting every reachable object

1.1.3. Compilation model

All stack based virtual machines, if they compile the method, they will compile not the bytecode directly, but they will make more optimizations on the intermediate representation (IR). Also this form is later converted into binary object code, and this code is executed. These optimizations are limited in scope because the compilation happens interactively with running application.

This means for practical applications, to reduce the startup time, and to get high level of interactivity, there are some solutions in literature like:

- using a JIT that optimize heavily but a linear (with no jumps) sequence of instructions, like tracing JITs as Android's Dalvik JIT
- having profiles of compilation as: interpreter, client and server compiler in Oracle's Java HotSpot implementation
- having a bit weaker optimizer (similar with HotSpot client) method based compiler as in .Net

Excluding Java server optimizer mode, all other are not getting the maximum performance that can be given by the compilation literature, and for example:

- tracing JITs do not take in account branch instructions
- .Net and Java client compiler are using most likely not optimum register allocation

1.1.4. "Stack Based" versus "Register based" instruction set

This means that operations between various entities are defined using an "evaluator stack" which means that instructions to be executed, they have to to read the last states (or to write them) before being able to define them.

In a stack based code, even for simplest operations like:

The code at the bytecode level is:

Source code	Intermediate code (Stack VMs)
int a = 5; int b = 3; int c = a + b;	1. push int 5 2. pop value from stack into “a” 3. push int 3 4. pop value from stack into “b” 5. push 'a' 6. push 'b' 7. add top two values on stack 8. pop value from stack into “c”

A register based virtual machine has already allocated number or local variables, named “registers” and the assignments of the operations, will not use stack as evaluator state, but the instructions will use the register index as source of data.

Even the number of instructions is the same in this simple sample code, a precompiler can optimize the register output and may rewrite as the following (column 3) in the following graph.

In a “register based” VM the code would be like following.

Source code	Intermediate code (Register VMs)	Optimized code (Register VM)
int a = 5; int b = 3; int c = a + b;	1. set reg_0 5 2. set into 'a' reg_0 3. set reg_1 3 4. set 'b from' reg_1 5. set reg_2 from 'a' 6. set reg_3 from 'b' 7. add reg_4 reg_2 reg_3 8. set 'c' from reg_4	1. set reg_0 5 2. set reg_1 3 3. add reg_2 reg_0 reg_1 4. set 'c' from reg_2

Which will remove the usages of “a” and “b” as not being necessary, and even instruction 3 can be removed (and evaluate reg_2 as value 8).

This is an important part as performance and optimization steps are concerned, if the stack VM can be rewritten as a Register based virtual machine, the optimizations can be applied more easily, because the instruction set contains more information to track the data.

Equally important is based on the instruction support to manipulate variables, makes Java and .Net VMs to be hybrid VMs (as the variable themselves are registers)

1.1.5. Specified instruction set

Both Java and .Net (CIL) have publicly defined instruction set. They can be understood from public made documentation. This is great for implementers and makes a case of making easier to test virtual machine compatibility[6]. This also can simplifies the testing of virtual machine’s semantics as writing a C#/Java program that gives the asked bytecode means at the end, if a bug arise, it is easier to check if the semantics of the bytecode reflect correctly the program (i.e. it can be a language compiler bug).

Also, it is good to know that both JVM and CIL try to fit (in most cases) in one byte, making these operations compact. Writing a virtual machine is easier starting from given high level source programs that use any specific instructions, as it is easier to write human readable code than binary bytecode.

[6] Look to references 1 and 6

1.2. Evaluation and implementation directions

1.2.1. Evaluating the execution model

Both Java and Net create a method graph of Callers and Callees, and all the logic can be extracted with reflection (a Java or .Net API to scan the runtime information) or with bytecode manipulation libraries.

Code Refractor as first compile step, will get the assembly file which needs to be scanned, and will read the bytecodes especially the calls ones. This “linker” will scan bytecodes, and every time will find a new **call** instruction, will track and if finds a new function, will scan it too, up-to the moment will create the closure of all functions.

A note-worthy implementation detail is that the “new object” (**newobj** CIL instruction) instruction includes an (implicit) constructor call (if this constructor is with no parameters), and this code is also scanned.

Code Refractor compiler should input and compile various methods online (without saving object files on disk) giving the advantage of having all necessary data accessible in memory. Even with big programs and with advent of 64 bit architectures the main issue is compilation time. At the end if the output C++ code will be compiled and the final output binary is generated. Look for section 2.1 of details.

1.2.2. Evaluating the memory model

An advanced (and fast final code) compilation and memory model is a very hard problem, garbage collectors are as of today improved.

The garbage collector can be implemented with C++'s “smart-pointer” class. This class defines a number to count all references to an instance of an object. When the references are zero, the object instance is automatically deleted.

This strategy of reference counting is well known in literature, the solving of the harder problems associated with ref-counting are described in the section: **Abstraction lowering and escape analysis (section 4.6)**.

Smart pointers, excluding their slow speed (and as CR resolved some slowness, as is described in the special section of it) they have some advantages:

- deallocation cost in C/C++ code is small, mostly is setting a bit of “saying” that the object is a free block

- dealocation is predictable, because (excluding there is no leak), when the method scope ends and if the object is not used outside the specified code, this can remove “shuttering” in objects made in a loop

- there is no need of pinning, and to specify that some objects are GC objects and some of objects are from “C/C++ world”, no moving of objects, etc.

1.2.3 Evaluating the compilation model

It is easier and practical to reuse solutions already existent. The very often used solution for AOT compilers for virtual machines is LLVM or another compiler intermediate representation (like in case of GCJ, using GIMPLE form), but they also require to implement a full runtime, which again is hard to be made practical.

As a solution to approximate the original virtual source, a C++ compiler can reflect fairly well the original bytecode (look to a later chapter where is defined how the C++ language reflects some of the semantics of the virtual machine, without introducing a memory or performance overhead). C++ has also another good side effect, C++ is supported in all operating systems, or even in platforms like Windows RT, so it is possible (given enough changes and platform support) to get access to all platforms which offer a C++ compiler, which includes embedded platforms, cars, etc.

Similarly, just a good register allocation, as the Register Allocation (RA) is an NP-hard problem, is not solvable and can be redefined as a coloring problem (which is again NP-hard), but C++ compilers do colorize registers very well.

1.2.4 Evaluating the Stack Based VM model

All operations in the .Net machine as they work against a stack which stores the states, it has to be implemented with main stack operations:

Introducing for a simple addition a full stack class, would make performance horrendous, but C++ (and C) pushes all (simple) variables to the CPU stack (which is very fast, as is implemented in the hardware of the CPU), so the solution is that every time we have a push of a value, we define a variable on stack to keep this variable, which is pushed on the process' stack, and when a Pop operation is called, this variable is read and will reset the counter to keep the correctness of the implementation.

Example:

CIL Instructions	C++ generated code
IL_0001: ldc.r8 0.0 IL_000a: stloc.0	`vreg_1 = 0.0;` `local_0 = vreg_1;`

The instructions set understanding is not important, but the left code does the following:

- Pushes a double (64 bit float) 0.0 on stack
- Stores into local variable zero what is on stack

C++ code (without optimizations) will:

- Create a stack store variable vreg_1 to store the pushed value
- When the second instruction of “store what’s on stack”, the Stack wrapper logic will know that vreg_1 keeps the top of the stack, so will give to the instruction interpreter the vreg_1 as the right argument

The main advantage with this implementation that it will change the virtual machine to be a Register-based VM, which at the end gives a lot of optimization opportunities (as it can be seen later).

Also, this stack implementation class gives an uniform model for variables, all variables being defined on stack: arguments, local variables and virtual registers.

1.2.5. Type tracking over stack evaluation

Type tracking is that every time when any operation is performed, the types of all components are tracked. This is a small improvement in itself (over the both register/stack virtual machines), but it simplifies the optimizations code and not only.

Most values are defined as: IdentifierValues, and every identifier can be a typed constant or a local variable (which in turn can be virtual register, local variable or a function argument). Every time an instruction is evaluated, the types are computed and evaluated.

When an instruction is evaluated as:

Vreg1 = add (const: 5) (const: 3)

We know that the Vreg1 is itself an System:Int32, and later when optimization passes happen, they can replace the entire expression with value 8.

In fact this type inference (albeit is somewhat not as advanced as a full type inference, but it works well enough to evaluate all instructions out of the CIL instruction set), enables through other things the:

- Constant folding: a = 2+3; => a =5;
- Partial Constant folding: a = 1*b; => a = b;
- Evaluating if a function is called only with constant parameters, and if the function is pure, to evaluate the function at compile time: a = Math.Cos(0); => a = 1.0;

- Class Hierarchy Analysis and early binding (remove virtual calls);
- Etc.

Type tracking doesn't guarantee performance improvements, but improves very much the accuracy of future optimization steps. Types tracking also imply that all small expressions that are reflected by the stack evaluator need to be typed. As C++ (11) allows type inference, a simple code generator could not do type tracking, and generate instead of: int a; a = 2+3; => to have auto a = 2+3;

Type tracking is done by every instruction kind by defining: ComputeType method.

For example:

ComputeType function in Binary operator

```
public class BinaryOperator :
OperatorBase
    {
        public BinaryOperator(string
name) : base(name)
        {
        }
```

```
public Type ComputedType()
{
    var leftType = Left.ComputedType();
    var rightType = Right.ComputedType();
    return leftType ?? rightType;
}(…)
```

1.2.6. Optimization overview

First of all, I will want to make a small comment about the word "optimization" which in many ways is considered to give an "optimum" of a starting program. In fact optimizations will never give the optimum program because:

- Optimum can be defined by more criteria, as most compilers (including Code Refractor one) optimize for runtime performance, will have some tradeoffs. One of them, is that it occupies more CPU stack (a memory tradeoff)

- Algorithm in the first place, that is defined in the user's code may not be optimal, and the compiler has to guarantee that will create the code with the same result

- Other factors, like but not limited to: some optimization passes in literature are not implemented as they are too complex to be made

Based on this, I would define an:

An **optimization step** is an algorithm which starts with the intermediate representation (IR) and will rewrite it as another IR (which is equivalent with the original IR's side effects and results) in case some conditions are met for that specific algorithm, which is in a form more advantageous as resources or strides to enable another optimization steps.

Definition: the **compiler optimizations** are the application of all optimization steps until no optimization step is possible to an IR.

Optimizations can be thought in some categories:

- Simplifications (like math simplifications);

- Propagation (like transitivity properties);

- Analysis steps (which would try to prove that some properties keep hold). If so, they do make possible that another optimization step to use these properties;

- Data flow optimizations (this will be a large section of the optimizations);

- Etc.

2. Code Refractor – implementation of an AOT virtual machines

2.1. Small overview of CodeRefractor

CodeRefractor does the following:

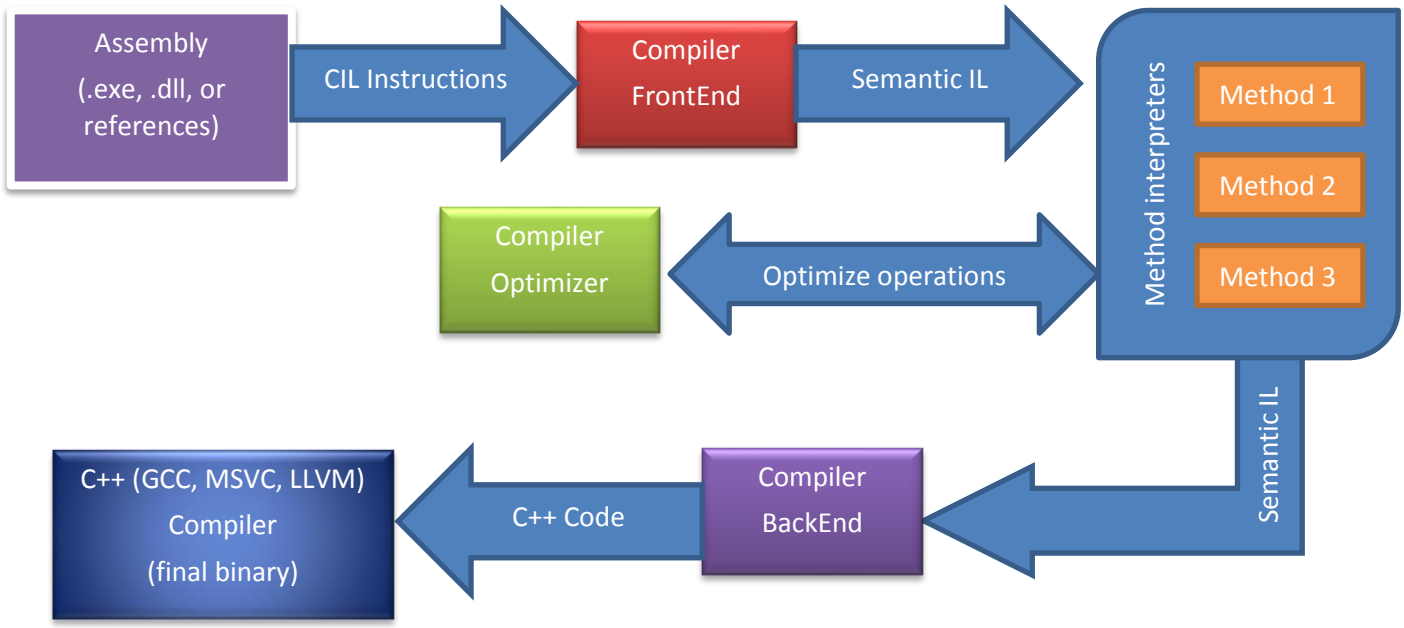
- reads the entry point of an assembly;

- A pre-scan of the method call tree is created;

- A second scan will take all methods and will transform their CIL bytecodes into intermediate representation

- Compiler optimizations are performed for every method

- Intermediate representation is written into C++.



CodeRefractor compiles to a final binary is an ahead-of-time (AOT) compiler with a lot of focus on performance of final executable.

2.2. Intermediary representation of Code Refractor

As the logic of most code is method based, CR does store the intermediate representation per every method as two parts:

- variable parts which are themselves split into:
- local variables
- virtual registers
- arguments (method's parameters)

Local operations which will use either the local variables or they will call other methods. These local operations are themselves split into:

- assignments
- operators:

  - binary operators
  - unary operators
- creation of arrays and objects
- accessing array elements or fields
- call of a separate methods
- branch operations
- labels

These operations are in fact a more compact form of the original CIL operations, and as a difference, the IR operation will keep unify the bytecodes that do a common semantic as a single operation kind, instead of having an individual bytecode for every operation.

Every LocalOperation is in fact a pair of an enum of:

LocalOperation reference class code

```
  public class LocalOperation
    {
        public enum Kinds
        {
            Call,

            Return,
            BranchOperator,
            AlwaysBranch,
            Label,

            NewObject,
            CallRuntime,
            Switch,

            GetField,
            SetField,
            GetStaticField,
            SetStaticField,

            GetArrayItem,
            SetArrayItem,
            NewArray,
            CopyArrayInitializer,

            BinaryOperator,
            UnaryOperator,


            Assignment,

            RefAssignment,
            DerefAssignment,

            LoadFunction,
            FieldRefAssignment
        }

        public Kinds Kind;
        public object Value;
(…)

}
```

Some important types are:

Some often used types

```
    public class IdentifierValue
    {
        public Type FixedType;

        public virtual Type
ComputedType()
        {
            return FixedType;
        }

        public string Name
        {
            get { return FormatVar(); }
        }(…)
}
public class ConstValue :
IdentifierValue
    {
public class Assignment : IClonableOperation
    {
        public LocalVariable AssignedTo;
        public IdentifierValue Right;
(…)
}
public class OperatorBase
    {
        public LocalVariable AssignedTo { get;
set; }
(…)
}
   public class UnaryOperator : OperatorBase
    {
                public IdentifierValue Left { get;
set; } (…)
}
   public class BinaryOperator : OperatorBase
```

```
(…)
}

    public class LocalVariable :
IdentifierValue
    {
        public VariableKind Kind; (…)
    }
{

    public IdentifierValue Left { get; set; }
    public IdentifierValue Right { get; set;
}
}
```

This means that if you make an **optimization step** that works against operators, it is easier to specify first a group of operations to target, and after that, the operations are inspected to see the specific operator you want to target.

Describing a simple optimization step with a code sample:

Merge Two labels which are consecutive

```
  internal class MergeConsecutiveLabels :
ResultingOptimizationPass
  {
    public override void
OptimizeOperations(MetaMidRepresentation
intermediateCode)
    {
      var operations =
intermediateCode.LocalOperations;

      var found = operations.Any(operation =>
operation.Kind == LocalOperation.Kinds.Label);
      if (!found)
        return;
      for (var i = 0; i < operations.Count - 2; i++)
      {
        var operation = operations[i];
        if (operation.Kind != LocalOperation.Kinds.Label)
          continue;

        var operation2 = operations[i + 1];
        if (operation2.Kind !=
LocalOperation.Kinds.Label)
          continue;
        var jumpId = (int) operation.Value;
        var jumpId2 = (int) operation2.Value;
        OptimizeConsecutiveLabels(operations, jumpId,
jumpId2);
        operations.RemoveAt(i + 1);
        Result = true;
      }
    }
    private static void
OptimizeConsecutiveLabels(List<LocalOperation>
operations, int jumpId, int jumpId2)
    {
      for (var i = 0; i < operations.Count - 2; i++)
      {
        var operation = operations[i];
        if (operation.IsBranchOperation())
          continue;
        switch (operation.Kind)
        {
          case LocalOperation.Kinds.AlwaysBranch:
            var jumpTo = (int) operation.Value;
            if (jumpId2 == jumpTo)
              operation.Value = jumpId;
            break;
          case LocalOperation.Kinds.BranchOperator:
            var destAssignment = (BranchOperator)
operation.Value;
            if (destAssignment.JumpTo == jumpId2)
              destAssignment.JumpTo = jumpId;
            break;
        }
      }
    }
  }
```

This *optimization step* will find if there are two consecutive instructions defined as label (target of jump instructions), and if they are found, the labels are merged into one, and all jumps that are targeting the deleted label, they will be redirected to the adjacent label.

Before starting optimizations, it is really great to know which optimization steps do make sense, as not all optimizations do have the same effect, but in short some rules can be easily discovered:

- When a program executes less instructions, is more likely it will run faster

- When the code that is not accessible in any execution path and is removed, this is also powerful as it possibly enable another optimizations

- When you replace a variable with a constant value (when possible) is faster as execution time

- inlining calls, will remove at least the performance of the call

- removing pointer assignments, this is crucial as an assignment requires smart pointer updating which is expensive

Some performance improvements (yet they are not optimization steps, as they don't work against the IR) can be done without any semantic analysis of the final code:

- Platform native libraries which are imported, can be merged as data structures as a compilation step. This will reduce the impact of reloading every pointer to function and reloading/searching for a DLL if is loaded already

- Duplicate array data initialization (that sometimes is initialized only with zeroes) can be merged, and the same can be done in all strings over the application

- String type can contain in a linear data set the length and the string data. This improves the memory accessing pattern, as there is no required L1/L2 cache misses if the string length is separated from the data

- make_shared: shared pointers do have cost of using and initializing. A smart way to improve shared-pointers in C++ 11 is to use make_shared function call, which will reduce the number of allocations. Please read section 4.6.2. for details

- this pointer is using: const & for smart pointers, which in case that this is not assigned, and is just used (like for accessing fields) will have no overhead at call

3. Code Refractor components

3.1. Overview

CR does have more components that it works with, like a runtime library, an underlying C++ compiler, and some various semantic differences (like some discussed previously, that it has to use smart-pointers even C/C++ codes have to work with raw pointers).

Some parts that worth mentioning are:

- **CR OpenRuntime** is a C# language written DLL that is used as a runtime for the underlying C++ implementation;
- System.Math class as an example of OpenRuntime implementation
- Platform Invoke
- string merging, array initialized byte data merging

3.2. CR OpenRuntime

As CodeRefractor uses C++ compiler backend, it needs to provide some implementations even for very simple program.

For an empty program we try to match the function by extracting the arguments and offer them as an array of strings.

Empty program generated code

```
#include "sloth.h"
namespace System {
struct Console {

}; }
#include "runtime_base.partcpp"
(…)
System::Void Figures_Program__Main(std::shared_ptr< Array < std::shared_ptr<System::String> > > args)
{

return;
}

void initializeRuntime();
int main(int argc, char**argv) {
auto argsAsList = System::getArgumentsAsList(argc, argv);
initializeRuntime();
Figures_Program__Main(argsAsList);
return 0;
}
void mapLibs() {
}

void RuntimeHelpersBuildConstantTable() {
}

void buildStringTable() {
} // buildStringTable
const wchar_t _stringTable[1] = {
```

```
0
}; // _stringTable
```

Also, as section 3.5 will explain, there is a string and constant table merging, and these tables should be initialized. Both these cases can be cleaned up in the future (to not write them at all if they are not needed, but the point still remains, mainly that there is C++ code that needs to be provided. So for this, there is a very small C++ code (written in both "sloth.h" and "runtime_base.partcpp" codes) and a lot of implementations that are provided by the scanning of the assembly OpenRuntime which provides a C++ runtime library either explicit (like providing C++ code) or implicit, like providing a .Net implementation and when methods are called as "System.dll" are in fact provided by these alternate implementations.

What is important to notice is that OpenRuntime also is made to not break license of the host virtual machine, as CR will not scan code of the system DLLs. As assemblies when are registered to system are named GAC assemblies, when we notice that a type reside inside a GAC assembly, we look for an alternate inside OpenRuntime, if not, the compilation will fail.

To create the alternates type table, we annotate with MapType annotation (look for Math class in the next section) to provide implementations for missing code.

3.3. System.Math

Math functions like Sine or Cosine functions are compiler intrinsics in most compilers, so for this reason, it is not possible a trivial implementation them at the CIL level, I should have access either to assembly code or to C/C++ code.

Because of this, the implementation is written in an annotation (similar how Platform Invoke works in .Net) but also I can decorate the method as PureMethod (look for **section 4.3.1.** where I take in account that some math functions are pure).

CR's Math class implementation

```
[MapType(typeof (Math))]
public class CrMath
{
    [PureMethod]
    [CppMethodBody(Header = "math.h", Code
= "return sin(a);")]
    public static double Sin(double a)
    {
        return 0;
    }

    [PureMethod]
    [CppMethodBody(Header = "math.h", Code
= "return cos(a);")]
    public static double Cos(double a)
    {
        return 0;
    }

    [PureMethod]
    [CppMethodBody(Header = "math.h", Code
= "return sqrt(d);")]
    public static double Sqrt(double d)
    {
        return 0;
    }
(…)
```

3.4. Platform invoke

C# code in particular depends on executing code that is written in other languages, and the default portable way to do it, is to use PInvoke calls. This way of doing it makes by annotating functions that are platform invoke.

CodeRefractor does handle platform invoke as DLL method calls as following

C# source code before optimization	C++ Generated code
[DllImport("test.dll")] private static extern int RandValue(); public static void Main() { var a = RandValue();	typedef System::Int32 (*dll_method_1_type)(); dll_method_1_type dll_method_1; System::Int32 SimpleAdditions_NBody__RandValue() { return dll_method_1(); } (…) void mapLibs() { auto lib_0 = LoadNativeLibrary(L"test.dll"); dll_method_1 = (dll_method_1_type)LoadNativeMethod(lib_0, "RandValue"); }

Platform invokes wrappers do take in account the following:

- If are many functions called from the same DLL/libSO, the library loading happen just once, and functions reuse the same handle

- Functions can use any convention calls (like __stdcall, __cdecl, etc. by default using compiler's one)

- The wrapping takes in account multiple parameters to make the behavior as a pointer function call (which is given from the external library)

- The methods LoadNativeLibrary, LoadNativeMethod, are OS independent with (for now) a Windows and a Linux implementation

Calling platform code can make your program to not run on other platforms and the compiler cannot decide and can block optimizations. For example if the program is compiled on 64 bit and the final dll is on 32 bit, the program will fail to load (this is also true for .Net) so extra work and care has to be taken in account when working with different platforms.

3.5. String merging

This is a micro-optimization of the CR compiler. This optimization in itself is not critical for most of programs, but it takes in account the idea that CPUs are limited by their L1/L2 caches sizes.

So, in CIL instruction set you can have multiple cases when you declare buffer constants (like strings and **ldtoken** CIL instruction which gives a buffer of bytes for an initialize declared array).

For LdToken instruction is that the code[7]:

Code that generated LdToken CIL instruction

```
var ints = new [] { 1, 2, 3 };
```

This in turn will create a code like:

Equivalent semantics of generated LdToken CIL instruction

```
var auxBuffer = new int[3];

IntPtr bufferAddress = (…);
RuntimeFieldHandle fieldHandle = new RuntimeFieldHandle();
fieldHandle.Value = bufferAddress;
 RuntimeHelpers::InitializeArray(auxBuffer, fieldHandle);

ints = auxBuffer;
```

InitializeArray is basically a memcpy (copy from source to destination) of a pointer of a source buffer.

[7] Look to reference 8

So every time when you declare strings, or arrays and it happens that these data arrays are duplicated, CR will look into all buffers, will see if there are duplicated, and will merge them. Big codebases tend to have duplicate data, and also strings in particular can be duplicated implicitly or explicitly.

4. Compiler optimization and optimization steps in the CR

4.1 Optimization short overview

Optimization passes are logically split in between:

- Local optimizations (the ones that go just over a basic block)

- Global optimizations (optimization passes that go over more than a basic block and can depend on the entire body of a function)

- Program-wide optimizations (optimizations that have the scope bigger than the body of a function)

4.1.1. Real life optimization strategies

Even it doesn't always appear to be in this way, optimizations that are the most effective are local optimizations, right after the global optimizations and at last the program-wide optimizations. The reason is that the .Net states defined by its specified stack based VM require to define many stack operations that after these states are transformed into a Register-Based VM (as it was described later), the impact of the higher order optimizations appear to be less important.

Android team decided to write a JIT compiler (just in time) and they compared their original design, an interpreter of a Register Based Virtual machine (named Dalvik) and they decided to see which strategies do impact most.

What they compared is firstly: their interpreter compared with "a competitor Java interpreter", where they found that the interpreter is like 2 times as fast (because register based operations are faster in themselves than stack based virtual machines). Also they found that on their applications at least, 2% of the code is "hottest of the hottest" using tracing JIT compared with method based compilation which sees as hot code 8%. The speedup over the interpreter was 2-5x times (would be like 4-10x times compared with a Java interpreter), but they also noted that a full compiler will improve the performance up-to 10 times (20 times compared with a Java interpreter).

Tracing JITs do optimize traces, which are parts of code that are made just by assignments, operators, array accesses, but not if/jumps. In math like coding, we can expect like a good performance with simple "local optimizations", which are sequence of instructions between jumps (like if(expr) goto; label_x:; or goto; instructions).

As we don't compile "live", we can see the range of optimizations as:

- **block based optimizations** (named also local optimizations), which do work in a sequence of instructions without jumps
- **global optimizations**: optimizations that do work for entire body of the method
- **dataflow analysis**: a form of global optimizations that visits various branches of the code to test various hypothesis
- **interprocedural optimizations**: mostly inlining but the most important part is that they require knowledge of entire program scope, by entire program it doesn't necessarily mean that any function will look into all program, but it means that a function **foo()** may access another function **bar()**, and this **bar()** function may be optimized previously, and inlining **bar()** into **foo()** may make some optimizations applied to foo to allow that: **foo_main()** which calls **foo()** to inline later this method.

Code Refractor offers some code helpers that are used by the various optimizations:

- for every instruction we can get usages and definitions
- for every instruction an usage can be replaced with a value (for example if we know that X variable is a constant 2, for all instructions that we know that X is used, the code allows to replace: X with 2)

- building a label table, which is crucial for some labels/goto optimizations and to not duplicate the code
- Loop detection code (for **4.3.5. Loop invariant code motion**)

4.2.Local optimizations (block based optimizations)

4.2.1. Constant folding optimizations

C# original code

```
    public static void Main()
    {
        int a = 30;
        int b = 9 - (a / 5);
        int c;

        c = b * 4;
        if (c > 10)
        {
            c = c - 10;
        }
        var d = c * (60 / a);
        Console.WriteLine(d);
     }
```

(based on Wikipedia's page of Constant folding)

Resulting C++ code (with no optimizations)

```
System::Void _Nbody__Main()
{
/*... local variables*/

vreg_2 = 30;
local_1 = vreg_2;
vreg_3 = 9;
vreg_4 = local_1;
vreg_5 = 5;
vreg_6 = vreg_4/vreg_5;
vreg_7 = vreg_3-vreg_6;
local_2 = vreg_7;
vreg_8 = local_2;
vreg_9 = 4;
vreg_10 = vreg_8*vreg_9;
local_3 = vreg_10;
vreg_11 = local_3;
vreg_12 = 10;
vreg_13 = (vreg_11 > vreg_12)?1:0;
vreg_14 = 0;
vreg_15 = (vreg_13 == vreg_14)?1:0;
local_5 = vreg_15;
vreg_16 = local_5;
if(vreg_16) goto label_42;
vreg_17 = local_3;
vreg_18 = 10;
vreg_19 = vreg_17-vreg_18;
local_3 = vreg_19;
label_42:
vreg_20 = local_3;
vreg_21 = 60;
vreg_22 = local_1;
vreg_23 = vreg_21/vreg_22;
vreg_24 = vreg_20*vreg_23;
local_4 = vreg_24;
vreg_25 = local_4;
System_Console__WriteLine(vreg_25);
return;
}
```

Resulting C++ code after constant folding optimizations:

```
System::Void _NBody__Main()
{

System_Console__WriteLine(24);
return;
}
```

To get this result, the following cases are found by the compiler as optimizable:

- assignment of identifier used next line
- assignment of expression
- evaluate constant expressions
- evaluate partial constant expressions

- evaluate conditional ifs
- dead store eliminations

Other optimization (that does not apply in this code, but is very powerful) is: common subexpression elimination.

4.2.2. Assignment of identifier used next line

Code before optimization	Code after optimization
var1 = Identifier (constant or local variable) var2 = var1	var1 = Identifier (constant or local variable) var2 = Identifier

This optimization allows us to remove many redundant vreg assignments. In fact one of the slowest part of the program being the part of updating references in the reference counted pointers, removing most of them using this simple optimization makes it so note-worty.

This optimization is a very simple propagation that goes over the entire block.

4.2.3. Assignment of expression

Code before optimization	Code after optimization
vreg1 = <expression> var2 = vreg1	var2 = <expression>

This looks like previous one, but the main difference is that implementation wise is that the second instruction is deleted. Similarly, this allows that expression to be more complex (like a function call), and propagating a funciton call lower may not be desirable.

If vreg1 is not used later, will allow to make a „reverse propagation" as expressions cannot be always cleanly propagated.

4.2.4. Evaluate constant expressions

Code before optimization	Code after optimization
var = constant1 (operator) constant2	var = result of the constant1 (operator) constant2

This is crucial to remove some redundant code, and this code matches the following cases:

- binary operators: all operations like +, -, *, /
- boolean operators: &&, ||
- unary operators: -a, !a
- conversion operators: **(float) 3.0** becomes **3.0f**
- conditional operators, as .Net has no other if statements than **ifTrue** or **ifFalse** the leading expression can be computed before evaluating this If value

4.2.5. Evaluate partial constant expressions

Code before optimization	Code after optimization
var = constant1 (operator) identifier	var = result of the constant1 (operator) identifier

Multiplication with zero, addition with zero, 0 divided by anything (but zero), give a value that can be computed. Sometimes it gives a constant (like, a = 0*b => a = 0) but sometimes it gives just another assignment (that may be removed later by optimization 4.2.2)

4.2.6. Evaluate conditional ifs

.Net offers by default basically just two if expressions, ifTrue and ifFalse, with an operand and a jump address.

If we have something like:

Code before optimization	Code after optimization
IfTrue(true) goto Label;	Goto Label;

IfFalse(false) goto Label;	Goto Label;
IfTrue(false) goto Label;	//delete instruction
IfFalse(true) goto Label;	//delete instruction

This optimization is very powerful, mostly in cases when you have logging and you have a global constant that is true or false to add logging in your application, which allows the code to remove a lot of tests and improve the performance of that section of the code

4.2.7. Dead store eliminations

Dead stores are variables that store a value that is not subsequently used.

If a variable is assigned twice but is not used over the entire block code, it can be safely removed.

Code before optimization	Code after optimization
var = expression with no side effects (...) //code where var is not used var = another expression	//delete the entire instruction (...) //code where var is not used var = another expression

If expression is with side effects, like an impure function, the assignment of the function is removed, and the function is called without any result.

4.2.8. Common Subexpression Elimination

Common subexpression elimination (CSE) is a very powerful optimization, as it removes a lot of redundancies in code. Even more important is that expensive function calls can be elided (not executed), if we know that some functions are pure (look for section 4.5.1. about purity of functions) we can merge all that use the same parameters.

Code before optimization	Code after optimization
Var1 = expression with no side effects (...) //code where parameters of expression are not //redefined Var2= same expression	cacheVariable = expression with no side effects Var1 = cacheVariable (...) //code where parameters of expression are not //redefined Var2= cacheVariable

CSE for simple operators is written by most compilers today, the noteworthy is just the function's CSE as most C++ compilers don't do it excluding for intrinsic methods.

CSE works for:

- unary operators
- binary operators
- field access
- pure function calls
- array read access

Some common subexpressions are generated by **4.3.5. Loop Invariant Code Motion**, and the optimizations, even if the user does not write a code specially to have redundant expressions, they may appear when hoisting the code outside of a loop can create duplicates the dependencies.

For example this code is written in the initialization part of NBody benchmark:

NBody initialization loop

```
pairs = new Pair[bodies.Length * (bodies.Length - 1) / 2];
int pi = 0;
for (int i = 0; i < bodies.Length - 1; i++)
   for (int j = i + 1; j < bodies.Length; j++)
      pairs[pi++] = new Pair() { bi = bodies[i], bj = bodies[j] };
```

Even at the first glance the bodies.Length appears to be duplicated, on the IL level it is not, because bodies.Length is given every time a different vreg by CodeRefractor. Similarly, some bodies fields are inside loops, and they are also stored in separate vregs.

Loop Invariant Code Motion will move the all: this->bodies calls outside the loops, and CSE will merge them into a new vreg (let's name it **newVreg)**. Later CSE will merge all: **newVreg->length** variables into a single variable.

4.3. Global optimizations (optimizations that work over more than a basic block)

4.3.0. Overview

Global optimizations look over patterns that go over the entire body of the function. They work mostly against jumps and variable usages/no usages.

4.3.1. Not used variables are deleted

As every function has local variables, if after optimizations we remove all usages to a variable, we safely remove it from the function local declarations.

Code before optimization	Code after optimization
Int vreg_1; //not used anymore	//delete local variable

It can be done safely by looking over all instructions for usages and definitions and compare with the entire declared variable list. If the variable list contains extra variables.

4.3.2. Variables that are assigned but not used can be deleted and the entire expression

Code before optimization	Code after optimization
local_1 = vreg_2 + vreg3; //not used anymore	//delete instruction

After some optimizations, it may happen that the result of a variable is never used, so the entire instruction is deleted. This may lead that vreg_2 and vreg_3 to not be used anymore, and they can be deleted subsequently too.

This optimization is done by looking for every variable that is assigned, and after that looking for it's usages. If is never used after, and the expression is with no side effects, we can remove it safely.

Code before optimization	Code after optimization
local_1 = Call_To_Method(...); //not used anymore	Call_To_Method(...)

A version of it, is that if the variable is assigned in a call, just the call is made, but the assignment is removed.

4.3.3. Remove unused labels

Code before optimization	Code after optimization
Label_3: //not used anymore	//delete instruction

After optimizations like 4.2.6 some labels may not be referenced anymore, so they can be deleted. A good consequence of this optimization is that sometimes it enlarges the basic block making possible that another optimizations to happen in place.

It is done by building a label table as candidates to be deleted. After that all jumps to labels will remove them from the candidates of removal. If are found labels that are not hit by jump instructions, they are removed.

4.3.3. Merge consecutive labels

Code before optimization	Code after optimization
Label_19: Label_20:	Label_19: //all jumps pointing to L20 are pointing now to L19

This reduces the number of labels and sometimes allows other optimizations, like the 4.3.4.

This is done by finding two instructions that are of type **label**.

If found, the algorithm is the following:

- the id of the both labels is stored

- the second label is deleted

- all jumps (conditional and not conditionals that are pointing to the second label) are redirected to the first label

4.3.4. Any goto to next line can be removed

Code before optimization	Code after optimization
Goto Label_20: Label_20:	//delete Goto Label_20:
ifTrue|ifFalse(test) Goto Label_20: Label_20:	//delete if Label_20:

This optimization allows sometimes optimization 4.3.3 (remove unreferenced labels) and can appear because other optimizations removed the instructions between the conditional or not conditional jump and the label itself.

4.3.5. Loop invariant code motion

Loops like For/While do sometimes have some parts that do not change so they are invariant.

Target loop code

```
public static void Main()
{
    var result = 0.0;
    var a = RandValue();
    var b = RandValue();
    var c = RandValue();
    var d = RandValue();
    for (var i = 0; i < 5000000; i++)
    {
        var auxC = Math.Sin(c);
        result = a + b + auxC + d + i;
    }
    Console.WriteLine(result);
}
```

The resulting code (with and without optimization)

Code before optimization	Code after optimization
`local_0 = 0;` `vreg_2 = SimpleAdditions_NBody__RandValue();` `vreg_3 = SimpleAdditions_NBody__RandValue();` `vreg_4 = SimpleAdditions_NBody__RandValue();` `vreg_5 = SimpleAdditions_NBody__RandValue();` `local_5 = 0;` **`goto label_74;`** **`label_41:`** ***`vreg_8 = (double)vreg_4;`*** ***`vreg_9 = System_Math__Sin(vreg_8);`*** ***`vreg_12 = vreg_2+vreg_3;`*** ***`vreg_13 = (double)vreg_12;`*** ***`vreg_15 = vreg_13+vreg_9;`*** ***`vreg_17 = (double)vreg_5;`*** ***`vreg_18 = vreg_15+vreg_17;`*** **`vreg_20 = (double)local_5;`** **`local_0 = vreg_18+vreg_20;`** **`local_5 = local_5+1;`** **`label_74:`** **`vreg_28 = (local_5 < 5000000)?1:0;`** **`if(vreg_28) goto label_41;`** `System_Console__WriteLine(local_0);`	`local_0 = 0;` `vreg_2 = SimpleAdditions_NBody__RandValue();` `vreg_3 = SimpleAdditions_NBody__RandValue();` `vreg_4 = SimpleAdditions_NBody__RandValue();` `vreg_5 = SimpleAdditions_NBody__RandValue();` `local_5 = 0;` *`vreg_17 = (double)vreg_5;`* *`vreg_12 = vreg_2+vreg_3;`* *`vreg_8 = (double)vreg_4;`* *`vreg_13 = (double)vreg_12;`* *`vreg_9 = System_Math__Sin(vreg_8);`* *`vreg_15 = vreg_13+vreg_9;`* *`vreg_18 = vreg_15+vreg_17;`* **`goto label_74;`** **`label_41:`** **`vreg_20 = (double)local_5;`** **`local_0 = vreg_18+vreg_20;`** **`local_5 = local_5+1;`** **`label_74:`** **`vreg_28 = (local_5 < 5000000)?1:0;`** **`if(vreg_28) goto label_41;`** `System_Console__WriteLine(local_0);`

This optimization will notice that (pure) function calls, expressions and assignments do not depend on anything is set in the loop iteration.

In the previous code sample the bold area is the loop, and the italicized text is invariant and we can notice that for this specific code, half of code is moved outside the main loop.

The algorithm is:

- Detect loops. This loop detection algorithm matches the C# pattern: a consecutive **goto** and **label** instructions (like **goto label_74** and **label_41**) and after that the loop will be as long as the latest conditional jump to this label
- For every loop, find which variables are reassigned so they are loop variant
- Detect any invariant expressions which do not depend on the variables which are reassigned
- Move expression at the start of loop (before the first **goto**)
- Repeat with the first step until no move is possible.

4.3.6. One assignment with constant

Code before optimization	Code after optimization
Var1 = constant: //not used anymore (...) Var2 = expression using Var1	//deleted (...) Var2 = expression using constant

This operation will look over all body of the function. This is lightweight compared with constant dataflow propagation, both having the same complexity but the DFA optimization for constant propagation will store more states for every instruction.

4.4 Dataflow analysis

4.4.1. Dataflow analysis overview

Dataflow optimizations allow to make more aggressive predictions about how code behaves by visiting all branches of the program and testing some speciffic hypothesis.

The DFA optimization passes do work like following:

- the analysis is done before any optimization is made
- the analysis starts with some hypothesis like: all variables are constant
- at every instructions and every branch the hypothesis is checked for accuracy
- if the hypothesis cannot be accepted as true, it is removed (or marked) in the hypothesis table
- after all analysis are done, and if the hypothesis table is not empty, will go over all instructions and will replace the consequence of these hypothesis based on the value

These optimizations are more powerful than block based optimizations as they can evaluate some hypothesis per lines of programs, and some complex DFA optimizations (which are not necessarily part of this project, but may be impemented later) can prove some optimizations regardless of jumps and loops.

4.4.2. Constant dataflow propagation

Code before optimization	Code after optimization
(...)	(...)
var1 = 3;	var1 = 3;
Goto 20;	Goto Label_20;
var1 = 2;	var1 = 2;
Label_20:	Label_20:
Local1=var1;	**Local1=3;**

This optimization happen because the visiting all nodes through all branches can conclude that some variables will not change their values in all branches (even they can be assigned on some other sections of code, if they are not reachable (look for the next section too), we can replace safely the var1 with it's constant value.

As this optimization cares about constants, this enables many local optimizations that were not possible because some branch expressions were in between the source and the final value.

4.4.3. Reachability lines analysis

Let's say we have the following (schematic) intermediary representation code:

Code before optimization	Code after optimization
(...)	(...)
var1 = 3;	var1 = 3;
Goto 20;	Goto Label_20;
var1 = 2;	**//delete code in between Goto Label_20 and Label_20**
Label_20:	Label_20:
Local1=var1;	Local1=var1;

Even it is obvious as looking to the code that will be always 3, in fact without Constant Dataflow Propagation, with previous analysis (even with 4.3.* section) we cannot decide and remove out this code.

This optimization will do the following:

1. will start with the first instruction and will follow every possible branch and will advance the code up-to the case either will find another instruction that was already visited or will find the final return statement;
2. The lines that are not marked are deleted

This optimization, when succeds, makes more potent optimizations like: **4.3.4. Any goto to next line can be removed**, and subsequently, if Label20 was not referenced later, will trigger **4.3.3. Remove unused labels**.

Another very important optimization as a result of Reachability lines Analysis is: 4.3.2. when some variables can become not used anymore, so they can be removed from the function.

4.5. Some inter-procedural optimizations

4.5.1. Annotation and evaluation

Functions are annotated to get better information so it can have:

- Is getter/setter
- Is empty
- Is pure
- Is readonly – works like pure functions, but it can read fields or global states, but the single effects are the result of function. For example any getter function is read only but not pure.

Pure functions are functions which respect the following properties:

- are functions that they depend on the parameter's input to give the result (so are non-void functions)

- don't have any visible side effects

- give the same answer for the same set of constants

Some mathematical functions do offer purity, and most functions from the class System.Math in the .Net framework guarantees purity.

Code Refractor's OpenRuntime implementation of Math's class does annotate all math functions as [Pure] and based on this, it can guarantee that all simple functions which:

- do not call unknown functions (by the CR's evaluation). If they are calling pure functions, they remain pure in evaluation;

- does not change/read static or not static fields

- does just simple math and assignments and/or jumps

are pure.

4.5.2. Optimizations based on analyisis

4.5.2.1. Call of pure function with constant parameters

Based on this purity evaluation, any function that is pure, if is called with constants, the CR compiler will replace the call of the function with the .Net's evaluation of the entire function, avoiding the function call all-together.

4.5.2.2. Call of pure function with the same parameters

Purity checking allows that calling a pure function twice with the same parameters to let the compiler cache into a local variable, and later to remove the function call for the second time.

4.5.2.3. Inlining

Inlining is almost always advantageous to be called, because it guarantees that the body of the function that is inlined can benefit from other optimizations, local for the host function in which the child function is inlined.

Anyway, inlining sometimes doesn't work (like in case of a recursive call), and sometimes it gives code-bloat, which the advantage of inlining (as performance), may get eventually a slower code because the code of the function combined with all inlined children would not fit in the CPU's cache.

So CR detects some cases when a function can be inlined and will always give a smaller overhead than calling the method and it knows it can do it safely:

- Empty functions;

- Getter functions;

- Setter functions.

4.5.2.3. Improved CSE

Common Subexpression Elimination benefits if it knows that a function is pure so it can be moved.

Code before optimization	Code after optimization
A = Call_To_Pure(c1, ..., cn); //... B = Call_To_Pure(c1, ..., cn);	A = Call_To_Pure(c1, ..., cn); C = A; //... B = C;

4.5.2.4. Improved DCE

A pure/readonly function which result is not used the calls can be removed. For this to work there is an optimization step that makes possible to remove the left side (look to Section 4.3.2.)

Code before optimization	Code after optimization
Call_To_ReadOnly(); Call_To_Pure();	//delete instruction

4.5.3. Class Hierarchy Analysis

Class Hierarchy Analysis (CHA) is based on a fairly simple premise: if the types are tracked, sometimes virtual calls, which have an overhead in calling, can be determined as a not virtual call. This is important because the first designs of CR do not implement virtual calls at all, and when a virtual call is made, the code generator will call the best match that matches closes the virtual call as a direct call.

Original code

```
using System;

namespace Figures
{
   class Figure{
      public virtual double Area(){
         return 0;
      }
   }
   class Circle : Figure{
      public override double Area()
      {
         return base.Area();
      }
   }
   class Program
   {
      public static void Main(string[] args)
      {
         var c = new Circle();
         Console.WriteLine(c.Area());
      }
   }
}
```

The CIL code is the following:

CIL intermediate code

```
.method public hidebysig static
   void Main (
      string[] args
   ) cil managed
{
   // Method begins at RVA 0x2094
   // Code size 20 (0x14)
   .maxstack 1
   .entrypoint
   .locals init (
      [0] class Figures.Circle c
   )
```

```
IL_0000: nop
IL_0001: newobj instance void Figures.Circle::.ctor()
IL_0006: stloc.0
IL_0007: ldloc.0
IL_0008: callvirt instance float64 Figures.Figure::Area()
IL_000d: call void [mscorlib]System.Console::WriteLine(float64)
IL_0012: nop
IL_0013: ret
} // end of method Program::Main
```

Notice the line:

IL_0008: callvirt instance float64 Figures.Figure::Area()

This line shows that there is a virtual call to the base method at the CIL level.

But the final C++ code (for the not optimized method is:

Use the closest method in hierarchy based on type

```
System::Void
Figures_Program__Main(std::shared_ptr< Array <
std::shared_ptr<System::String> > > args)
{
std::shared_ptr<Figures::Circle> local_0;
System::Double local_1;
std::shared_ptr<Figures::Circle> vreg_1;
std::shared_ptr<Figures::Circle> vreg_2;
System::Double vreg_3;
}
```

```
vreg_1 = std::make_shared<Figures::Circle>();
Figures_Circle__Circle_ctor(vreg_1);
local_0 = vreg_1;
vreg_2 = local_0;
vreg_3 = Figures_Circle__Area(vreg_2);
local_1 = vreg_3;
return;
```

Notice the call:

Vreg3 = Figures_Circle__Area(vreg2);

Which is a mapping to Figure.Circle.Area(Circle c) method.

4.6. Abstraction lowering and escape analysis

4.6.1. Escape analysis information

We say that an object does not escape if the compiler can decide the reference counting usages of that object to all points of the program where the object lives.

Examples of programs with objects that do escape

```
    class SampleEscaping
    {
        public NBodySystem
CreateSystem()
        {
            return new NBodySystem();
        }
```

```
public void AddToList(
     List<Body> bodies,
     Body body)
     {
         bodies.Add(body);
     }
```

First of all it is important to see why these methods do have escaping behavior and where it is.

The CreateSystem method does return an object, and we know that the “reference counting” to a new creatly object is “one” so why does the method we say it is escaping? It is because we don’t know to all usages of the object the reference count. Because the method can be called by another method which can change the reference count, the method is escaping and the reference count cannot be determined.

Similarly, even we know that AddToList class is able to add into bodies a reference of body into the list, this implies that the reference counting of Body body has been changed and in short it implies that the Body instance escapes.

4.6.2. Why escape analysis is important?

There are many optimizations escape analysis is really important and it is more efficient to have it as a part of CR/C++ design than even in .Net?

The short answer is: reference counting is very slow in practice, and the second slowest part in using a C/C++ allocator in a memory intensive program is the memory allocation. Escape analysis can reduce reference counting updates and it can improve the application allocation rate.

Reference counting performance depends on two components; even a simple assignment of two smart pointers is really complex:

C++ smart pointers and their C logic

```
// C++ code

std::shared_ptr<Class> a, b;

(…)
a =b;
```

```
// C code
a.RefCount--;
if(a.RefCount == 0)
        delete a.Ptr;
a.Ptr=b
b.RefCount++
```

Depending of runtime, reference counting has to be protected against concurent access, making the code even slower.

Allocation is also much slower because the most allocators in C++ do use lists of allocation items, and they have to loop into them to look for a free memory area. Most (generational) garbage collectors do

allocate in a small heap of memory and they are moving this heap into the big heap (named old generation) only if this small heap is done, but in short, the allocation using a GC is very fast.

So using a reference counting will also mean that two allocations are done at once, once for the referring object (which keeps the reference count) the other one for the object itself, making it even slower.

C++ smart pointers' C-like logic

```
a = new SmartPtr<A>();
a.Ptr = new A();
a.Ref=1;
```

Using **make_shared<A>()** function, will reduce the overhead of allocation to just one. So the generated (logical code) of the makeshared for the same code is:

C++ smart pointers' C-like logic using make_shared

```
smartPtrAddress = Allocate(sizeof(SmartPtr>() + sizeOf(A));
a.Ptr = &smartPtrAddress + sizeof(SmartPtr);
a.Ref = 1;
```

The difference is that there are two allocations and just one assignment in the first implementation, the second implementation are two assignments but just one allocation, and allocation is much more expensive than the assignment.

4.6.3. Improving reference counting by using escape analysis

As a variable is not escaping, some optimizations can be done, stack allocation is the key one to reduce the allocation cost (for variables that we know that they don't escape) and using direct pointers for all usages of non escaping use-cases.

Examples:

Simple getter function where the parameter does not escape

```
public static bool GetResult(ObjectType _this)
{
    return _this.Result;
}
```

Resulting code (reduced, no escape analysis):

Simple getter function where the parameter does not escape

```
System::Double
SimpleAdditions_NBody__GetResult(std::shared_ptr<SimpleAdditions::NBodySystem> _this)
{
System::Double vreg_2;

vreg_2 = _this->Result;
return vreg_2;
}
```

With escape analysis the code can be changed slightly to not use smart pointers but raw pointers like this:

Simple getter function where the parameter does not escape

```
System::Double SimpleAdditions_NBody__GetResult(SimpleAdditions::NBodySystem* _this)
{
```

```
System::Double vreg_2;

vreg_2 = _this->Result;
return vreg_2;
}
```

As it can be noticed, instead of using std::shared_ptr template class (which wraps the smart pointers), the code is using a raw pointer. This as a side effect, will make no increment/decrement reference counting for _this parameter if the function is called with any external object.

4.6.4. Evaluating a variable is not escaping algorithm

A variable is not escaping if:

- is not a reference variable (so we can remove the computation of escapness for non referenced variables)
- is never assigned to an array item, to a field or to a variable
- is not the result of a return instruction
- is never called to a function where it escapes the parameter

Given this, the final variable is marked as not escaping.

4.6.5. Generating non escaping mode

Variables that are not escaping are defined as „pointer" variables, the non escaping allocation variables are named „stack" variables.

After this if there is an assignment from an escaping variable to a non-escaping one, the .get method is called.

Assignment from a shared pointer to a raw pointer

```
std::shared_ptr<SimpleAdditions::NBodySystem> _this;
SimpleAdditions::NBodySystem* vreg_2;
vreg_2 = _this.get();
```

Similarly, a stack allocated variable will be assigned to a raw pointer

Assignment from a stack variable pointer to a raw pointer

```
SimpleAdditions::NBodySystem _this;
SimpleAdditions::NBodySystem* vreg_2;
vreg_2 = &_this;
```

Call of **.get()** function and using of ampersand & are used in all other cases like, but not limited to:

- array item assignment
- function calls

4.7. Aggressive inter-procedural optimizations

4.7.1. Remove unused parameters over whole program

This optimization does what it says: it looks for a parameter is unused in a function and marks it as: unused. This will mean that in all other functions that call the method, the parameter is set as unused, so it can be removed at the call site.

So if you have a call:

```
var c = 5;
var result = Max(a, b, c);
```

And "c" is not used in method Max, the C variable will be removed initially in call, and later will be removed from entire function!

4.7.2. Optimizing the constant arguments over whole program

This optimization will see all call sites, and if they are called with the same constant, the program will replace in the body of the function with the constant and will make the variable unused. So for the same code as in previous example,

```
var c = 5;
var result = Max(a, b, c);
```

From the previous optimizations (section 4.3.6) the code will become:

```
var c = 5;
var result = Max(a, b, 5);
```

Value of 5 is inserted in the body of method Max and Max method will not depend on value 5.

Later, because of optimization 4.7.1. the code will become:

```
var result = Max(a, b);
```

when 5 and c are removed.

This code is somewhat not so complex, but this kind of coding is very common when working with logging: “if (debug)printf …” .

5.Compiler implementation and runtime optimization

5.1. Overview

Compilation time itself has to be taken into account when optimizing the program, as programs do grow and the optimization steps hit grows biggeer and bigger. So, based on this, there have to be taken into account some principles:

- linear reads are faster than jumping all-over into memory
- most optimizations will not be „true" for a sequence of code, so is better that every time when a code change happens, this have to change the in-memory state and to do it expensive, but the rest of optimizations should not change anything (or almost anything)
- List<T> (the high level Generics class that makes easy to add/remove/substract – equivalent with std::vector<T> in C++) is slower than the T[] (array of T)

Right now (compared with the initial designs), compilation step is much faster. Optimizations can work (by uncommenting some lines of code) even by using multiple cores. As for example, an OpenGL application was taking like 600-700 milliseconds on an Intel i5-540M. Right now it takes for CR to generate the code around 170-220 milliseconds (the first time is slower, as spinning disks are slower at access). But by all measures is much faster.

Most optimization steps do the following: look for a pattern of code that matches a property, after that if it matches, it tries to perform the optimization by impacting some instructions; it notifies the compiler that some changes are done. CR after this will try to perform all optimizations up to the point no optimization can be done. CR in a typical case will apply (using the default codebase optimizations) 35+ optimization steps for every function, every instruction, etc.

In typical case let's say there is just one some optimizations that can be done, for example: a variable is nowhere used, so it can be removed. Before noticing that the variable can be removed, the compiler performs other optimizations, and right after will perform another pattern matching. At last when the optimization of the unused variable is match, for every instruction CR will track all declared variables and CR will compare with all used variables. At the end what remains, they can be removed.

But as someone can notice, the step that makes that one optimization succeed is based on some most common knowledge:
- variable usages per instructions
- the instruction kind: if one optimization will remove a declared but never used variable, it will have to take in account if the instruction is a call to a function or if is a simple math operation
- jumps: some optimizations do work just for a sequence of instructions that have no branches and jumps, so looking for jumps and labels are important delimiters for these instructions

Based on this, every time an optimization is performed, the optimization framework in code will recalculate this information so it can be reused. So if the first optimization step needs to check variable usages, and doesn't perform any optimization, the second optimization can use the same usages data, as correct as no change is done.

5.2. Indexing of instructions

Compilation time itself has to be taken into account when optimizing the program, most optimizations will:

- interact with other optimizations

- will not be hit more than once in a function or maybe never

This makes to define an instance named: UseDef (UsagesDefinitions) that for an intermediate instruction set will keep track of:

- for every instruction the usage count

- for every instruction the definition count

- for entire function will index the instructions, so if an optimization pass will optimize let's say just call operations, if the function has no call, it is possible to ask for the index of call instructions directly or to skip them fully

- all instructions are stored as an array of instructions, this step itself improves by 5-10% the optimization's speed because the .Net optimizer will simplify the array iterations to not have checked Index-Out-Of-Bound which would not be necessary

From optimization steps, every time when an optimization changes something, will notify the optimization framework (by returning true value, instead of false) and the optimization framework will rebuild this index.

5.2. Program Closure and closure resolving of methods

Programs do depend on various parts of code. Initially, when CR design started, there was a MetaLinker class, which will link everything up-to the point of .Net classes that are not linked (first of all because of license issues). Also, the rest of C++ code for missing .Net methods was written by hand in a separate include.

In an medium step, there was defined as described in Section 3.2 there was defined a CROpenRuntime assembly that will match .Net runtime classes with backend classes. This design has some problems in special two that will be explained here:

- CR has limited support for .Net constructions, this doesn't mean that cannot create full programs, it certainly can, but for example there is no support for generic methods, or a full virtual methods call (some components are implemented, but they are not fully supported, and it is very likely that the compiler will crash on them, or the code will run unexpectedly)
- as both components: MetaLinker and CodeRuntime are in fact singletons (they are unique for program) they cannot easily be replaced or have two „programs" inside two programs. In future, it would be great if OpenCL support will be added for simple functions. This will mean that all OpenCL (which is a variant of C) that is generated will need an CrOpenCLRuntime kind of assembly, which supports just other methods

To solve these issues, the code right now is handled in a split steps component. The analysis of the program starts with a ProgramClosure that is initialized given a runtime, and it stores the list of types and methods and types are referenced by an entry method.

The ProgramClosure has some concerns of the previous MetaLinker logic, but most of logic is basically to give high level management of closure:

- it calculates initial methods/types closure

- it optimizes the methods

- it recalculates the methods/types closure (because some methods after optimizations may not used anymore)

- it writes the C++ output

As optimizations are explained at alrge in section 4 of this thesis, what it remains to be basically described is the part of how we calculate the closure of types and methods.

5.2.1. Calculate type closure algorithm

CR knows which are primitive types (like int, double, char) and it will start from a closure of methods and will add the following types:

- declaring type of a method: NbodySystem.Advance requires to add NbodySystem even if the class is static and all values are static

- types of parameters, return types
- types of fields of a class
- types of variables in the methods

It also, keeps a mapping of types so the String type, a fairly common types is in fact CrString type and the compiler will replace logically all the occurences of String to be CrString.

The User Defined Resolver (described in section 5.2.3) let the user to replace at implementation level some classes so the TypeClosure will not hit unknown types (for example interfaces or generic types), but is given a stub class

5.2.2. Calculate method closure algorithm

The program closure starts from an entry point and will visit all Call instructions.

The single notable part is that there is a virtual method implementation (very limited, but to describe the problems) that will do the following: every time when you call a virtual method and you have in type closure a type that implement a method, all realizations of virtual method are added

Vtable implementation is per-method but not per-type as in a typical C++ implementation. In a C++ class, if you would „dereference this" you will go directly to a virtual table, which contains all pointers to A::f1(), A::f2(), etc.

CR actual implementation does have f1Vtable in the form: A::f1(), B1::f1(), ... being a more CPU cache friendly, but as implementation of virtual methods is not finished, I can just speculate that a bigger class hierarchy where the optimizer will not be able to optimize the virtual calls (like in Section 4.5.3), a virtual call will be executed faster.

5.2.3. The resolver and user-defined resolver

Ever y time when a new type or method is hit by the ProgramClosure, will ask this method to be resolved as a MethodInterpreter. This method will be either: Default (CIL translated to CR intermediate representation), CppRuntime or Runtime method (if is a C++ method, or a CR OpenRuntime C# method), PInvoke (calls a Windows dll using platform defined services).

Anyway, the user is allowed to before the final resolution is given to a MethodInterpreter based on a method, to define the MethodInterpreter content and to resolve it.

Here is an example of user-defined resolver:

Type resolver user defined code

```
    public class CrGlu
    {
    }

    public class CrGl
    {
    }
    public class CrSdl
    {
    }
    public class TypeResolver :
CrTypeResolver
    {
        public TypeResolver()
        {
            MapType<CrGl>(typeof(Gl));
            MapType<CrGlu>(typeof(Glu));
            MapType<CrSdl>(typeof(Sdl));
            if (method.DeclaringType == typeof(Glu))
            {
             ResolveAsPinvoke(methodInterpreter,
"glu32.dll", CallingConvention.StdCall);
                    return true;
             }
             if (method.DeclaringType == typeof (Gl))
             {
              ResolveAsPinvoke(methodInterpreter,
"opengl32.dll", CallingConvention.StdCall);
                return true;
             }
             if (method.DeclaringType == typeof(Sdl))
             {
               ResolveAsPinvoke(methodInterpreter,
"sdl.dll", CallingConvention.Cdecl);
               return true;
             }
```

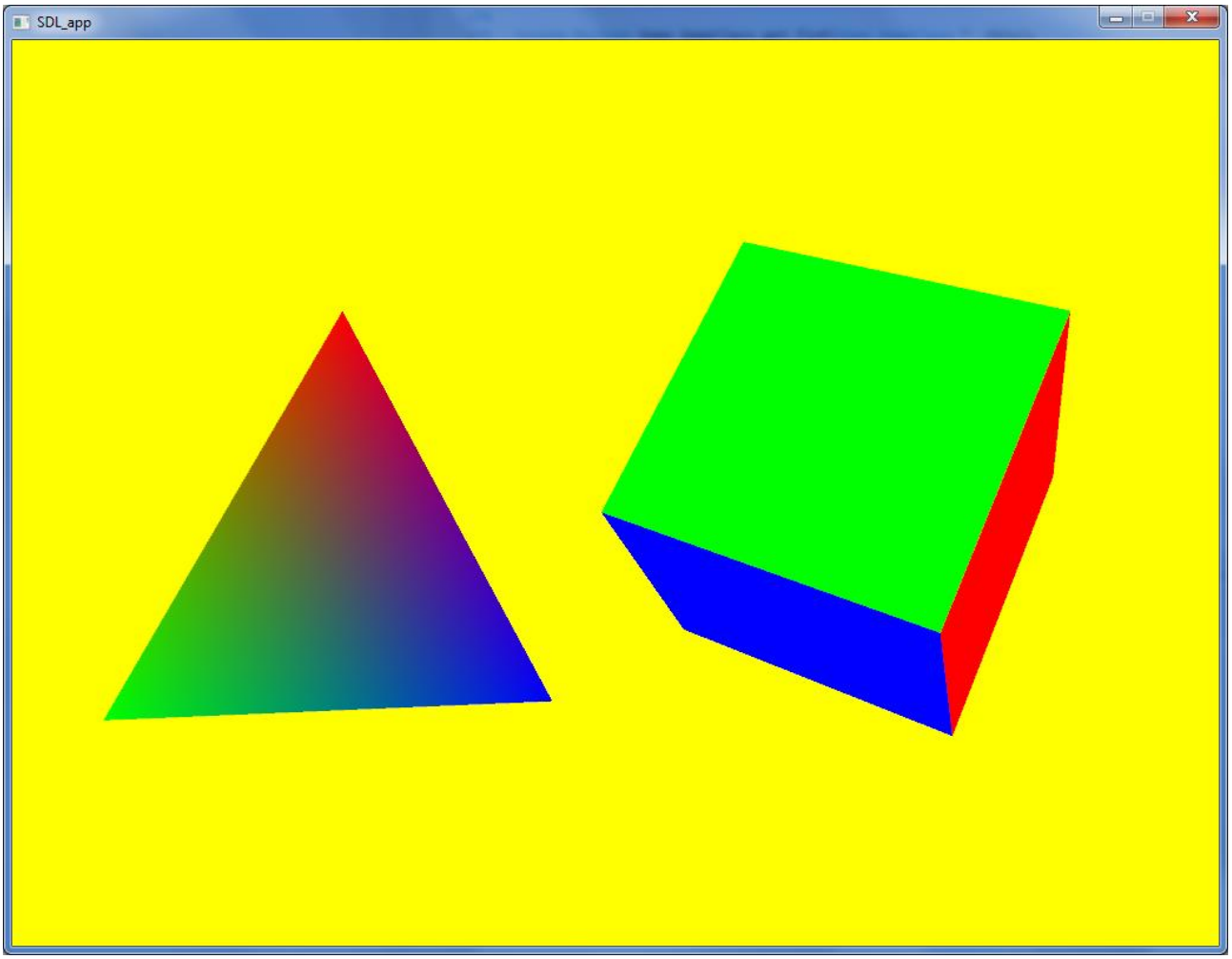
```
        }                                                   return false;
public override bool                                    }
Resolve(MethodInterpreter methodInterpreter)        }
{
 method = methodInterpreter.Method;
```

The code may be overwhelming at start, but lets see what it does:

Every time when a the type Gl is found, it should be repalced with CrGl's definition, similarly, Glu is replaced with CrGlu and Sdl with CrSdl.

And every time when an OpenGL, GLU or SDL call is happening, the class will replace the call to methods in class with PInvoke calls, and it will set the calling convention and DLL.

As a result, starting with a full C#/OpenGL sample, this application is running correctly:



6. Benchmarks

6.1. Overview

Benchmarks are good to evaluate performance, but also have some shortcommings:

- compilers (including CoreRefractor) optimize just for some scenarious, this means in short that every time when you benchmark, you may get a very good result because it happens that the compiler to have a proper optimization pass that matches the code pattern;
- modern CPUs are complex, which makes the same program to execute depending on the instruction blend faster on one family of processors and slower on others;
- most modern CPUs have CPU frequency scaling, which makes that for short running benchmarks the time that the CPU scales to maximum frequency to not happen in some configurations. This means that if a CPU takes 0.1 seconds to switch the frequency, and program A supposedly is 2 times slower than program B, if program B finishes in just 0.1 seconds, the program A can finish in 0.17 seconds (not in 0.2 seconds) because CPU will boost the frequency for a half of running program;
- benchmarks reflect various use-cases, but for CPU bound benchmarks, it may happen that the applications that most users do use to not depend on CPU, but on other computer component: games' performance is mainly dependent on the video card and GPU computing power, database engines depend on disk IO (mostly sequential performance), office applications depend very often on the memory bandwidth and the size of the CPU cache;
- some benchmarks do test how fast is the memory allocator (a runtime component, not a compiler improvement);
- at last, some popular benchmarks do make that compilers optimize for benchmarks, with less regard about real life tests.

6.2. Bad benchmarks: OS News 2004 article

I will present next a very bad benchmark, where CR happens to work really well, the OSNews' 2004 „Nine Language Performance Round-up: Benchmarking Math & File I/O"[8].

In short these benchmarks compute the following:

- int 32 math
- int 64 math
- floating point math
- trigonometric math
- IO bound code

As we said in introduction, the IO bound code makes that even Python to run in the same league (like 20% slower than C code) as the fastest implementation, as it will be limited eventually by the particular disk/OS implementation.

What about the rest of the benchmarks:

Quoting from the article:

"…" **32-bit integer math**: using a 32-bit integer loop counter and 32-bit integer operands, alternate among the four arithmetic functions while working through a loop from one to one billion. That is, calculate the following (while discarding any remainders):
1 - 1 + 2 * 3 / 4 - 5 + 6 * 7 / 8 - ... - 999,999,997 + 999,999,998 * 999,999,999 / 1,000,000,000
64-bit integer math: same algorithm as above, but use a 64-bit integer loop counter and operands. Start at ten billion and end at eleven billion so the compiler doesn't knock the data types down to 32-bit." (…)

[8] Look to reference 9

It just happen that all CPU benchmarks of OS News trigger PureMethod evaluation of Code Refractor and makes that all the codes are evaluated at compile time. Some reader may notice that this also means that instead of benchmarking the program, you move the execution time inside compiler, so the execution time is felt somewhere, so to make this benchmark fair will mean basically to make sure I disable compiler optimizations.

6.3. Better benchmarks: Nbody test

NBody benchmark tests a simulated solar system and tries to compute the position of the planets. Even this benchmark is considered to be a memory bandwidth program, it has some good characteristics:

- the complexity of the problem is $O(n^2)$ with a big part set around accessing memory arrays
- the data is small and fits in all today's CPU caches, meaning that it works fairly well as a computation kernel
- it uses a compiler intrinsic (sqrt) so a compiler should be able to optimize it as a simple instruction, not a method call
- for big values the results are reproductible and the range of execution time is always in some miliseconds
- as for CodeRefractor, the baseline code (with no optimizations) is around 8 times slower than .Net, making it a great baseline to optimize from

I don't pretend that Nbody is a great benchmark, but compared with previous one, is a much better one. In fact, because it has what most programs do have:

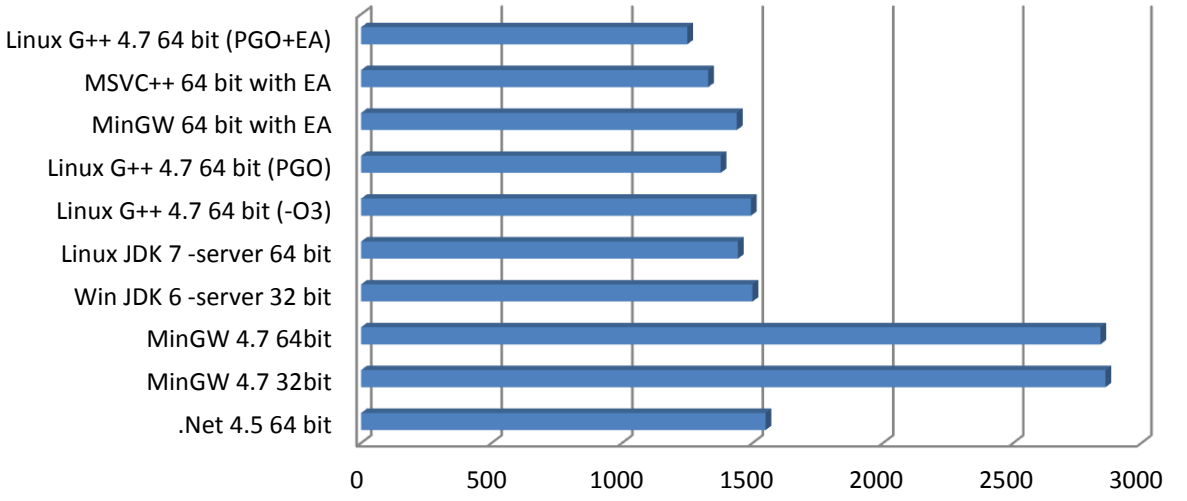
- field accesses
- array accesses
- math
- loops

makes it that the optimization of all these operations will reflect into a lot of applications improvements.

The benchmark results are: baseline (no optimizations) for 5.000.000 iterations - all configurations tested are:

Runtime	Time(ms)
.Net 4.5 64 bit	*1550*
MinGW 4.7 32bit	*2860*
MinGW 4.7 64bit	*2840*
Win JDK 6 -server 32 bit	*1500*
Linux JDK 7 -server 64 bit	*1444*
Linux G++ 4.7 64 bit (-O3)	*1494*
Linux G++ 4.7 64 bit (PGO)	*1378*
MinGW 64 bit with EA	*1440*
MSVC++ 64 bit with EA	*1310*
Linux G++ 4.7 64 bit (PGO+EA)	*1250*
Linux G++ 4.7 64 bit (PGO+EA+LICM)	*1240*
Mono (Win32)	*1910*
Mono –O=all (all optimizations)	*1610*
Mono –llvm	*1710*
Mono (Linux 64bit)	*1951*
Mono --llvm (Linux 64 bit)	*1885*
Baseline	*27350*

The Java tested are rewritten code into Java, translation line by line with similar semantic and code characteristics.



	.Net 4.5 64 bit	MinGW 4.7 32bit	MinGW 4.7 64bit	Win JDK 6 -server 32 bit	Linux JDK 7 -server 64 bit	Linux G++ 4.7 64 bit (-O3)	Linux G++ 4.7 64 bit (PGO)	MinGW 64 bit with EA	MSVC++ 64 bit with EA	Linux G++ 4.7 64 bit (PGO+EA)
Time	1550	2860	2840	1500	1444	1494	1378	1440	1330	1250

Conclusion

As an evaluation of this project that comes with this paper, it can be said that it is possible to make a static compiler to compile a stack based virtual machine using ahead of time techniques.

Using compiler optimization passes it is possible at least in some cases to generate a faster code than the desktop/server class compiler of the virtual machine would generate(in Nbody benchmark to be 25% faster with the GCC compiler vs the best .Net implementation), and it can be removed the dependency to the host runtime (here .Net).

This thesis presents how critical components of a compilation and runtime systems are targetted and implemented and how various semantic differences can be work-arounded and solved by using speciffic optimizations (like escape analysis, or using merging of data).

The actual design allows some use-cases where either .Net or Java would not work as well as an ahead of time (AOT) compiler:

- **for memory constrained systems, the memory overhead of GC may not be preferred to a C like memory model**: As of time of writing, the entire memory for a smartphone is around 512 MB or 1 GB, but this memory is shared with the OS. The GC applications typically are using like 2x more memory than steady state.
- **for slow systems that need responsive close-to-realtime requirements** – like a game – that is not acceptable to have breaks because full GC collection happens, a reference counting memory model may make breaks into manageable pauses. Having escape analysis makes the reference counting to not be that expensive in execution time and still give a reduced pauses
- **for embedded or slow machines where the cost of JIT compilation is visible**, a precompiled binary will make application to performe as fast (or maybe faster, as the benchmarks show) but with no startup overhead. Also, making compilation before execution, can let the developer to let the compiler to try making as many compilations as are needed with no worry that the application has to remain interactive (a concern in JIT compilation)
- **it makes possible to run final executable with the same semantics without the original VM**. This is really important for some AppStores like Apple's Iphones and Ipad ones, where for adding support for Xamarin Mono will add some (few) megabytes by default to any application as „MonoRuntime". This is around 4MB which every user will have to download on top of the application's assets

At last, but not at least, this project is opensource/free and the compiler is under GPL2 and the runtime under Mit/X11 license, and the shortcommings and improving support of various .Net features can be extended and addressed based on the future needs of its users.

Trivia

Here are some facts about CodeRefractor:

- The original name of the project was SlothVM, and it was a reference to RoboVM where the main maintainer/developer was nicknamed Sloth, and also as for being a very slow but promising implementation of a VM
- The CodeRefractor name was proposed by author's twin brother, because „it will look like .Net but it will never run in the very same" like a refracted image
- The development of the project progressed with no code changes from two incompatible binary versions of .Net (4.0 and 4.5) and the code should be compilable in .Net 3.5 (VS 2008)
- At least for the first 6 months of the project, GCC was hardcoded using the DevC++ distribution on Windows. Linux was always a small fix but was never a target, even the changes to make it work are minimal
- The author contributed to MonoDevelop's IDE on C# code semantics fixes (named Nrefactory) and he found that the project (Xamarin Studio, the commercial counter part) was too pricey to ever be affordable for common folks, making a motivation to start the project as looking for free .Net/Mono runtime
- Compared with level zero of optimizations (that implement with smartpointers 1:1 the semantics of the IL code for Nbody but in C++) at the time of writing this thesis, using GCC/Windows, and Linux on 64 bit with all optimizations , there is an over **20x speedup.** Anyway, even if the CR compilation time increases like 10x from 60 ms to 600 ms (less than 400 ms in Release mode), the total compilation time is shorter , because CR removes like 600 lines (from 1200 to 600 lines in the final generated code), and the C++ compiler will compile faster (with more than 1 second faster).
- Memory usage of the compiler is fairly small. Many small objects are allocated for various states, mostly tracking usages and definitions. With no caching the compiler generates like 6 MB of allocations for NBody benchmark, and calling **GC.Collect()** in middle of optimization steps, makes that the compiler to use 7.2 MB of memory, which includes the .Net runtime, various representations that are stored, so it may work for small programs on a Pentium 2 class with 64 MB machine with Windows 2000. GCC compilation (or any C++ compiler) may be much memory hungry.
- The project was targetted to match use-cases of game development and phone development, and a simple SDL application (on Windows) was running (but no game ported yet)
- The author understands compiler theory, and some of the ideas are based from the Standford's Compiler course, which was made public by Coursera at the end of 2012

References

1. CIL infrastructure, ECMA standard

   http://www.ecma-international.org/publications/standards/Ecma-335.htm

2. Alex Aiken's page: http://theory.stanford.edu/~aiken/

3. Alex Aiken's Compiler Course on Coursera (2012): https://class.coursera.org/compilers-2012-002

4. Paper: **Virtual Machine Showdown: Stack Versus Registers** (Yunhe Shi, David Gregg, Andrew Beatty): *We found that a register architecture requires an average of 47% fewer executed VM instructions, and that the resulting register code is 25% larger than the correpsonding stack code. The increased cost of fetching more VM code due to larger code size involves only 1.07% extra real machine loads per VM instruction eliminated. On a Pentium 4 machine, the register machine required 32.3% less time to execute standard benchmarks if dispatch is performed using a C switch statement. Even if more efficient threaded dispatch is available (which requires labels as first class values), the reduction in running time is still around 26.5% for the register architecture.*

5. Paul Biggar – „Compiling and Optimizing Scripting Languages"

   http://youtube.com/watch?v=kKySEUrP7LA

Slides: http://www.slideshare.net/LittleBIGRuby/compiling-and-optimizing-scripting-languages

6. Wikipedia's listing of JVM bytecodes:

   http://en.wikipedia.org/wiki/Java_bytecode_instruction_listings

7. Eric Brumer – Native Code Performance and Memory: The Elephant in the CPU http://channel9.msdn.com/Events/Build/2013/4-329

8. Bart De Smet [MVP Visual C#] - How C# Array Initializers Work

http://bartdesmet.net/blogs/bart/archive/2008/08/21/how-c-array-initializers-work.aspx

9. OSNews 2004 – Nine Language Performance Round-up: Benchmarking Math & File I/O

http://www.osnews.com/story/5602/Nine_Language_Performance_Round-up_Benchmarking_Math_File_I_O/page2/

10. Google I/O 2010 - A JIT Compiler for Android's Dalvik VM

@4:00 „Host interpreter is 2x faster" (than Java)

@6:20 „But this good enough for most applications, doesn't mean it is perfect. For some applications you do really, really feel the pain of interpretation. Applications in which you do a lot of computation. And that gets painful because you experience the slowdown of the interpreter, which is often on the order of five to ten times."