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

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

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(original 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

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(original 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 paper, 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

[Claim about originality and respecting intelectual property and copyright 3](#_Toc366349184)

[Declaration of acceptance 4](#_Toc366349185)

[Agreement about rights of ownership 5](#_Toc366349186)

[0. Introduction 8](#_Toc366349187)

[0.1. Abstract 8](#_Toc366349188)

[0.2. Introduction 8](#_Toc366349189)

[1. AOT description and evaluation 10](#_Toc366349190)

[1.1. Stack based Virtual Machines Overview 10](#_Toc366349191)

[1.1.1. Execution model 10](#_Toc366349192)

[1.1.2. Memory model 10](#_Toc366349193)

[1.1.3. Compilation model 10](#_Toc366349194)

[1.1.4. “Stack Based” versus “Register based” instruction set 10](#_Toc366349195)

[1.1.5. Specified instruction set 11](#_Toc366349196)

[1.2. Evaluation and implementation directions 12](#_Toc366349197)

[1.2.1. Evaluating the execution model 12](#_Toc366349198)

[1.2.2. Evaluating the memory model 12](#_Toc366349199)

[1.2.3 Evaluating the compilation model 12](#_Toc366349200)

[1.2.4 Evaluating the Stack Based VM model 12](#_Toc366349201)

[1.2.5. Type tracking because of using stack evaluation 13](#_Toc366349202)

[1.2.6. Optimization overview 13](#_Toc366349203)

[2. Code Refractor – implementation of an AOT virtual machines 15](#_Toc366349204)

[2.1. Small overview of CodeRefractor 15](#_Toc366349205)

[2.2. Intermediary representation of Code Refractor 15](#_Toc366349206)

[3. Code Refractor components 19](#_Toc366349207)

[3.2. CR OpenRuntime 19](#_Toc366349208)

[3.3. System.Math 20](#_Toc366349209)

[3.4. Platform invoke 21](#_Toc366349210)

[3.5. String merging 21](#_Toc366349211)

[4. Optimization and optimization steps in the CR 22](#_Toc366349212)

[4.1 Optimization short overview 22](#_Toc366349213)

[4.1.1. Real life optimization strategies 22](#_Toc366349214)

[4.2.Local optimizations (block based optimizations) 22](#_Toc366349215)

[4.2.1. Constant folding optimizations 23](#_Toc366349216)

[4.2.2. Assignment of identifier used next line 23](#_Toc366349217)

[4.2.3. Assignment of expression 24](#_Toc366349218)

[4.2.4. Evaluate constant expressions 24](#_Toc366349219)

[4.2.5. Evaluate partial constant expressions 24](#_Toc366349220)

[4.2.6. Evaluate conditional ifs 24](#_Toc366349221)

[4.3. Global optimizations (optimizations that work over more than a basic block) 24](#_Toc366349222)

[4.3.1. Not used variables are deleted 25](#_Toc366349223)

[4.3.2. Variables that are assigned but not used can be deleted and the entire expression 25](#_Toc366349224)

[4.3.3. Remove unused labels 25](#_Toc366349225)

[4.3.3. Merge consecutive labels 25](#_Toc366349226)

[4.3.4. Any goto to next line can be removed 25](#_Toc366349227)

[4.4 Dataflow analysis 26](#_Toc366349228)

[4.4.1 Constant dataflow propagation 26](#_Toc366349229)

[4.4.2. Reachability lines analysis 26](#_Toc366349230)

[4.5. Some inter-procedural optimizations 27](#_Toc366349231)

[4.5.1. Purity annotation and evaluation 27](#_Toc366349232)

[4.5.2. Inlining 27](#_Toc366349233)

[4.5.3. Class Hierarchy Analysis 27](#_Toc366349234)

[5. Benchmarks 30](#_Toc366349235)

[5.1. Overview 30](#_Toc366349236)

[5.2. Bad benchmarks: OS News 2004 article 30](#_Toc366349237)

[5.3. Better benchmarks: Nbody test 31](#_Toc366349238)

[References 32](#_Toc366349239)

# 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.

A virtual machine like Java (JVM = Java Virtual Machine) or .Net executes “on demand” an intermediate form (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]](#footnote-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]](#footnote-2) and GCJ[[3]](#footnote-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]](#footnote-4) has two Ahead-of-time compilers, one is their made LinearIR[[5]](#footnote-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. AOT 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.

## 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;

- right after that will read the bytecode as operations;

- depending of the design, will start executing that method 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.

### 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 reprezentation (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 mostly means, that the Register Allocation step is not optimal (for example in HotSpt Client version is using Linear Scan Allocator)

### 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.

### 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]](#footnote-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 (so 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, so many operations are compact. So 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.

## 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.

So the 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” instruction includes an (implicit) constructor call (if this constructor is with no parameters), and this code is also scanned.

### 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. 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).

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.

### 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 GCC and major LLVM platforms, 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.

### 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:

Push

Pop

Top / Peek

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.

The another 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.

### 1.2.5. Type tracking because of using 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.

So 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.

So 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 inferences (albeit is somewhat not so 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 to 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

- the compiler optimizations are performed for every method

- the intermediate representation is written into C++

## 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:

1. variable parts which are themselves split into:
   1. local variables
   2. virtual registers
   3. arguments (method's parameters)
2. local operations which will use either the local variables or will call another methods. These local operations, are themselves split into:
   1. assignments
   2. operators:
      1. binary operators
      2. unary operators
      3. conditional operators
   3. creation of arrays and objects
   4. accessing array elements or fields
   5. call of a separate methods
   6. branch operations
   7. 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 LocalVariable : IdentifierValue  {  public VariableKind Kind; (…)  } | 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 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

- removing calls, and replacing it with the equivalent code will save at least the performance of the final code

- removing pointer assignments, this is crucial as an assignment requires more 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

- 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 wrote 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 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.

## 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]](#footnote-7):

|  |
| --- |
| Code that generated LdToken CIL instruction |
| var ints = new [] { 1, 2, 3 }; |

Which 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.

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 explicitl0y.

# 4. 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 moth

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.

## 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:

### 4.2.2. Assignment of identifier used next line

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

### 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.

### 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

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

### 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 |

### 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.

### 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.

### 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.

### 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.

## 4.4 Dataflow analysis

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:

* they do analyze a hypothesis for every possible branch of the program
* if the optimization hypothesis holds and is discovered a case when it is happened
* a replacement is made

The power of these optimizations is that can define properties that are not always safe to make for the entire procedure flow.

### 4.4.1 Constant dataflow propagation

|  |  |
| --- | --- |
| Code before optimization | Code after optimization |
| (...)  var1 = 3;  Goto 20;  var1 = 2;  Label\_20:  Local1=var1; | (...)  var1 = 3;  Goto Label\_20;  var1 = 2;  Label\_20:  **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.

### 4.4.2. Reachability lines analysis

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

|  |  |
| --- | --- |
| Code before optimization | Code after optimization |
| (...)  var1 = 3;  Goto 20;  var1 = 2;  Label\_20:  Local1=var1; | (...)  var1 = 3;  Goto Label\_20;  **//delete code in between Goto Label\_20 and Label\_20**  Label\_20:  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.

So 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. Purity annotation and evaluation

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 does 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)

- does not change/read static or not static fields

- does just simple math and assignments and/or jumps

are pure.

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. 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.

These cases happen in cases even where user is not aware of them, like auto-properties.

### 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

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)

# 5. Benchmarks

## 5.1. 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.

## 5.2. 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]](#footnote-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.” (…) |

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.

## 5.3. 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(n2) 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.

# 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>

1. Wikipedia’s listing of JVM bytecodes: <http://en.wikipedia.org/wiki/Java_bytecode_instruction_listings>
2. Eric Brumer – Native Code Performance and Memory: The Elephant in the CPU <http://channel9.msdn.com/Events/Build/2013/4-329>
3. 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>

1. 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/>

1. 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.”

1. <http://www.excelsior-usa.com/jet.html> [↑](#footnote-ref-1)
2. <http://www.robovm.org/> [↑](#footnote-ref-2)
3. <http://gcc.gnu.org/java/> [↑](#footnote-ref-3)
4. <http://www.mono-project.com/Main_Page> [↑](#footnote-ref-4)
5. <http://mono-project.com/Linear_IR> [↑](#footnote-ref-5)
6. Look to references 1 and 6 [↑](#footnote-ref-6)
7. Look to reference 8 [↑](#footnote-ref-7)
8. Look to reference 9 [↑](#footnote-ref-8)