Code Refractor

Optimizing Stack-based Vms

Student
Khlud Ciprian
Coordinator
PhD Ferucio Laurențiu Țiplea

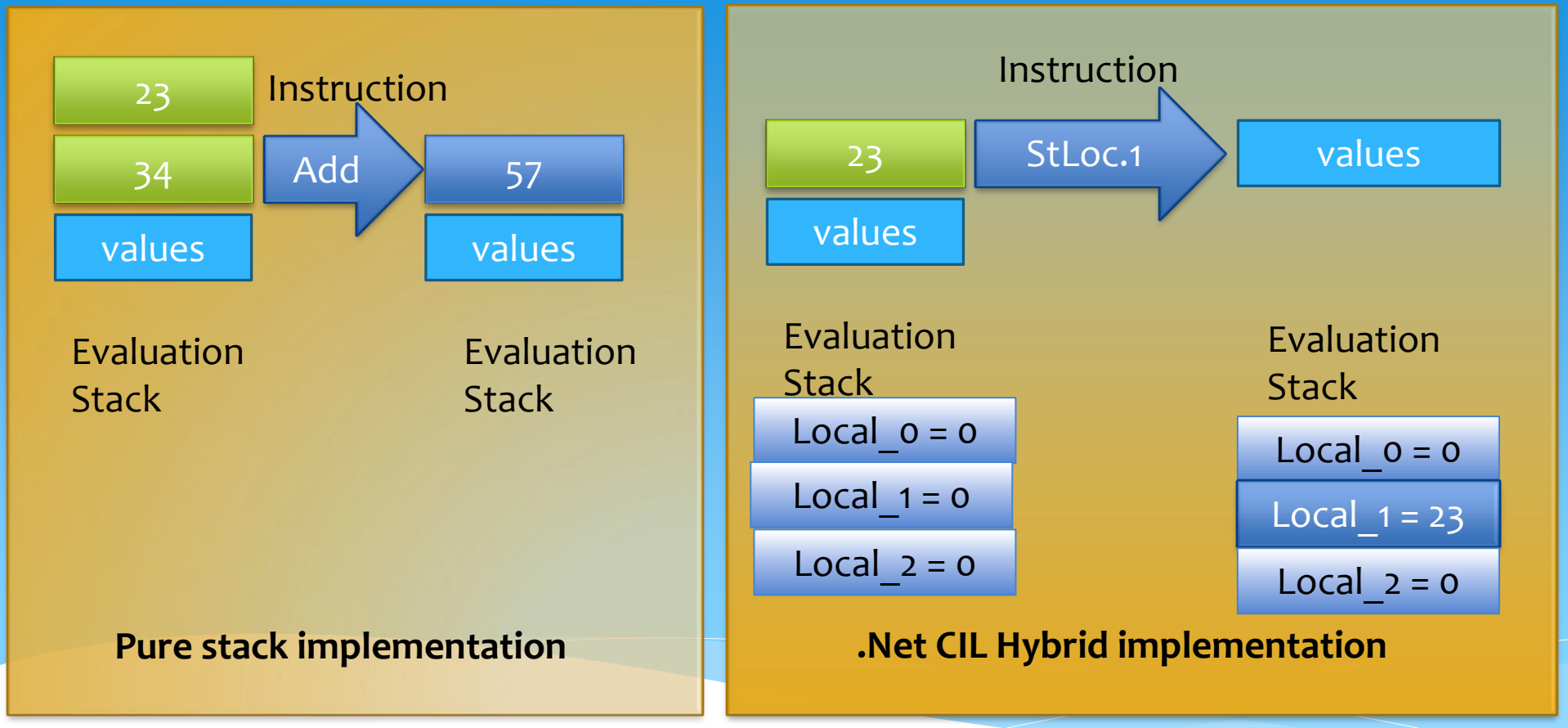
Code Refractor - Content

1. - Stack-based Vms
2. - Code Refractor architecture
3. - FrontEnd
4. - Optimization overview
5. - Local Optimizations
6. - Use-Def optimizations
7. - DataFlow optimizations
8 - Purity and Escape analysis

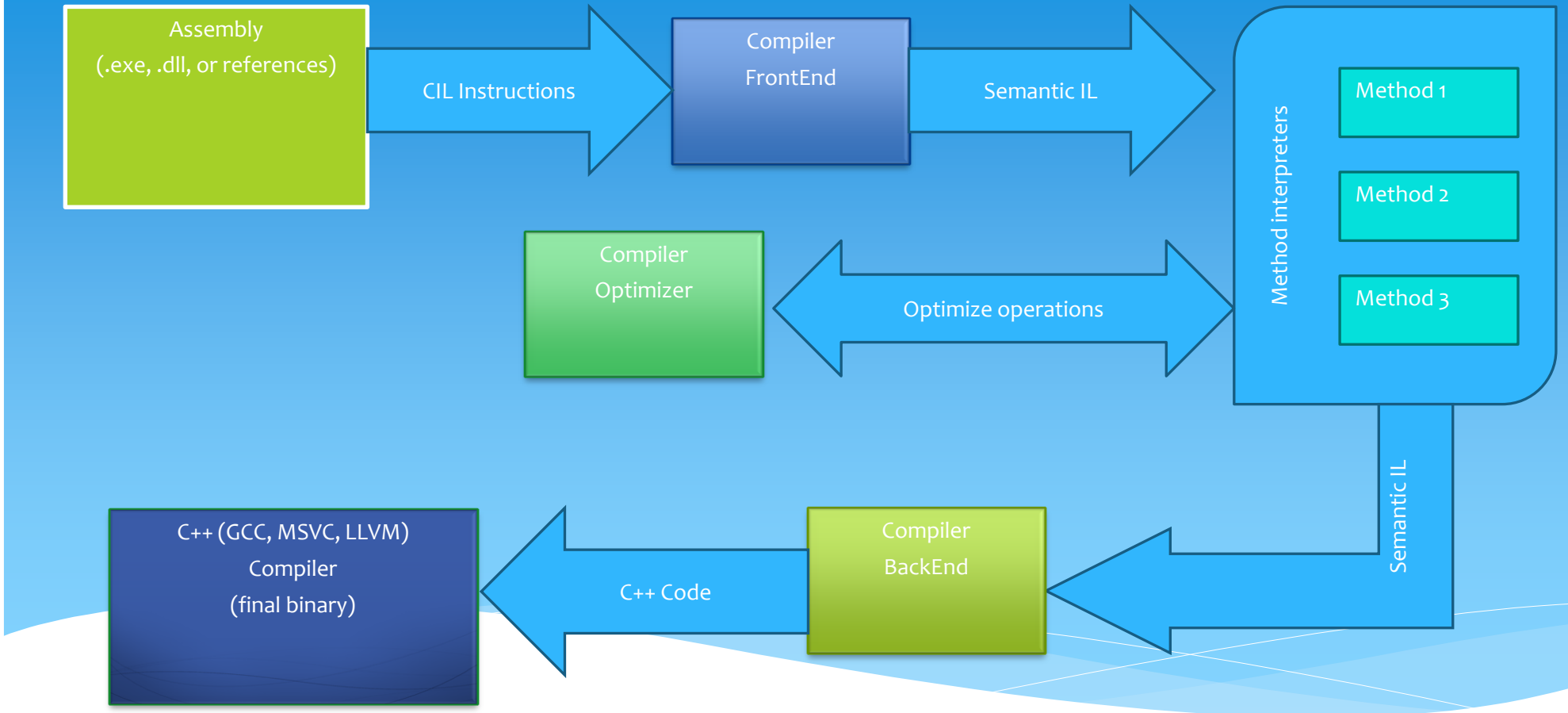
Conclusions
1. Performance vs .Net
2. Questions !?

Note: put questions at any moment
if things are not clear, is better to interrupt.

1. - Stack-based Vms



2. - Code Refractor architecture



3. - FrontEnd

```
IL_0000: nop
IL_0001: ldc.i4 5000000
IL_0006: stloc.0
IL_0007: newobj instance void SimpleAdditions.NBodySystem::.ctor()
IL_000c: stloc.1
IL_000d: ldloc.1
IL_000e: callvirt instance float64 SimpleAdditions.NBodySystem::Energ
IL_0013: call void [mscorlib]System.Console::WriteLine(float64)
IL_0018: nop
IL_0019: ldc.i4.0
IL_001a: stloc.2
IL_001b: br.s IL_0031
// loop start (head: IL_0031)
    IL_001d: ldloc.1
    IL_001e: ldc.r8 0.01
    IL_0027: callvirt instance void SimpleAdditions.NBodySystem::Adva
    IL_002c: nop
    IL_002d: ldloc.2
    IL_002e: ldc.i4.1
    IL_002f: add
    IL_0030: stloc.2

    IL_0031: ldloc.2
    IL_0032: ldloc.0
    IL_0033: clt
    IL_0035: stloc.3
    IL_0036: ldloc.3
    IL_0037: brtrue.s IL_001d
// end loop

IL_0039: ldloc.1
IL_003a: callvirt instance float64 SimpleAdditions.NBodySystem::Energ
IL_003f: call void [mscorlib]System.Console::WriteLine(float64)
IL_0044: nop
IL_0045: ret
```

Disassemble IL
Rewrite as IR

intermediateCode.LocalOpe	Count = 31	System
[0]	{Assignment: vreg_1 = 5000000}	CodeR
[1]	{Assignment: local_0 = vreg_1:Int32}	CodeR
[2]	{NewObject: vreg_2 = unknown}	CodeR
[3]	{Call: Call void = .ctor(vreg_2:NBodySystem);}	CodeR
[4]	{Assignment: vreg_3 = vreg_2:NBodySystem}	CodeR
[5]	{Assignment: local_1 = vreg_3:NBodySystem}	CodeR
[6]	{Assignment: vreg_4 = local_1:NBodySystem}	CodeR
[7]	{Call: Call vreg_5 = Energy(vreg_4:NBodySystem);}	CodeR
[8]	{Call: Call void = WriteLine(vreg_5:Double);}	CodeR
[9]	{Assignment: vreg_6 = 0}	CodeR
[10]	{Assignment: local_2 = vreg_6:Int32}	CodeR
[11]	{AlwaysBranch: 49}	CodeR
[12]	{Label: 29}	CodeR
[13]	{Assignment: vreg_7 = local_1:NBodySystem}	CodeR
[14]	{Assignment: vreg_8 = 0.01}	CodeR
[15]	{Call: Call void = Advance(vreg_7:NBodySystem, vreg_8:Double);}	CodeR
[16]	{Assignment: vreg_9 = local_2:Int32}	CodeR
[17]	{Assignment: vreg_10 = 1}	CodeR
[18]	{BinaryOperator: vreg_11 = vreg_9 add vreg_10}	CodeR
[19]	{Assignment: local_2 = vreg_11:Int32}	CodeR
[20]	{Label: 49}	CodeR
[21]	{Assignment: vreg_12 = local_2:Int32}	CodeR
[22]	{Assignment: vreg_13 = local_0:Int32}	CodeR
[23]	{BinaryOperator: vreg_14 = vreg_12 clt vreg_13}	CodeR
[24]	{Assignment: local_3 = vreg_14:Int32}	CodeR
[25]	{Assignment: vreg_15 = local_3:Int32}	CodeR
[26]	{BranchOperator: Branch operator vreg_15:Int32 brtrue ? jump label_2	CodeR
[27]	{Assignment: vreg_16 = local_1:NBodySystem}	CodeR
[28]	{Call: Call vreg_17 = Energy(vreg_16:NBodySystem);}	CodeR
[29]	{Call: Call void = WriteLine(vreg_17:Double);}	CodeR
[30]	{Return: }	CodeR

Converts CIL code into intermediate representation, buids call graph

4. - Optimization overview

One instruction

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

Block based

Code before optimizations	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

Global optimizations

Code before optimizations	Code after optimization
Over whole body of function	(various)

Program wide optimizations

Code before optimizations	Code after optimization
Across functions	(various)

5. – Block based optimizations

Assignment of identifier used next line
Assignment of expression
Evaluate constant expressions
Evaluate partial constant expressions
Evaluate conditional ifs
Dead store eliminations
Common Subexpression Elimination

6 – USE-DEF optimizations

- Not used variables are deleted
- Label optimizations
  - remove unused,
  - merge consecutive,
  - remove goto to labels on the next line
- Dead store eliminations (over all function)
- One assignment with constants propagated with all funciton

7. – Global optimizations

DFA Optimizations

Methodology:

- Make some startup assumptions
- Visit all flow of function to see if they hold
- Go over all branches until assumptions stabilize

Implementations

- Reachability lines (Dead Code Elimination)
- Constant DFA propagation

8 - Purity and Escape analysis

Purity Analysis
Checks if a function does not have any side effects and depend only from parameters

Resulting optimizations
- Pure functions calls with constants are evaluated as constants
- More aggressive CSE
- LICM works by moving functions too

Escape Analysis
Checks if an instance does have the reference counting usages of other object changing in an known way.

Resulting optimizations
- Allocations can be done on stack
- Smart pointer usages can be made as raw pointers

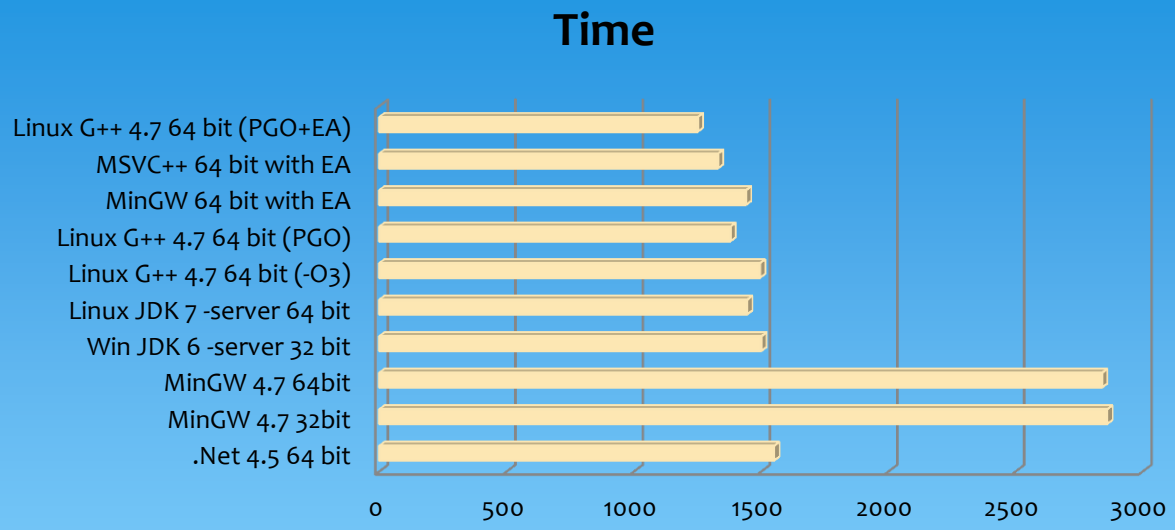
Conclusions 1. Performance vs .Net

Bad benchmarks:
OS News 2004 benchmark

Good Benchmarks
NBody Benchmark

- In math computations
- in memory usage

CR runs good in both cases



	.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

Questions?

```
IL_004c: clt
IL_004e: stloc.s CS$4$0000
IL_0050: ldloc.s CS$4$0000
IL_0052: brtrue.s IL_0036
```

THANK YOU

Optimizing Stack-based Vms
Khlud Ciprian

Coordinator
PhD Ferucio Laurențiu Țiplea