An Optimizing Just-in-Time Compiler for Rotor

João H. Trindade
jtrindade.net@iol.pt
Instituto Superior de Engenharia de Lisboa

José C. Silva
jsilva.net@iol.pt

Porto
June 2005
Rotor – Shared Source CLI

- implementation of ECMA-335 (CLI) and ECMA-334 (C#) standards
- source code available (shared source)
- MS Windows, FreeBSD; IA-32, PowerPC
- code base derived from Microsoft’s commercial CLI implementation (CLR)
  - different garbage collector
  - different just-in-time compiler
FJIT – Rotor’s JIT Compiler

- Translates CIL to native code in runtime
- Invoked the first time each method is called
- The same compiler is used for IA-32 and PowerPC
  - specific code emission layer for each processor family
Invoking the JIT Compiler
Invoking the JIT Compiler

Object

MethodTable *

MethodTable

Instance data

JIT compiler invocation

Prestubs

JIT

Compiled method

jump to the compiled method
JIT Compiler - Time consumed

- For each method:
  \[ T = C + n_0 \times E(n_1, n_2, ..., n_L) \]

  - \( T \): total consumed time
  - \( C \): compilation time
  - \( E \): execution time
  - \( L \): number of loops
  - \( n_0 \): number of executions
  - \( n_x, x > 0 \): average number of iterations for loop \( x \)

- For **small** values of \( n \) → reduce \( C \)
- For **large** values of \( n \) → reduce \( E \)
FJIT – Rotor’s JIT Compiler

- FJIT is a simple single-pass compiler
- In loop, it reads one CIL instruction and emits the corresponding native code.
- No optimizations are performed
- Fast and portable, but resulting code is slow
- $C$ has been minimized, but $E$ is large
- High penalty for methods with a large $n$
Supporting Multiple JITters

using JHTJCS.System;

public class Test {
    [UserPreferedJitter(UserDefinedJITNumber.UserDefJIT3)]
    public static int Main() {
        int a = 1;
        int b = 2;
        return a + b;
    }
}

2005-06-22 An Optimizing JIT Compiler for Rotor
Compiler Structure

- **Front-end**: Translates CIL to the compiler’s internal intermediate representation.

- **Optimizer**: Modifies the original program, without changing its behaviour, in order to improve the efficiency of generated code.

- **Code Generator**: Translates the optimized code from the compiler’s internal intermediate representation to executable native code.
Front-end

- divides the program in basic blocks and determines the connections between them
- detects existing loops (to focus expensive optimizations)
- collects data dependence between instructions (not explicit in CIL)
Control Flow Analysis

IL_0000: ldarg.2
IL_0001: stloc.0
IL_0002: ldarg.1
IL_0003: brfalse.s IL_0009
IL_0005: ldarg.1
IL_0006: ldarg.2
IL_0007: or
IL_0008: stloc.0
IL_0009: ldloc.0
IL_000a: ldarg.2
IL_000b: xor
IL_000c: stloc.0
IL_000d: ldloc.0
IL_000e: brtrue.s IL_0002
IL_0010: ret
END
Loops

- Set of basic blocks $C$, including an *entry basic block* $E$, such that:
  - there is a path from any node in $C$ to $E$;
  - there is a path from $E$ to any node in $C$;
  - there is no path from any node outside $C$ to a node of $C$ except through $E$.

- In a control flow graph, with an initial node $r$, node $d$ dominates node $n$ iff any path between $r$ and $n$ must go through $d$. By definition, a node dominates itself.

- **Loop Detection:**
  - *backedges* $n \rightarrow c$, with $c$ dominating $n$ (c dominates a predecessor)
Dominators

- Cooper, Harvey e Kennedy, “A Simple Fast Dominance Algorithm” (2001):
  - graph nodes numbered in postorder;
  - Immediate dominators represented in a dominance vector.
  - Efficient intersection of immediate dominators’ sets.
Data Dependence Analysis I

- Translates CIL to intermediate representation
- Builds a data dependence graph

IL_0000: ldloc.1
IL_0001: ldloc.2
IL_0002: add
IL_0003: ldloc.2
IL_0004: mul

- Optimizations are performed simultaneously
Data Dependence Analysis II

- IL_0000: ldloc.0
- IL_0001: ldloc.1
- IL_0002: add
- IL_0003: stloc.0
- IL_0004: ldloc.0
- IL_0005: ldloc.2
- IL_0006: sub
- IL_0007: stloc.0

2005-06-22
This simple approach allows:

- Redundant load elimination (after loads and/or stores over the same variable/argument);
- Elimination of intermediate stores (*dead stores*);
- Copy and constant propagation;
- Null reference check elimination.
Optimizations II

- Each instruction’s result has an associated usage counter:
  - Single usage loads are identified (*embeddable load / instruction folding*)
  - Stores of values not used within the basic block are identified (*embeddable store / copy elimination*)
  - Elimination of instructions with an usage counter value of zero (*dead code elimination*)
Optimizations III

- Arithmetic operations with constant operand(s):
  - Constant Folding
  - Algebraic Simplification
  - Invalid operation detection (throws an exception)
Code Generation I

- Register Allocation
  - assigns registers to the edges of the data dependence graph

- IA-32 code emission
Code Generation II
Code Generation III

1. IA-32 instruction types:
   - register - register;
   - register - immediate;
   - register - memory;
   - memory - register;
   - memory - immediate.

2. Register allocation and propagation.

3. Additional data moves:
   - embeddable loads;
   - retrieve spilled values;
   - register moves.
public class X {
    private int[] v = new int[8];
    public int m(int a, int b) {
        return v[a * b + b];
    }
}

FJIT

mov eax, [ebp+0x8]
push eax
mov eax, 0x4
push eax
mov eax, LDFLD_REF
call eax
add esp, 0x8
push eax
mov eax, [ebp+0xc]
pop ecx
imul ecx
push eax
mov eax, [ebp+0x10]
pop ecx
add eax, ecx
push eax
mov eax, LDELEM_I4
call eax
add esp, 0x8
mov [ebp-0x8], eax
mov [ebp-0x10], eax

public class X {
    private int[] v = new int[8];
    public int m(int a, int b) {
        return v[a * b + b];
    }
}

AJIT

mov eax, [ebp+0x8]
test eax, eax
jne x1
mov eax, ThrowNullRef
call eax

x1: mov ecx, [eax+0x4]
mov eax, [ebp+0x10]
imul edx, eax
add edx, eax
test ecx, eax
jne x2
mov ecx, ThrowNullRef
call ecx

x2: cmp [ecx+0x4], edx
jnbe x3
mov ecx, ThrowOutOfBounds
call ecx

x3: mov eax, [ecx+edx*4+0x8]
xe:
Tests

- In evaluation:
  - compilation time
  - execution time of compiled methods
  - compilation time distribution by phases

- Benchmark
  - RC4
  - Quicksort
Results I

Total compilation time

QuickSort

RC4

AJIT

FJIT

[ms]
0,0
0,1
0,2
0,3
0,4
0,5

2005-06-22
An Optimizing JIT Compiler for Rotor
Results II

Execution time

T(*) / T(CLR)

CLR
AJIT
FJIT

2005-06-22
An Optimizing JIT Compiler for Rotor
Conclusions I

- Ability to select JITters.
- Implementation of the front-end and the code generator, including optimizations.
- Control flow graph construction in 1 pass
- Dominators
- Loop detection
Conclusions II

- **Data Dependence Graph construction (1 pass) including:**
  - constant propagation; copy propagation; constant folding; algebraic simplifications; dead code elimination; redundant load and store elimination; redundant null reference check elimination.

- **Code Generation (1 pass) including:**
  - register allocation; IA-32 native code emission taking advantage of the available addressing modes.
To Do

- Fully support CIL:
  - 64 bit instructions
  - floating-point instructions
  - exceptions
  - value types

- Full integration with Rotor
  - Garbage collector (write barrier, polling, tracing roots in the stack)
Future Work

- Implement global analysis
- **Additional optimizations** (method inlining, loop optimizations, devirtualization, redundant check elimination)
- **Improve register allocation** (Linear Scan Register Allocation)
- Develop a dynamic multicomilation system
Thank you.

jtrindade.net@iol.pt
jsilva.net@iol.pt