Programming Language Laboratory

Code Optimization for Trace Compilers

Rodrigo Sol

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INCIPIT VITA NOVA
7 DE SETEMBRO DE 1927

Importance of Script Languages

30%

Position Nov 2009	Position Nov 2008	Delta in Position	Programming Language	Ratings Nov 2009	De[illegible] Nov 2008	
1	1	=	Java	18.373%	-1.93%	A
2	2	=	C	17.315%	+2.04%	A
3	5	↑↑	PHP	10.176%	+1.24%	A
4	3	↓	C++	10.002%	-0.36%	A
5	4	↓	(Visual) Basic	8.171%	-1.10%	A
6	7	↑	C#	5.346%	+1.32%	A
7	6	↓	Python	4.672%	-0.47%	A
8	9	↑	Perl	3.490%	-0.39%	A
9	10	↑	JavaScript	2.916%	-0.01%	A
10	11	↑	Ruby	2.404%	-0.47%	A
11	8	↓↓↓	Delphi	2.127%	-1.88%	A

Efficiency Challenges

- To produce machine code out of scripting languages is difficult:
  - No concrete type information is available
  - Dynamic code inclusion
  - Meta-programming
- Just-in-time compilers are a feasible and useful alternative

Traditional Just-in-time

- Code is interpreted.
- The methods that are more often called are completely compiled to machine code.
- Folk knowledge: in general about 20% of the code will account for 80% of the execution time.
  - But the JIT compiler compiles the whole method…

Trace Compilation

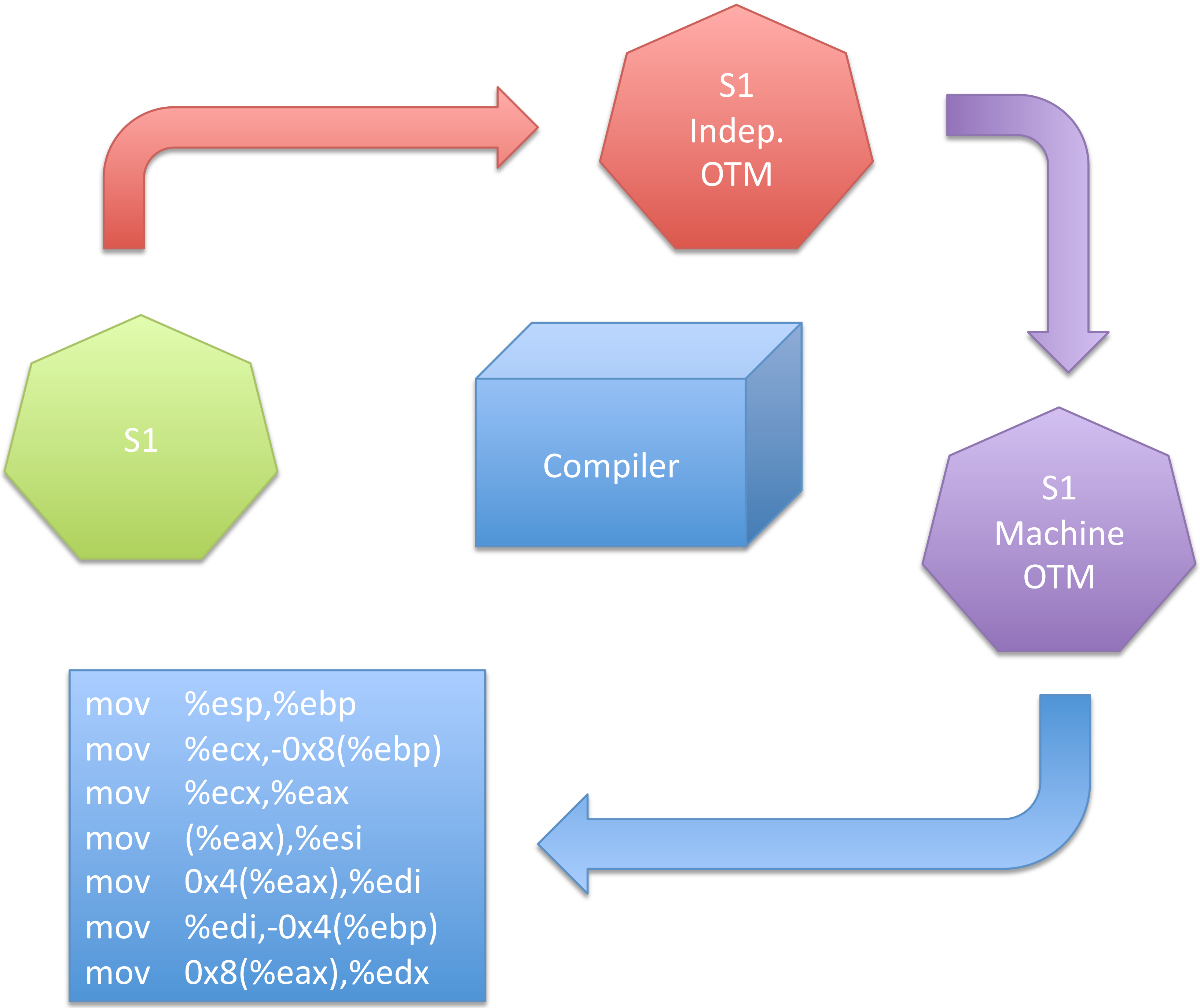
- A more granular JIT
- Only the most executed parts of the code are compiled to machine code.
- New approach: 2007
- Used in the Mozilla Firefox.
  - Now also used in Lua JIT
  - Many proposals for other languages.

What is a program trace?

- A sequence of program instructions with no branches.
  - May span many basic blocks
  - May span multiple functions
- A trace has only one entry point, but may have many exit points.

How does a trace compile work?

```
Start
state = param 0 ecx
sp = ld state[0]
rp = ld state[4]
cx = ld state[8]
0001eos = ld state[12]
eor = ld state[16]
ld1 = ld cx[0]
sti sp[0] = globalObj
ld2 = ld cx[152]
…
 add2 = add ld5, 8
sti state[732] = add2
ld9 = ld cx[152]
ld10 = ld ld9[60]
ld11 = ld ld10[0]
sti sp[0] = add2
sti sp[8] = 30
sti state[732] = add2
loop
```


S1
S1 Indep. OTM
Compiler
S1 Machine OTM
mov %esp,%ebp
mov %ecx,-0x8(%ebp)
mov %ecx,%eax
mov (%eax),%esi
mov 0x4(%eax),%edi
mov %edi,-0x4(%ebp)
mov 0x8(%eax),%edx

Challenges

- Is it possible to produce fast native code without spending much time doing code optimization?
- Can we use user input as meta-data for optimizations?
- What standard code optimizations can we use in trace compilation?

What We Want to Do

- Create a back-end for testing new optimizations on top of TraceMonkey
- Implement two optimizations:
  - Loop unrolling
  - Overflow test elimination

Loop Unrolling

- Many program loops contain only a few instructions.
- Goals
  - Decrease the number of control hazards in the total run of the loop
  - Fill the unavoidable stall spots with independent instructions

Loop Unrolling Optimization

Original Code

```
for (i=0;i<x;i++){
  sum+=1
}
```

x=6

```
If(x >= 4){
   for (i=0;i<x;i+=4){
        sum+=1
        sum+=1
        sum+=1
        sum+=1
        sum+=1
        sum+=1
                    +){
   }
}else{
 for (i=0;i<x;i++){
      sum+=1    }
}
```

What We Want to Do

- Create a back-end for testing new optimizations on top of TraceMonkey
- Implement two optimizations:
  - Loop unrolling
  - Elimination of overflow tests

Elimination of overflow tests

- JavaScript has no integer type
  - 64-bit IEEE- 754 floating-pointer numbers
- Many JavaScript instructions use only integer data
  - Array accesses and bitwise operators
- An optimization is to convert doubles to integers whenever possible
- Can overflow tests be avoided?

```
state = param 0 ecx
sp = ld state[0]
cx = ld state[8]
ld1 = ld cx[0]
eq1 = eq ld1, 0
xf1: xf eq1 -> pc=0x30d9e7 imacpc=0x0 sp+0 rp+0
sti sp[0] = globalObj
ld5 = ld state[732]
sti sp[8] = ld5
sti sp[16] = 7
add1 = add ld5, 7
ov1 = ov add1
xt1: xt ov1 -> pc=0x30d9f1 imacpc=0x0 sp+24 rp+0
sti state[732] = add1
add2 = add ld5, 8
ov2 = ov add2
xt2: xt ov2 -> pc=0x30d9f6 imacpc=0x0 sp+0 rp+0
sti state[732] = add2
sti sp[0] = add2
sti sp[8] = 30
lt1 = lt add2, 30
xf2: xf lt1 -> pc=0x30d9ff imacpc=0x0 sp+16 rp+0
```

```
state = param 0 ecx
sp = ld state[0]
cx = ld state[8]
ld1 = ld cx[0]
eq1 = eq ld1, 0
sti sp[0] = globalObj
ld5 = ld state[732]
sti sp[8] = ld5
sti sp[16] = 7
add1 = add ld5, 7
sti state[732] = add1
add2 = add ld5, 8
sti state[732] = add2
sti sp[0] = add2
sti sp[8] = 30
lt1 = lt add2, 30
```

```
state = param 0 ecx
sp = ld state[0]
cx = ld state[8]
ld1 = ld cx[0]
eq1 = eq ld1, 0
xf1: xf eq1 -> pc=0x30d9e7 imacpc=0x0 sp+0 rp+0
sti sp[0] = globalObj
ld5 = ld state[732]
sti sp[8] = ld5
sti sp[16] = 7
add1 = add ld5, 7
xt1: xt ov1 -> pc=0x30d9f1 imacpc=0x0 sp+24 rp+0
sti state[732] = add1
add2 = add ld5, 8
xt2: xt ov2 -> pc=0x30d9f6 imacpc=0x0 sp+0 rp+0
sti state[732] = add2
sti sp[0] = add2
sti sp[8] = 30
lt1 = lt add2, 30
xf2: xf lt1 -> pc=0x30d9ff imacpc=0x0 sp+16 rp+0
```

Programming Language Laboratory

Code Optimization for Trace Compilers

Rodrigo Sol