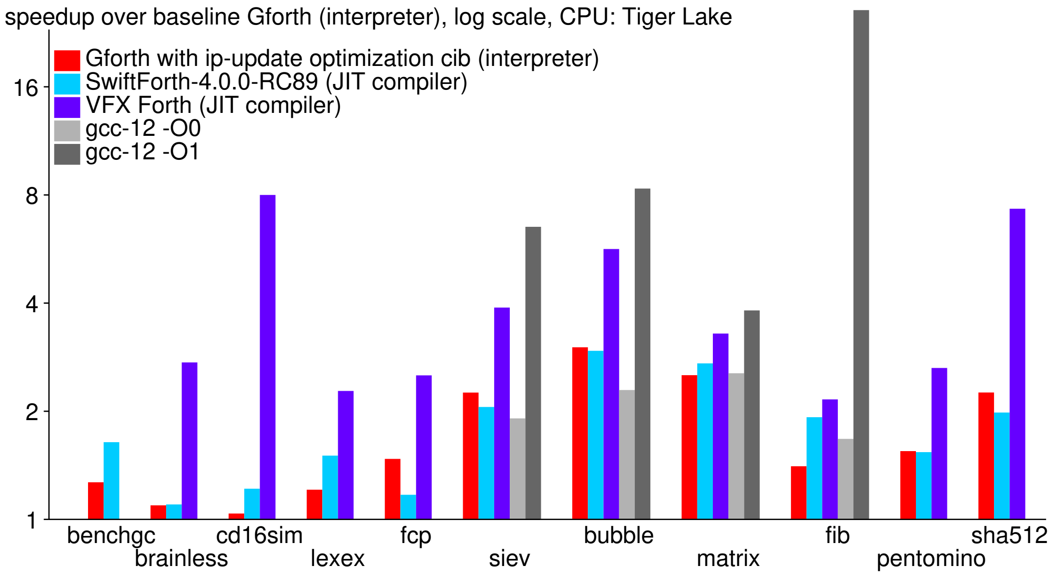
The Performance Effects of Virtual-Machine Instruction Pointer Updates 2024 update

M. Anton Ertl, TU Wien
Bernd Paysan, net2o

Overview

- Background: Virtual-machine interpreter with code copying
- Every VM instruction increments the VM instruction pointer (IP)
- Question: How relevant are IP updates for performance?
- Answer: on some programs critical latency path
- Method: optimize most IP updates away

Is interpreter performance relevant? What about JITs?



Running example: inner loop of siev

Forth Source:

```
do
  0 i c! dup +loop
```

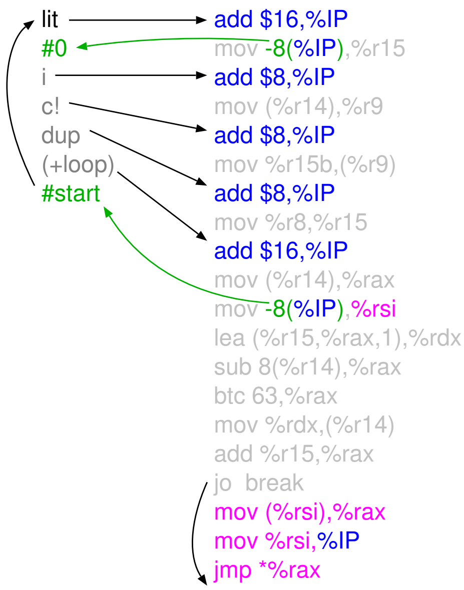
C Source:

```
for(p = ... ; p <= ... ; p += prime)
  *p = 0;
```

Virtual-Machine code (Gforth):

```
        (do)
start:  lit
        #0
        i
        c!
        dup
        (+loop)
        #start
```


Baseline: Code-copying interpreter with static stack caching



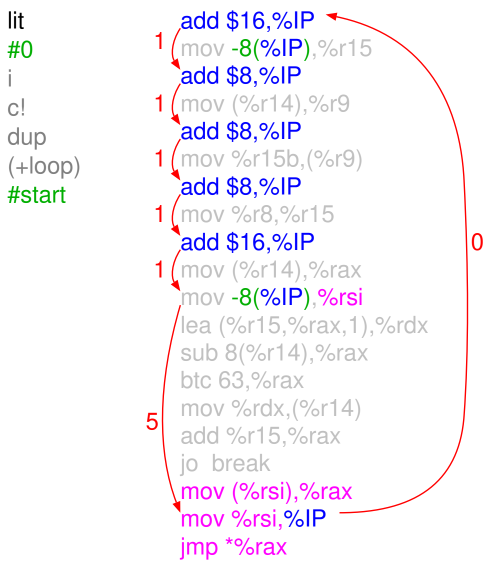
It's a JIT compiler!

+ Copies native code

It's an interpreter!

+ Portable
  gcc generates code snippets
+ Fallback option
  to threaded-code interpreter
  without code copying
+ VM code is still needed
  for immediate values
  for control flow
+ ⇒ VM instruction pointer needed

Instruction pointer updates limit execution rate



c: combine instruction pointer updates

```
lit
#0
i
c!
dup
(+loop)
#start
```

```
add $16,%IP
mov -8(%IP),%r15
add $8,%IP
mov (%r14),%r9
add $8,%IP
mov %r15b,(%r9)
add $8,%IP
mov %r8,%r15
add $16,%IP
mov (%r14),%rax
mov -8(%IP),%rsi
lea (%r15,%rax,1),%rdx
sub 8(%r14),%rax
btc 63,%rax
mov %rdx,(%r14)
add %r15,%rax
jo  break
mov (%rsi),%rax
mov %rsi,%IP
jmp *%rax
```

```
add $16,%IP
mov -8(%IP),%r15

mov (%r14),%r9

mov %r15b,(%r9)

mov %r8,%r15
add $40,%IP
mov (%r14),%rax
mov -8(%IP),%rsi
lea (%r15,%rax,1),%rdx
sub 8(%r14),%rax
btc 63,%rax
mov %rdx,(%r14)
add %r15,%rax
jo  break
mov (%rsi),%rax
mov %rsi,%IP
jmp *%rax
```


ci: ... and optimize immediate VM instructions

```
lit
#0
i
c!
dup
(+loop)
#start
```

```
add $16,%IP
mov -8(%IP),%r15

mov (%r14),%r9

mov %r15b,(%r9)

mov %r8,%r15
add $40,%IP
mov (%r14),%rax
mov -8(%IP),%rsi
lea (%r15,%rax,1),%rdx
sub 8(%r14),%rax
btc 63,%rax
mov %rdx,(%r14)
add %r15,%rax
jo  break
mov (%rsi),%rax
mov %rsi,%IP
jmp *%rax
```

```
mov 8(%IP),%r15

mov (%r14),%r9

mov %r15b,(%r9)

mov %r8,%r15
add $56,%IP
mov (%r14),%rax
mov -8(%IP),%rsi
lea (%r15,%rax,1),%rdx
sub 8(%r14),%rax
btc 63,%rax
mov %rdx,(%r14)
add %r15,%rax
jo  break
mov (%rsi),%rax
mov %rsi,%IP
jmp *%rax
```


cib: ... and optimize VM branch instructions

```
lit
#0
i
c!
dup
(+loop)
#start
```

```
mov 8(%IP),%r15

mov (%r14),%r9

mov %r15b,(%r9)

mov %r8,%r15
add $56,%IP
mov (%r14),%rax
mov -8(%IP),%rsi
lea (%r15,%rax,1),%rdx
sub 8(%r14),%rax
btc 63,%rax
mov %rdx,(%r14)
add %r15,%rax
jo  break
mov (%rsi),%rax
mov %rsi,%IP
jmp *%rax
```

```
mov 8(%IP),%r15

mov (%r14),%r9

mov %r15b,(%r9)

mov %r8,%r15

mov   (%r14),%rax

lea   (%r15,%rax,1),%rdx
sub   8(%r14),%rax
btc   $63,%rax
mov   %rdx,(%r14)
add   %r15,%rax
jo    break
mov   (%IP),%rax

jmp   *%rax
```

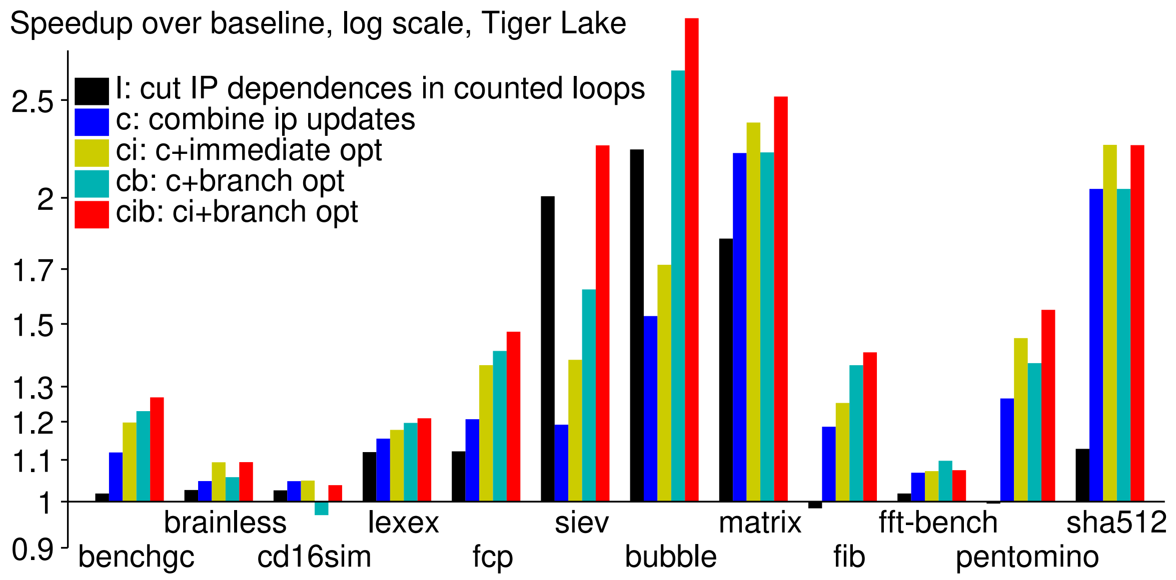
I: break loop dependencies

```
lit
#0
i
c!
dup
(+loop)
#start
```

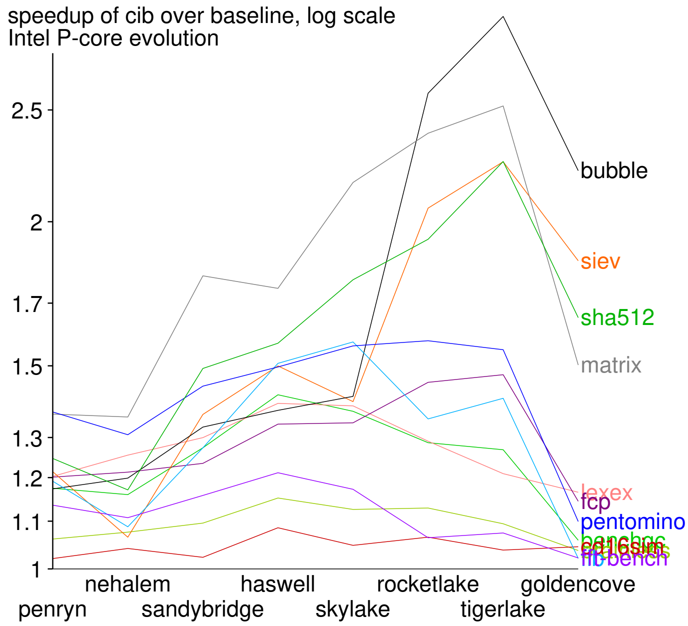
```
add $16,%IP
mov -8(%IP),%r15
add $8,%IP
mov (%RP),%r9
add $8,%IP
mov %r15b,(%r9)
add $8,%IP
mov %r8,%r15
add $16,%IP
mov (%RP),%rax
mov -8(%IP),%rsi
lea (%r15,%rax,1),%rdx
sub 8(%RP),%rax
btc 63,%rax
mov %rdx,(%RP)
add %r15,%rax
jo  break
mov (%rsi),%rax
mov %rsi,%IP
jmp *%rax
```

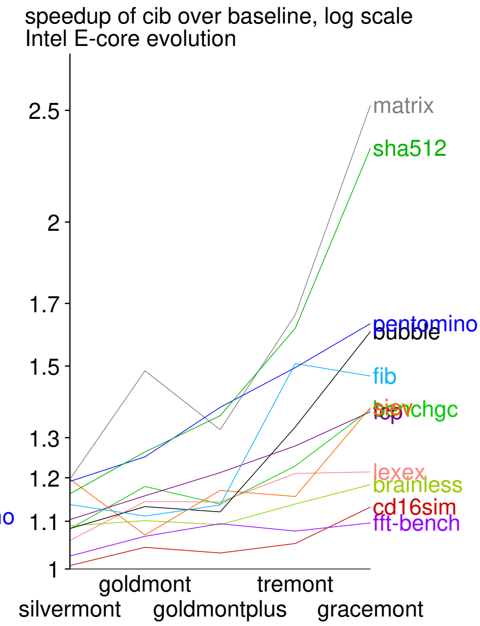
```
add $16,%IP
mov -8(%IP),%r15
add $8,%IP
mov (%RP),%r9
add $8,%IP
mov %r15b,(%r9)
add $8,%IP
mov %r8,%r15
add $8,%IP
mov (%RP),%rax

lea (%r15,%rax,1),%rdx
sub 8(%RP),%rax
btc 63,%rax
mov %rdx,(%RP)
add %r15,%rax
jo  break
mov 16(%RP),%IP
mov 0(%IP),%rax
jmp *%rax
```

9



10





11

Conclusion

- Problem: VM instruction-pointer updates can be a performance bottleneck
- Solution: Optimize instruction-pointer updates
  combine them
  immediate operand variants
  branch to (adjusted) instruction pointer
  load loop start address without using the instruction pointer
- Results
  speedup factors > 2 on loop-dominated benchmarks: critical path
  speedup factors 1.1–1.3 on call-dominated benchmarks
- Paper: DOI: 10.4230/LIPIcs.ECOOP.2024.14
  `https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2024.14`

Gforth (interpreter) vs. SwiftForth (JIT)

Gforth VM	Gforth machine code	source	SwiftForth

lit
#0
	mov 0x8(%rbx),%r15	0	lea -0x8(%rbp),%rbp
			mov %rbx,0x0(%rbp)
			mov $0x0,%ebx

i	mov (%r14),%r9	i	lea -0x8(%rbp),%rbp
			mov %rbx,0x0(%rbp)
			mov %r14,%rbx
			add %r15,%rbx

c!	mov %r15b,(%r9)	c!	mov 0x0(%rbp),%eax
			mov %al,(%rbx)
			mov 0x8(%rbp),%rbx
			lea 0x10(%rbp),%rbp

dup	mov %r8,%r15	dup	lea -0x8(%rbp),%rbp mov %rbx,0x0(%rbp)

(+loop)
#start
	mov (%r14),%rax
	lea (%r15,%rax,1),%rdx
	sub 0x8(%r14),%rax
	btc $0x3f,%rax
	mov %rdx,(%r14)
	add %r15,%rax
	jo end
	mov (%rbx),%rax
	jmp *%rax
		+loop	add %rbx,%r14
			mov 0x0(%rbp),%rbx
			lea 0x8(%rbp),%rbp
			jno start

Why is gcc -O3 so slow for *bubble*?

```
     gcc -O1                        gcc -O3

1c:  add   $0x4,%rax           c0:  movq   (%rax),%xmm0
     cmp   %rsi,%rax                add    $0x1,%edx
     je    35                       pshufd $0xe5,%xmm0,%xmm1
25:  mov   (%rax),%edx              movd   %xmm0,%edi
     mov   0x4(%rax),%ecx           movd   %xmm1,%ecx
     cmp   %ecx,%edx                cmp    %ecx,%edi
     jle   1c                       jle    e1
     mov   %ecx,(%rax)              pshufd $0xe1,%xmm0,%xmm0
     mov   %edx,0x4(%rax)           movq   %xmm0,(%rax)
     jmp   1c                  e1:  add    $0x4,%rax
35:                                 cmp    %r8d,%edx
                                    jl     c0
```