Montgomery Multiply from SSH

gcc -O3 (29 LOC)

.L0:
    movq rsi, r9
    movl ecx, ecx
    shrq 32, rsi
    andl 0xffffffff, r9d
    movq rcx, rax
    movl edx, edx
    imulq r9, rax
    imulq rdx, r9
    imulq rsi, rdx
    imulq rsi, rcx
    addq rdx, rax
    jae .L2
    movabsq 0x100000000, rdx
    addq rdx, rcx

.L2:
    movq rax, rsi
    movq rax, rdx
    shrq 32, rsi
    salq 32, rdx
    addq rsi, rcx
    addq r9, rdx
    adcq 0, rcx
    addq r8, rdx
    adcq 0, rcx
    addq rdi, rdx
    adcq 0, rcx
    movq rcx, r8
    movq rdx, rdi

STOKE (11 LOC)

.L0:
    shlq 32, rcx
    movl edx, edx
    xorq rdx, rdx
    movq rcx, rax
    mulq rsi
    addq r8, rdi
    adcq 9, rdx
    addq rdi, rax
    adcq 0, rdx
    movq rdx, r8
    movq rax, rdi
The Issue

Compiler optimizers solve a search problem ... 

... but without doing any search.
Step Back

What if we started over?

What would a search-based optimizer look like?
Hmmmmm....

• Irregular, high-dimensional search space
  • Lots of complicated interactions between different instructions, machine resources, bits of state.

• And really, really big

• Solution:
  • Use Markov Chain Monte Carlo sampling
MCMC Optimizer

1. Start with some program

2. Repeat (millions of times)
   • Make a random change and evaluate cost
   • If (cost decreased)
     {accept (i.e., keep the change)}
   • If (cost increased)
     {with some probability accept anyway}
Transformations

**Insert**

```assembly
... movl ecx, ecx shrq 32, rsi andl 0xffffffff, r9d movq rcx, rax movl edx, edx imulq r9, rax imulq rsi, rdx ...
```

**Delete**

```assembly
... movl ecx, ecx shrq 32, rsi andl 0xffffffff, r9d movq rcx, rax movl edx, edx imulq r9, rax ...
```

**Opcode**

```assembly
... movl ecx, ecx shrq 32, rsi andl 0xffffffff, r9d movq rcx, rax movl edx, edx imulq r9, rax ...
```

**Instruction**

```assembly
... movl ecx, ecx shrq 32, rsi andl 0xffffffff, r9d movq rcx, rax movl edx, edx imulq r9, rax ...
```

**Swap**

```assembly
... movl ecx, ecx shrq 32, rsi andl 0xffffffff, r9d movq rcx, rax movl edx, edx imulq r9, rax ...
```

**Operand**

```assembly
... movl ecx, ecx shrq 32, rsi andl 0xffffffff, r9d movq rcx, rax movl edx, edx imulq r9, rax ...
```
Synthesis

cost(r,t) = equal(r,t)
Optimization

\[ \text{cost}(r,t) = \text{eq}(r,t) + \text{perf}(r,t) \]
Montgomery Multiply, Revisited
The (Correctness) Cost Function

• Idea 1:
  \[ \text{Cost}(P) = \# \text{ of incorrect bits of output on a test suite} \]

• Idea 2:
  \textit{Reward the right answer in the wrong place.} \\
  Consider all ways of adding a suffix of move instructions to the program and choose the one with lowest cost \\
  (+ a small penalty for the moves)
Example

Synthesize a program to increment every element of a vector of integers.

\%rax holds the base address of the vector
\%si holds the # of vector elements
Synthesizing Vector Increment

Cost vs. Iterations

- Iterations: 0 to 7
- Cost (in Millions): 0 to 250

Graph shows a steady decline in cost from 250 million to near zero over iterations.
Synthesizing Vector Increment

inc:
imull $0x7, %esi, %eax
cmpxchgb %al, %al
.L_start:
salb $0x5, %sil
leaq 0x7(%edi,%edi,2), %rdx
jne .L_start
retq
Synthesizing Vector Increment

inc:
  movq %rdi, %rax
  xchgb %dil, %al
  testb $0x4, %al
  .L_start:
  setnp %r10b
  jne .L_start
  incb (%rax)
  retq

Cost

03-Nov-15

STOKE Overview
Synthesizing Vector Increment

```
inc:
.L_start:
    movq %rdi, %rax  
cmpw (%rax), %di  
    incw (%rax)  
    decw %si  
    cmovngw %si, %dx  
    addq $0x4, %rdi  
    testb $0xfffffffffffffff, %sil  
    jne .L_start  
    jne .L_start  
    retq
```
Synthesizing Vector Increment

Inc:
.L_start:
    movq %rdi, %rax
    incw (%rax)
    incq (%rax)
    addq $0x4, %rdi
    decw %si
    jne .L_start
    retq

03-Nov-15

STOKE Overview
Engineering Constraints

• The cost function needs to be inexpensive
  • Because we will be evaluating it billions of times

• Cost functions are split into two parts
  • A fast approximate test in the inner loop
  • A slow exact test we do rarely
Two Cost Functions

Correctness

• Fast: Use test cases

• Slow: Use SMT solver to check for program equivalence

Performance

• Fast: Sum instruction latencies

• Slow: Benchmark code on the target processor
Application: Program Optimization

• Start with a program $T$
  • Run $T$ on some test cases, save results

• Use STOKE to find a program $R$ that
  • Matches $T$ on the tests
  • Is faster than $T$

• Check $R = T$
But ...

• How do we check $R = T$?

• $R$ was discovered by a random process
  • Unrelated to the structure of $T$
  • There is no given translation from $T$ to $R$

• A hard program equivalence problem
The Easy Case

• If \( T \) and \( R \)
  • Are loop-free
  • And have no floating point operations
  • (and some other things)

• Then
  • Encode \( T = R \) as an SMT formula

• Bonus
  • Can get a counterexample if \( T \) is not equal to \( R \)
  • Use as a new test case in another search
Handling Loops

• SMT solvers can prove properties of loop-free code

• We want to prove equivalence of loops

• Decompose proof for loops into subproofs
  • Subproofs about loop free fragment, query SMT solvers
  • Cutpoints: break the loops
  • Invariants: relationships between $T$ and $R$ at cutpoints
Proof Decomposition Example

while(i!=0) i--; 

Rewrite R

$I_a$: states equal $a'$. 

$I_b$: $8(rsp)=rdi=r9'$. 

$I_c$: live out equal. 

$I_b$: $8(rsp)=rdi=r9'$. 

Simulation Relation

STOKE Overview

03-Nov-15
Inference

• Given a simulation relation, proofs for loops reduce to subproofs for loop free fragments
  • Query SMT solvers

• Main challenge: infer a simulation relation
  • Infer cutpoints
  • Infer invariants

• Mine relations from data
  • program executions
Cutpoints From Data

• Attempt to detect cutpoints
  • Number of times program points are executed

Data!

Target $T$

Rewrite $R$

$\text{movq 8(rsp), rdi}$
$\#\text{rdi} != 0$

$n+1$

$\text{movq 8(rsp), rdi}$
$\text{decq rdi}$
$\text{movq rdi, 8(rsp)}$

$n$

$\text{movq 8(rsp), r9}$
$\#\text{r9} != 0$

$n+1$

$\text{decq r9}$
$\text{retq}$

$\text{retq}$
Invariants

• Invariants are restricted to equalities
  • Infer invariants from observed data values

Target $T$

```
movq 8(rsp), rdi
#rdi != 0
```

```
movq 8(rsp), rdi
decq rdi
movq rdi, 8(rsp)
```

```
retq
```

Data!

<table>
<thead>
<tr>
<th>8(rsp)</th>
<th>rdi</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Invariants

- Invariants are restricted to equalities
  - Infer invariants from observed data values

**Rewrite** $R$

```
movq 8(rsp), r9
#r9 != 0
decq r9  
retq
```

<table>
<thead>
<tr>
<th>8(rsp)</th>
<th>rdi</th>
<th>r9'</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Linear algebra

\[ A \equiv \begin{array}{ccc}
8(rsp) & rdi & r9' \\
2 & 2 & 2 \\
1 & 1 & 1 \\
0 & 0 & 0
\end{array} \]

- Mine all equalities
- Find all \( w \) s.t. \( Aw = 0 \)
- Nullspace
  - \( w_1 = [-1,1,0] \)
  - \( w_2 = [0,1,-1] \)

\[ I \equiv (8(rsp) = rdi) \land (rdi = r9') \]
Check Simulation Relation

- Query SMT solvers
  - Incorporate counter-examples in relations

- Sound but not complete
  - If checking succeeds then equivalent
  - Can fail to infer a correct simulation relation
  - Learn from counter-examples
Benchmarks

- **Synthesis Kernels:** 25 loop-free kernels taken from A Hacker’s Delight (Gulwani el al. ‘11)

- **SSL:** Montgomery multiplication kernel

- **Heap Modifying:** Linked list traversal

- **Linear Algebra:** SAXPY from BLAS
Results (ASPLOS’ 13, OOPSLA’ 13)

- STOKE is comparable or outperforms gcc and icc while maintaining correctness w.r.t. unoptimized code
Observation

• Speedups are good: 70% over production compiler

• However, x86 experts produce much better code

• Why can’t we get 2X, 3X speedups over gcc –O3?

• Checker rejects many “good” programs
  • Works perfectly with the application
  • Fails on corner cases that cannot arise
Example

• Target

```c
foo(int32_t* a, int32_t* b)
    *a++=*b++;    
    *a++=*b++;    
    *a++=*b++;    
    return;
```

• Expert Rewrite

```c
MOVAPD xmm0, [b]
MOVAPD [a], xmm0
RET
```

• Incorrect, if `a` and `b` overlap

• Segfault if `a` and `b` are not 16 byte aligned
Conditions

• Cannot use the fast rewrite in an arbitrary context

• Developers know conditions on the context
  • Used in adding unchecked annotations, writing assembly

• Unconditionally correct optimizations are effective
  • But fall short of the fastest code desired

• For performance, we need conditional equivalence
Conditional Correctness

• Target

```c
foo(int32_t* a,
    int32_t* b)
*a++=*b++;
*a++=*b++;
*a++=*b++;
*a++=*b++;
return;
```

• Rewrite

```assembly
MOVAPD xmm0, [b]
MOVAPD [a], xmm0
RET
```

• Conditionally correct
  • restrict(a), restrict(b)
  • a, b are 16 byte aligned
Conditionally correct optimizer

Random Programs → STOKE → Equivalence Checker → Perf Tests → Conditionally Correct Programs → Fast Conditionally Correct Program

Condition C
Conditionally correct optimizer

Random Programs

STOKE

Equivalence Checker

Condition C

Data!

Conditionally Correct Programs

Perf Tests

Fast Conditionally Correct Program

03-Nov-15
STOKE Overview
Condition Inference

• Tests encode the conditions on contexts implicitly

• Learn facts about tests
  • Keep adding facts until the proof succeeds

• Aliasing: two memory dereferences do not alias
• Alignment: an address is x-byte aligned
• Equalities, inequalities, floating point specific
Conditions

1. Analyze A to verify C
2. Manually verify C
3. Runtime check
Conditional GCC

Annotations can help a compiler significantly

Relative Performance

annot. gcc -O3  gcc -O3

03-Nov-15  STOKE Overview
Conditional STOKE

STOKE with condition inference is comparable or outperforms gcc with or without annotations.
Runtime Checks

Overhead of condition checking can be bearable
Why Does STOKE Work?

• Larger repertoire
  • gcc only knows a few instructions in the X86 instruction set. STOKE knows all 3,000+ instructions.

• More effort
  • Throw lots of computation at the problem.

• Can prove correctness
  • Allows experimentation with incorrect code
Why Does STOKE Work?

• Optimization is a natural application
  • Standard compiler transformations are small, local changes

• Division into search and verification
  • Search is very general and highly parallel
  • Only ask SMT solver to do checking, not inference
  • All examples verify in under 1 second
Related Work

• Massalin [ASPLOS 87]

• Joshi, Nelson and Randall [PLDI 02]

• Bansal and Aiken [ASPLOS 06]

• Tate, Stepp, Tatlock and Lerner [POPL 09]

• Gulwani. Jha, Tiwari, Venkatesan [PLDI 11]
Questions?

stoke.stanford.edu