Stochastic Program Optimization

Eric Schkufza, Rahul Sharma, and Alex Aiken, CACM’16

Presented by Ajay Jain

Reading group | April 4th, 2018

Program synthesis & induction

- Target program: $\vec{y} = P(\vec{x})$
- Generated program: $\hat{P}(\vec{x}) \approx \vec{y}$
- Generated using:
    - input output pairs
    - an example program *P* (e.g. for performance optimization or translation/compilation)
- In **program synthesis**, $\hat{P}$ is interpretable
    - e.g. stored as explicit code
- In **program induction**, $\hat{P}$ can be evaluated, but we don’t know its internals

MACHINE INTELLIGENCE COMMUNITY

Program synthesis & induction **for compilers**

- Compilers need to be **correct** (mostly - according to a spec) in translating a program to output machine instructions

- Typically, you hope to **minimize runtime**

- In a reasonable amount of time (compilation is already pretty slow…)

Superoptimization

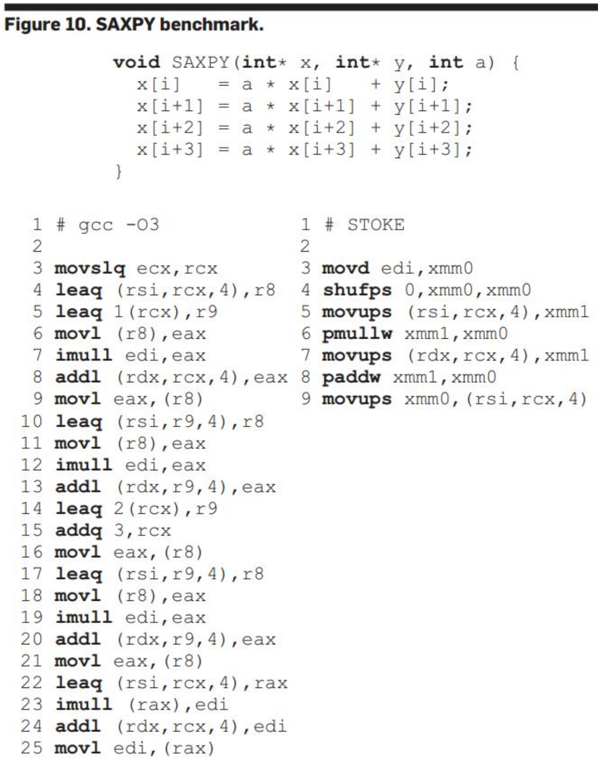


Figure 10. SAXPY benchmark.

```
void SAXPY(int* x, int* y, int a) {
  x[i]   = a * x[i]   + y[i];
  x[i+1] = a * x[i+1] + y[i+1];
  x[i+2] = a * x[i+2] + y[i+2];
  x[i+3] = a * x[i+3] + y[i+3];
}
```

```
# gcc -O3

movslq ecx,rcx
leaq (rsi,rcx,4),r8
leaq 1(rcx),r9
movl (r8),eax
imull edi,eax
addl (rdx,rcx,4),eax
movl eax,(r8)
leaq (rsi,r9,4),r8
movl (r8),eax
imull edi,eax
addl (rdx,r9,4),eax
leaq 2(rcx),r9
addq 3,rcx
movl eax,(r8)
leaq (rsi,r9,4),r8
movl (r8),eax
imull edi,eax
addl (rdx,r9,4),eax
movl eax,(r8)
leaq (rsi,rcx,4),rax
imull (rax),edi
addl (rdx,rcx,4),edi
movl edi,(rax)
```

```
# STOKE

movd edi,xmm0
shufps 0,xmm0,xmm0
movups (rsi,rcx,4),xmm1
pmullw xmm1,xmm0
movups (rdx,rcx,4),xmm1
paddw xmm1,xmm0
movups xmm0,(rsi,rcx,4)
```

- **Basic block:** A sequence of instructions without branches (jumps).
    - Also known as straight-line code

- **Superoptimization:** Translate straight-line code into a more optimized form

Superoptimization

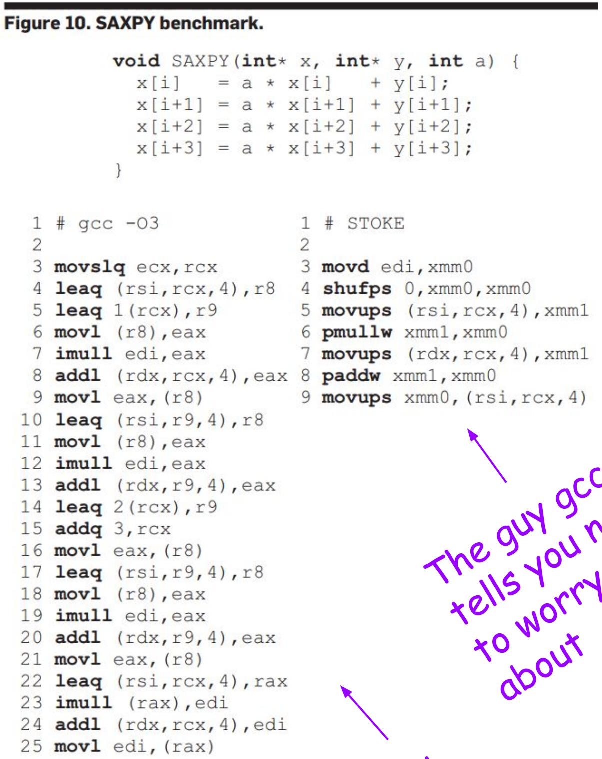


Figure 10. SAXPY benchmark.

```
void SAXPY(int* x, int* y, int a) {
  x[i]   = a * x[i]   + y[i];
  x[i+1] = a * x[i+1] + y[i+1];
  x[i+2] = a * x[i+2] + y[i+2];
  x[i+3] = a * x[i+3] + y[i+3];
}
```

```
# gcc -O3

movslq ecx,rcx
leaq (rsi,rcx,4),r8
leaq 1(rcx),r9
movl (r8),eax
imull edi,eax
addl (rdx,rcx,4),eax
movl eax,(r8)
leaq (rsi,r9,4),r8
movl (r8),eax
imull edi,eax
addl (rdx,r9,4),eax
leaq 2(rcx),r9
addq 3,rcx
movl eax,(r8)
leaq (rsi,r9,4),r8
movl (r8),eax
imull edi,eax
addl (rdx,r9,4),eax
movl eax,(r8)
leaq (rsi,rcx,4),rax
imull (rax),edi
addl (rdx,rcx,4),edi
movl edi,(rax)
```

```
# STOKE

movd edi,xmm0
shufps 0,xmm0,xmm0
movups (rsi,rcx,4),xmm1
pmullw xmm1,xmm0
movups (rdx,rcx,4),xmm1
paddw xmm1,xmm0
movups xmm0,(rsi,rcx,4)
```

The guy gcc tells you not to worry about

You

- **Basic block:** A sequence of instructions without branches (jumps).
    - Also known as straight-line code

- **Superoptimization:** Translate straight-line code into a more optimized form

To be fair to gcc…

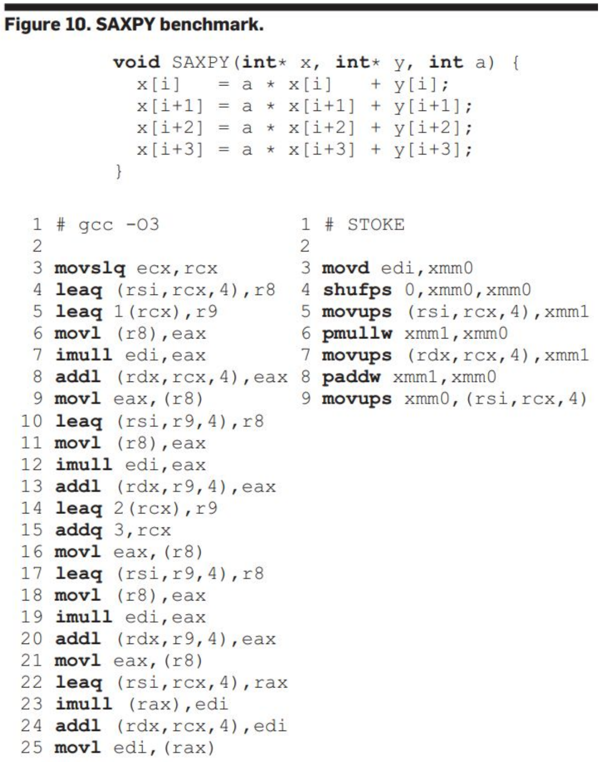


Figure 10. SAXPY benchmark.

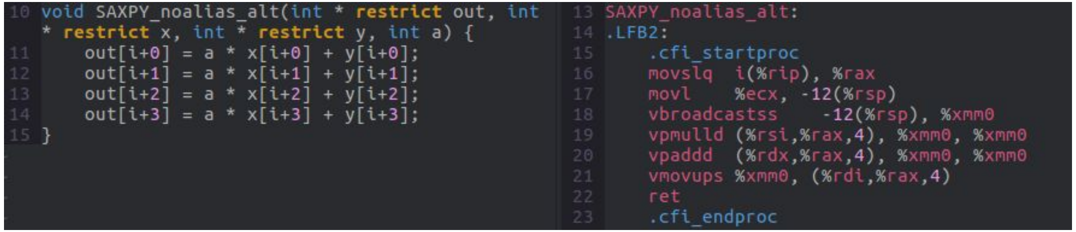
```
void SAXPY(int* x, int* y, int a) {
  x[i]   = a * x[i]   + y[i];
  x[i+1] = a * x[i+1] + y[i+1];
  x[i+2] = a * x[i+2] + y[i+2];
  x[i+3] = a * x[i+3] + y[i+3];
}
```

```
# gcc -O3

movslq ecx,rcx
leaq (rsi,rcx,4),r8
leaq 1(rcx),r9
movl (r8),eax
imull edi,eax
addl (rdx,rcx,4),eax
movl eax,(r8)
leaq (rsi,r9,4),r8
movl (r8),eax
imull edi,eax
addl (rdx,r9,4),eax
leaq 2(rcx),r9
addq 3,rcx
movl eax,(r8)
leaq (rsi,r9,4),r8
movl (r8),eax
imull edi,eax
addl (rdx,r9,4),eax
movl eax,(r8)
leaq (rsi,rcx,4),rax
imull (rax),edi
addl (rdx,rcx,4),edi
movl edi,(rax)
```

```
# STOKE

movd edi,xmm0
shufps 0,xmm0,xmm0
movups (rsi,rcx,4),xmm1
pmullw xmm1,xmm0
movups (rdx,rcx,4),xmm1
paddw xmm1,xmm0
movups xmm0,(rsi,rcx,4)
```



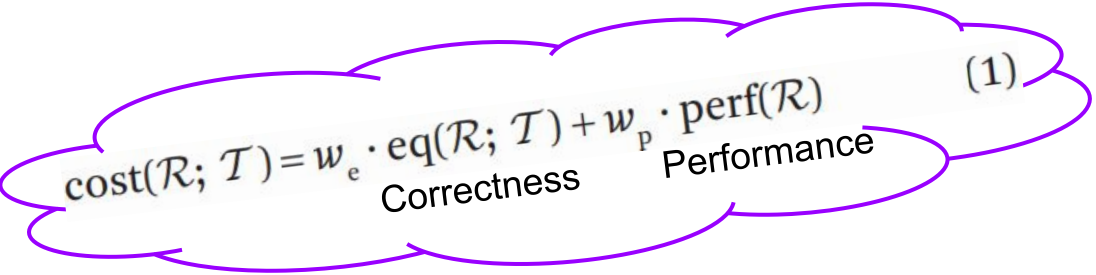
```
void SAXPY_noalias_alt(int * restrict out, int * restrict x, int * restrict y, int a) {
    out[i+0] = a * x[i+0] + y[i+0];
    out[i+1] = a * x[i+1] + y[i+1];
    out[i+2] = a * x[i+2] + y[i+2];
    out[i+3] = a * x[i+3] + y[i+3];
}
```

```
SAXPY_noalias_alt:
.LFB2:
    .cfi_startproc
    movslq  i(%rip), %rax
    movl    %ecx, -12(%rsp)
    vbroadcastss    -12(%rsp), %xmm0
    vpmulld (%rsi,%rax,4), %xmm0, %xmm0
    vpaddd  (%rdx,%rax,4), %xmm0, %xmm0
    vmovups %xmm0, (%rdi,%rax,4)
    ret
    .cfi_endproc
```

Originally, gcc translated the SAXPY function into a series of *scalar operations* for correctness, possibly due to aliasing and alignment issues. Annotations help.

Compiled with `gcc -O3 saxpy.c -S -march=native` *with sse2 support*

Cost to minimize



$$\mathrm{cost}(\mathcal{R}; \mathcal{T}) = \underbrace{w_{\mathrm{e}} \cdot \mathrm{eq}(\mathcal{R}; \mathcal{T})}_{\text{Correctness}} + \underbrace{w_{\mathrm{p}} \cdot \mathrm{perf}(\mathcal{R})}_{\text{Performance}} \qquad (1)$$

- **eq(Rewrite; Target)**: measures similarity eq(R,T) = 0 if R and T result in the same output registers and memory given the same input *(*live inputs and outputs)*

Cost to minimize

$$\mathrm{cost}(\mathcal{R};\mathcal{T}) = \underbrace{w_{\mathrm{e}} \cdot \mathrm{eq}(\mathcal{R};\mathcal{T})}_{\text{Correctness}} + \underbrace{w_{\mathrm{p}} \cdot \mathrm{perf}(\mathcal{R})}_{\text{Performance}} \tag{1}$$

$$\mathrm{eq}^{*}(\mathcal{R};\mathcal{T},\tau) = \begin{cases} \mathrm{eq}(\mathcal{R};\mathcal{T}), & \text{if } \mathrm{eq}'(\mathcal{R};\mathcal{T},\tau) = 0 \\ \mathrm{eq}'(\mathcal{R};\mathcal{T},\tau), & \text{otherwise} \end{cases}$$

- **eq(Rewrite; Target)**: measures similarity eq(R,T) = 0 if R and T result in the same output registers and memory given the same input *(*live inputs and outputs)*
  - *With a symbolic validator:* only 1000 evaluations per second
  - → *Solution:* approximate based on test cases

Cost to minimize

$$\mathrm{cost}(\mathcal{R};\mathcal{T}) = \underbrace{w_{\mathrm{e}} \cdot \mathrm{eq}(\mathcal{R};\mathcal{T})}_{\text{Correctness}} + \underbrace{w_{\mathrm{p}} \cdot \mathrm{perf}(\mathcal{R})}_{\text{Performance}} \qquad (1)$$

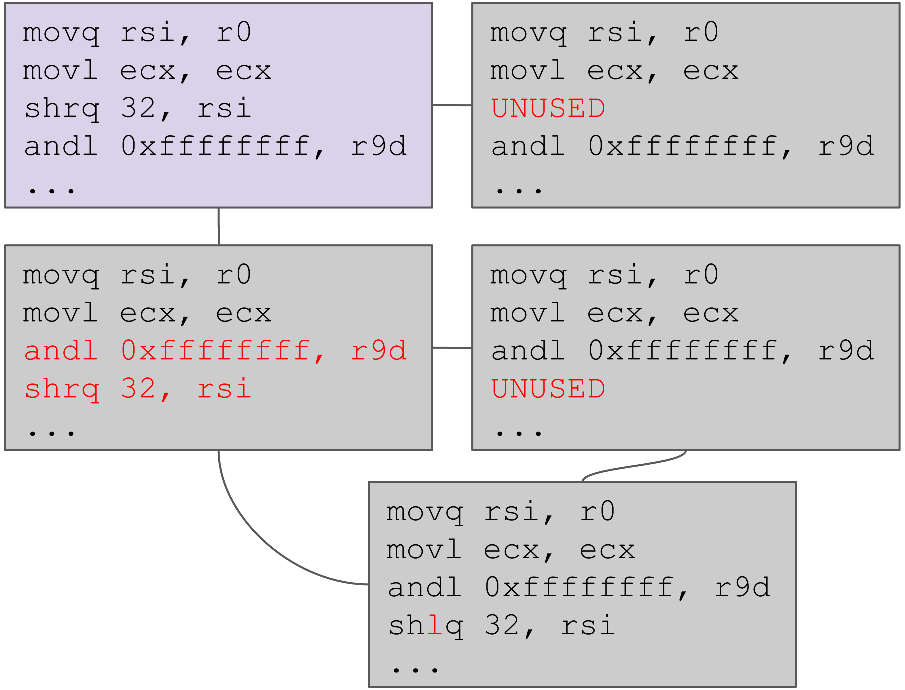
$$\mathrm{eq}^*(\mathcal{R};\mathcal{T},\tau) = \begin{cases} \mathrm{eq}(\mathcal{R};\mathcal{T}), & \text{if } \mathrm{eq}'(\mathcal{R};\mathcal{T},\tau) = 0 \\ \mathrm{eq}'(\mathcal{R};\mathcal{T},\tau), & \text{otherwise} \end{cases}$$

$$\mathrm{perf}(\mathcal{R}) = \sum_{i \in \mathrm{inst}(\mathcal{R})} \mathrm{lat}(i)$$

- **eq(Rewrite; Target)**: measures similarity eq(R,T) = 0 if R and T result in the same output registers and memory given the same input *(*live inputs and outputs)*
  - *With a symbolic validator:* only 1000 evaluations per second
  - → *Solution:* approximate based on test cases

- **perf:** Quantifies speedup of the rewrite
  - (heuristic, sum of instruction latencies)

Search space - Markov Chain, graph for program space



```
movq rsi, r0
movl ecx, ecx
shrq 32, rsi
andl 0xffffffff, r9d
...
```

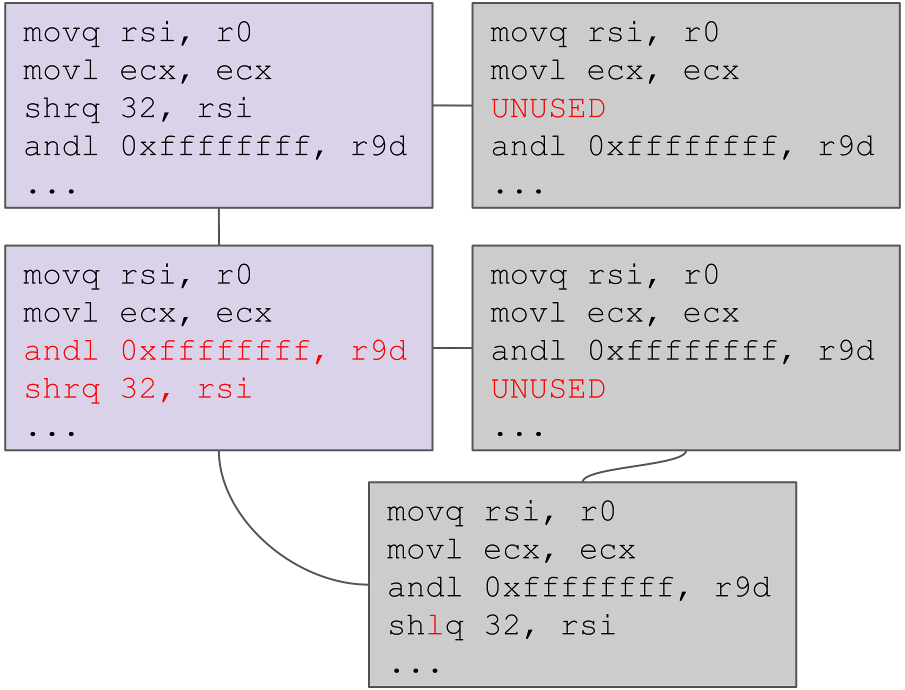
```
movq rsi, r0
movl ecx, ecx
UNUSED
andl 0xffffffff, r9d
...
```

```
movq rsi, r0
movl ecx, ecx
andl 0xffffffff, r9d
shrq 32, rsi
...
```

```
movq rsi, r0
movl ecx, ecx
andl 0xffffffff, r9d
UNUSED
...
```

```
movq rsi, r0
movl ecx, ecx
andl 0xffffffff, r9d
shlq 32, rsi
...
```

Search space - Markov Chain, graph for program space



```
movq rsi, r0
movl ecx, ecx
shrq 32, rsi
andl 0xffffffff, r9d
...
```

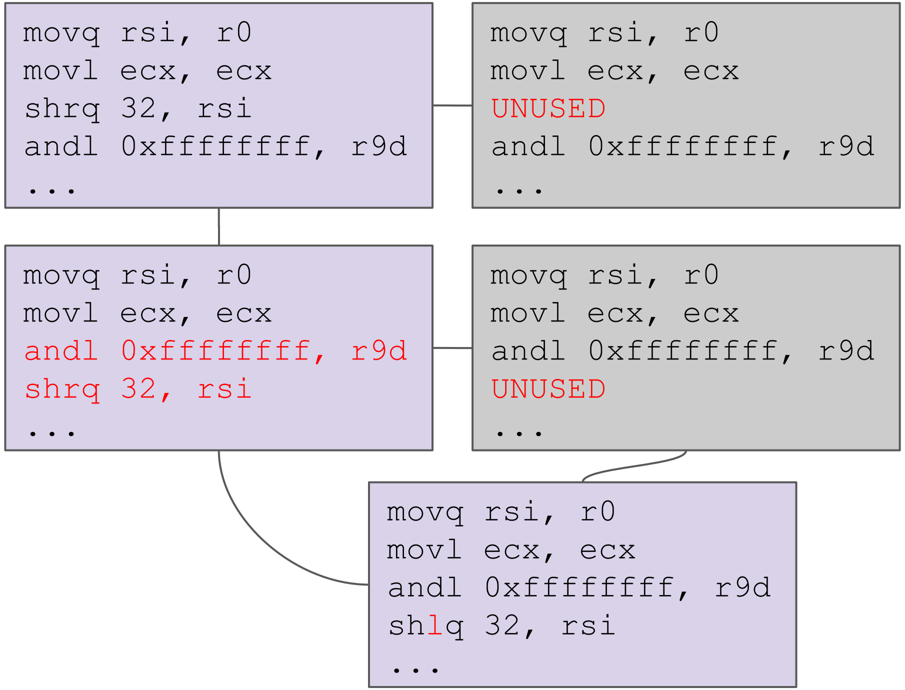
```
movq rsi, r0
movl ecx, ecx
UNUSED
andl 0xffffffff, r9d
...
```

```
movq rsi, r0
movl ecx, ecx
andl 0xffffffff, r9d
shrq 32, rsi
...
```

```
movq rsi, r0
movl ecx, ecx
andl 0xffffffff, r9d
UNUSED
...
```

```
movq rsi, r0
movl ecx, ecx
andl 0xffffffff, r9d
shlq 32, rsi
...
```

Search space - Markov Chain, graph for program space



```
movq rsi, r0
movl ecx, ecx
shrq 32, rsi
andl 0xffffffff, r9d
...
```

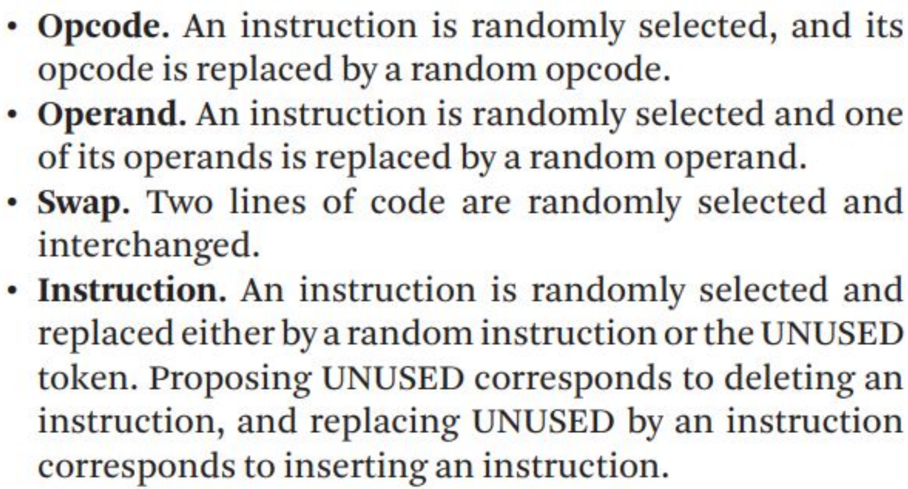
```
movq rsi, r0
movl ecx, ecx
UNUSED
andl 0xffffffff, r9d
...
```

```
movq rsi, r0
movl ecx, ecx
andl 0xffffffff, r9d
shrq 32, rsi
...
```

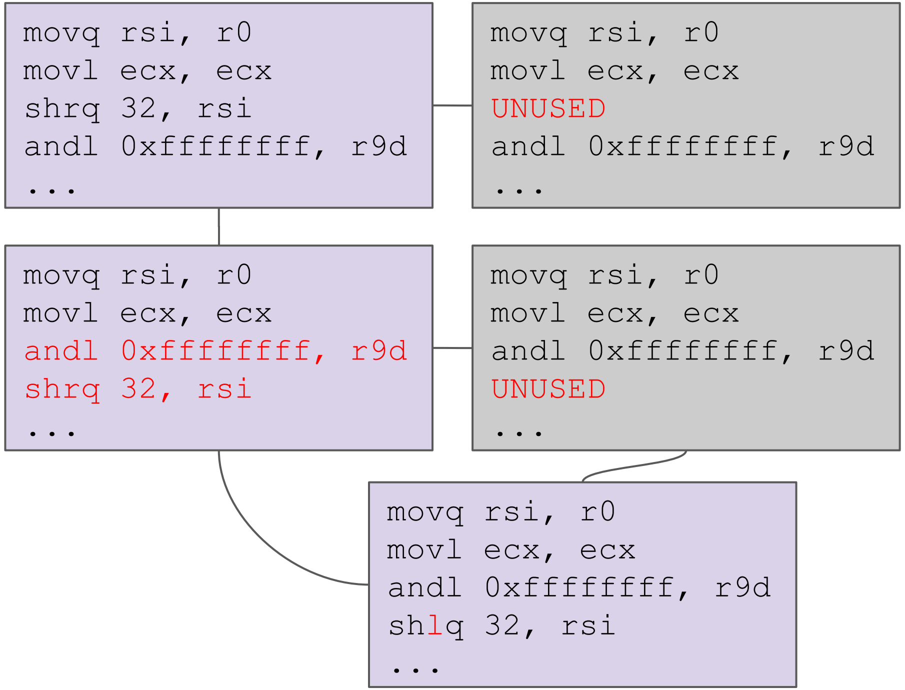
```
movq rsi, r0
movl ecx, ecx
andl 0xffffffff, r9d
UNUSED
...
```

```
movq rsi, r0
movl ecx, ecx
andl 0xffffffff, r9d
shlq 32, rsi
...
```

Search space - Markov Chain, graph for program space



- **Opcode.** An instruction is randomly selected, and its opcode is replaced by a random opcode.
- **Operand.** An instruction is randomly selected and one of its operands is replaced by a random operand.
- **Swap.** Two lines of code are randomly selected and interchanged.
- **Instruction.** An instruction is randomly selected and replaced either by a random instruction or the UNUSED token. Proposing UNUSED corresponds to deleting an instruction, and replacing UNUSED by an instruction corresponds to inserting an instruction.



```
movq rsi, r0
movl ecx, ecx
shrq 32, rsi
andl 0xffffffff, r9d
...
```

```
movq rsi, r0
movl ecx, ecx
UNUSED
andl 0xffffffff, r9d
...
```

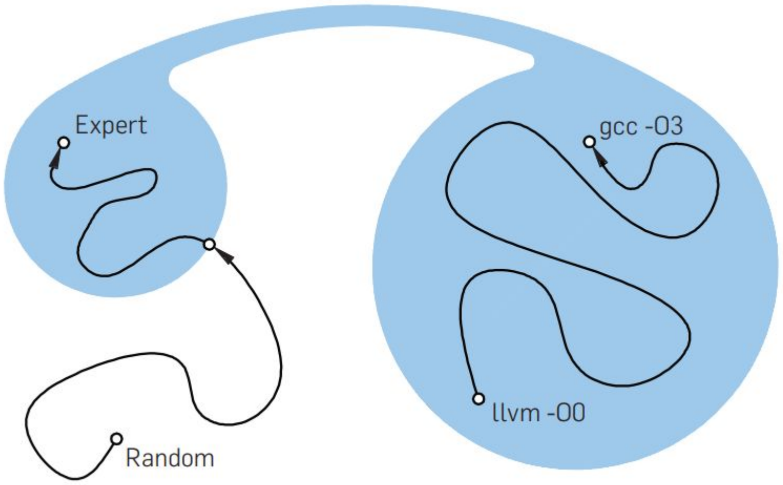
```
movq rsi, r0
movl ecx, ecx
andl 0xffffffff, r9d
shrq 32, rsi
...
```

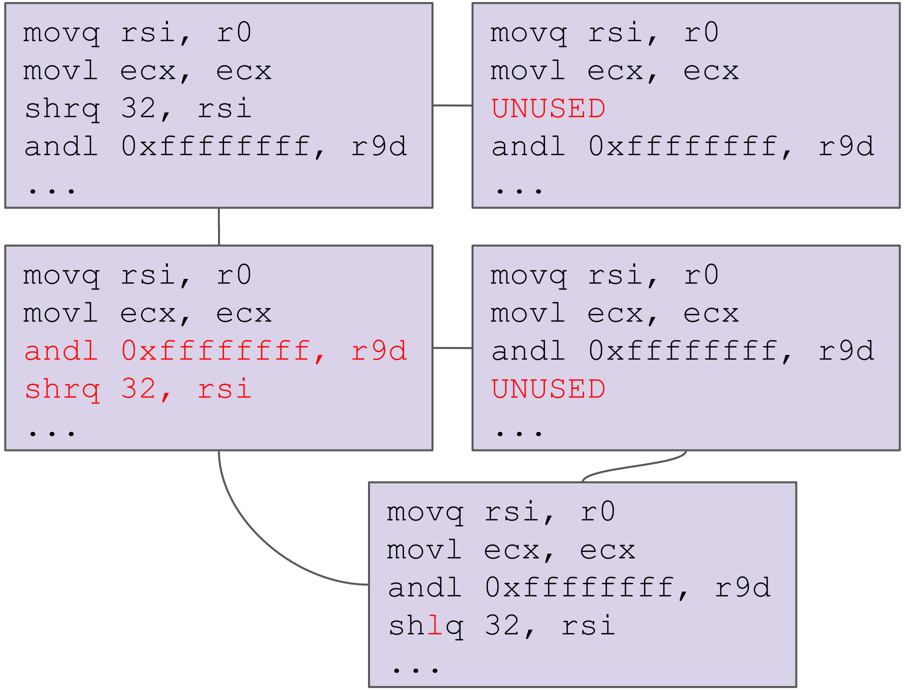
```
movq rsi, r0
movl ecx, ecx
andl 0xffffffff, r9d
UNUSED
...
```

```
movq rsi, r0
movl ecx, ecx
andl 0xffffffff, r9d
shlq 32, rsi
...
```

Search space - Markov Chain, graph for program space

Figure 3. Search space for the Montgomery multiplication benchmark: O0 and O3 codes are densely connected, whereas expert code is reachable only by an extremely low probability path.





```
movq rsi, r0
movl ecx, ecx
shrq 32, rsi
andl 0xffffffff, r9d
...
```

```
movq rsi, r0
movl ecx, ecx
UNUSED
andl 0xffffffff, r9d
...
```

```
movq rsi, r0
movl ecx, ecx
andl 0xffffffff, r9d
shrq 32, rsi
...
```

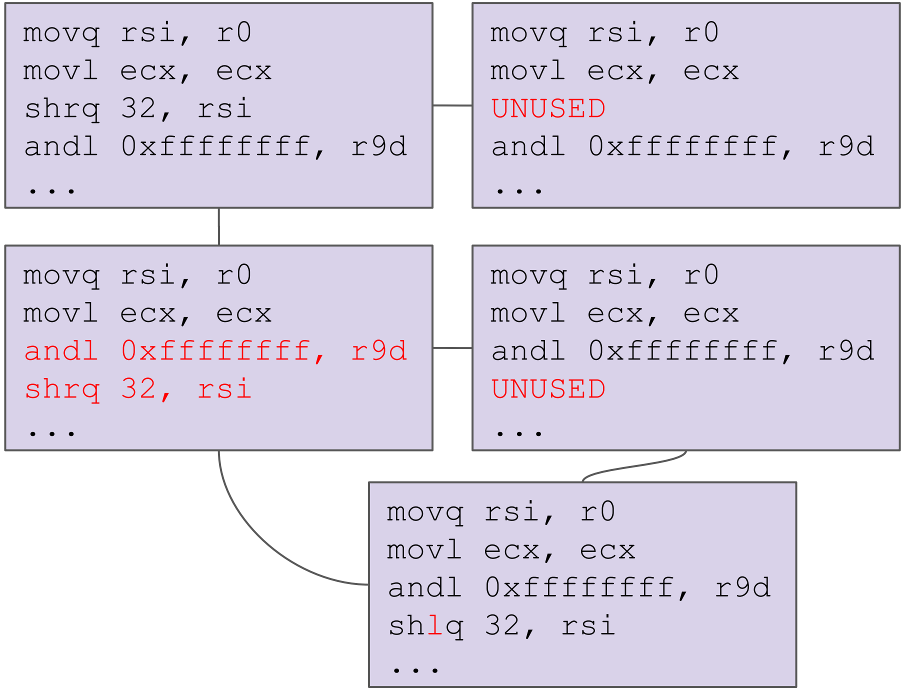
```
movq rsi, r0
movl ecx, ecx
andl 0xffffffff, r9d
UNUSED
...
```

```
movq rsi, r0
movl ecx, ecx
andl 0xffffffff, r9d
shlq 32, rsi
...
```

Markov-Chain Monte-Carlo sampling

- Rewrite the cost as a probability distribution

$$p(\mathcal{R};\mathcal{T}) = \frac{1}{Z}\exp\left(-\beta\cdot\mathrm{cost}(\mathcal{R};\mathcal{T})\right)$$



```
movq rsi, r0
movl ecx, ecx
shrq 32, rsi
andl 0xffffffff, r9d
...
```

```
movq rsi, r0
movl ecx, ecx
UNUSED
andl 0xffffffff, r9d
...
```

```
movq rsi, r0
movl ecx, ecx
andl 0xffffffff, r9d
shrq 32, rsi
...
```

```
movq rsi, r0
movl ecx, ecx
andl 0xffffffff, r9d
UNUSED
...
```

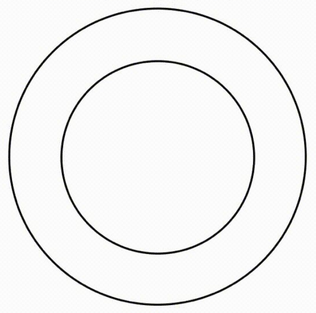
```
movq rsi, r0
movl ecx, ecx
andl 0xffffffff, r9d
shlq 32, rsi
...
```

Markov-Chain Monte-Carlo sampling

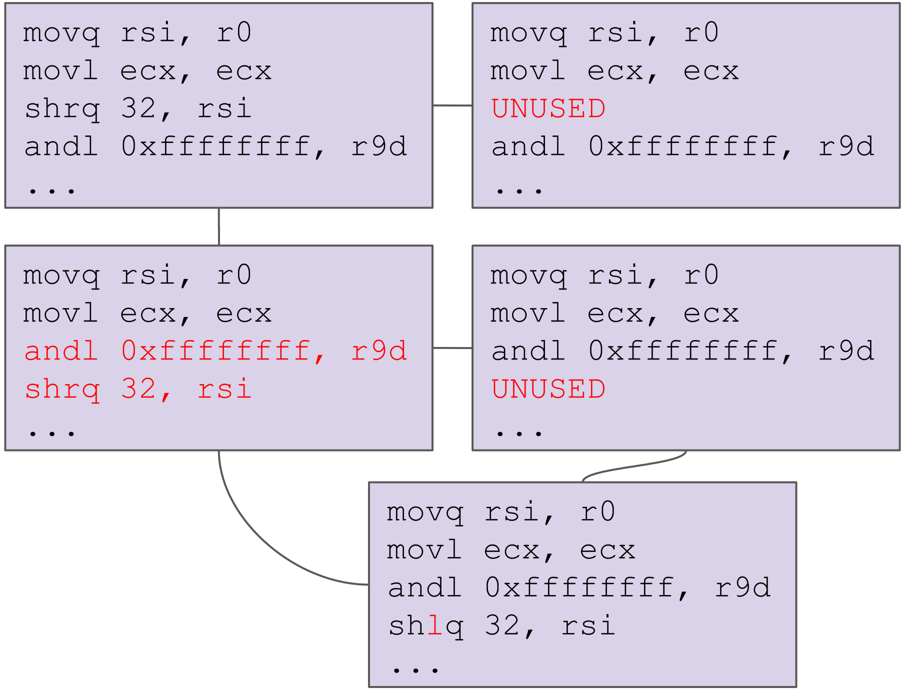
- Rewrite the cost as a probability distribution

$$p(\mathcal{R};\mathcal{T})=\frac{1}{Z}\exp\left(-\beta\cdot\text{cost}(\mathcal{R};\mathcal{T})\right)$$

- Metropolis-Hastings algorithm: Sample from complicated distributions



Picture credit: Colin Carroll
[Source]



```
movq rsi, r0
movl ecx, ecx
shrq 32, rsi
andl 0xffffffff, r9d
...
```

```
movq rsi, r0
movl ecx, ecx
UNUSED
andl 0xffffffff, r9d
...
```

```
movq rsi, r0
movl ecx, ecx
andl 0xffffffff, r9d
shrq 32, rsi
...
```

```
movq rsi, r0
movl ecx, ecx
andl 0xffffffff, r9d
UNUSED
...
```

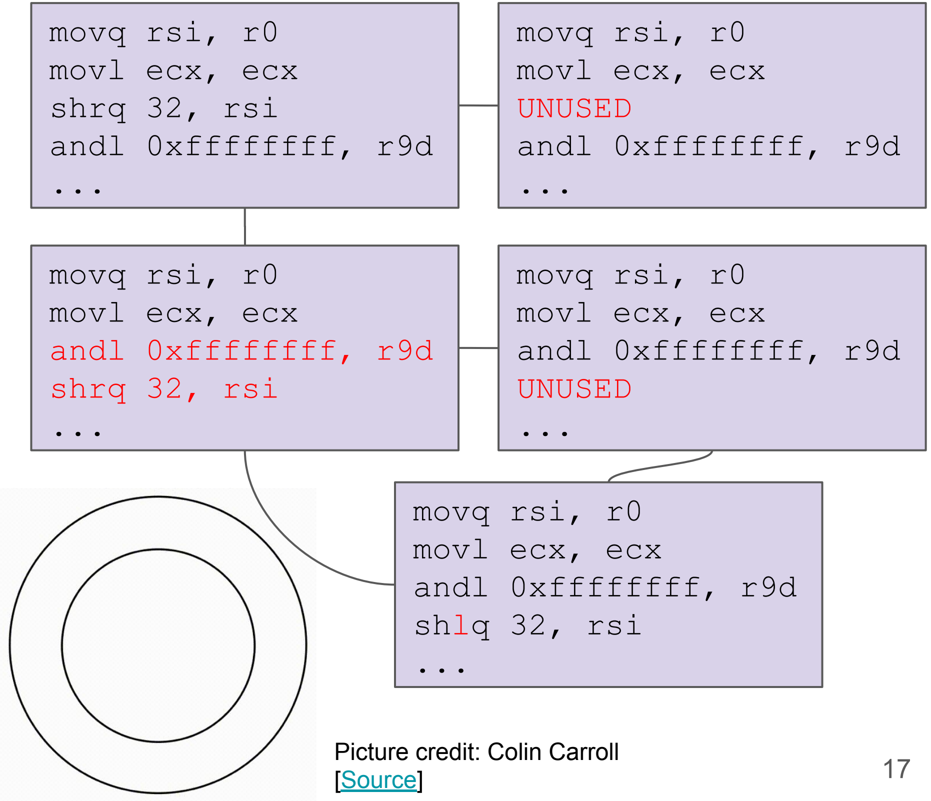
```
movq rsi, r0
movl ecx, ecx
andl 0xffffffff, r9d
shlq 32, rsi
...
```

Markov-Chain Monte-Carlo sampling

- Rewrite the cost as a probability distribution

$$p(\mathcal{R};\mathcal{T}) = \frac{1}{Z}\exp\left(-\beta \cdot \mathrm{cost}(\mathcal{R};\mathcal{T})\right)$$

- Metropolis-Hastings algorithm: Sample from complicated distributions
  - Probabilities don’t need to be scaled!
  - Based on local decisions in the search space
    - Last rewrite R
    - Proposal rewrite R*



```
movq rsi, r0
movl ecx, ecx
shrq 32, rsi
andl 0xffffffff, r9d
...
```

```
movq rsi, r0
movl ecx, ecx
UNUSED
andl 0xffffffff, r9d
...
```

```
movq rsi, r0
movl ecx, ecx
andl 0xffffffff, r9d
shrq 32, rsi
...
```

```
movq rsi, r0
movl ecx, ecx
andl 0xffffffff, r9d
UNUSED
...
```

```
movq rsi, r0
movl ecx, ecx
andl 0xffffffff, r9d
shlq 32, rsi
...
```

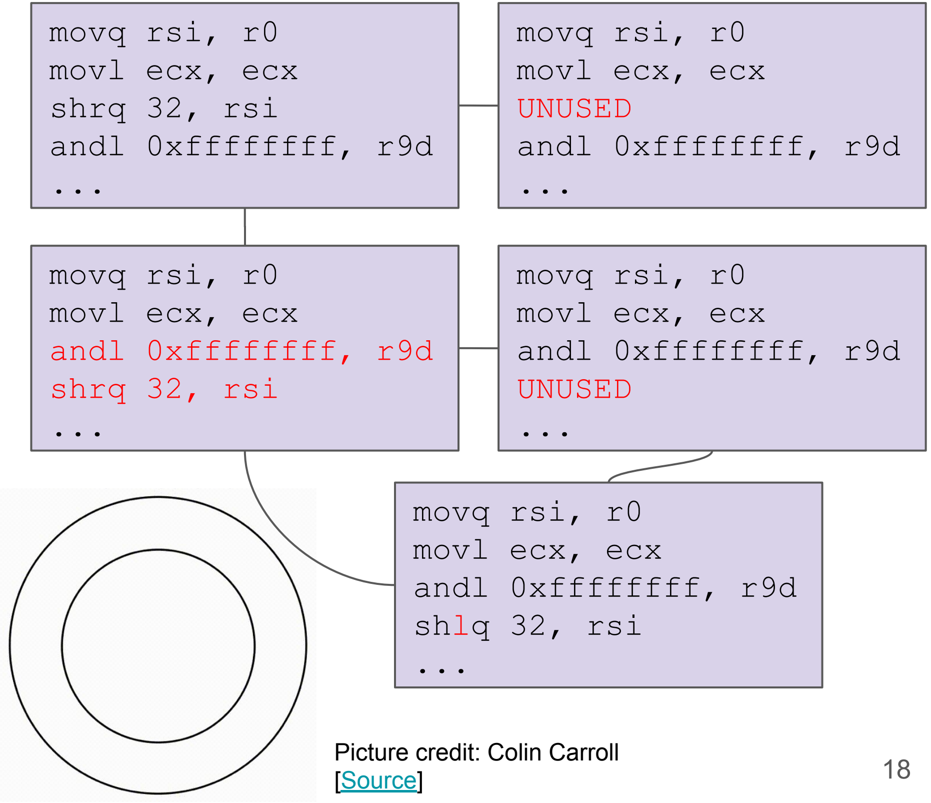
Picture credit: Colin Carroll
[Source]

Markov-Chain Monte-Carlo sampling

- Rewrite the cost as a probability distribution

$$p(\mathcal{R};\mathcal{T}) = \frac{1}{Z}\exp\left(-\beta\cdot\mathrm{cost}(\mathcal{R};\mathcal{T})\right)$$

- Metropolis-Hastings algorithm: Sample from complicated distributions
    - Probabilities don’t need to be scaled!
    - Based on local decisions in the search space
        - Last rewrite R
        - Proposal rewrite R*
- Make a proposal from easy distribution *q*, accept or reject with probability



```
movq rsi, r0
movl ecx, ecx
shrq 32, rsi
andl 0xffffffff, r9d
...
```

```
movq rsi, r0
movl ecx, ecx
UNUSED
andl 0xffffffff, r9d
...
```

```
movq rsi, r0
movl ecx, ecx
andl 0xffffffff, r9d
shrq 32, rsi
...
```

```
movq rsi, r0
movl ecx, ecx
andl 0xffffffff, r9d
UNUSED
...
```

```
movq rsi, r0
movl ecx, ecx
andl 0xffffffff, r9d
shlq 32, rsi
...
```

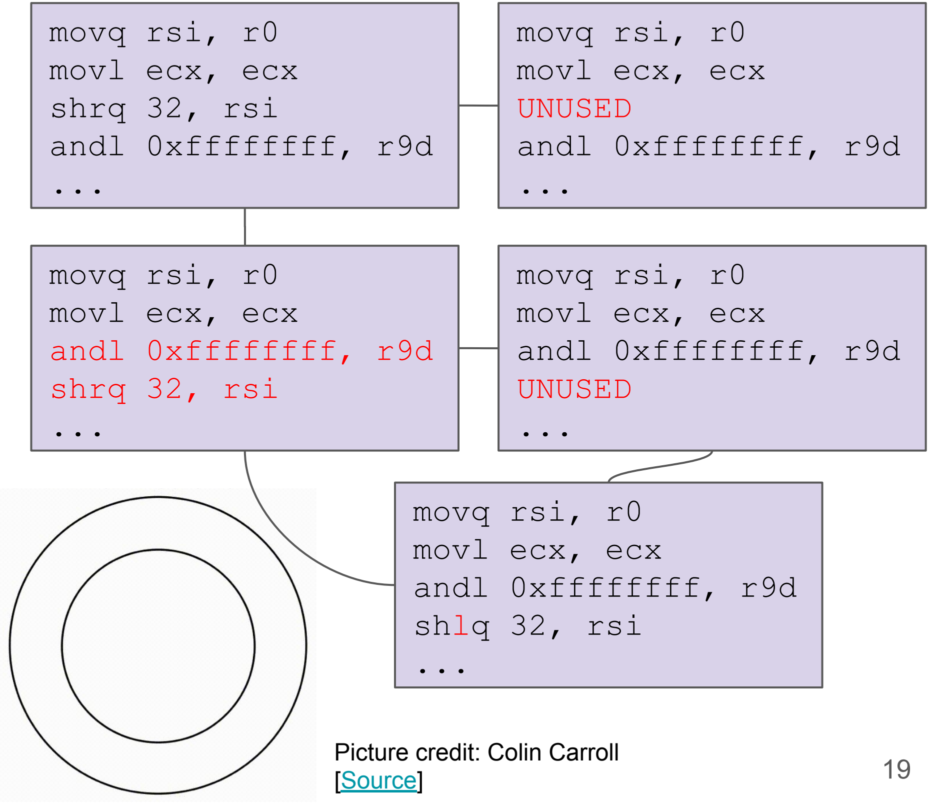
Picture credit: Colin Carroll
[Source]

Markov-Chain Monte-Carlo sampling

- Rewrite the cost as a probability distribution

$$p(\mathcal{R};\mathcal{T}) = \frac{1}{Z}\exp\left(-\beta\cdot\mathrm{cost}(\mathcal{R};\mathcal{T})\right)$$

- Metropolis-Hastings algorithm: Sample from complicated distributions
  - Probabilities don’t need to be scaled!
  - Based on local decisions in the search space
    - Last rewrite R
    - Proposal rewrite R*
- Make a proposal from easy distribution *q*, accept or reject with probability
  - *If cost(R*; T) < cost(R; T), always accept
  - Else, use the *Metropolis ratio* based on unscaled probabilities



```
movq rsi, r0
movl ecx, ecx
shrq 32, rsi
andl 0xffffffff, r9d
...
```

```
movq rsi, r0
movl ecx, ecx
UNUSED
andl 0xffffffff, r9d
...
```

```
movq rsi, r0
movl ecx, ecx
andl 0xffffffff, r9d
shrq 32, rsi
...
```

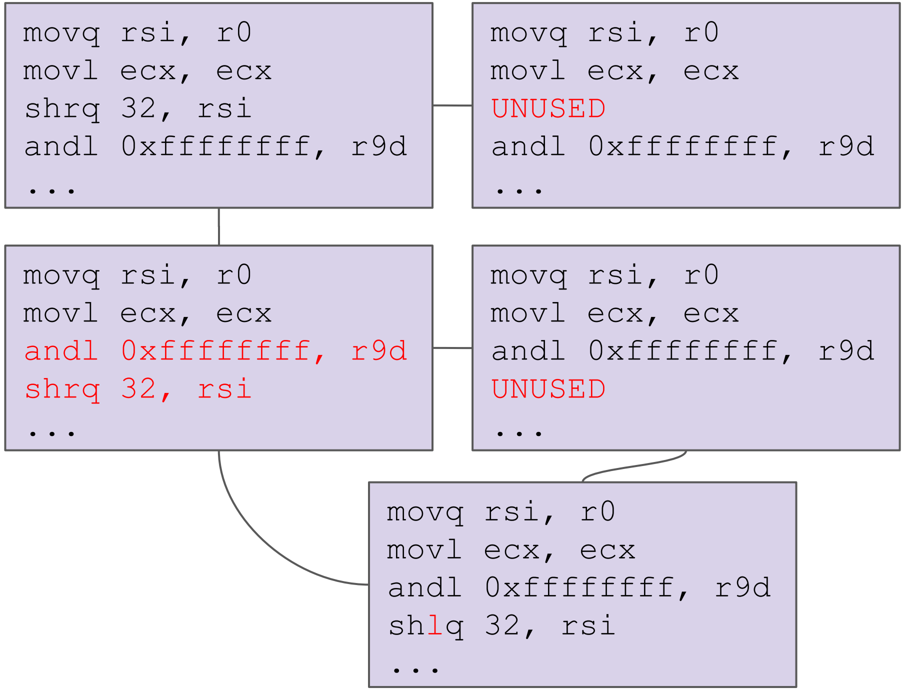
```
movq rsi, r0
movl ecx, ecx
andl 0xffffffff, r9d
UNUSED
...
```

```
movq rsi, r0
movl ecx, ecx
andl 0xffffffff, r9d
shlq 32, rsi
...
```

Picture credit: Colin Carroll
[Source]

Markov-Chain Monte-Carlo sampling

- In the paper: Synthesis of program from random initialization works well when incremental progress can be made to solution
  - Pretty wild that a program can be synthesized from scratch via sampling!



```
movq rsi, r0
movl ecx, ecx
shrq 32, rsi
andl 0xffffffff, r9d
...
```

```
movq rsi, r0
movl ecx, ecx
UNUSED
andl 0xffffffff, r9d
...
```

```
movq rsi, r0
movl ecx, ecx
andl 0xffffffff, r9d
shrq 32, rsi
...
```

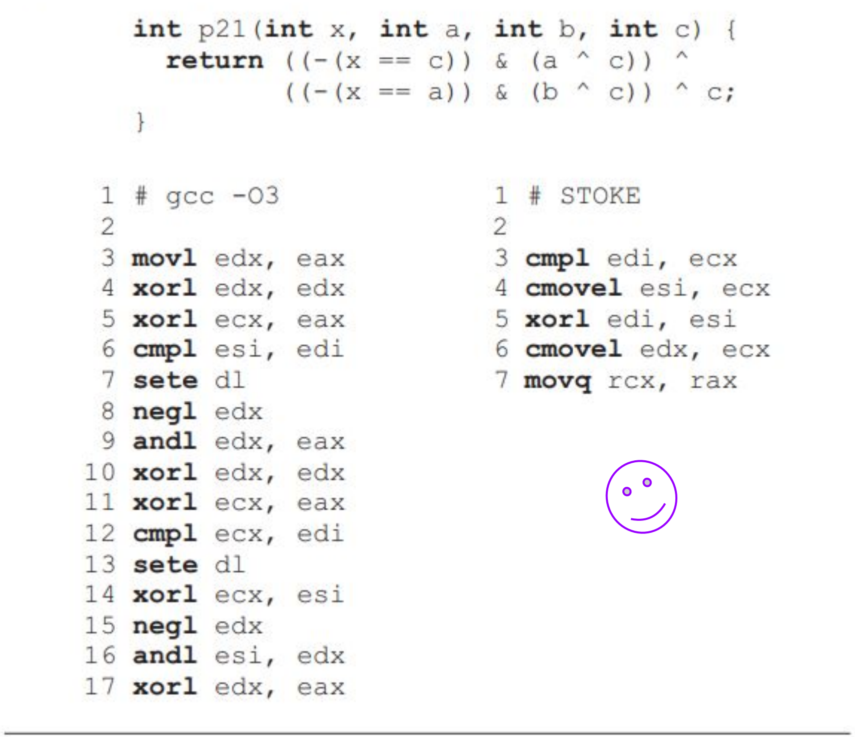
```
movq rsi, r0
movl ecx, ecx
andl 0xffffffff, r9d
UNUSED
...
```

```
movq rsi, r0
movl ecx, ecx
andl 0xffffffff, r9d
shlq 32, rsi
...
```

MACHINE INTELLIGENCE COMMUNITY

Results

Figure 9.Cycling Through 3 Values benchmark.



```
int p21(int x, int a, int b, int c) {
  return ((-(x == c)) & (a ^ c)) ^
         ((-(x == a)) & (b ^ c)) ^ c;
}
```

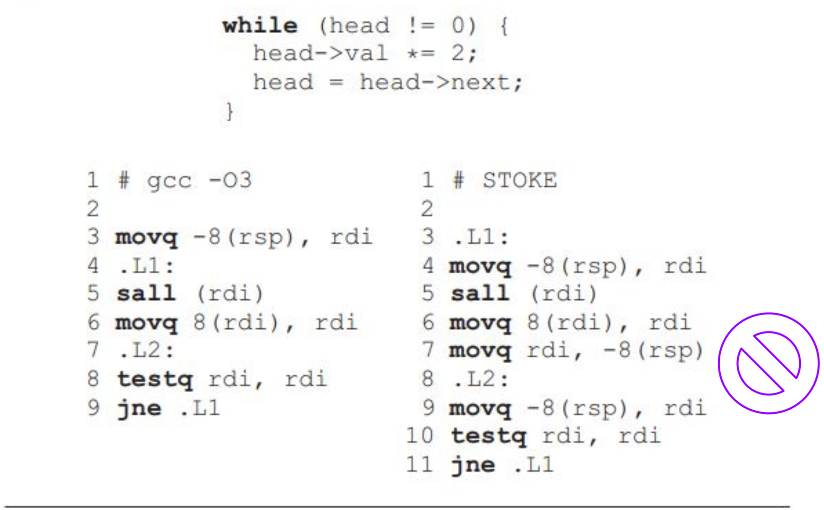
```
# gcc -O3

movl edx, eax
xorl edx, edx
xorl ecx, eax
cmpl esi, edi
sete dl
negl edx
andl edx, eax
xorl edx, edx
xorl ecx, eax
cmpl ecx, edi
sete dl
xorl ecx, esi
negl edx
andl esi, edx
xorl edx, eax
```

```
# STOKE

cmpl edi, ecx
cmovel esi, ecx
xorl edi, esi
cmovel edx, ecx
movq rcx, rax
```

Figure 11. Linked List Traversal benchmark.



```
while (head != 0) {
  head->val *= 2;
  head = head->next;
}
```

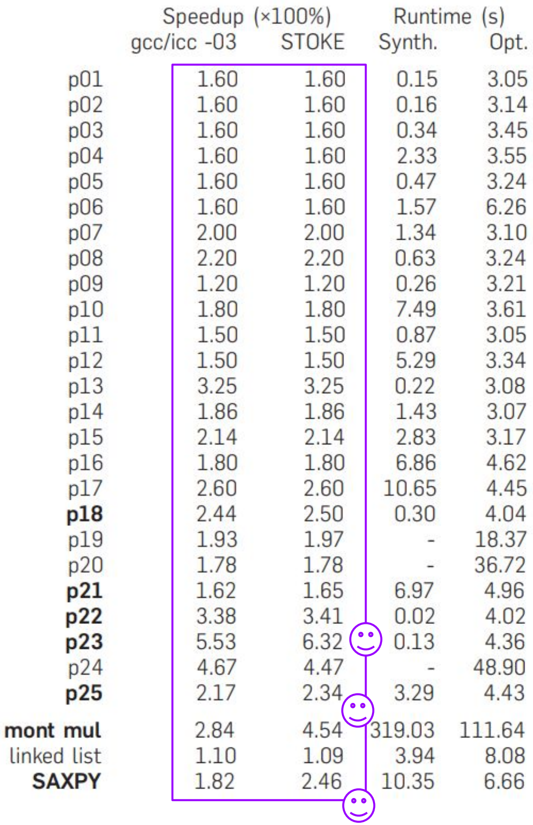
```
# gcc -O3

movq -8(rsp), rdi
.L1:
sall (rdi)
movq 8(rdi), rdi
.L2:
testq rdi, rdi
jne .L1
```

```
# STOKE

.L1:
movq -8(rsp), rdi
sall (rdi)
movq 8(rdi), rdi
movq rdi, -8(rsp)
.L2:
movq -8(rsp), rdi
testq rdi, rdi
jne .L1
```

Results



	Speedup (×100%)		Runtime (s)	
	gcc/icc -O3	STOKE	Synth.	Opt.
p01	1.60	1.60	0.15	3.05
p02	1.60	1.60	0.16	3.14
p03	1.60	1.60	0.34	3.45
p04	1.60	1.60	2.33	3.55
p05	1.60	1.60	0.47	3.24
p06	1.60	1.60	1.57	6.26
p07	2.00	2.00	1.34	3.10
p08	2.20	2.20	0.63	3.24
p09	1.20	1.20	0.26	3.21
p10	1.80	1.80	7.49	3.61
p11	1.50	1.50	0.87	3.05
p12	1.50	1.50	5.29	3.34
p13	3.25	3.25	0.22	3.08
p14	1.86	1.86	1.43	3.07
p15	2.14	2.14	2.83	3.17
p16	1.80	1.80	6.86	4.62
p17	2.60	2.60	10.65	4.45
p18	2.44	2.50	0.30	4.04
p19	1.93	1.97	-	18.37
p20	1.78	1.78	-	36.72
p21	1.62	1.65	6.97	4.96
p22	3.38	3.41	0.02	4.02
p23	5.53	6.32	0.13	4.36
p24	4.67	4.47	-	48.90
p25	2.17	2.34	3.29	4.43
mont mul	2.84	4.54	319.03	111.64
linked list	1.10	1.09	3.94	8.08
SAXPY	1.82	2.46	10.35	6.66

References & Further Reading

[1] http://stoke.stanford.edu/

[2] Stochastic Program Optimization – CACM 2016

[3] Inference and Metropolis-Hastings in Python: https://colcarroll.github.io/hamiltonian_monte_carlo_talk/bayes_talk.html