Superoptimization

Lecture 20

Montgomery (Modular) Multiplication

- Fast method for performing modular multiplication ab mod N
- Steps:
  - Transform a, b into *Montgomery form*
    - aR mod N for some R that depends only on N and the underlying architecture
  - Multiply Montgomery forms to get abR mod N
    - Montgomery multiplication is a fast algorithm for computing this Montgomery product
  - Transform abR mod N to classic ab mod N to get product

Montgomery (Modular) Multiplication

- Conversions take time, so using method to do one modular multiplication is slower than classic method
- BUT, when many multiplications are involved, then the conversion overhead becomes a negligible piece of the cost
- When are many multiplications involved?
  - RSA: modular exponentiation
  - Diffie-Hellman key exchange
  - In practice Montgomery multiplication used for these

LLVM

- Originally: Lower Level Virtual Machine
- Now collection of compiler technologies
  - Language agnostic tool for dynamic optimization
  - Originally a research project at U Illinois designed to study dynamic compilation techniques for static and dynamic languages
  - Now a large part of Apple development systems
  - Also front end (Clang) used by Sony in the PS4

Example: Montgomery Multiply from SSH

llvm -O0 (100 LOC) **gcc -O3** (29 LOC)

```
L0:
movq rdi, -8(rsp)
movq rsi, -16(rsp)
movl edx, -20(rsp)
movl ecx, -24(rsp)
movq r8, -32(rsp)
movq -16(rsp), rsi
movq rsi, -48(rsp)
movq -48(rsp), rsi
movabsq 0xffffffff, rdi
andq rsi, rdi
movq rdi, -40(rsp)
movq -48(rsp), rsi
shrq 32, rsi
movabsq 0xffffffff, rdi
andq rsi, rdi
movq rdi, -48(rsp)
movq -40(rsp), rsi
movq rsi, -72(rsp)
movq -48(rsp), rsi
movq rsi, -80(rsp)
movl -24(rsp), esi
imulq -72(rsp), rsi
movq rsi, -56(rsp)
movl -20(rsp), esi
imulq -72(rsp), rsi
movq rsi, -72(rsp)
movl -20(rsp), esi
...
```

```
.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
```

LOC is "lines of code"

Notes

- O3 is the highest level of optimization provided by gcc
  - And the slowest

- Does
  - Instruction scheduling
  - Register allocation
  - And many others …

Instruction Scheduling

- Modern processors are *pipelined*
  - Some instructions take more than one cycle
  - Have more than one instruction executing at the same time

load r1, 0(r2)	load r1, 0(r2)
addi r3, r1, 1	load r4, 0(r5)
load r4, 0(r5)	addi r3, r1, 1

- Bottom line: order of instructions matters

Register Allocation

- Assign registers to variables
  - Such that variables that are live simultaneously are in different registers

- Observation
  - Register allocation is sensitive to the live range of variables

Instruction Scheduling vs. Register Allocation

- Register allocation can add dependencies between instructions
  - Limits instruction scheduling

- Instruction scheduling can increase the live range of variables
  - Limits register allocation

- Which should be done first?

The Phase Ordering Problem

- Each optimization is a *phase*
- The *phase ordering problem* is selecting a best order for the optimizations to execute
- But there is no single best order for every application
  - Optimizations can interfere with one another

Phase Ordering

- Optimizing compilers have a *lot* of phases
- Each solves a problem in isolation
- But the solutions don't always compose well
  - Phases are ordered heuristically
  - Implies some optimizations are missed

Individual Phases are Limited, Too

- Phases try to capture the most important and easiest cases
  - Ignore the rest

- Common subexpression elimination
  - How complicated can two equivalent expressions be and still be recognized as equivalent?

Reprise

So how good is the code produced by gcc -O3?

STOKE

- Stochastic Superoptimizer and Program Sythesizer
  - Uses random search to "explore" the very large space of all possible program transformations
  - Repeatedly uses random sequences of millions of transformations
    - Kind of reminiscent in some ways of genetic algorithms
  - Produces "novel and non-obvious code sequences"
    - Code generally outperforms any other compiler, and often faster than the best hand-coded optimizations

Example: Montgomery Multiply from SSH

llvm -O0 (100 LOC)

```
L0:
movq rdi, -8(rsp)
movq rsi, -16(rsp)
movl edx, -20(rsp)
movl ecx, -24(rsp)
movq r8, -32(rsp)
movq -16(rsp), rsi
movq rsi, -48(rsp)
movq -48(rsp), rsi
movabsq 0xffffffff, rdi
andq rsi, rdi
movq rdi, -40(rsp)
movq -48(rsp), rsi
shrq 32, rsi
movabsq 0xffffffff, rdi
andq rsi, rdi
movq rdi, -48(rsp)
movq -40(rsp), rsi
movq rsi, -72(rsp)
movq -48(rsp), rsi
movq rsi, -80(rsp)
movl -24(rsp), esi
imulq -72(rsp), rsi
movq rsi, -56(rsp)
movl -20(rsp), esi
imulq -72(rsp), rsi
movq rsi, -72(rsp)
movl -20(rsp), esi
...
```

gcc -O3 (29 LOC)

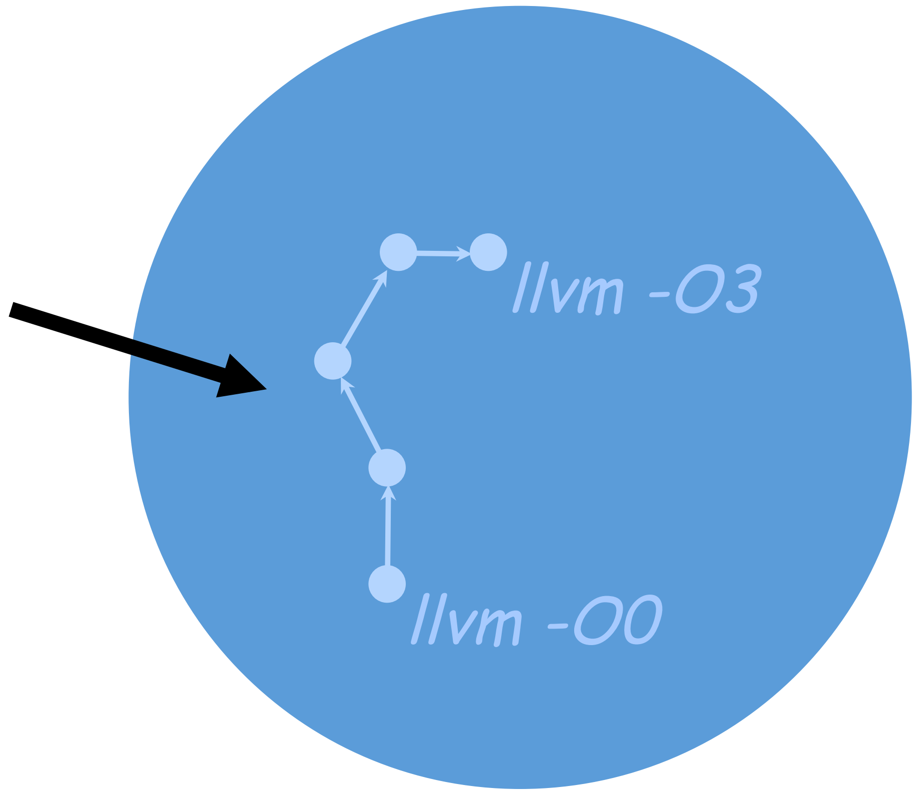
```
.L0:
movq rsi, r9
movl ecx, ecx
shrq 32, rsi
andl 0xffffffff, r9d
movq rcx, rax
movl edx, edx
imulq r9, rax
imulq rdx, r9
imulq rsi, rdx
imulq rsi, rcx
addq rdx, rax
jae .L2
movabsq 0x100000000, rdx
addq rdx, rcx
.L2:
movq rax, rsi
movq rax, rdx
shrq 32, rsi
salq 32, rdx
addq rsi, rcx
addq r9, rdx
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, rcx
movq rcx, rax
mulq rsi
addq r8, rdi
adcq 9, rdx
addq rdi, rax
adcq 0, rdx
movq rdx, r8
movq rax, rdi
```

A Picture

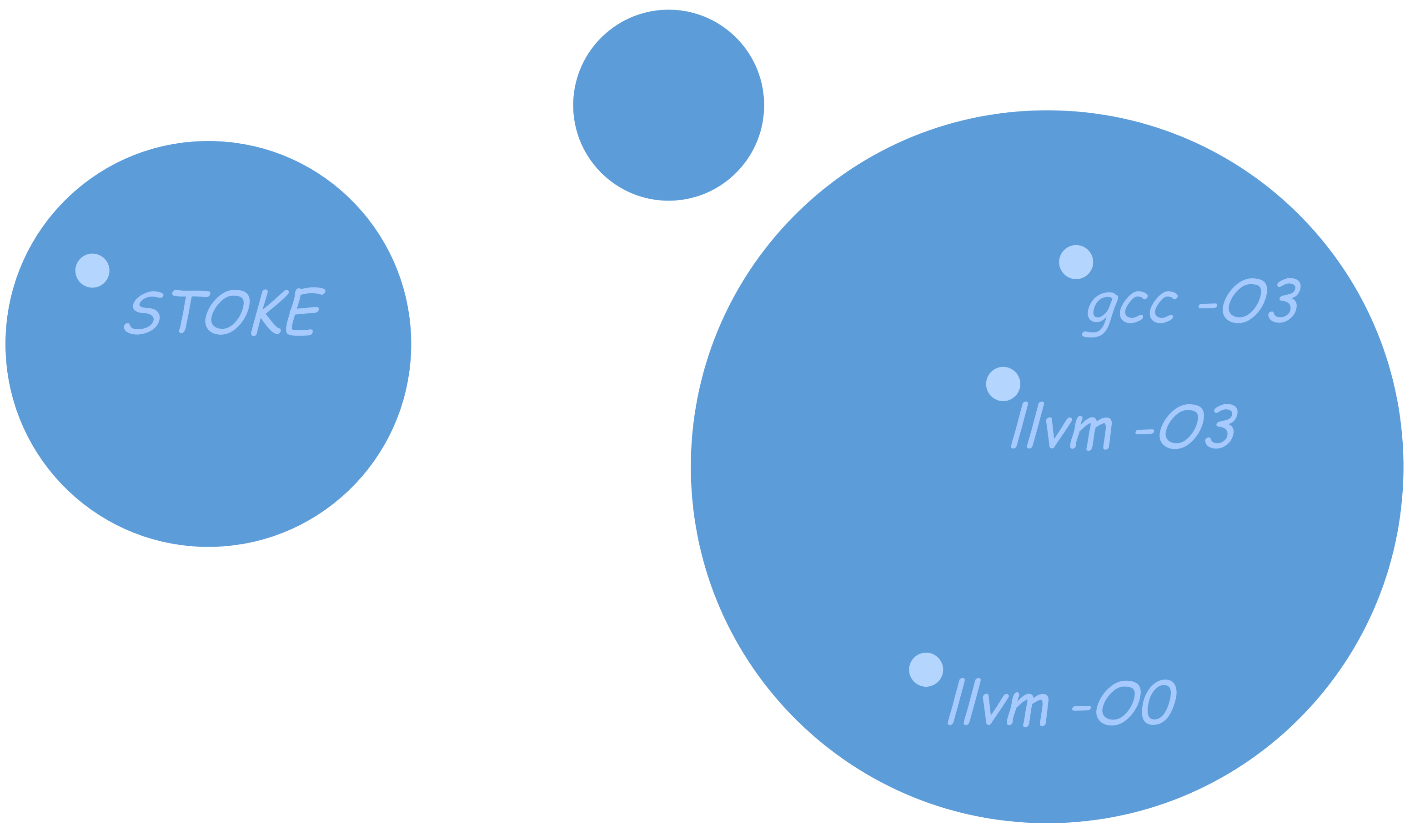
Region of equivalent programs



As in, all programs in the blue circle are equivalent

- **Traditional Compilers:** Consistently good, but not optimal

Another Picture



What Happened?

- Compilers are complex systems
  - Must find ways to decompose the problem

- Standard design
  - Identify optimization subproblems that are tractable (phases)
  - Try to cover all aspects with some phase

Why Do We Care?

- There are many systems where code performance matters
  - Compute-bound
  - Repeatedly executed

- Scientific computing
- Graphics
- Low-latency server code
- Encryption/decryption
- …

Montgomery Multiply, Revisited

- SSH does not use llvm or gcc for the Montgomery Multipy kernel
- SSH ships with a hand-written assembly MM kernel
- Which is slightly *worse* than the code produced by STOKE …

Another View

- Optimization is a search problem
  - Start with an initial program
  - Through a sequence of transformations find a better code

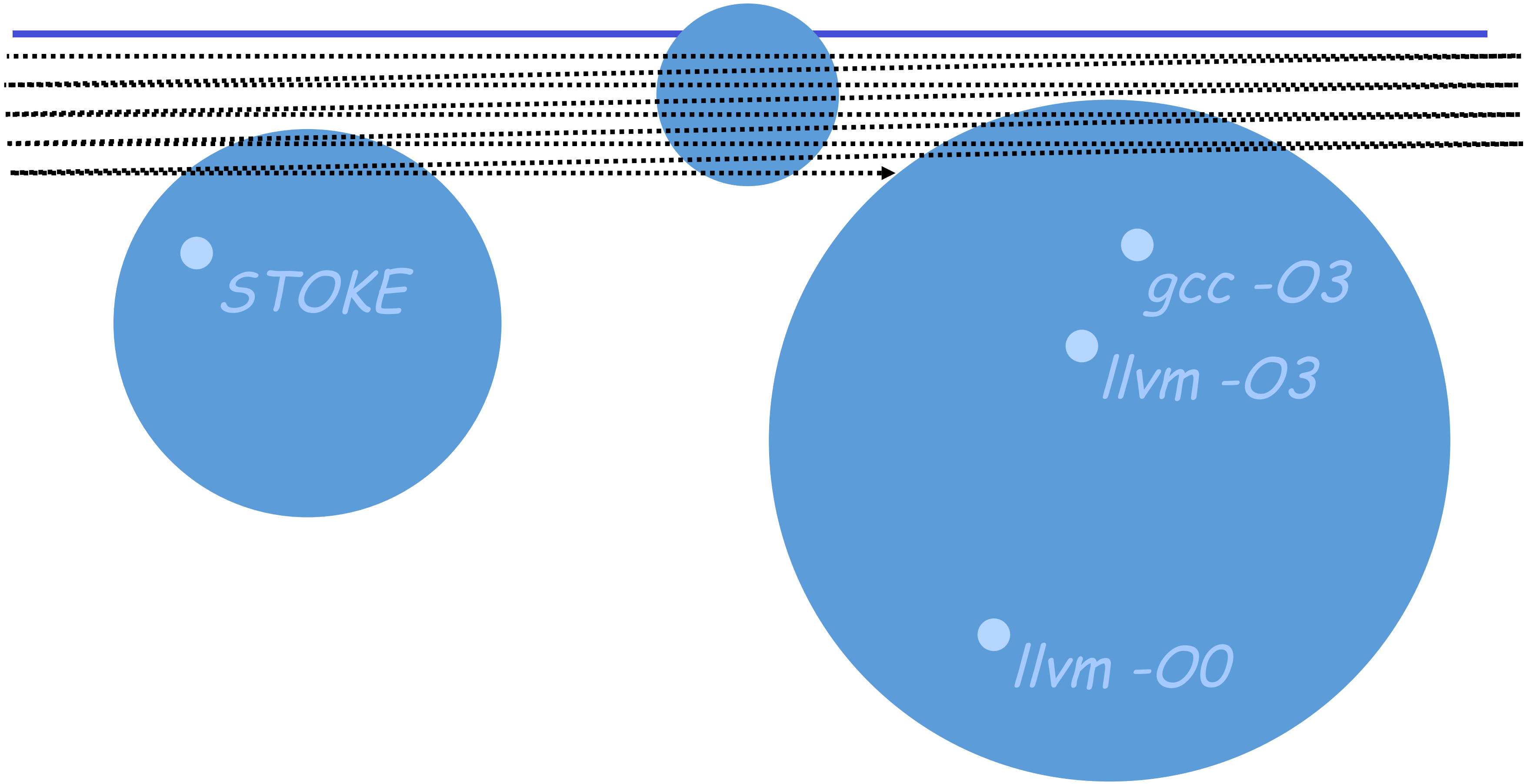
- So compilers solve a search problem
  - But don't do any search!

Superoptimization

- A family of techniques that perform optimization by searching over programs

- Why the awful name?
  - Because the term "optimization" was already taken
  - And we want to do better than "optimizing"

History



Bruteforce Enumeration

- Enumerate all programs, one at a time
  - Usually in order of increasing length

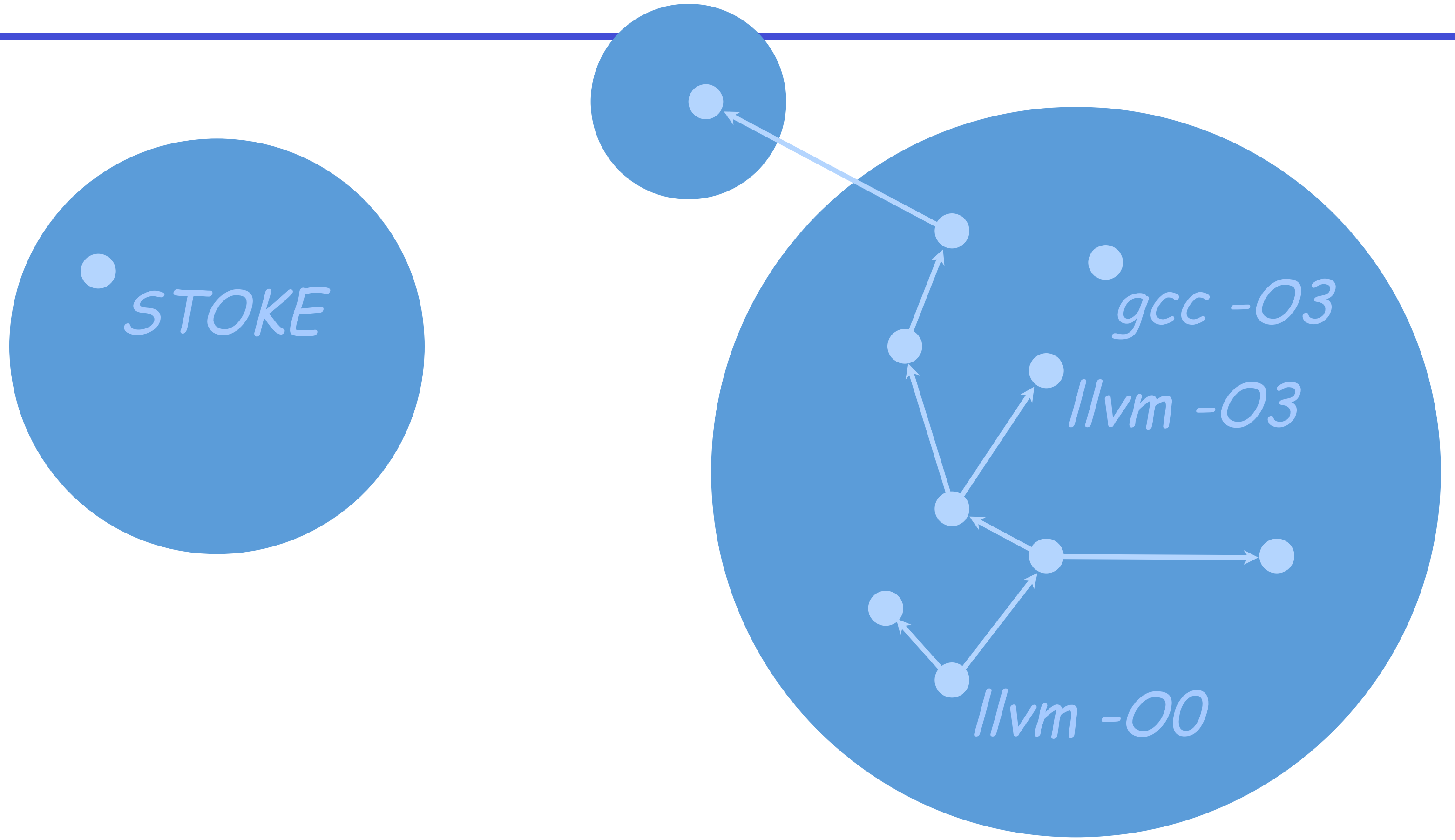
- [Massalin '87]
  - 10's of register instructions
  - could enumerate programs of length ~15

- [Bansal '06][Bansal '08]
  - Full x86 instruction set
  - Could enumerate programs of length ~3

Downsides

- Most enumerated programs are worthless
  - Not correct implementations of the program
- Enumeration is slow …

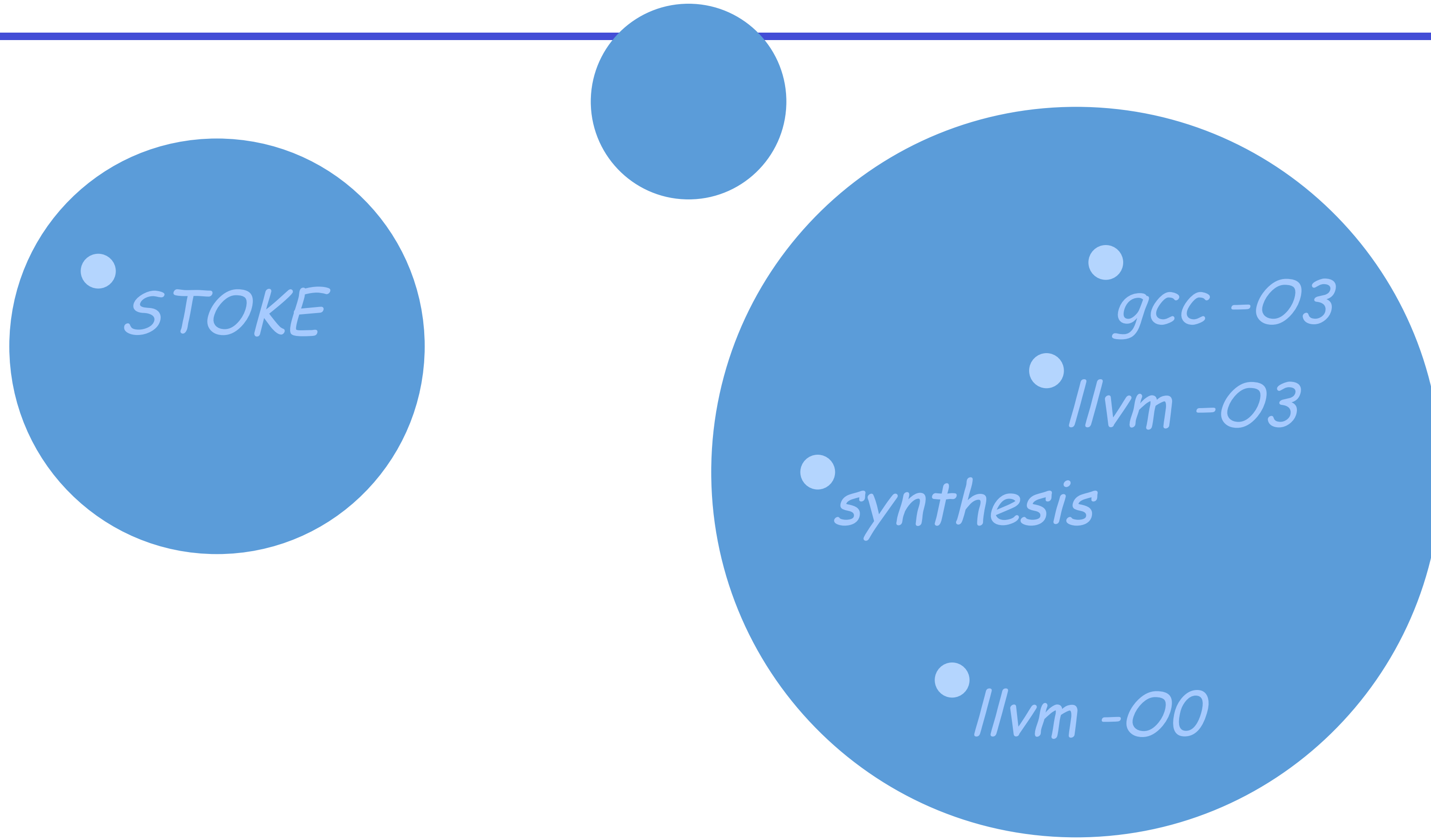
History



Equality Preserving Rules

- Expert-written rules for traversing the space of correct implementations
  - [Joshi '02][Tate '09]
- Problem
  - Someone has to write down all the possible equivalences of interest

History

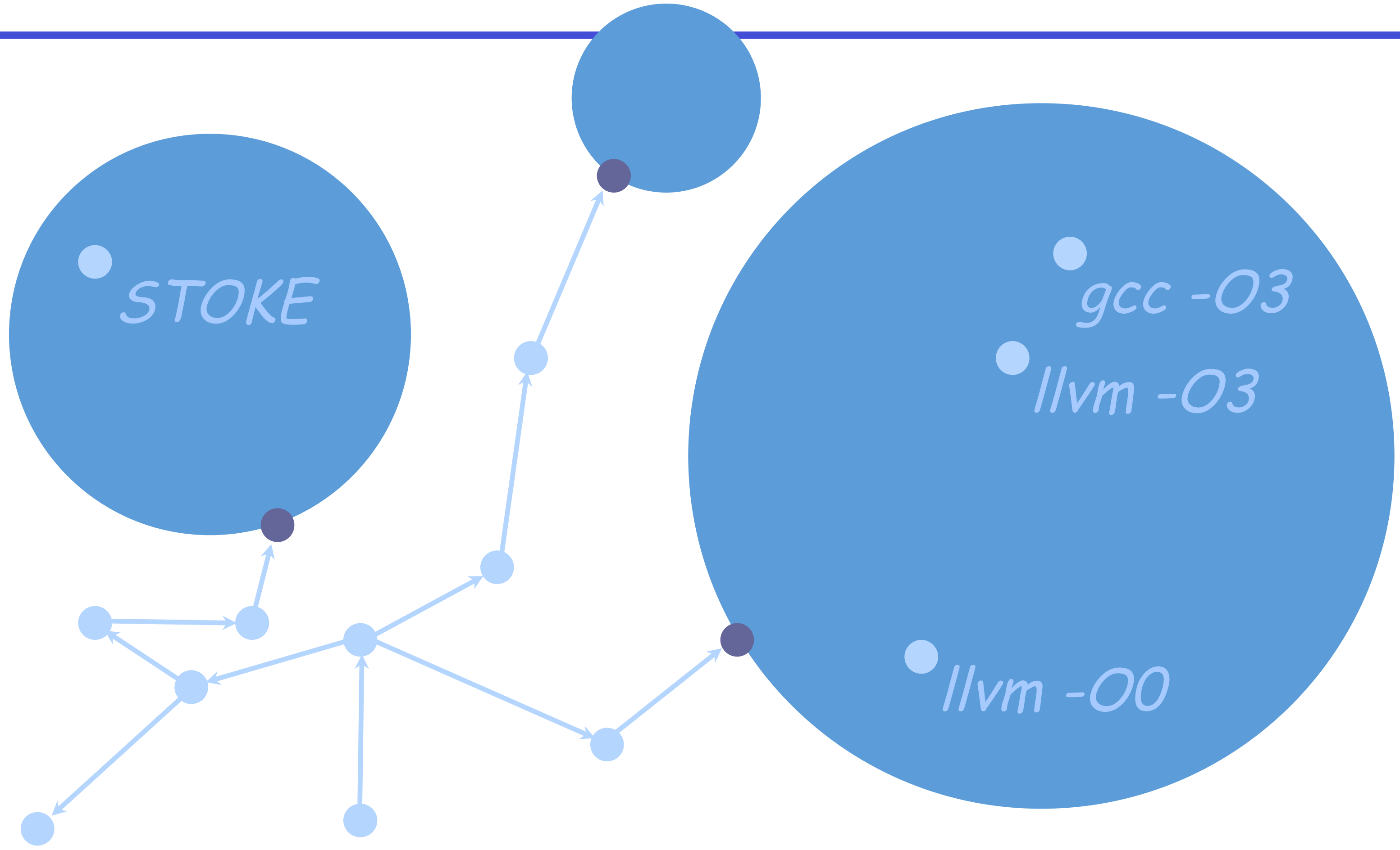


- **Program Synthesis:** Write constraints, produce one correct implementation [gulwani 11][solar-lezama 06][liang 10]

Step Back

- What if we were going to start over?
- What would a search-based optimizer look like?

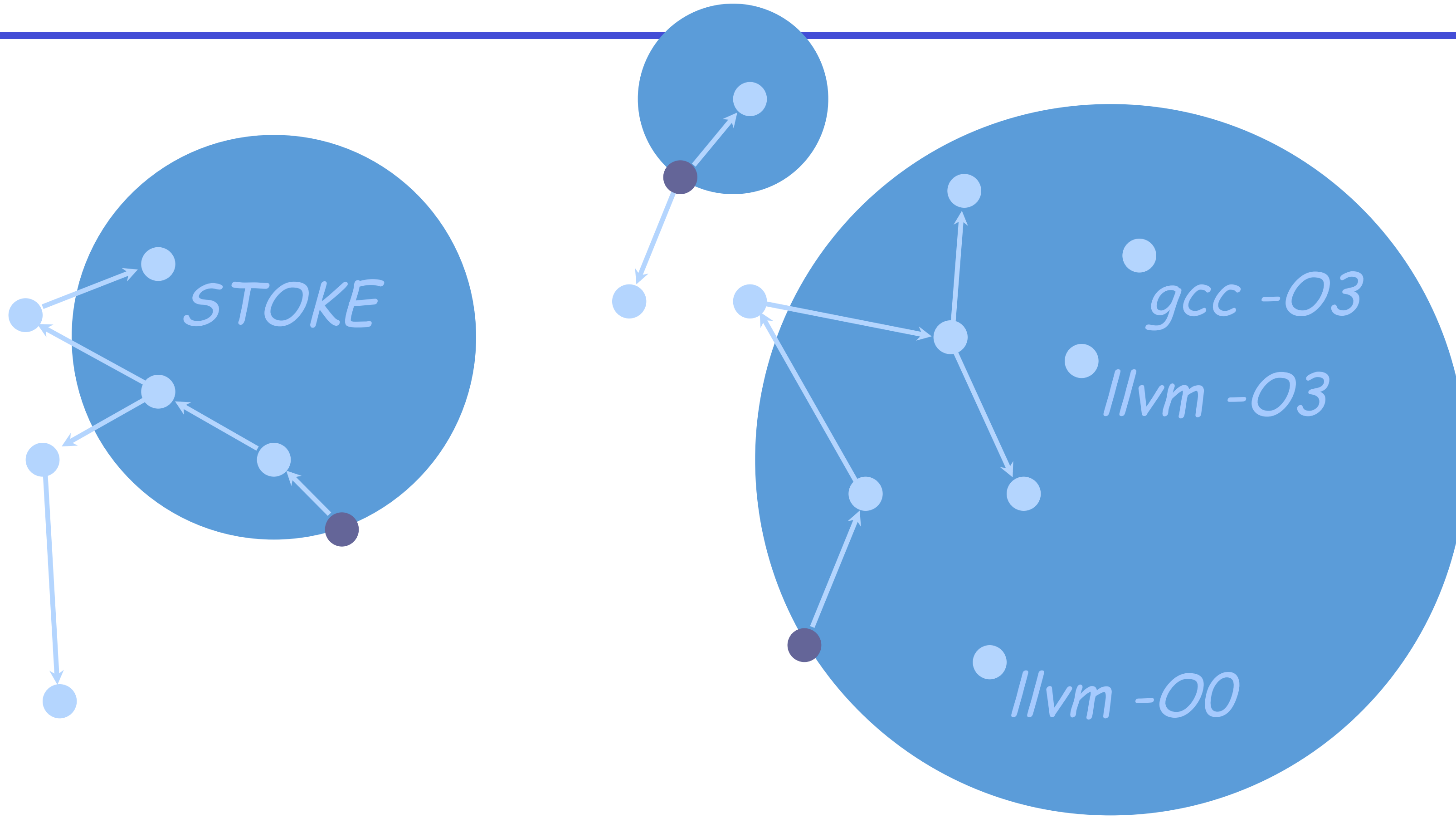
Stochastic Superoptimization



Randomized Search, Part I

- Begin at a random code
  - Somewhere in program space
- Make random moves
  - Looking for regions of correct implementation of the function of interest
  - The *target*

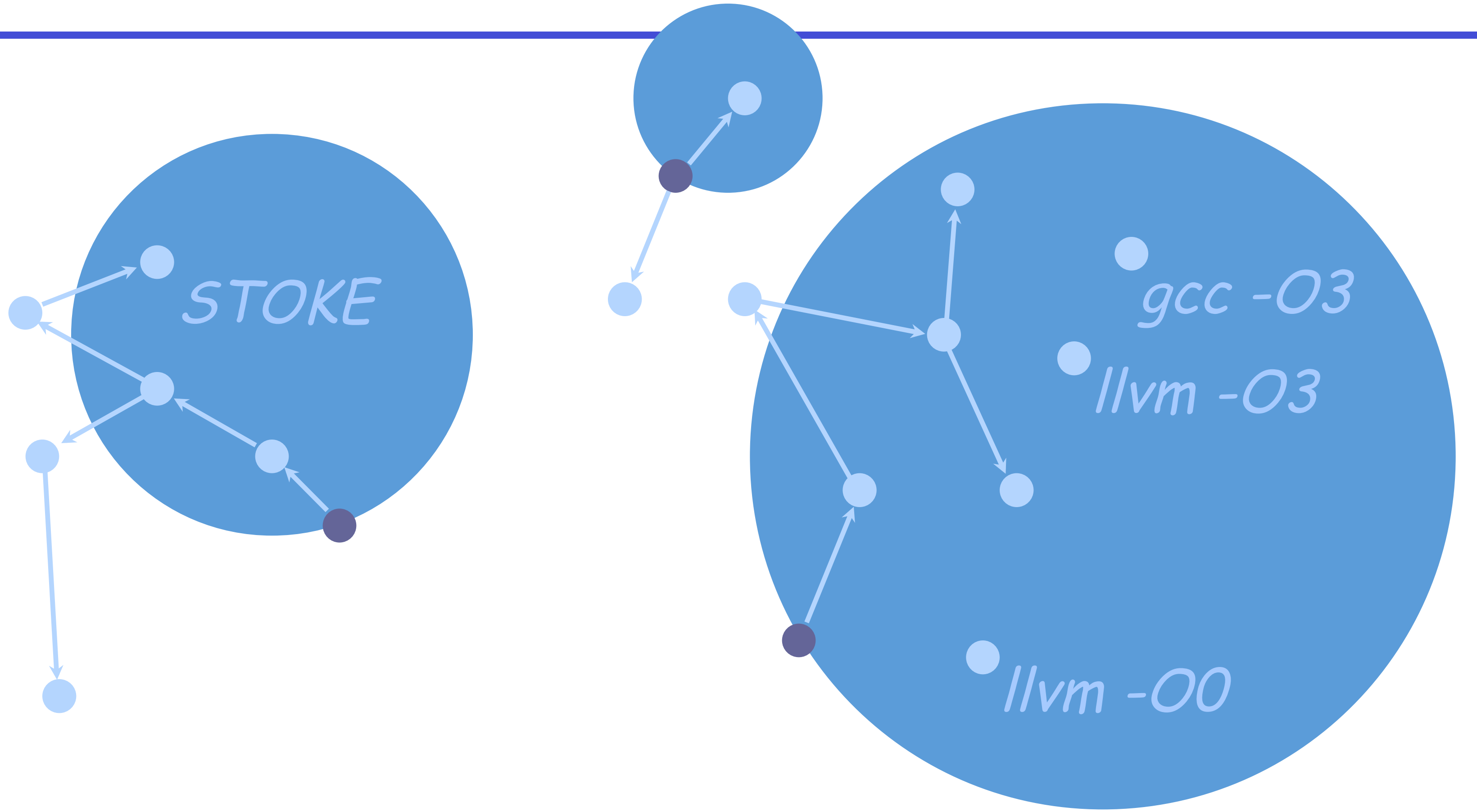
Stochastic Superoptimization



Randomized Search, Part II

- Run optimization threads for each correct program found
- Try to find more correct programs that run faster
  - Again by making randomized moves

Stochastic Superoptimization



- **Result:** A superoptimization technique that scales beyond all previous approaches to interesting real world kernels

What Do We Need?

- Search procedure
  - Program space too large for brute force enumeration
- Random search
  - Guaranteed not to get stuck
  - Might not find a nearby great program
- Hill climbing
  - Guaranteed to find the best program in the vicinity
  - Likely to get stuck in local minima

MCMC

- A compromise
  - Markov Chain Monte Carlo sampling
  - The only known tractable solution method for high dimensional irregular search spaces
  - [andrieu 03][chenney 00]

- Best of both worlds
  - An intelligent hill climbing method
  - Sometimes takes random steps out of local minima

MCMC Sampling Algorithm

1. Select an initial program

2. Repeat (billions of times)
   i. Propose a random modification and evaluate cost
   ii. If (cost decreased)
       { accept }
   i. If (cost increased)
       { with some probability accept anyway }

Technical Details

- **Ergodicity**
  - Random transformations should be sufficient to cover entire search space.
- **Symmetry**
  - Probability of transformation equals probability of undoing it
- **Throughput**
  - Runtime cost to propose and evaluate should be minimal

Theoretical Properties

- Limiting behavior
  - Guaranteed in the limit to examine every point in the space at least once
  - Will spend the most time in and around the best points in the space

Transformations

- Simple
  - No expert knowledge
- Balance between “coarse” and “fine” moves
  - Experience with MCMC suggests successful applications need both

Transformations

- **original**

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

Transformations

- **original**

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

insert

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

Transformations

- **original**

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

insert

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

delete

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

Transformations

- **original**

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

insert

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

delete

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

instruction

```
...
movl ecx, ecx
shrq 32, rsi
salq 16, rcx
movq rcx, rax
movl edx, edx
imulq r9, rax
...
```

Transformations

- **original**

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

insert

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

opcode

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

delete

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

instruction

```
...
movl ecx, ecx
shrq 32, rsi
salq 16, rcx
movq rcx, rax
movl edx, edx
imulq r9, rax
...
```

Transformations

- **original**

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

insert

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

opcode

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

delete

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

instruction

```
...
movl ecx, ecx
shrq 32, rsi
salq 16, rcx
movq rcx, rax
movl edx, edx
imulq r9, rax
...
```

operand

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

Transformations

- **original**
- ...
- movl ecx, ecx
- shrq 32, rsi
- andl 0xffffffff, r9d
- movq rcx, rax
- movl edx, edx
- imulq r9, rax
- ...

insert

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

opcode

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

delete

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

instruction

...
movl ecx, ecx
shrq 32, rsi
salq 16, rcx
movq rcx, rax
movl edx, edx
imulq r9, rax
...

swap

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

operand

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

The Secret Sauce: The Cost Function

- Measures the quality of a *rewrite* with respect to the *target*
  - **Synthesis:** cost(r; t) = eq(r; t)
  - **Optimization:** cost(r; t) = eq(r; t) + perf(r; t)

- Lower cost codes should be better codes
  - Better cost functions -> better results

Engineering Constraints

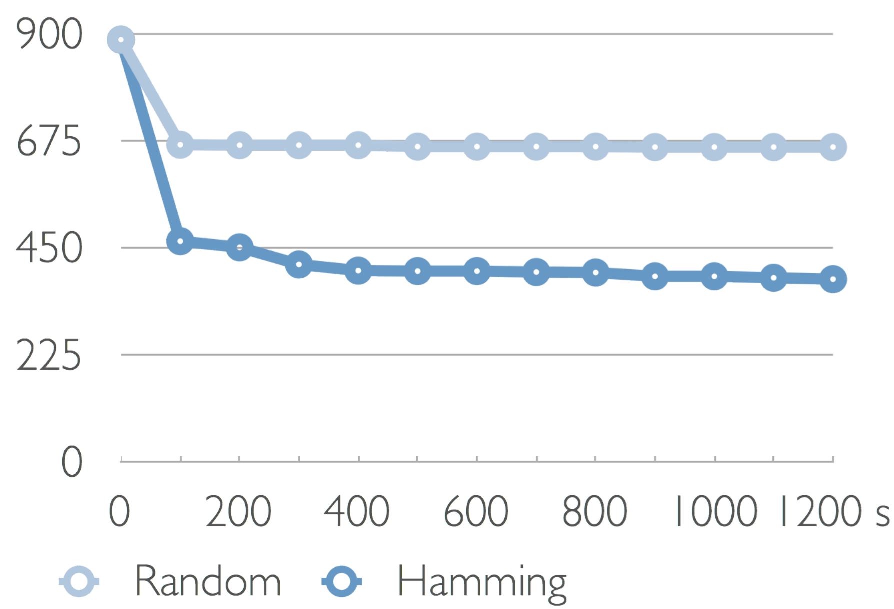
- The cost function needs to be inexpensive
  - Because we will be evaluating it billions of times

- Idea: Use test cases
  - Compare output of target and rewrite on small set of test inputs
  - Typically 16

Cost Function, Version One

- Hamming Distance
  - Of output of target and rewrite of test cases
  - # of bits where they disagree
  - Provides useful notion of partial correctness

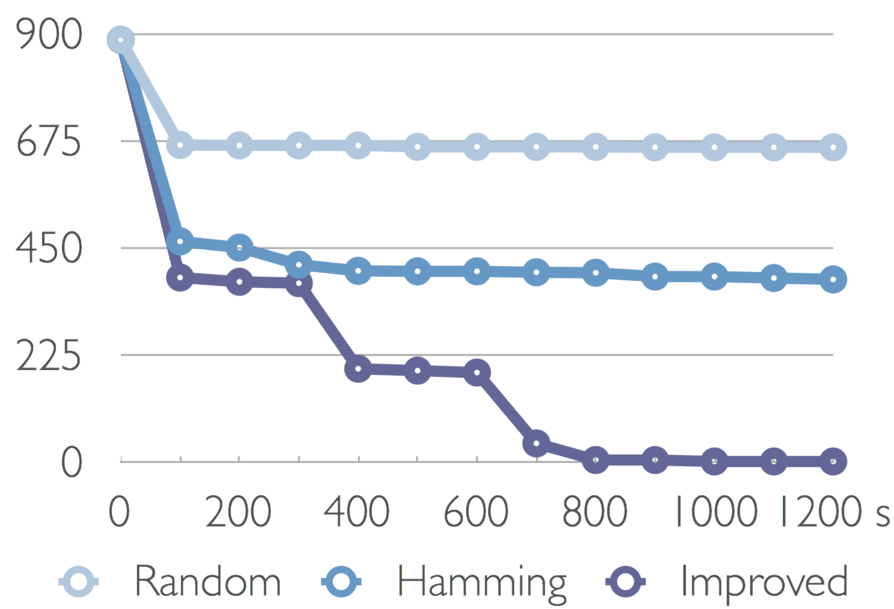
	ax	*bx*	*cx*	*dx*
T	*1111*	*0000*	*0000*	*0000*
R	*0010*	*1000*	*1100*	*1111*
	3			



Cost Function, Version Two

- Reward the right answer in the wrong place
- For each output value of the target, Hamming distance to closest matching output of the rewrite

	ax	*bx*	*cx*	*dx*
T	1111	0000	0000	0000
R	0010	1000	1100	1111
min (	*3 + 0*	*3 + 1*	*2 + 1*	*0 + 1)*
				1



Correctness and Optimization

- Measuring correctness
  - Hamming distance on outputs
  - Plus: Fast!
  - Minus: Matching a few test cases doesn't guarantee rewrite is correct

- Next: Performance

Performance Metric

- Latency Approximation
  - Approximate the runtime of a program by summing the average latencies of its instructions
- Positive
  - Fast!
- Negative
  - Gross oversimplification
  - Ignores almost all the interesting architectural details of a modern CISC machine

Doing It Right

- Both the correctness and performance metrics are fast to compute
  - But both are also approximations

- Want to guarantee
  - We get a correct program
  - We get the fastest program we find

- Observation
  - These checks can be more expensive if we don't do them for every rewrite

Formal Correctness

- Prove formally that target = rewrite
  - For all inputs
  - Can be done using a *theorem prover*
- Encode target and rewrite as logical formulas
  - Compare the formulas for equality
  - Equal formulas => Equal programs
  - If formulas are not equal, theorem prover produces a counterexample input

Theorem Prover Example

Target:
neg %eax

Rewrite:
movq 0xffffffff, %eax

- Target negates register %eax
- Rewrite fills %eax with ones
- Why?
  - Maybe we only have a single testcase with %eax equal to zero

Theorem Prover Example

Target:
neg %eax

Rewrite:
movq 0xffffffff, %eax

$eax_o[31] = \sim eax_i[31]$ &
$eax_o[30] = \sim eax_i[30]$ &
… &
$eax_o[0] = \sim eax_i[0]$

$eax'_o[31] = 1$ &
$eax'_o[30] = 1$ &
… &
$eax'_o[0] = 1$

- Define variables for the bits of the machine state after every instruction executes
- Write formulae describing the effects produced by every instruction

Counterexample

Target:
neg %eax

Rewrite:
movq 0xffffffff, %eax

eax_i = 0xffffffff
eax_o = 0x00000000

eax'_i = 0xffffffff
eax'_o = 0xffffffff

- A theorem prover will discover these codes are different
- And produce an example input proving they are different

Theorem Prover Example

- If theorem prover succeeds, the two programs are guaranteed to be equivalent
- If the theorem prover fails, it produces a counterexample input
  - Can be added to the test suite and the search procedure repeated

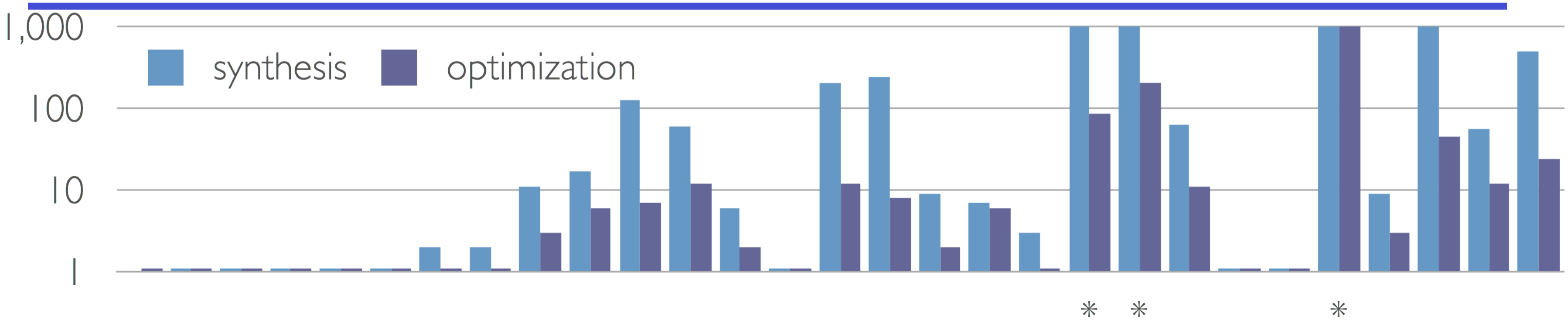
Performance Guarantee

- Assemble and run rewrite on inputs
  - And measure the results
  - But this is too expensive to do all the time
- Idea: Preserve the top-n most performant results
  - rerank based on actual runtime behavior

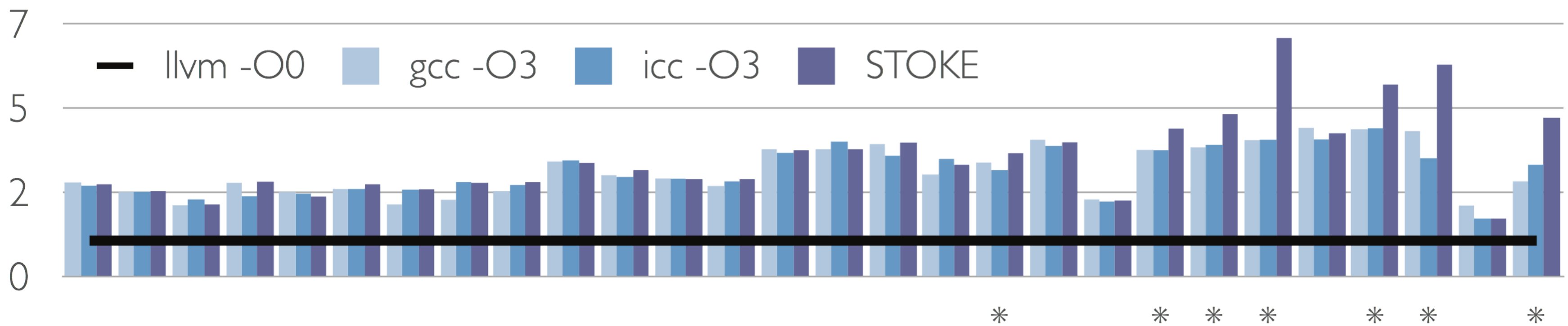
Benchmarks

- **Synthesis Kernels:** 25 loop-free kernels taken from A Hacker's Delight [gulwani 11]
- **Real World:** OpenSSL 128-bit integer multiplication montgomery multiplication kernel
- **Vector Intrinsics:** BLAS Level 1 SAXPY
- **Heap Modifying:** Linked List Traversal [bansal 06]

Benchmarks



Runtime



Speedup

- **Experiments:** Target codes compiled using llvm -O0, STOKE matches or outperforms gcc and icc with full optimizations

Limitations

- All of these experiments are on loop-free kernels
  - But extending the approach to loops is possible
- All of these experiments are on fixed point values
  - Need to extend to floating point as well

Conclusions

- Search-based techniques can generate much better code!
- Very different basis from current optimizing compilers
  - Perform real search
  - Alow experimentation with incorrect but fast code

Thanks!