Stochastic Optimization of x86-64 Binaries

eric schkufza
stanford university

Overview

- **Optimizing compute bound kernels is a pain:** Once data movement is orchestrated correctly, many interesting apps still compute bound
- **Examples:** Simulation, Cryptography, Rendering…
- **Getting the best performance:** Requires full exploitation of the dark corners of an instruction set and how it interacts with machine resources

Overview

- **Stochastic Optimization:** Automated method for generating high-performance kernels that match or outperform expert handwritten alternatives

Outline

- Loop-free fixed-point optimization [asplos 13]
  - **Intuitions**
  - Technical Detail / Evaluation
- Extensions to floating-point [pldi 14]
  - Intuitions
  - Technical Detail / Evaluation

Example

```
#include “mont_mul.h”

void mont_mul(/* … */) {
  // …
  // …
  // …
  // …
  // …
}
```

gcc -O3

```
      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

```
.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
```

Optimization Notes

- **Smaller kernel:** 29 LOC reduced to 11 LOC, no control flow divergence

- **Performance improvement:** 60% speedup over gcc -O3, slightly faster than handwritten assembly

- **Highly specialized:** Reorganizes input data at the bit-level to take advantage of hardware intrinsics

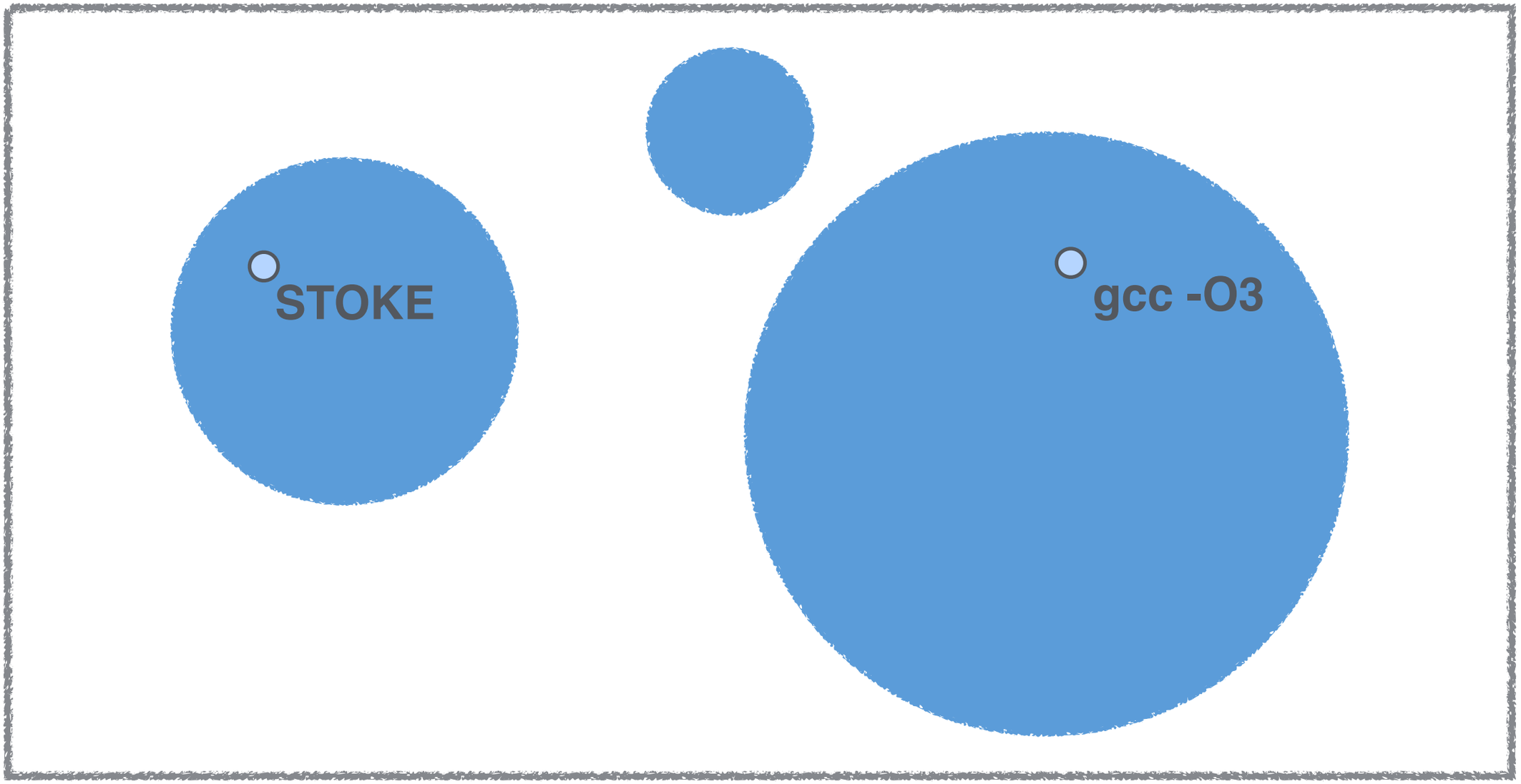
Intuition

Abstract space of all programs (*rewrites)*: Extremely high-dimensional

Intuition

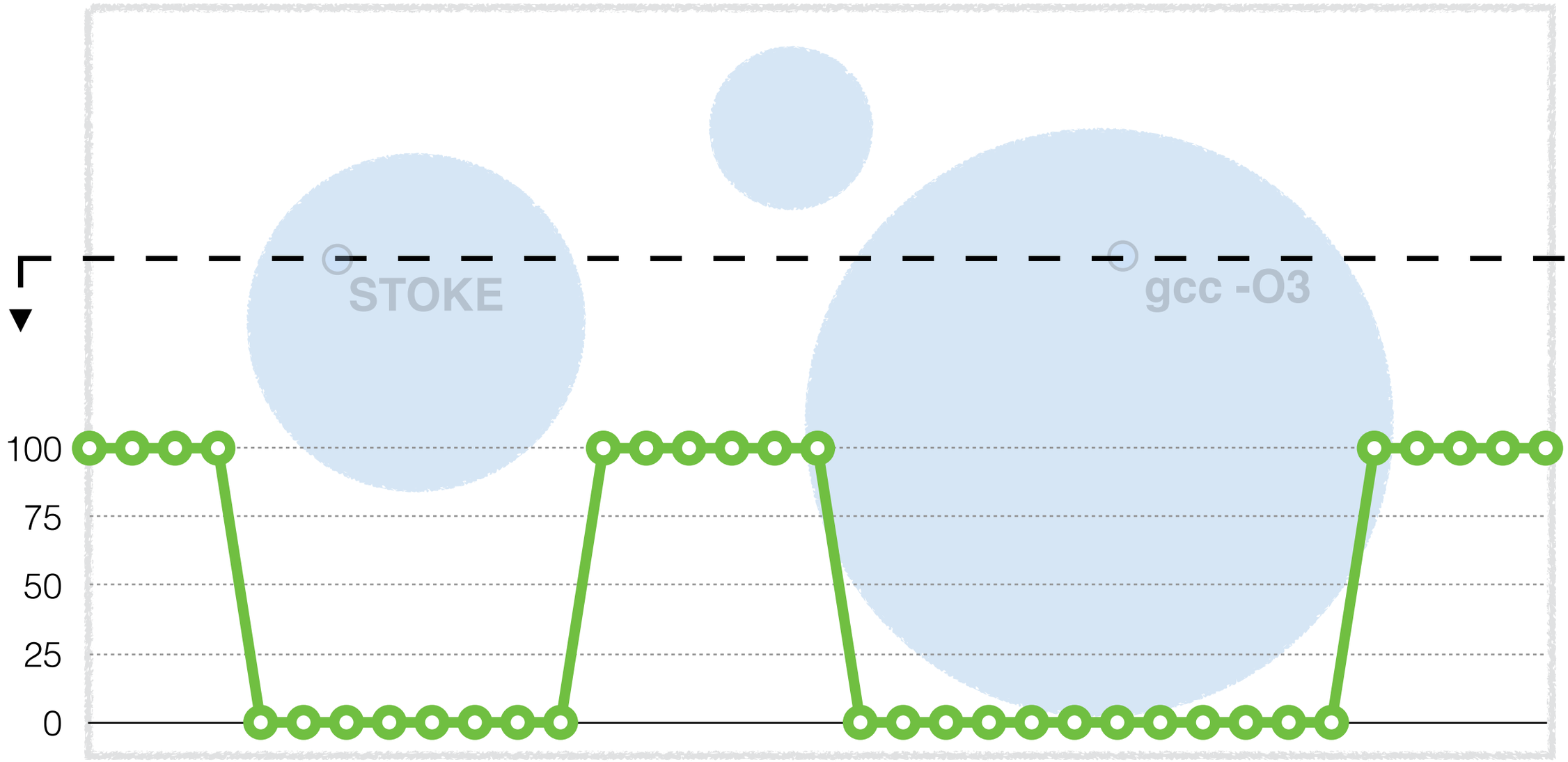
Input Program (*target*): A distinguished point in this space

Intuition



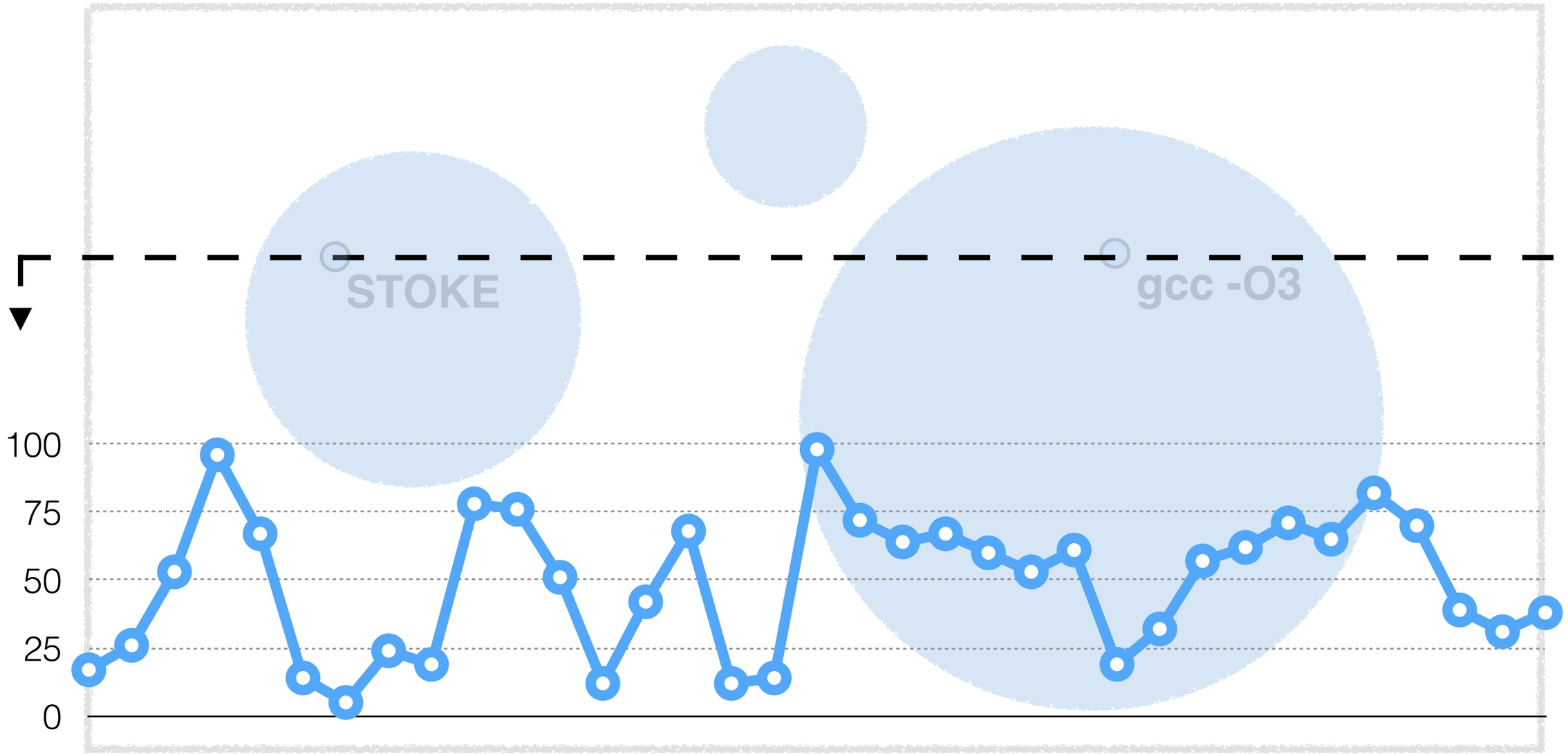
Correctness: Implicit partition separates rewrites into two disjoint classes: correct (blue) and incorrect (white)

Intuition



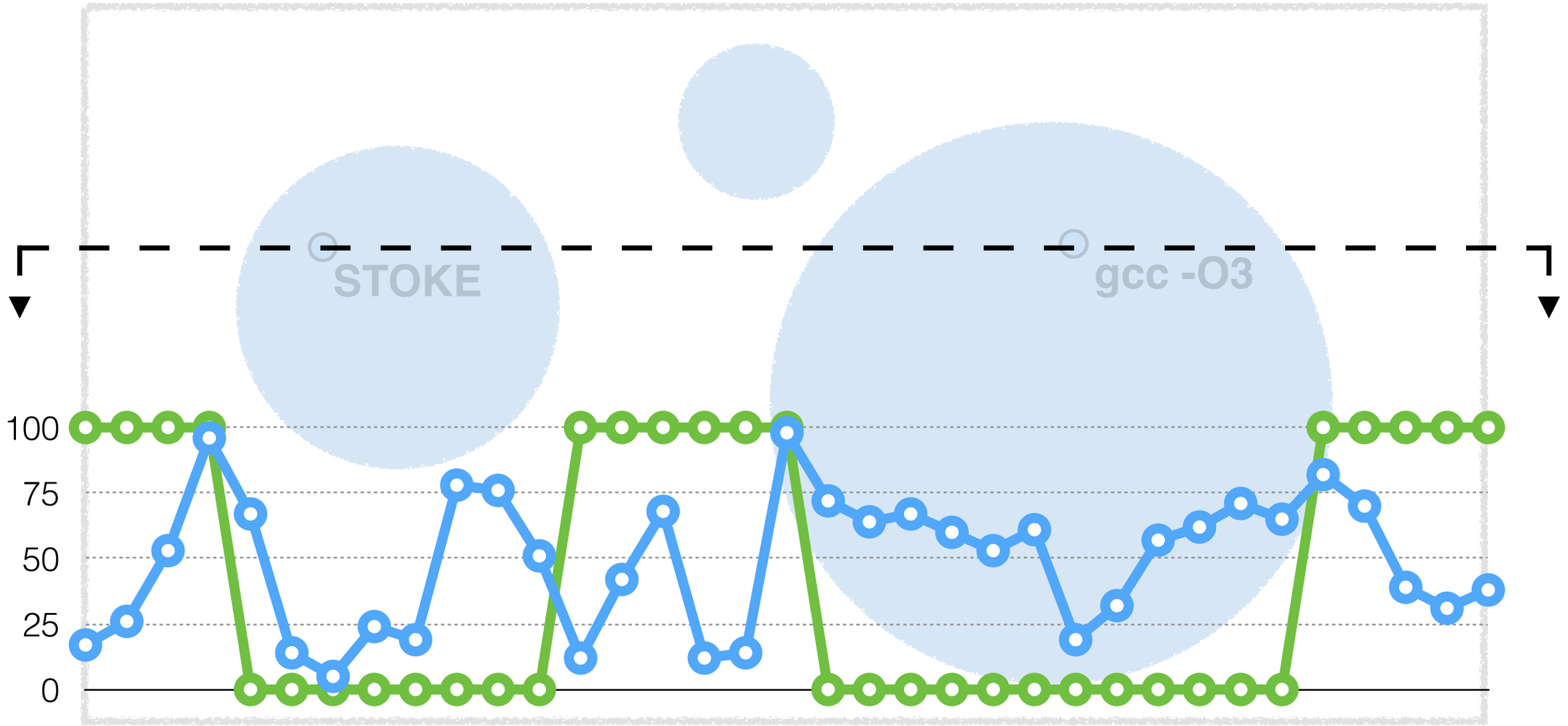
Correctness: Equivalent to a binary indicator function

Intuition



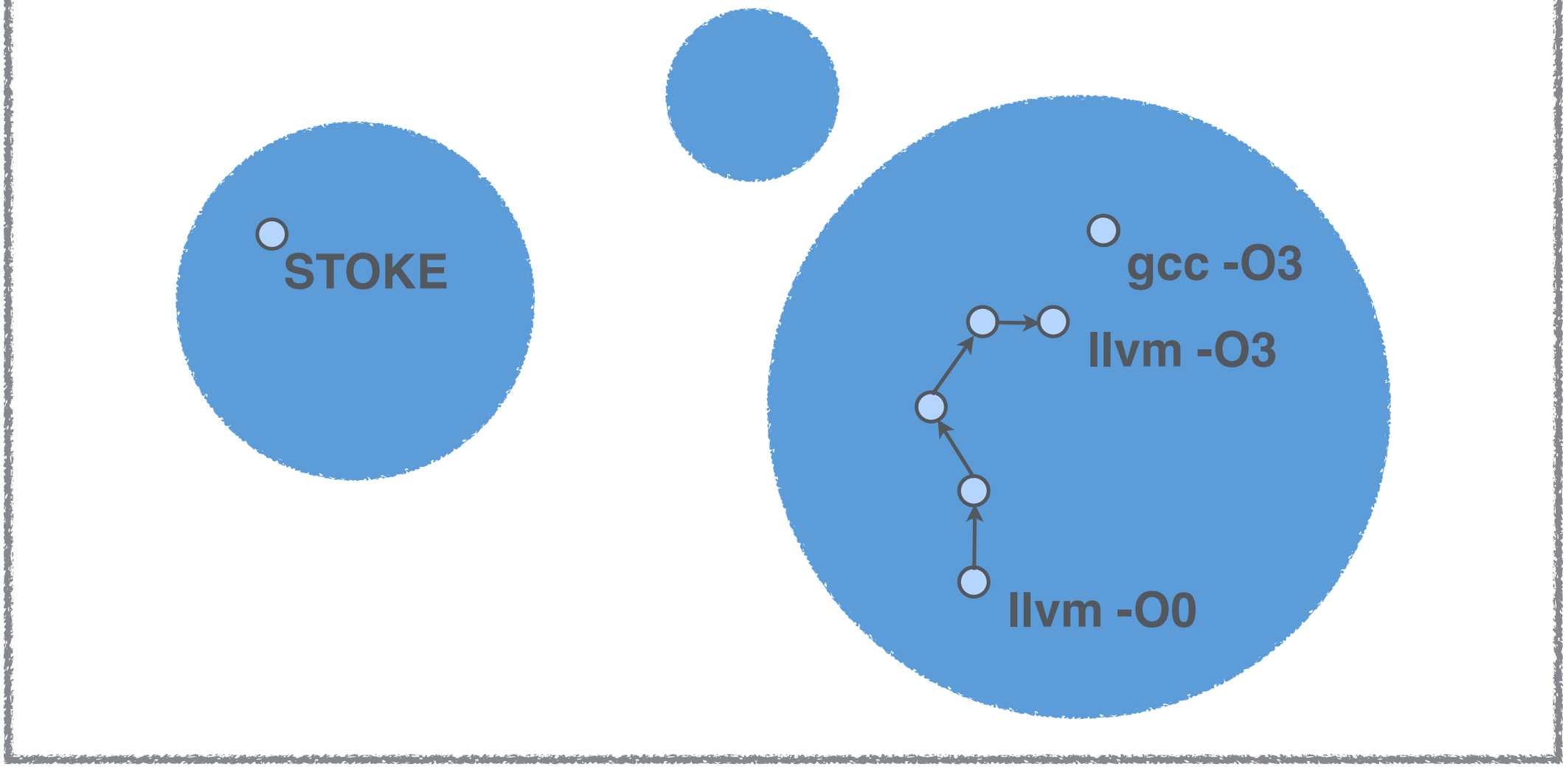
Performance: An orthogonal cost function quantifies the expected quality of a rewrite (lower is better)

Intuition



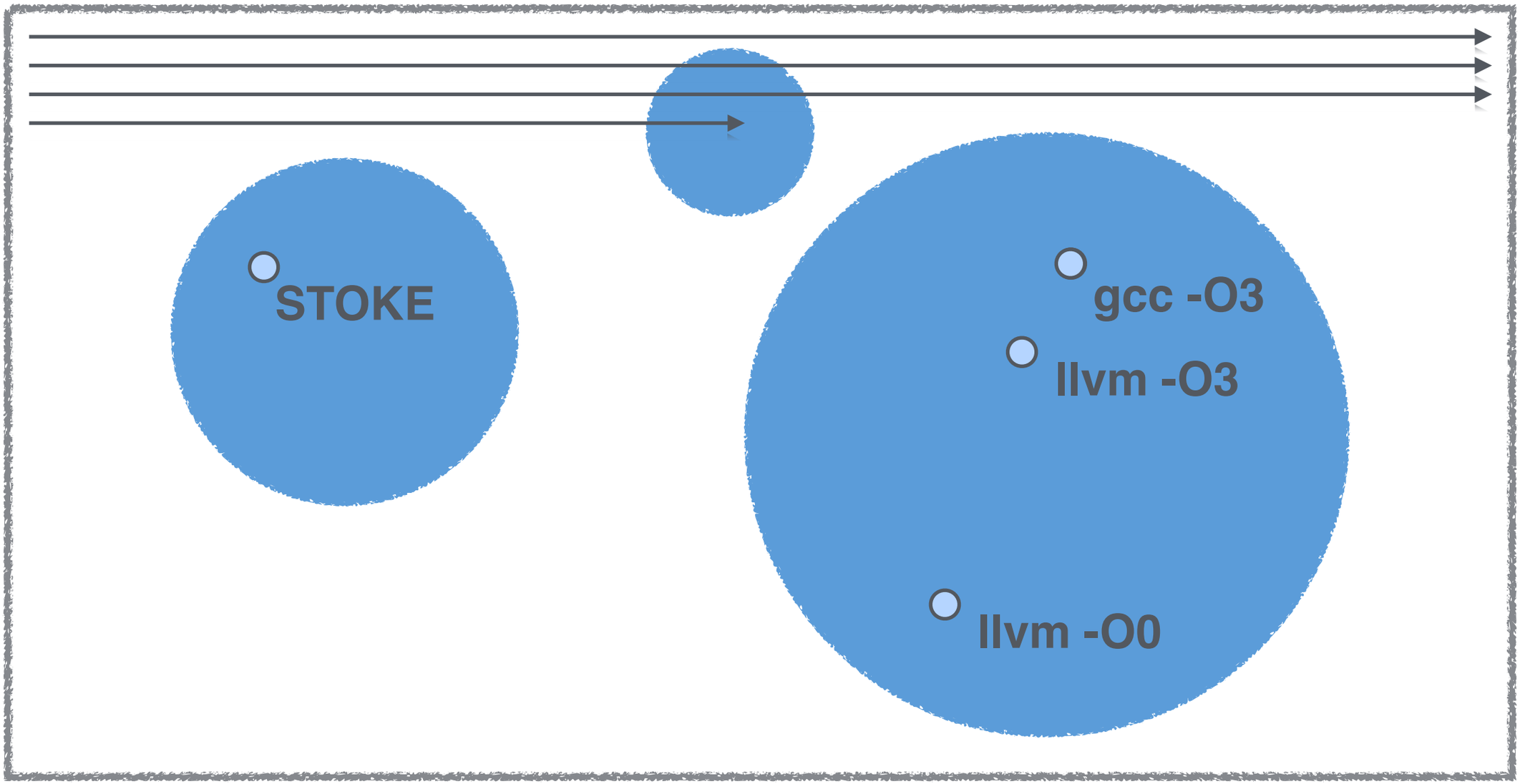
Optimization is a search problem: Find the lowest cost (most performant) code that is also correct

Related Work



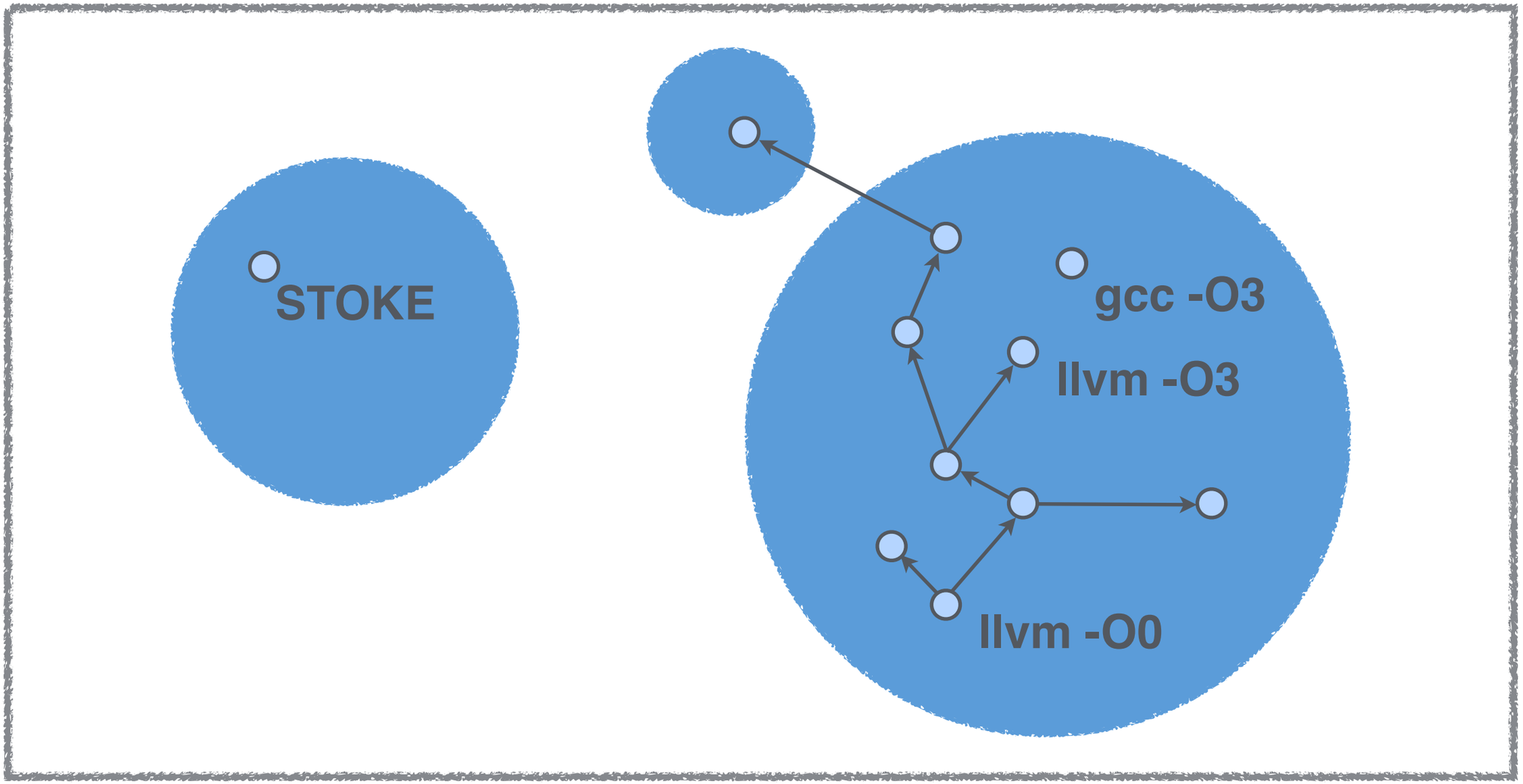
Traditional compilers: Designed to produce code that is consistently very good, but not necessarily optimal

Related Work



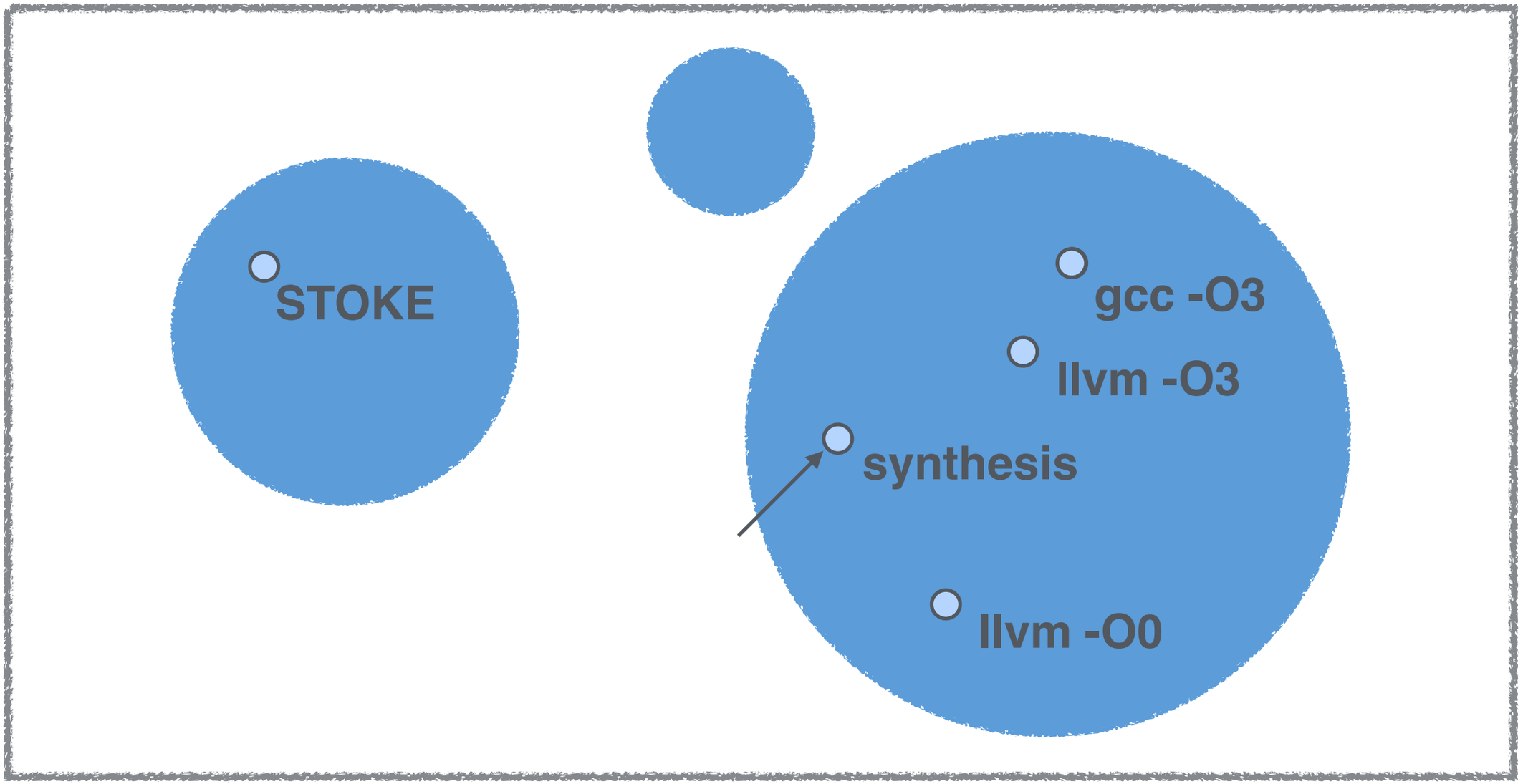
Bruteforce enumeration: Guess and check all possible implementations [massalin 87] [bansal 06/08]

Related Work



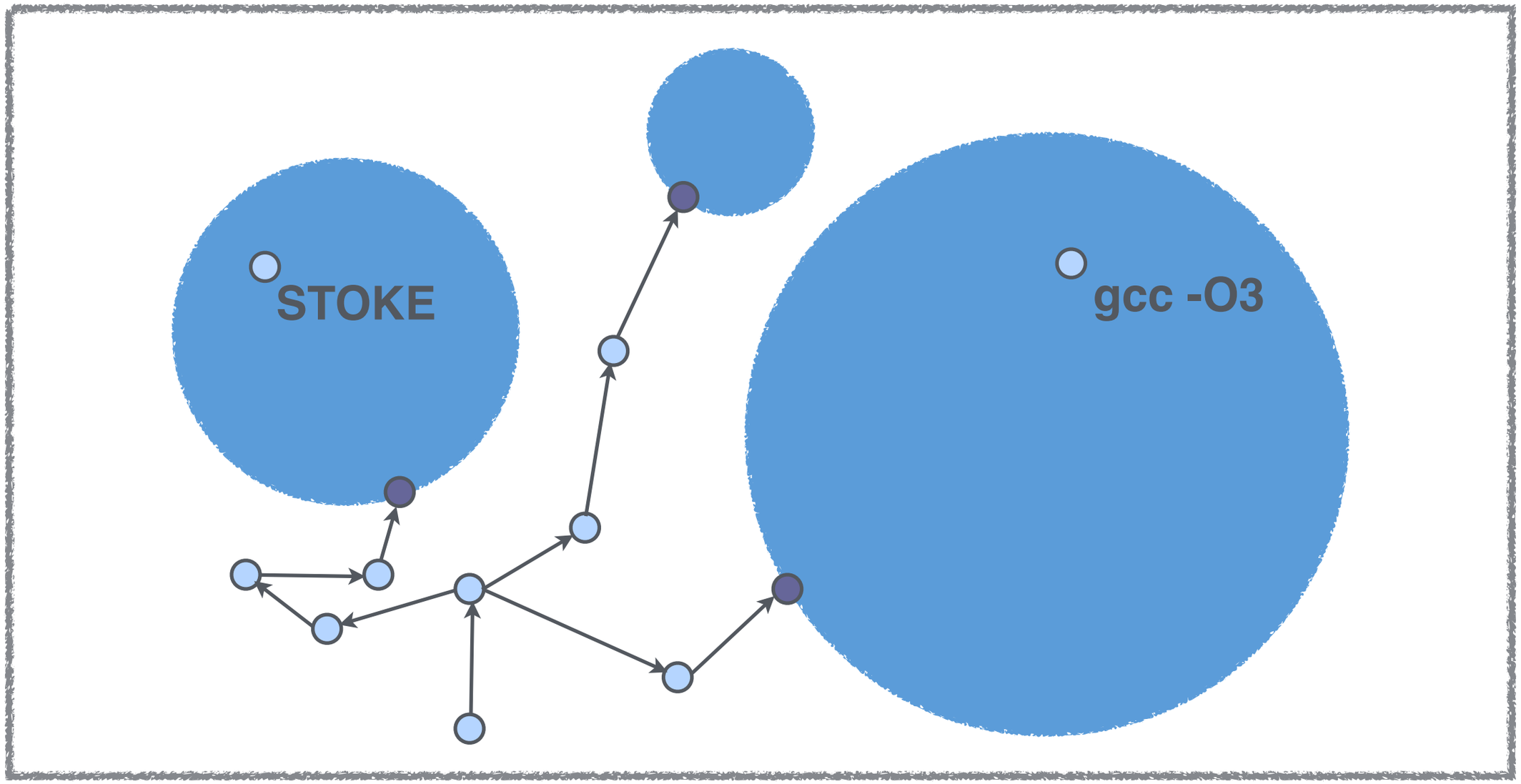
Equality preservation: Expert-written rules for traversing the space of correct implementations [joshi 02] [tate 09]

Related Work



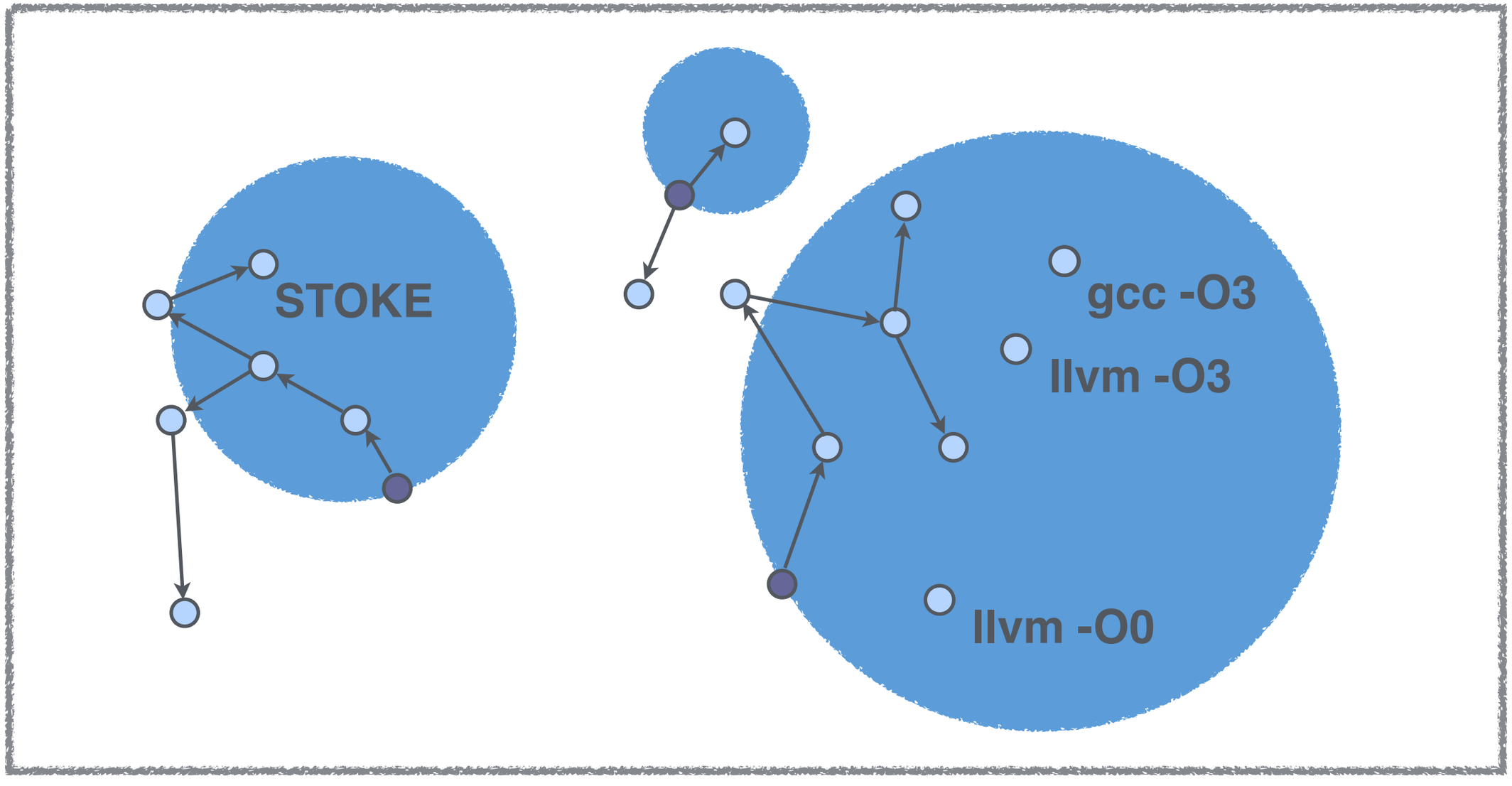
Program synthesis: Intermediate-level RTL, produce one correct implementation [gulwani 11] [solar-lezama 06]

Stochastic Optimization



Random search: Synthesis threads begin from random codes, attempt to discover regions of correct optimizations

Stochastic Optimization



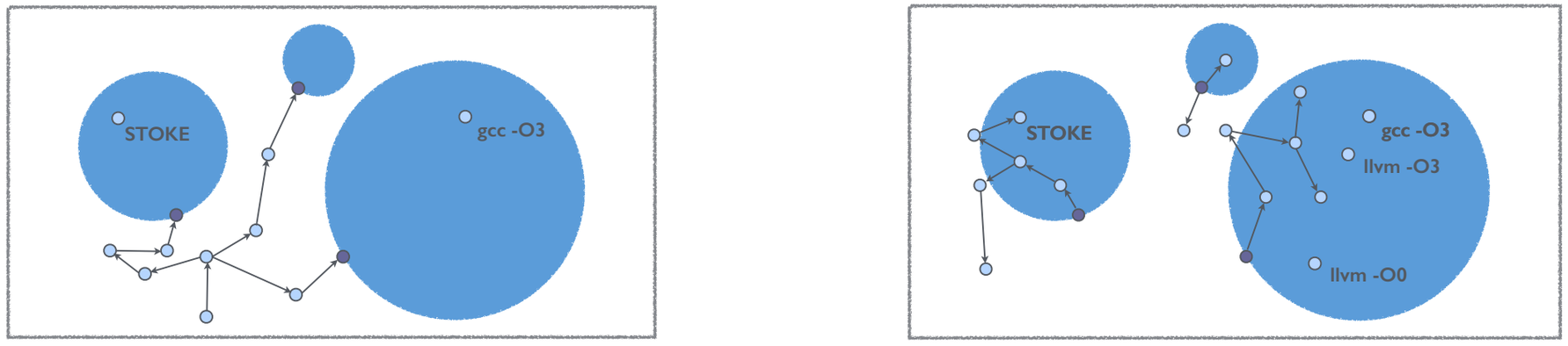
Random search: Optimization threads search each region and improve code, sometimes ignoring correctness

Outline

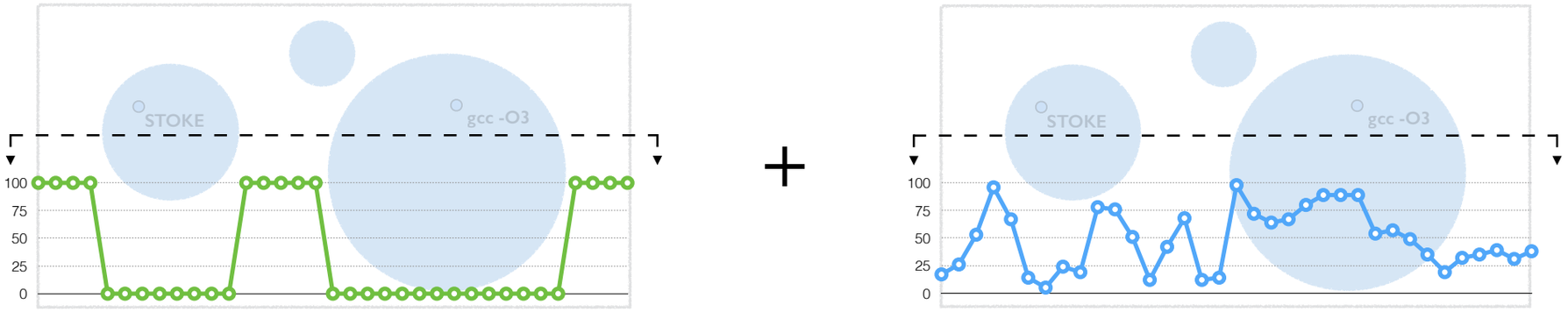
- Loop-free fixed-point optimization [asplos 13]
  - Intuitions
  - **Technical Detail / Evaluation**
- Extensions to floating-point [pldi 14]
  - Intuitions
  - Technical Detail / Evaluation

What's Required

1. **Search procedure:** MCMC Sampling



2. **Cost Function:** Formally encode the competing constraints of correctness and performance



MCMC Sampling

- **Widely used:** For many domains, the only known tractable solution method for high dimensional irregular search spaces [andrieu 03][chenney 00]

- **Random process:** Draw samples from a function; does not require a closed form representation of that function

MCMC Sampling

- **Guarantees:** Draws samples in proportion to their value; higher value points are sampled more frequently

- **No claim of convergence rate:** But works well in practice for the benchmarks that we consider

MCMC Sampling

Algorithm:

1. Select an initial program
2. Repeat (**millions** to **billions** of times)
   A. Propose a random change and evaluate cost
   B. If (decreased) { accept }
   C. If (increased) { maybe accept anyway }

Technical Details

- **Ergodicity:** Random transformations should be sufficient to cover entire search space.

- **Throughput:** Runtime cost to propose and evaluate should be minimal

- **Symmetry:** Probability of transformation equals probability of undoing it

Empirical Results

- **Result:** Intelligent hill “climbing” method, robust against local minima, desirable limiting properties

- **Strikes a balance:** Nothing prevents the use of other search methods. Offers a tradeoff between performance and implementation complexity

Transformations

original

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

Transformations

original

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

insert

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

Transformations

original

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

insert

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

delete

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

Transformations

original

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

insert

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

delete

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

insert

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

opcode

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

delete

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

insert

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

opcode

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

delete

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

Transformations

original

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

insert

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

opcode

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

delete

```
...
movl ecx, ecx
shrq 32, rsi
andl ff, 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

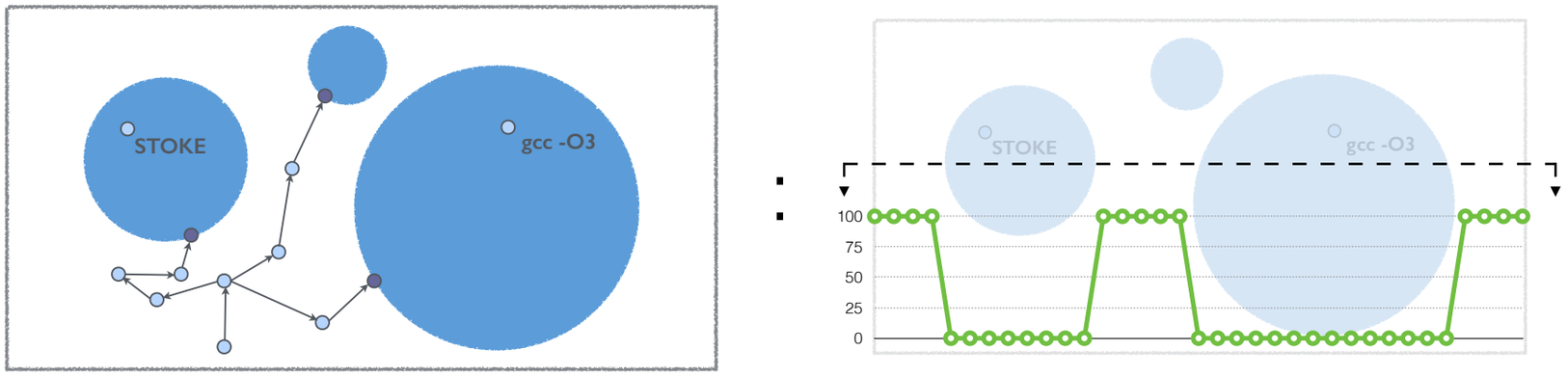
```
...
movl ecx, ecx
movl edx, edx
andl ff, r9d
movq rcx, rax
shrq 32, rsi
imulq r9, rax
...
```

operand

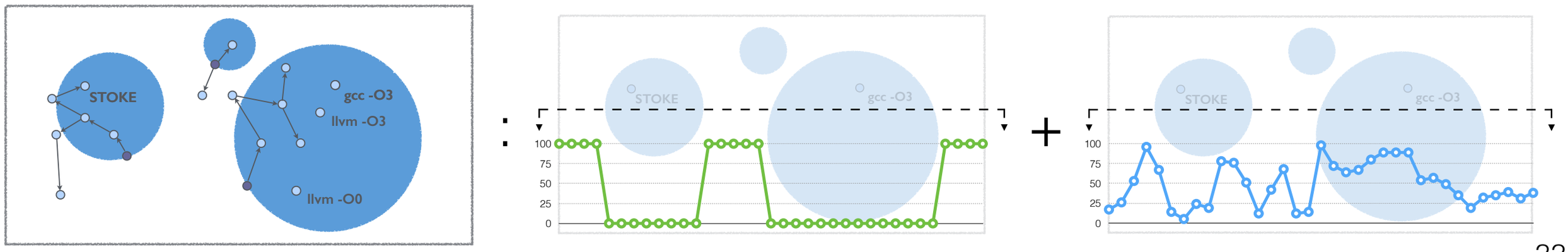
```
...
movl ecx, ecx
shrq 32, rcx
andl ff, r9d
movq rcx, rax
movl edx, edx
imulq r9, rax
...
```

Cost Function

- **Synthesis:** Search only for correct implementations

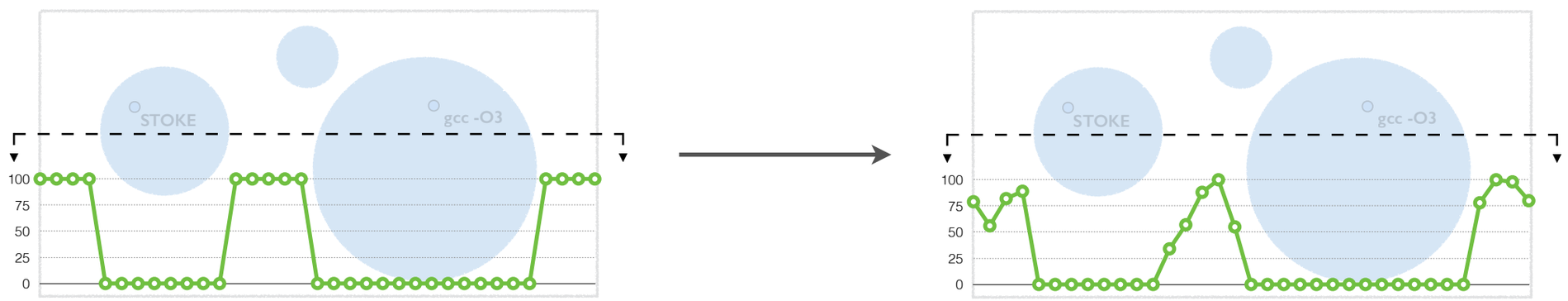


- **Optimization:** Simultaneously optimize correctness and performance using a weighted sum

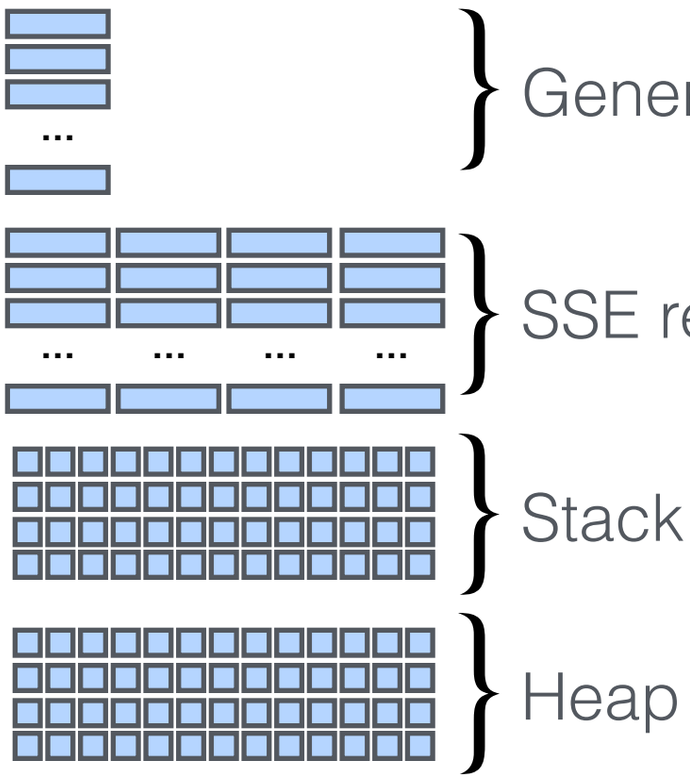


Correctness Function

- **Requirements:** Return zero iff a rewrite is functionally equivalent to the target
- **Smooth metric:** Helpful to provide a notion of partial correctness to smooth the search space

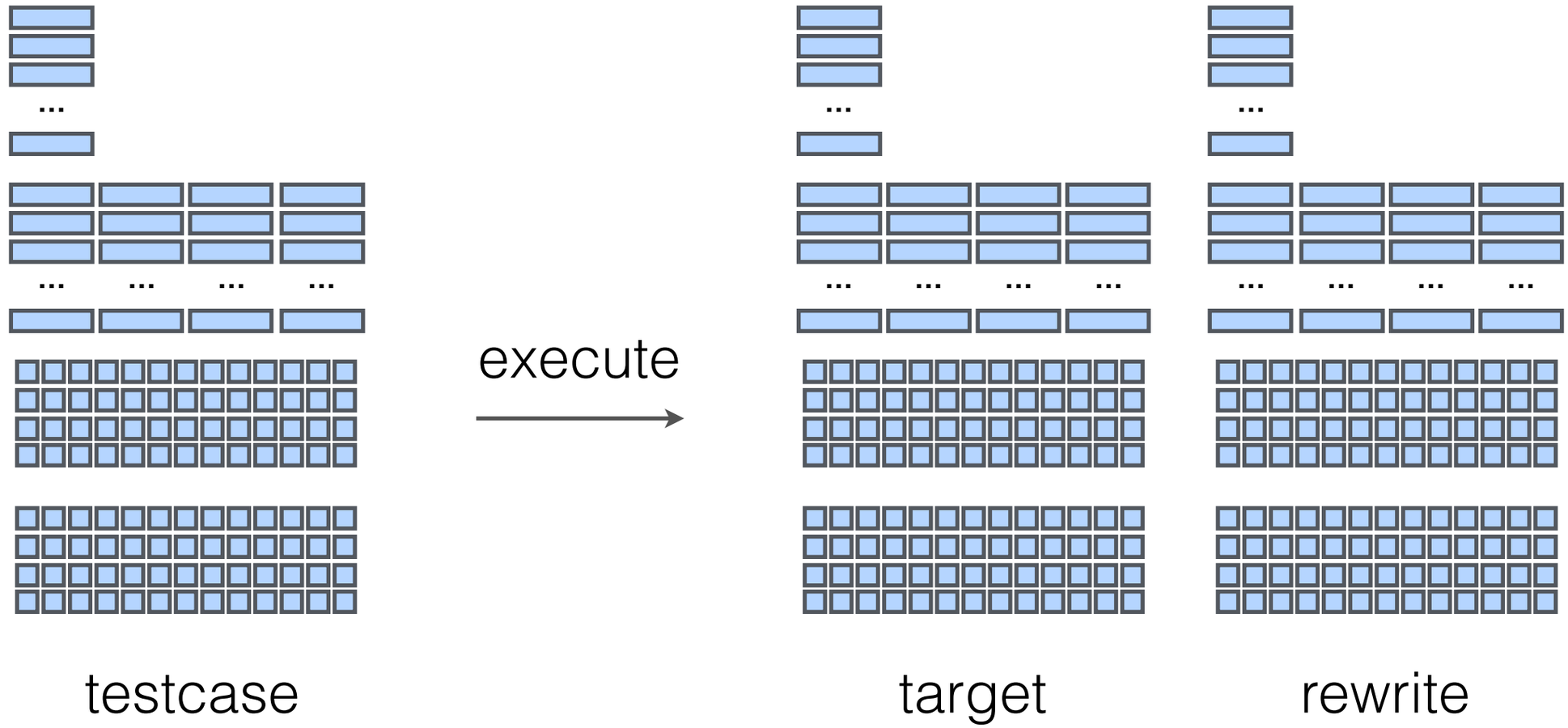


Correctness Function



Test case: Snapshot of machine state; assignments to registers and memory locations used by target

Correctness Function



Comparison: Execute target and rewrite on identical state and compare live outputs

Correctness Function

- **Hamming distance:** Count bit error across live out registers and memory locations

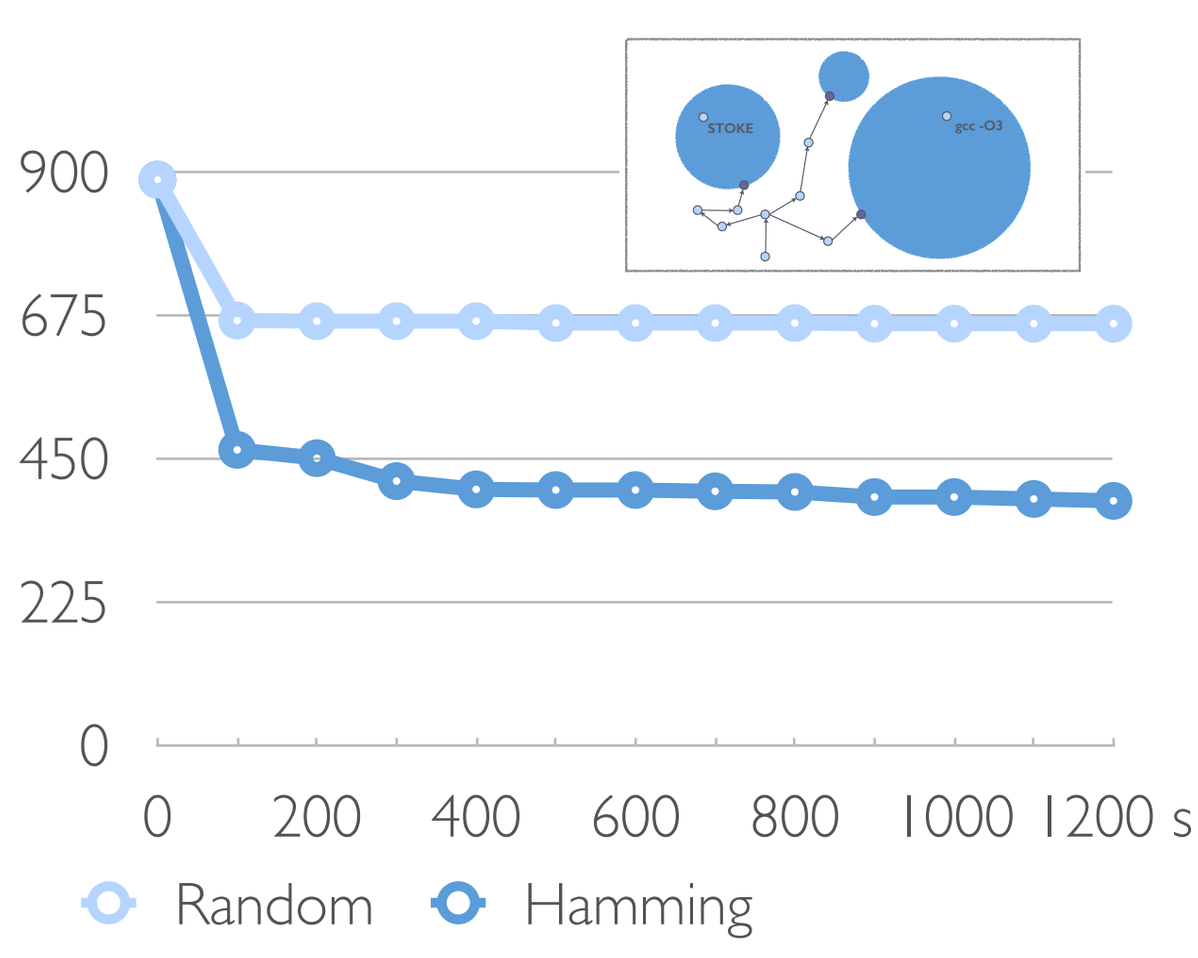
ax	bx	cx	dx
1111	0000	0000	0000

target

0010	1000	1100	1111

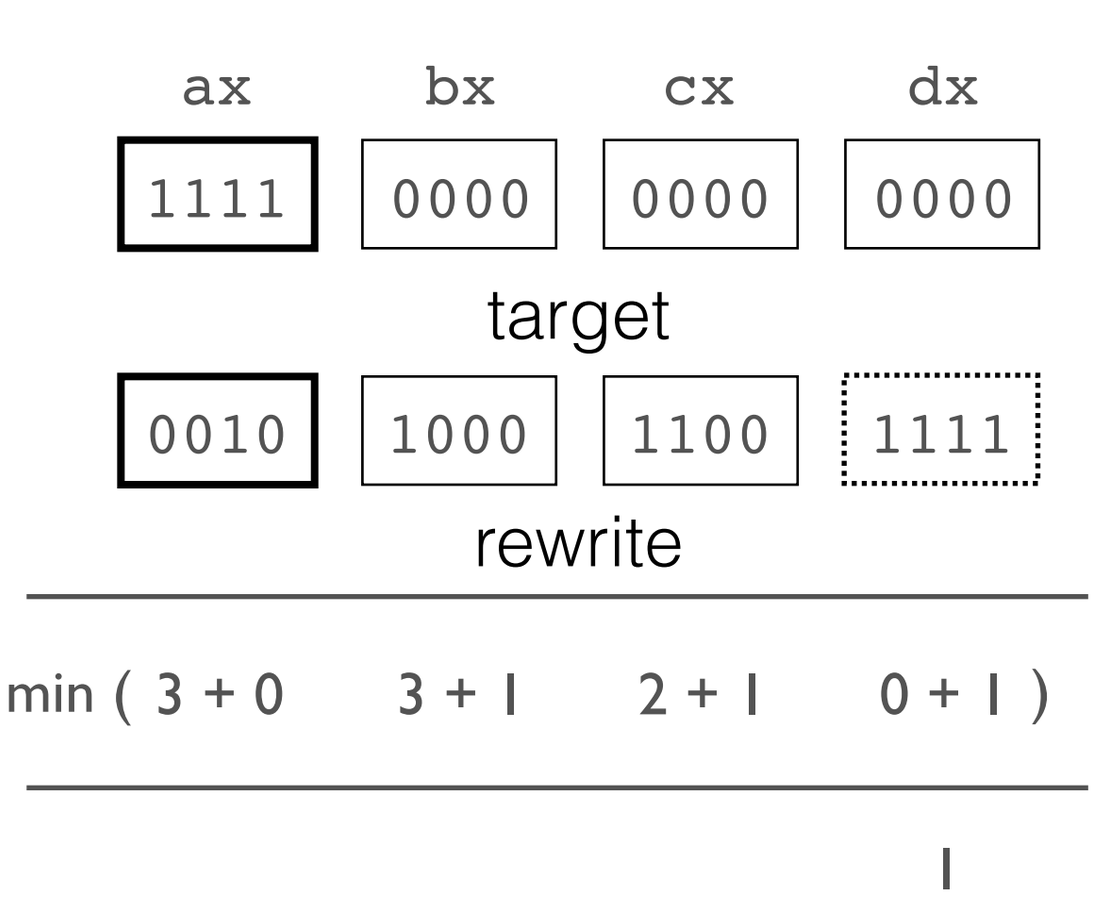
rewrite

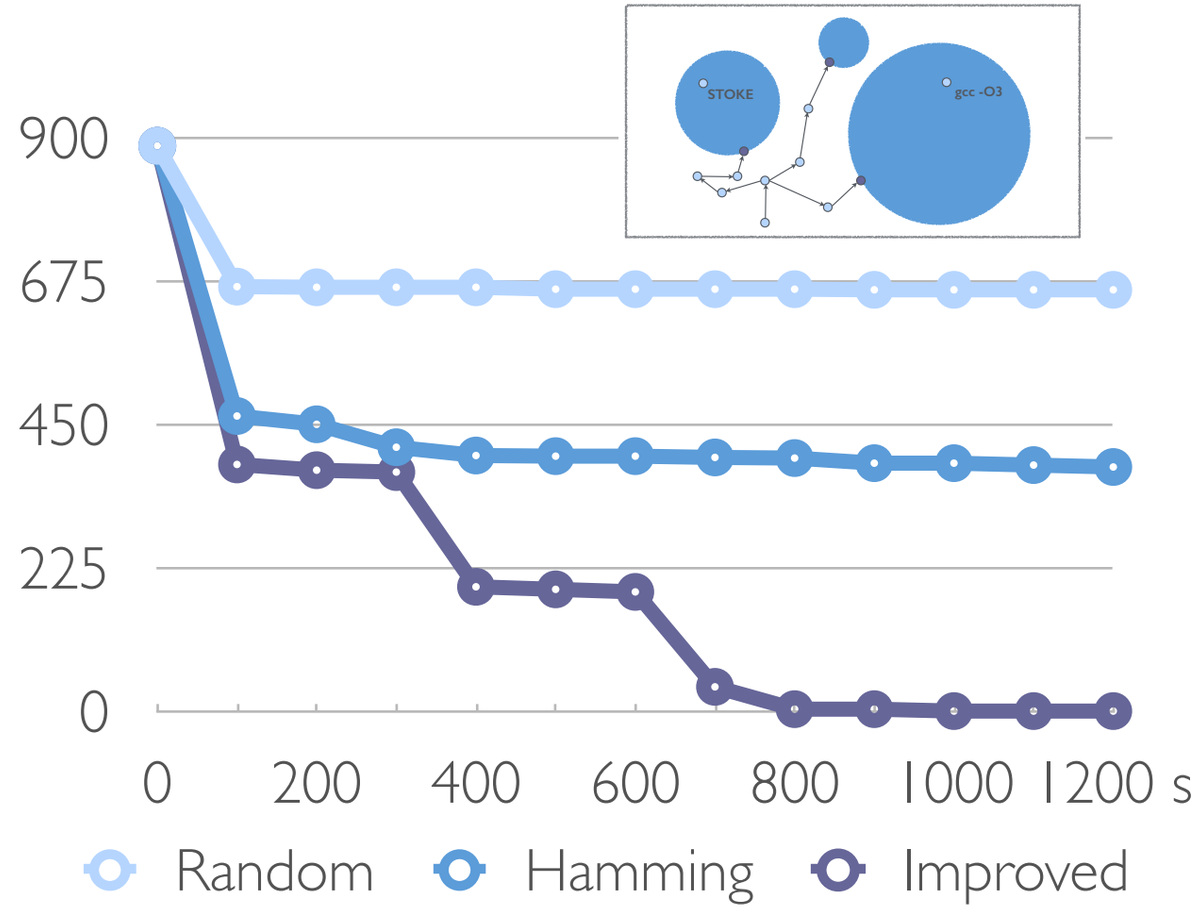
3



Correctness Function

- **Improvement:** Relax the constraint that correct results appear in correct locations





Testcase Evaluation

- **JIT Assembly:** Execute test cases natively
- **Guaranteed Sound:** No emulation errors
- **Dynamic Instrumentation:** Catch signals before they happen and execute unsound rewrites safely without crashing STOKE
- **Throughput:** Dispatch over one million test case evaluations per second

Performance Function

- **Fast estimate:** Approximate the runtime of a program by summing the average latencies of its instructions

- **Better than it sounds:** Ignores nearly all the complexity of a modern out-of-order processor, but works well in practice at this scale

Formal Guarantees

- **Correctness:** Encode both codes as SMT formulae and query the existence of inputs which will produce distinct outputs

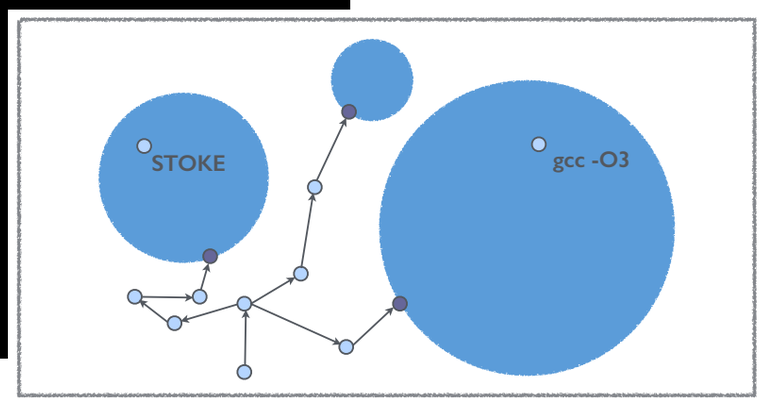
- **Conflict-driven test cases:** Failures produce counter-examples which refine test cases

- **Performance:** Preserve the top-n most performant results and rerank using compilation and evaluation on representative workloads

Demo

```
COST = 5013

retq
```

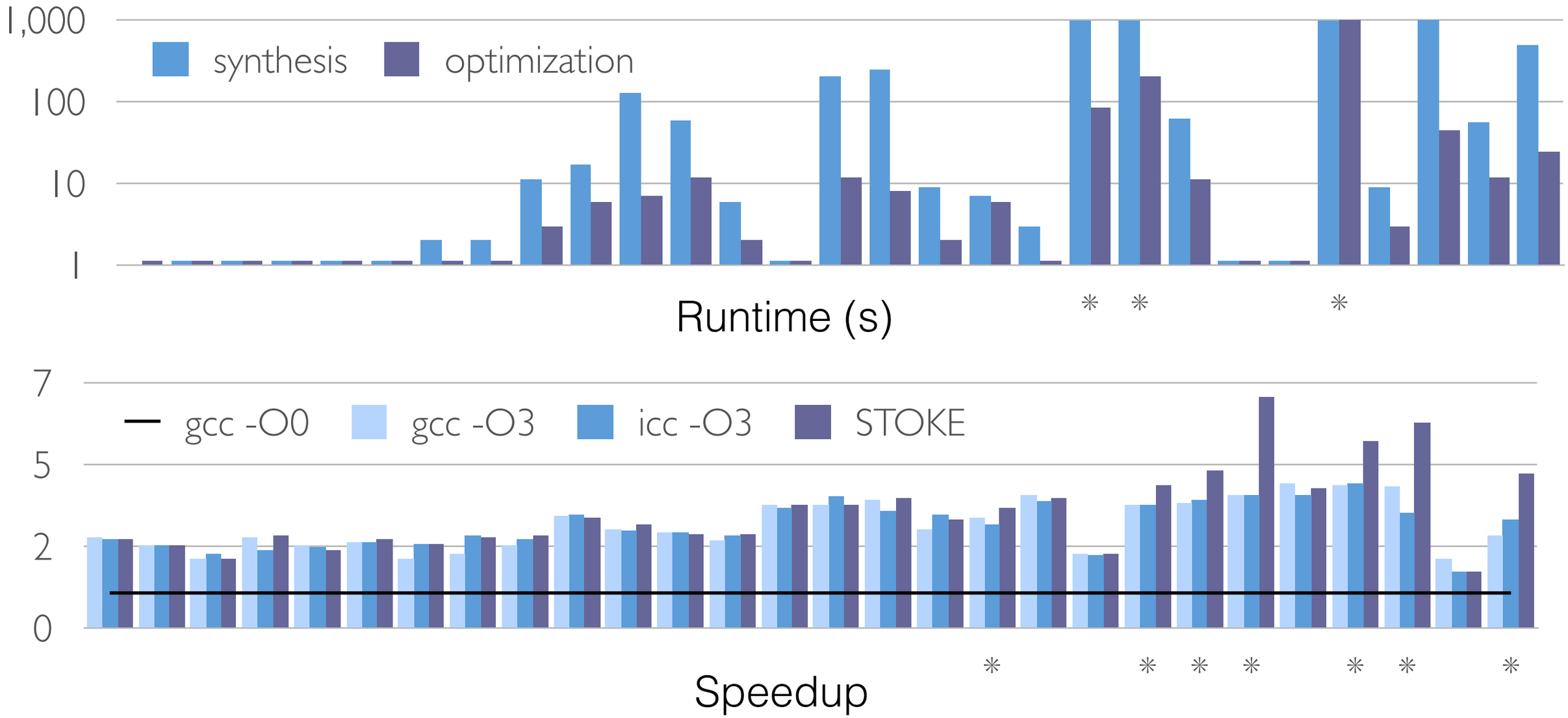
```
COST = 5013

retq
```



Evaluation

- **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]

Results

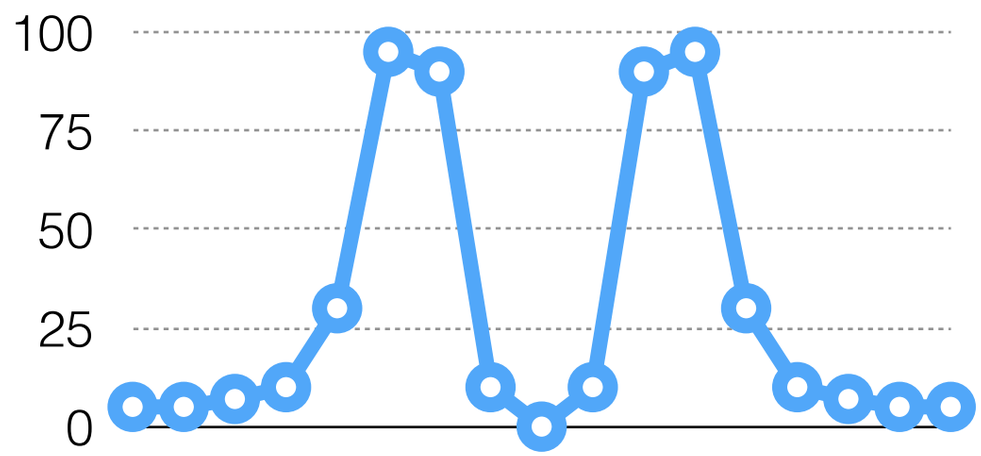


Takeaway: Beginning from gcc -O0, match or outperform the code produced by gcc -O3 and icc -O3

Failure Case

- **Difficult code:** Round input up to next highest power of two
- **General issue:** Difficult to synthesize codes that distill inputs down to few bits of result

Input	Correct	Incorrect
0001	0010	0000
0110	1000	0000
1100	0000	0000



Outline

- Loop-free fixed-point optimization [asplos 13]
  - Intuitions
  - Technical Detail / Evaluation
- Extensions to floating-point [pldi 14]
  - **Intuitions**
  - Technical Detail / Evaluation

Example

expert

```
vmovddup   %xmm0, %xmm0
vmulpd     (%rdi), %xmm0, %xmm2
vroundpd   $0, %xmm2, %xmm2
vmulpd     0x10(%rdi), %xmm2, %xmm1
vcvtpd2dqx %xmm2, %xmm3
vmulpd     0x20(%rdi), %xmm2, %xmm2
vaddpd     %xmm1, %xmm0, %xmm1
vmovapd    0x30(%rdi), %xmm0
vpaddd     0x40(%rdi), %xmm3, %xmm3
vpslld     $20, %xmm3, %xmm3
vpshufd    $114, %xmm3, %xmm3
vaddpd     %xmm2, %xmm1, %xmm1
vmulpd     0x50(%rdi), %xmm1, %xmm2
vaddpd     0x60(%rdi), %xmm2, %xmm2
vmulpd     %xmm1, %xmm2, %xmm2
vaddpd     0x70(%rdi), %xmm2, %xmm2
vmulpd     %xmm1, %xmm2, %xmm2
vaddpd     0x80(%rdi), %xmm2, %xmm2
vmulpd     %xmm1, %xmm2, %xmm2
vaddpd     0x90(%rdi), %xmm2, %xmm2
vmulpd     %xmm1, %xmm2, %xmm2
vaddpd     0xa0(%rdi), %xmm2, %xmm2
vmulpd     %xmm1, %xmm2, %xmm2
vaddpd     0xb0(%rdi), %xmm2, %xmm2
vmulpd     %xmm1, %xmm2, %xmm2
vaddpd     0xc0(%rdi), %xmm2, %xmm2
vmulpd     %xmm1, %xmm2, %xmm2
vaddpd     0xd0(%rdi), %xmm2, %xmm2
vmulpd     %xmm1, %xmm2, %xmm2
vaddpd     0xe0(%rdi), %xmm2, %xmm2
vmulpd     %xmm1, %xmm2, %xmm2
vaddpd     0xf0(%rdi), %xmm2, %xmm2
vmulpd     %xmm1, %xmm2, %xmm2
vaddpd     %xmm0, %xmm2, %xmm2
vmulpd     %xmm1, %xmm2, %xmm1
vaddpd     %xmm0, %xmm1, %xmm0
vmulpd     %xmm3, %xmm0, %xmm0
retq
```

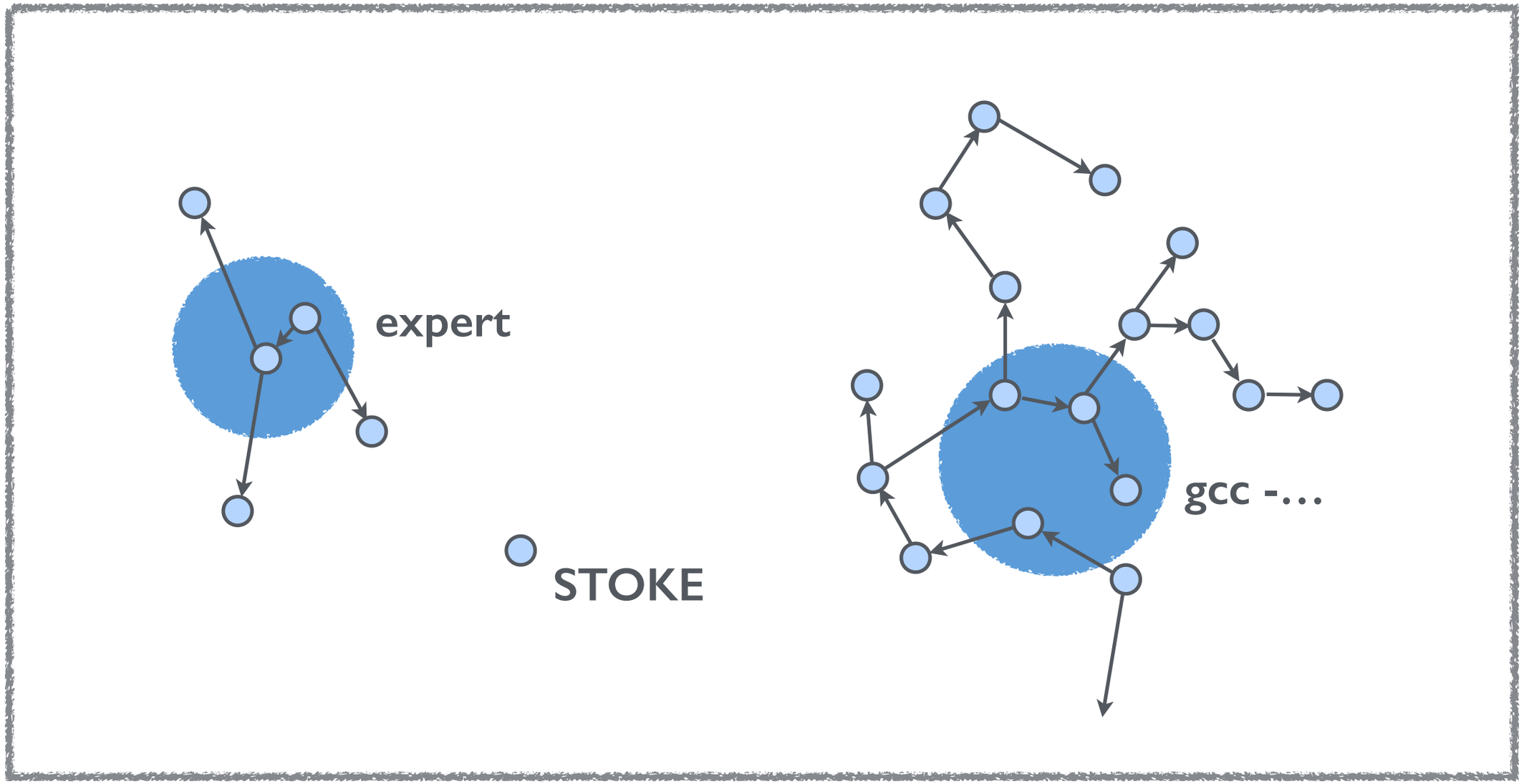
STOKE

```
vmulpd      (%rdi), %xmm0, %xmm2
vroundpd    $0xfffffffffffffffe, %xmm2, %xmm2
vcvttpd2dq  %xmm2, %xmm3
vmulpd      0x10(%rdi), %xmm2, %xmm1
vlddqu      0x90(%rdi), %xmm2
vaddpd      %xmm1, %xmm0, %xmm1
vmulpd      %xmm1, %xmm2, %xmm2
vpaddw      0x40(%rdi), %xmm3, %xmm3
vmovapd     0x30(%rdi), %xmm0
vpsllq      $0x14, %xmm3, %xmm3
vaddpd      0xa0(%rdi), %xmm2, %xmm2
vmulpd      %xmm1, %xmm2, %xmm2
vaddpd      0xb0(%rdi), %xmm2, %xmm2
vmulsd      %xmm1, %xmm2, %xmm2
vaddpd      0xc0(%rdi), %xmm2, %xmm2
vmulpd      %xmm1, %xmm2, %xmm2
vaddpd      0xd0(%rdi), %xmm2, %xmm2
vmulpd      %xmm1, %xmm2, %xmm2
vaddpd      0xe0(%rdi), %xmm2, %xmm2
vmulsd      %xmm1, %xmm2, %xmm2
vaddpd      0xf0(%rdi), %xmm2, %xmm2
vmulsd      %xmm1, %xmm2, %xmm2
vaddsd      %xmm0, %xmm2, %xmm2
vpshufd     $0x3, %xmm3, %xmm3
vmulpd      %xmm1, %xmm2, %xmm1
vaddsd      %xmm0, %xmm1, %xmm0
vmulpd      %xmm3, %xmm0, %xmm0
retq
```

Optimization Notes

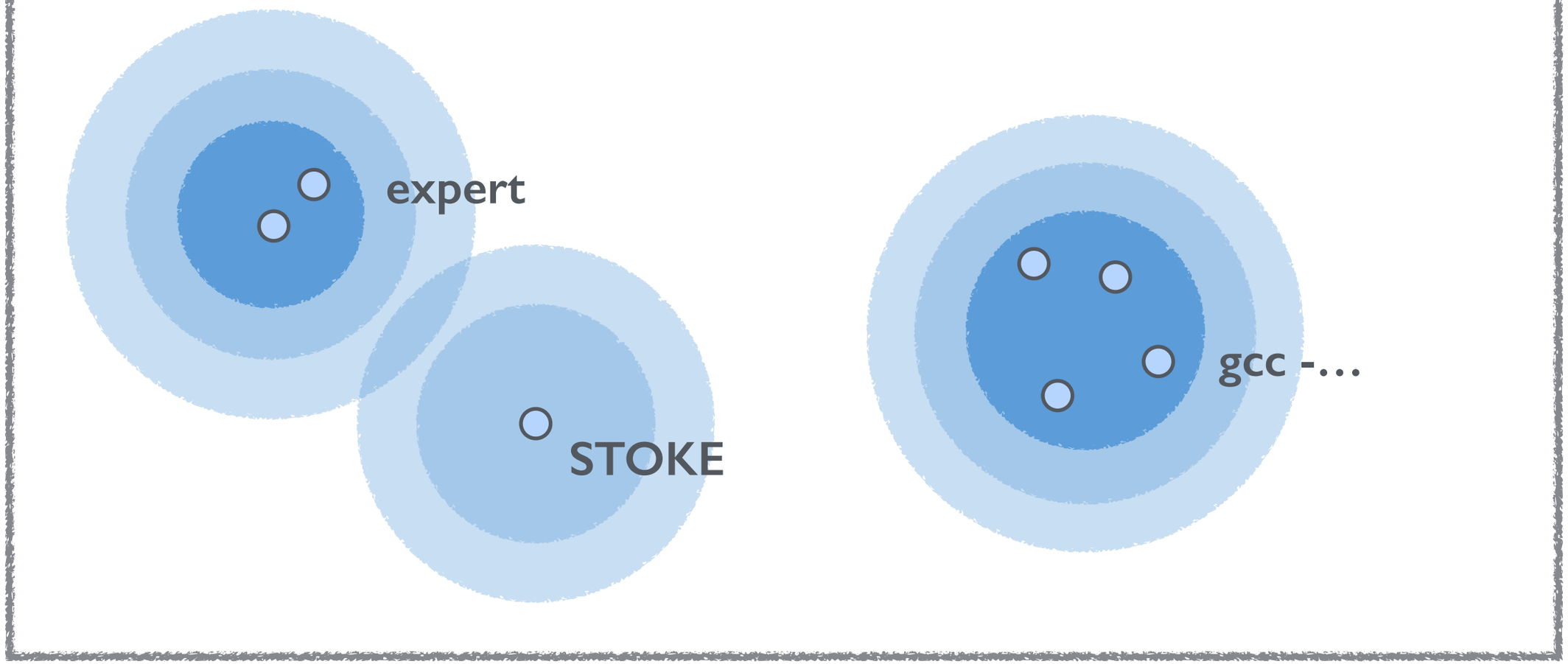
- **Smaller kernel:** 38 LOC reduced to 28 LOC
- **Performance improvement:** 57% kernel speedup, produces a 27% overall task speedup
- **Highly specialized:** Obeys application-specific error bound requirements for all inputs between -3.0 and 0

Intuition



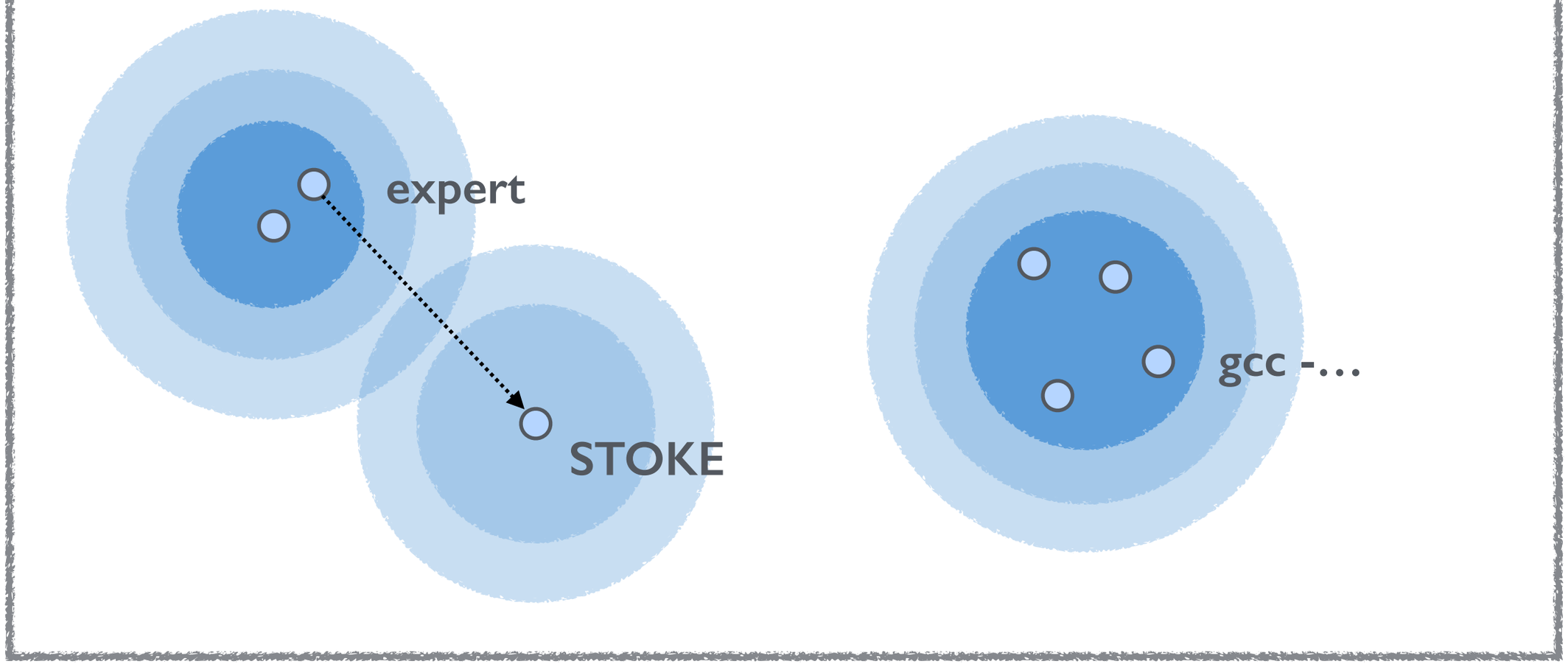
Few opportunities: Floating-point instruction set semantics complicate optimization

Intuition



Relax correctness: Most applications don't require full precision results for all possible inputs

Intuition



New opportunities: Gain access to high performance optimizations that were previous inaccessible

Outline

- Loop-free fixed-point optimization [asplos 13]
  - Intuitions
  - Technical Detail / Evaluation
- Extensions to floating-point [pldi 14]
  - Intuitions
  - **Technical Detail / Evaluation**

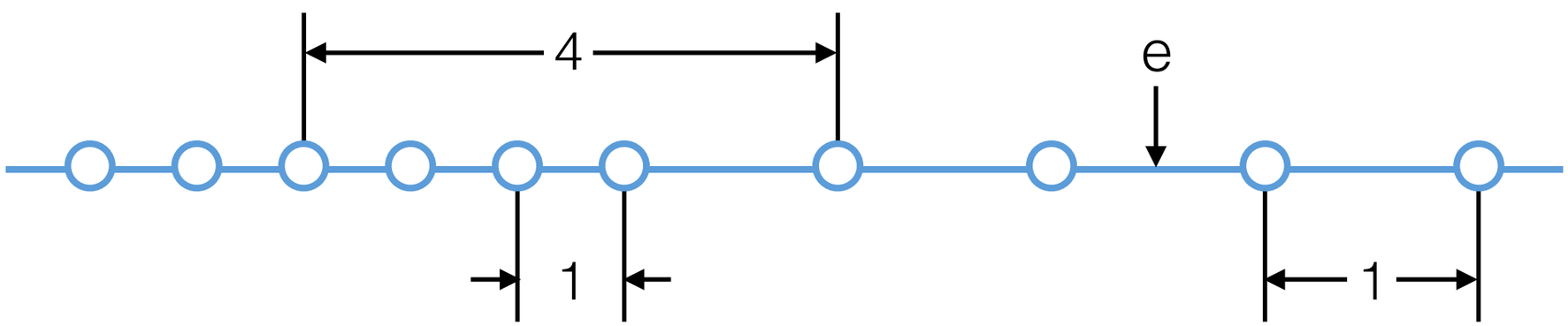
What's Required?

1. **Relaxed correctness function:** Formal definition of what it means for a floating-point rewrite to be "sort of correct"

2. **Improved formal check:** Procedure to guarantee that a rewrite is "sort of correct" for all application-dependent expected inputs

Correctness Function

- **Uncertainty in Last Place:** Measures the distance between a real number and the closest representable floating-point value
- **Widely Used:** Most scientific applications measure correctness in terms of ULPs; 0.5 is the gold standard but very expensive to obtain, most settle for 1 to 2



Verification

- **Guarantees:** How do we know that an optimization is "sort of correct" for all possible inputs?

- **Decision Procedures:** Don't scale beyond 5 lines of code; can't handle mixed fixed- and floating-point codes

- **Abstract Interpretation:** Can't prove even bitwise equality in many common cases; can't handle mixed fixed- and floating-point codes

- **Bottom Line:** No standard techniques can do this

Validation

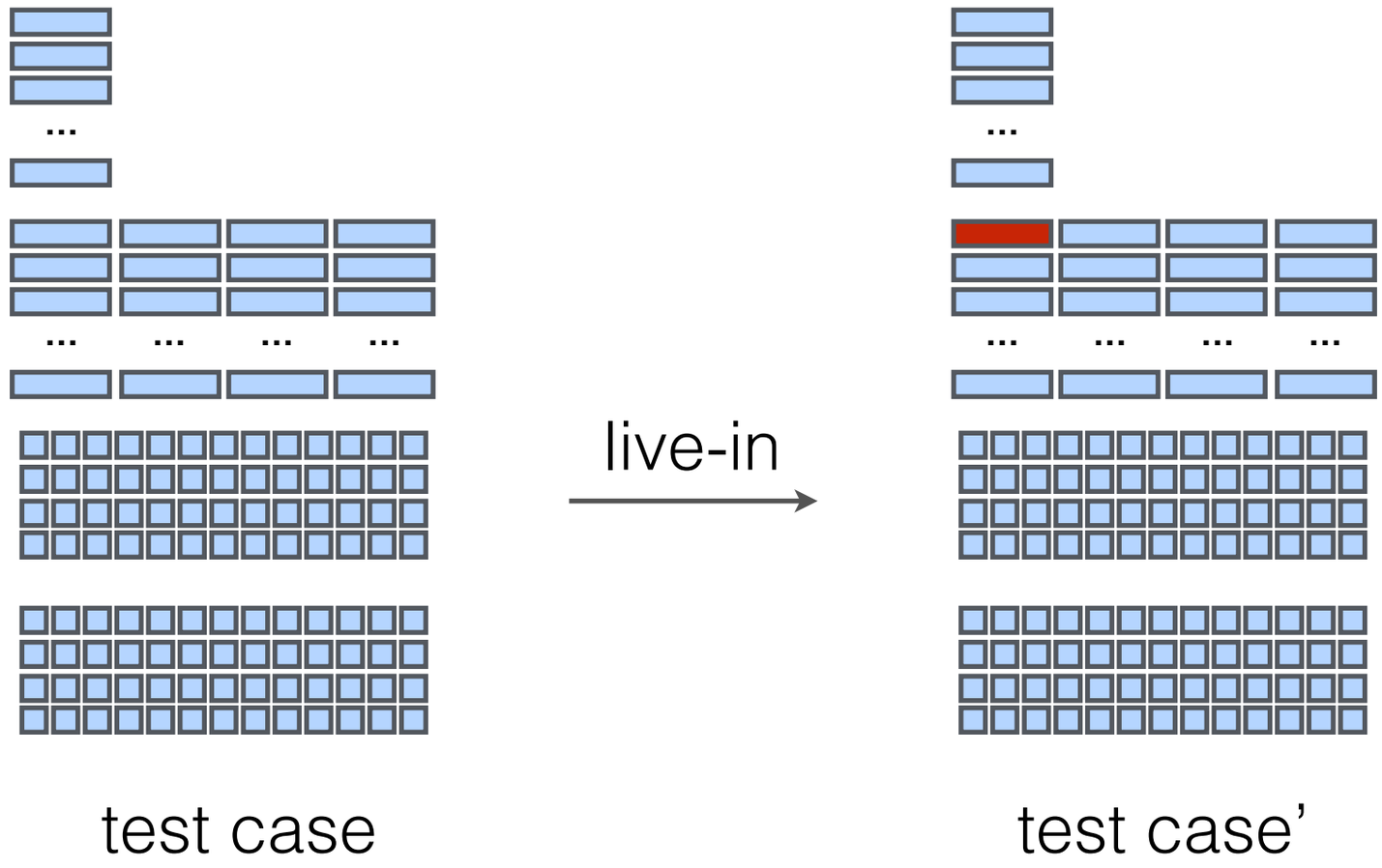
- **Relaxed problem statement:** Claim with high confidence that there is no input that will cause an error in excess of maximum user bound

- **Error function:** Run original and optimized code on identical inputs and measure ULP Error

  error(x) = **ULP**(**eval**(target, x), **eval**(rewrite, x))

- **Goal:** Show max error is below user-defined bound

Validation

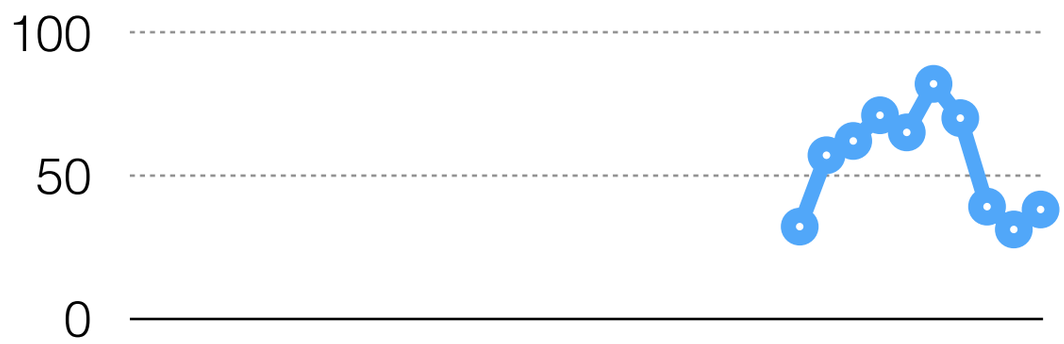


Search: Use MCMC sampling to search for test case inputs that maximize the error function; just one transform

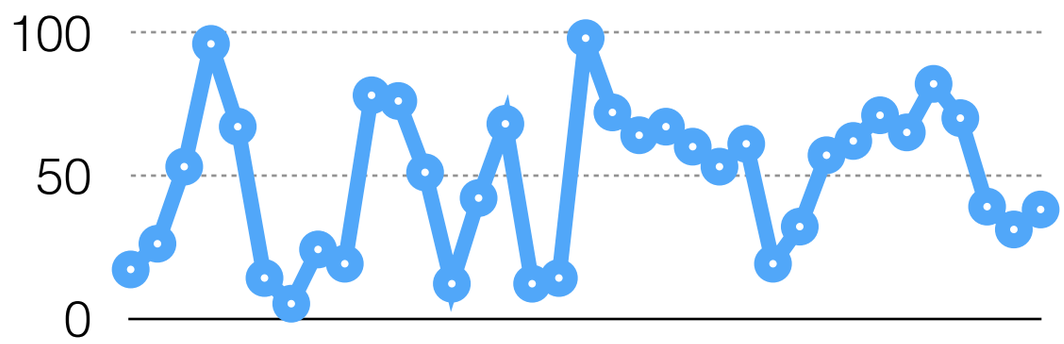
Termination

- **Mixing Tests:** Statistical tests [geweke 92] for producing a high-confidence guarantee that search has sampled uniformly across the domain of a function

- **Therefore:** High confidence that search has discovered all local maxima implies high confidence that it has discovered the global maximum



not converged



converged

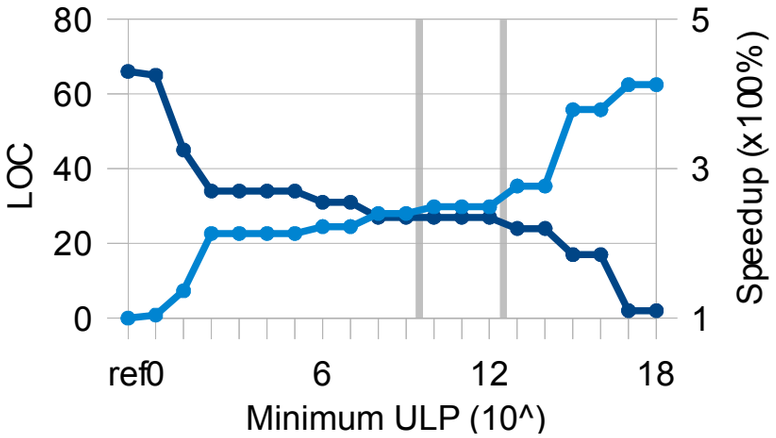
Evaluation

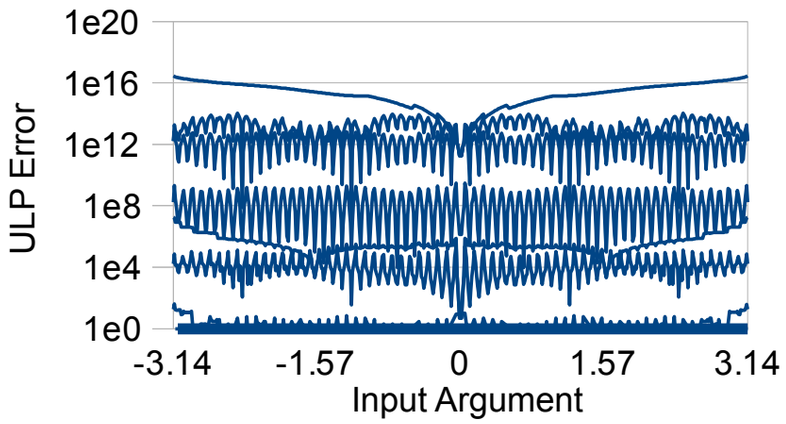
- **Numeric simulation:** S3D, 3-dimensional direct numerical solver for HCCI combustion

- **C library:** libimf, Intel's hand-written implementation of the C numerics library math.h

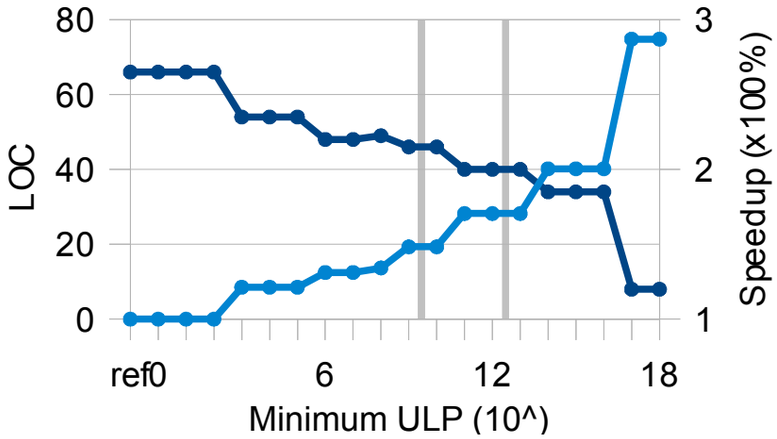
- **Computer graphics:** A ray tracer

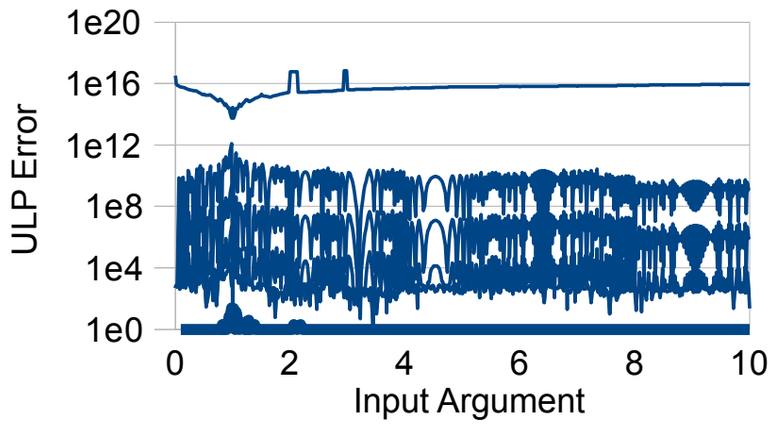
libimf



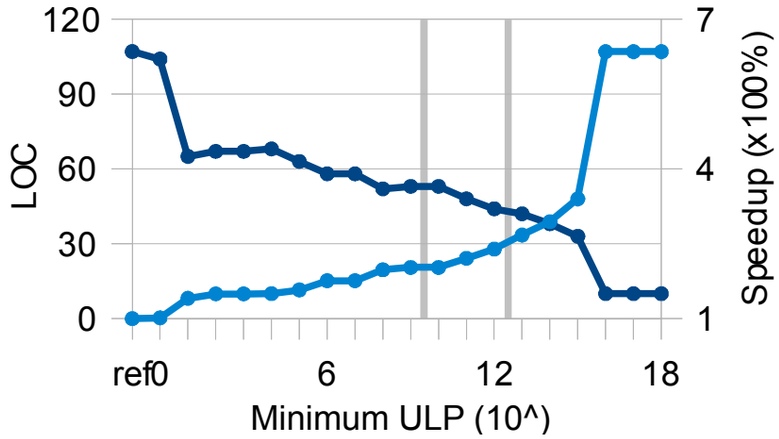


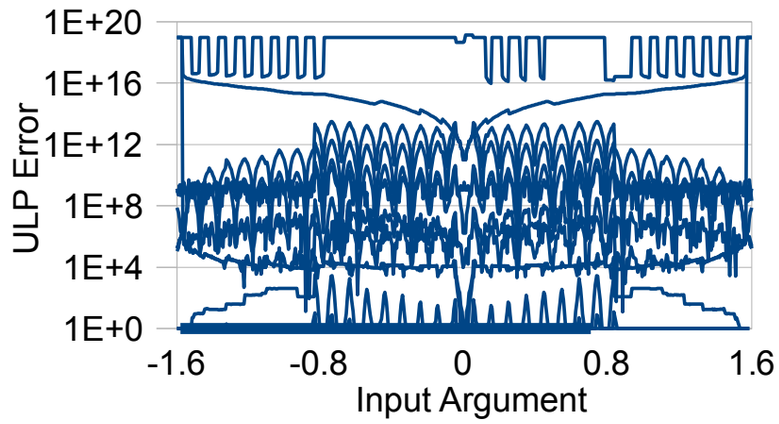
sin()





log()

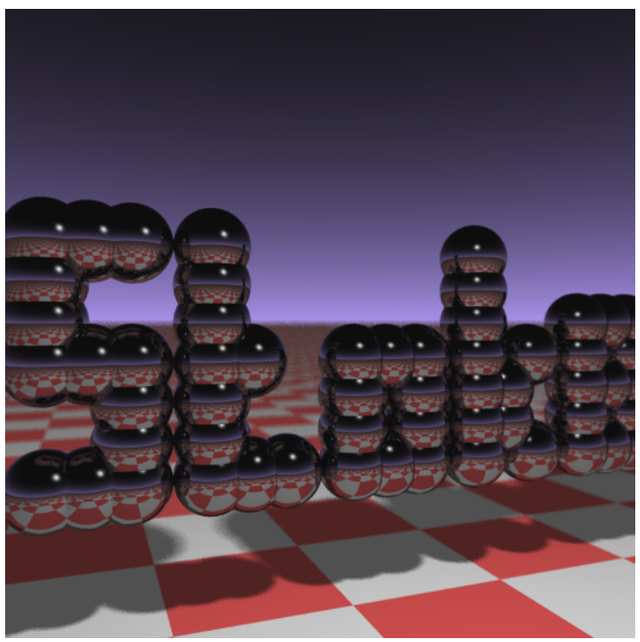




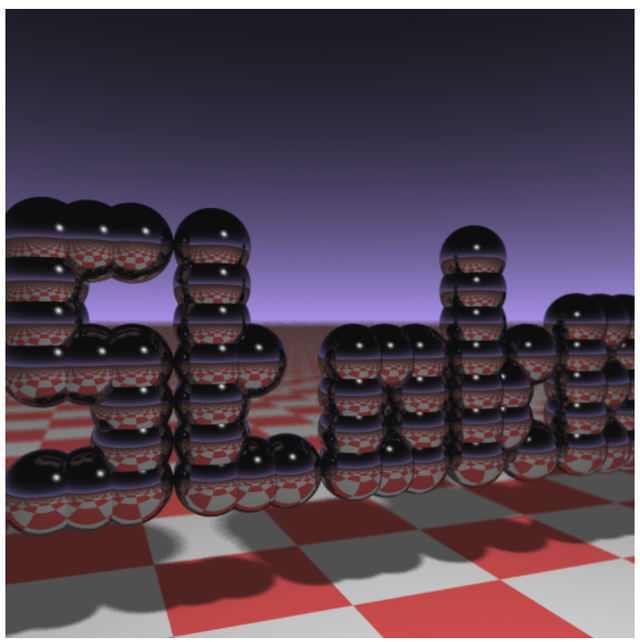
tan()

Ray Tracer

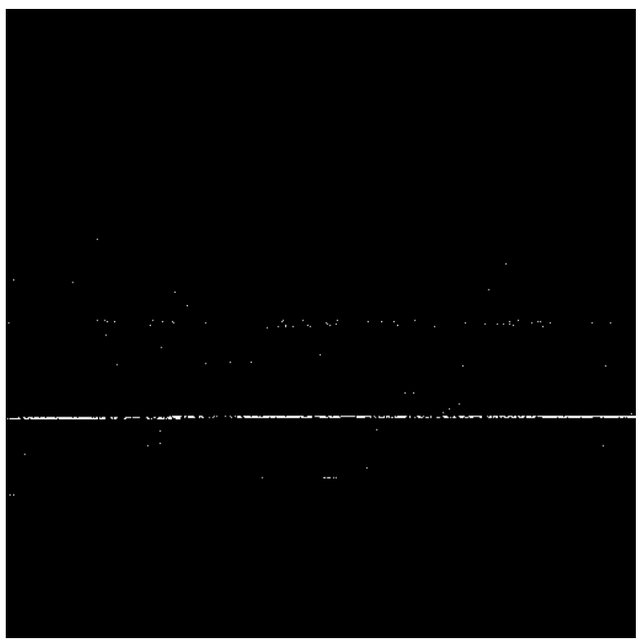
- **Bit-wise correct:** 30% speedup optimizing vector kernels
- **Depth of field blur:** Random perturbations made to viewing camera angle; 6% speedup by relaxing precision requirements



bit-wise correct



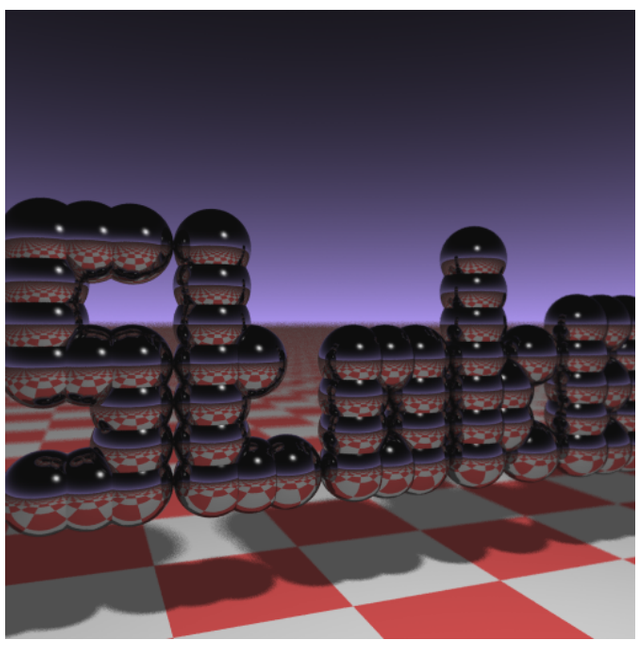
relaxed precision



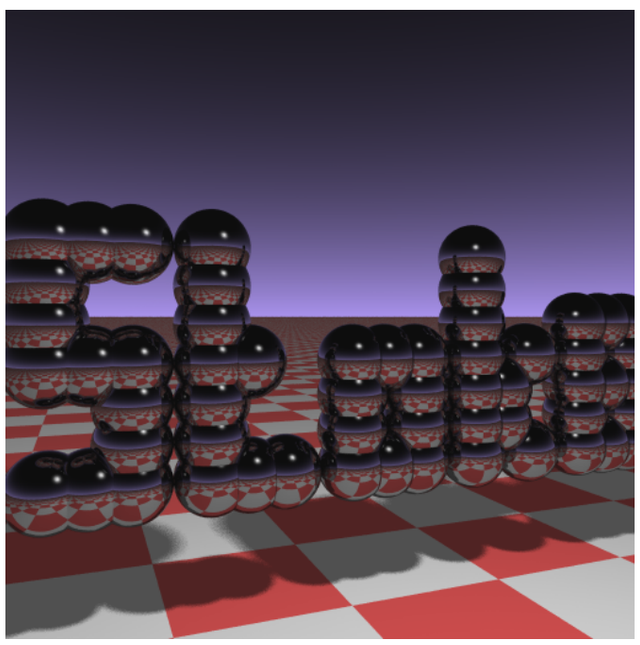
error pixels (white)

Failure Case

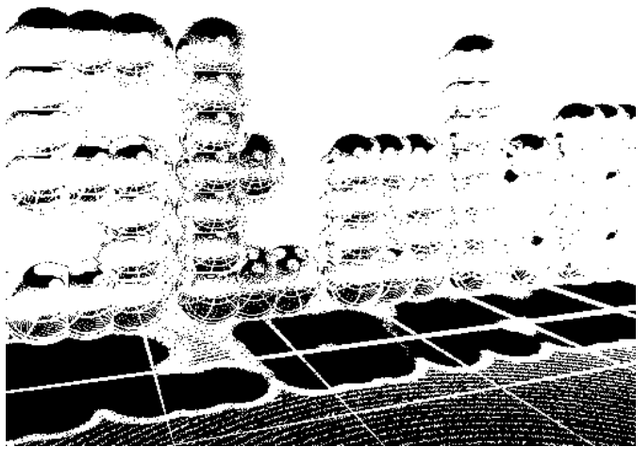
- **Over-relaxation:** If minimum tolerable error exceeds the variance of random perturbations, they are removed altogether
- **Faster still:** But depth of field blur has been removed



bit-wise correct



over-relaxed precision



error pixels (white)

Summary

- **Micro-optimization:** For many interesting application domains, once data movement is orchestrated correctly, even a single instruction can make a difference

- **New approach:** Use random search to experiment with incorrect intermediate optimizations and obtain optimizations that outperform handwritten expert code

- **New opportunities for optimization:** Identify applications that can tolerate a loss of precision and produce code that is specialized to that domain