
Automatic Generation of Peephole Superoptimizers

Sorav Bansal and Alex Aiken
Stanford University

`{sbansal, aiken}@cs.stanford.edu`

Compiler Folks



Architecture Folks



Compiler Folks



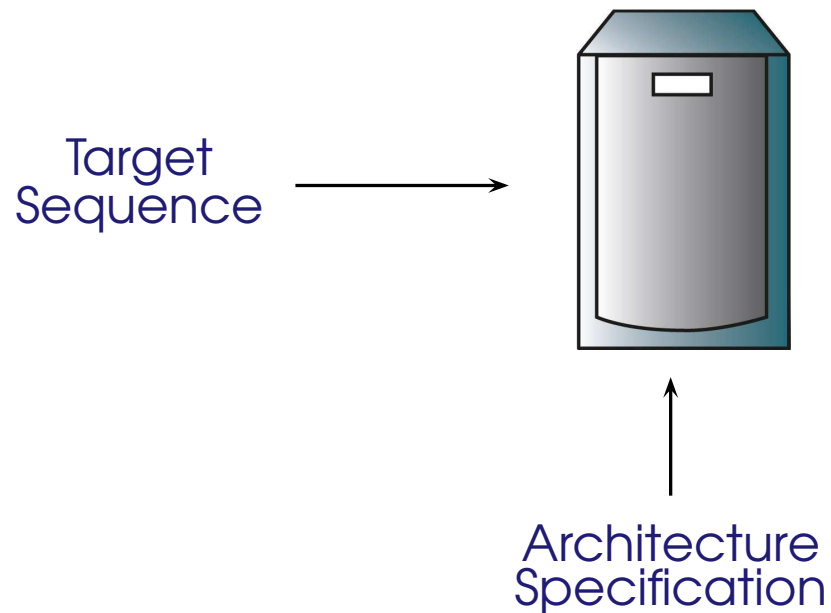
Architecture Folks



SUPE^ROPTIMIZATION

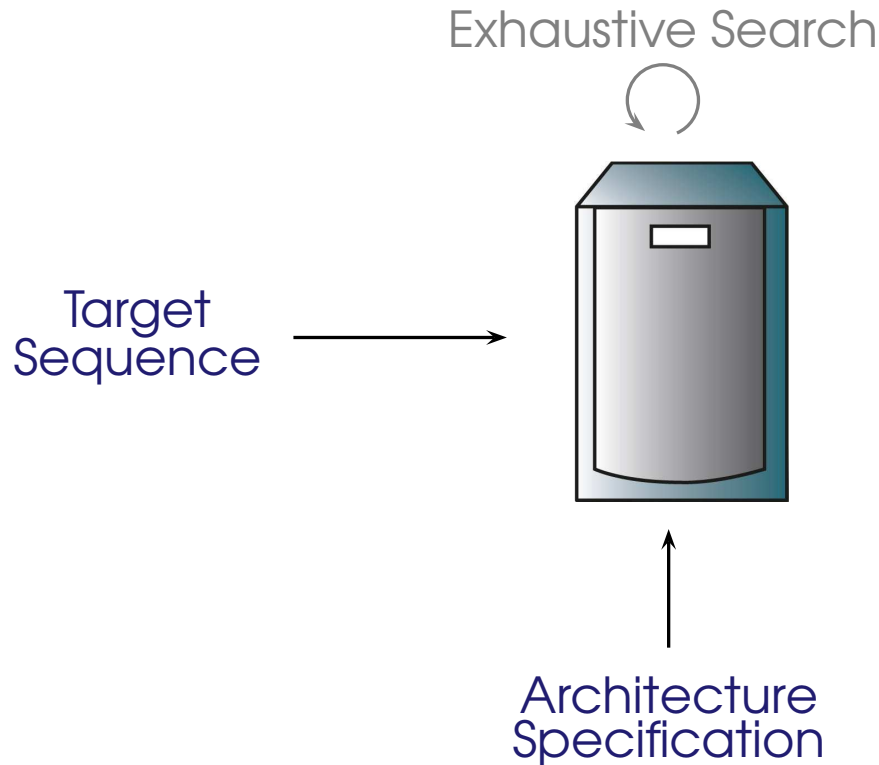
SUPEROPTIMIZATION

Classical Meaning: To find the optimal code sequence for a single, loop-free assembly sequence of instructions



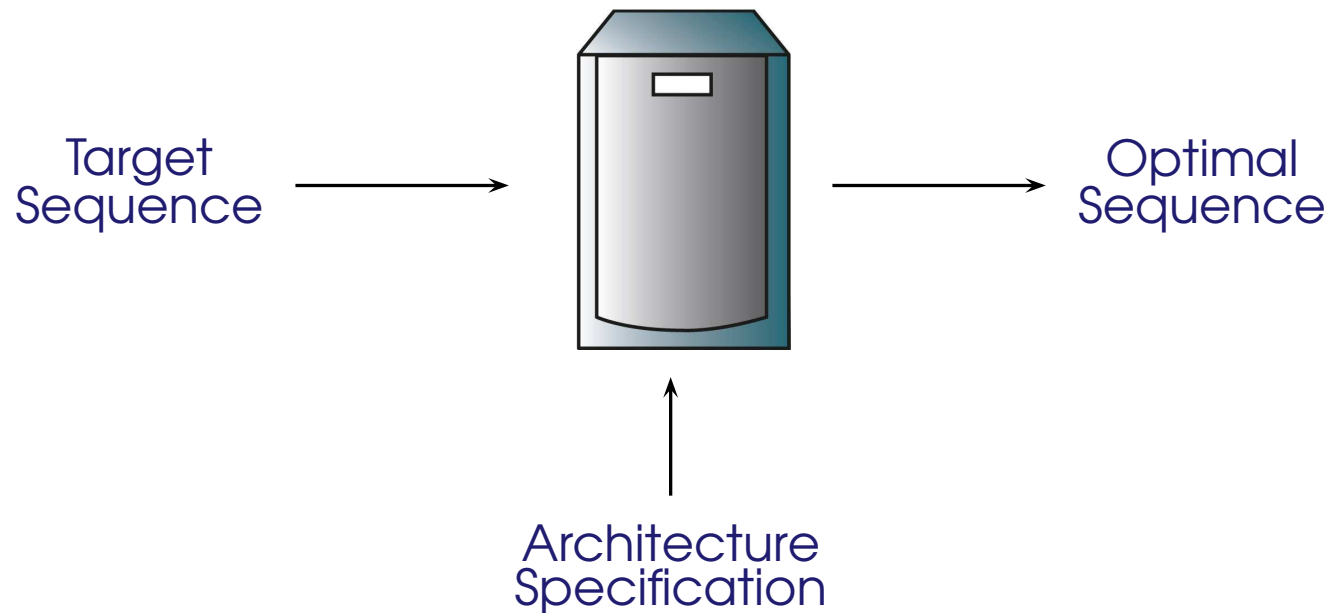
SUPEROPTIMIZATION

Classical Meaning: To find the optimal code sequence for a single, loop-free assembly sequence of instructions



SUPEROPTIMIZATION

Classical Meaning: To find the optimal code sequence for a single, loop-free assembly sequence of instructions



SUPEROPTIMIZATION

Classical Meaning: To find the optimal code sequence for a single, loop-free assembly sequence of instructions

Common Point-of-View

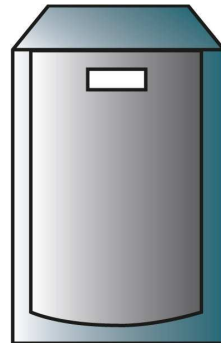
- Superoptimization is Sloowww..
- Superoptimization is useful only for occasional optimization of the critical inner loop

SUPEROPTIMIZATION

```
movl %eax , %ecx
in  %al ,
shll %eax ,
leal (%ecx), %eax
push (%ebx),
movl %ecx , (%ebx)
shll (%ebx),
addl $9 , (%ebx)
call func ,
mov %eax , %ecx

shll %al ,
out %al ,
push (%ebx),
mov %ecx , (%ebx)
push %eax ,
mov %eax , %ecx
```

a.out



SUPEROPTIMIZATION

```
movl %eax , %ecx
in  %al ,
shll %eax ,
leal (%ecx), %eax
push (%ebx),
movl %ecx , (%ebx)
shll (%ebx),
addl $9 , (%ebx)
call func ,
mov %eax , %ecx

shll %al ,
out %al ,
push (%ebx),
mov %ecx , (%ebx)
push %eax ,
mov %eax , %ecx
```

a.out

Exhaustive Search

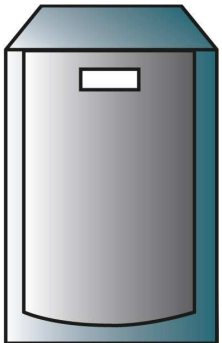


SUPEROPTIMIZATION

```
movl %eax , %ecx
in  %al ,
shll %eax ,
leal (%ecx), %eax
push (%ebx),
movl %ecx , (%ebx)
shll (%ebx),
addl $9 , (%ebx)
call func ,
mov %eax , %ecx

shll %al ,
out %al ,
push (%ebx),
mov %ecx , (%ebx)
push %eax ,
mov %eax , %ecx
```

a.out



Optimization Table

	target	optimal
1.	movl %ecx (%ebx) shll (%ebx) addl \$9 (%ebx)	leal 9(%ecx) %ebx movl %ebx (%ebx)

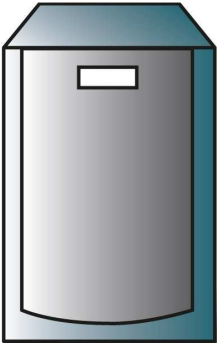
SUPEROPTIMIZATION

```

movl %eax , %ecx
in %al ,
shll %eax ,
leal (%ecx), %eax
push (%ebx),
movl %ecx , (%ebx)
shll (%ebx),
addl $9 , (%ebx)
call func ,
mov %eax , %ecx

shll %al ,
out %al ,
push (%ebx),
mov %ecx , (%ebx)
push %eax ,
mov %eax , %ecx
    
```

a.out



Optimization Table

	target	optimal
1.	<pre> movl %ecx (%ebx) shll (%ebx) addl \$9 (%ebx) </pre>	<pre> leal 9(%ecx) %ebx movl %ebx (%ebx) </pre>
2.	<pre> shll %eax leal (%ecx) (%eax) </pre>	<pre> movl %ecx %eax </pre>

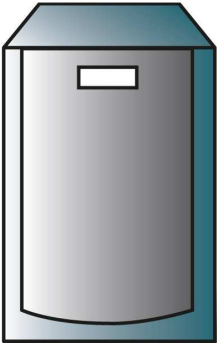
SUPEROPTIMIZATION

```

mov %eax ,%ecx
in %al ,
shll %eax ,
leal (%ecx),%eax
push (%ebx),
movl %ecx ,%eax
shll %ecx ,
movl %ecx ,%eax
call func ,
mov %eax ,%ecx

shll %al ,
out %al ,
push (%ebx),
mov %ecx , (%ebx)
push %eax ,
mov %eax ,%ecx
    
```

b.out



Optimization Table

	target	optimal
1.	movl %ecx (%ebx) shll (%ebx) addl \$9 (%ebx)	leal 9(%ecx) %ebx movl %ebx (%ebx)
2.	shll %eax leal (%ecx) (%eax)	movl %ecx %eax
3.	movl %ecx %eax shll %ecx movl %ecx %eax	shll %ecx movl %ecx %eax

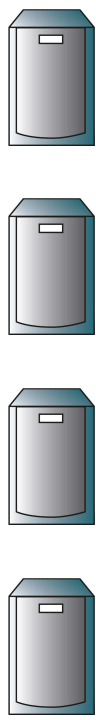
SUPEROPTIMIZATION

```

mov %eax ,%ecx
in %al ,
shll %eax ,
leal (%ecx),%eax
push (%ebx),
movl %ecx ,%eax
shll %ecx ,
movl %ecx ,%eax
call func ,
mov %eax ,%ecx

shll %al ,
out %al ,
push (%ebx),
mov %ecx , (%ebx)
push %eax ,
mov %eax ,%ecx
    
```

b.out



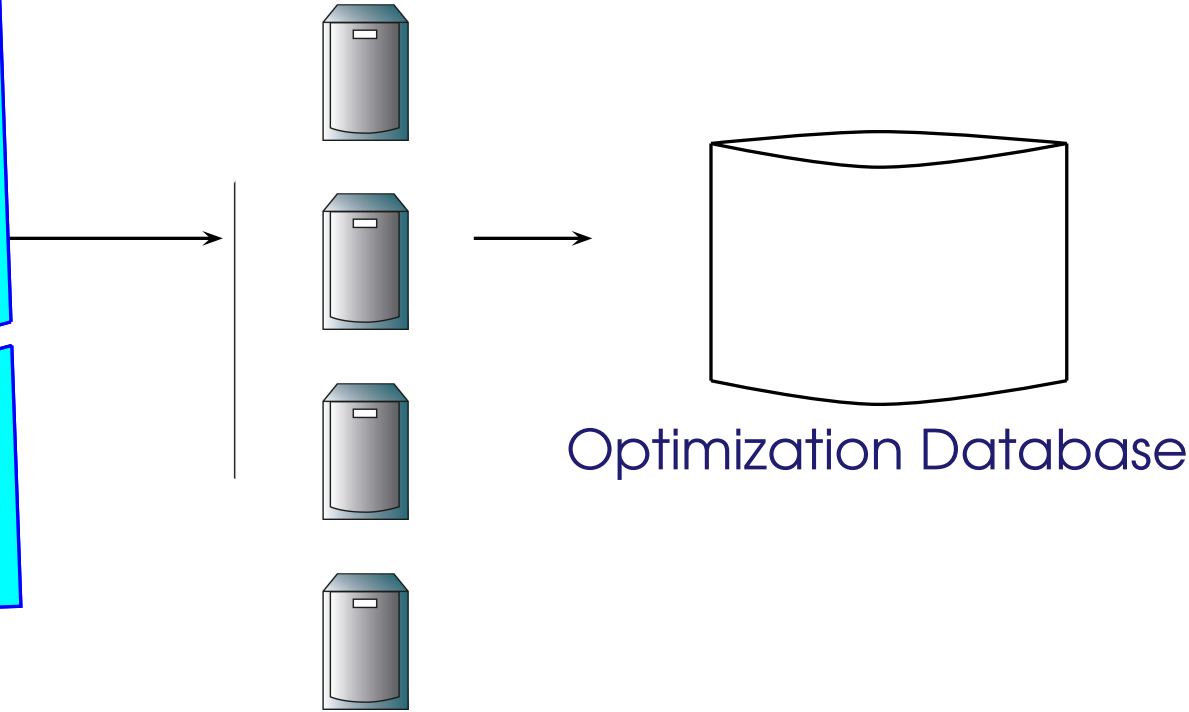
Optimization Table

	target	optimal
1.	movl %ecx (%ebx) shll (%ebx) addl \$9 (%ebx)	leal 9(%ecx) %ebx movl %ebx (%ebx)
2.	shll %eax leal (%ecx) (%eax)	movl %ecx %eax
3.	movl %ecx %eax shll %ecx movl %ecx %eax	shll %ecx movl %ecx %eax

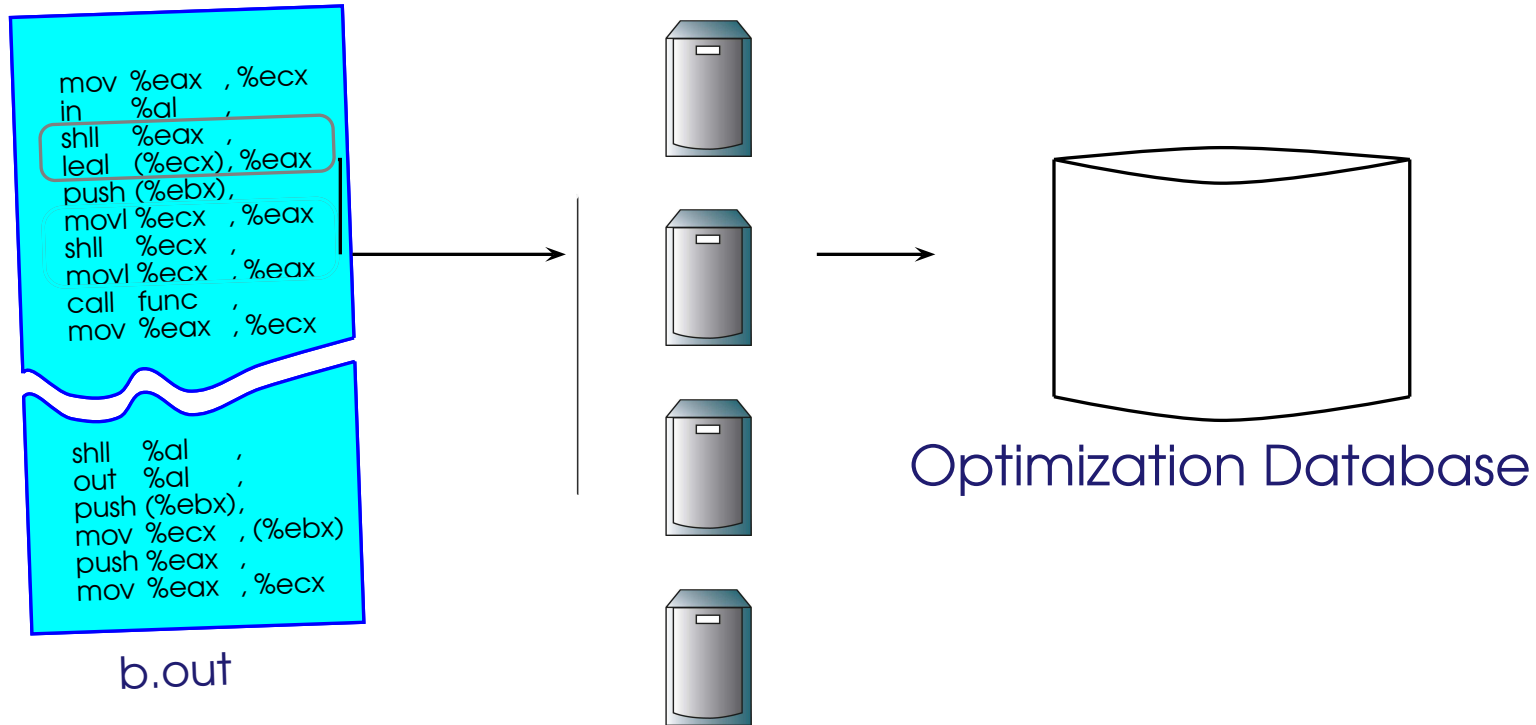
SUPEROPTIMIZATION

```
mov %eax ,%ecx  
in %al ,  
shll %eax ,  
leal (%ecx),%eax  
push (%ebx),  
movl %ecx ,%eax  
shll %ecx ,  
movl %ecx ,%eax  
call func ,  
mov %eax ,%ecx  
  
shll %al ,  
out %al ,  
push (%ebx),  
mov %ecx , (%ebx)  
push %eax ,  
mov %eax ,%ecx
```

b.out

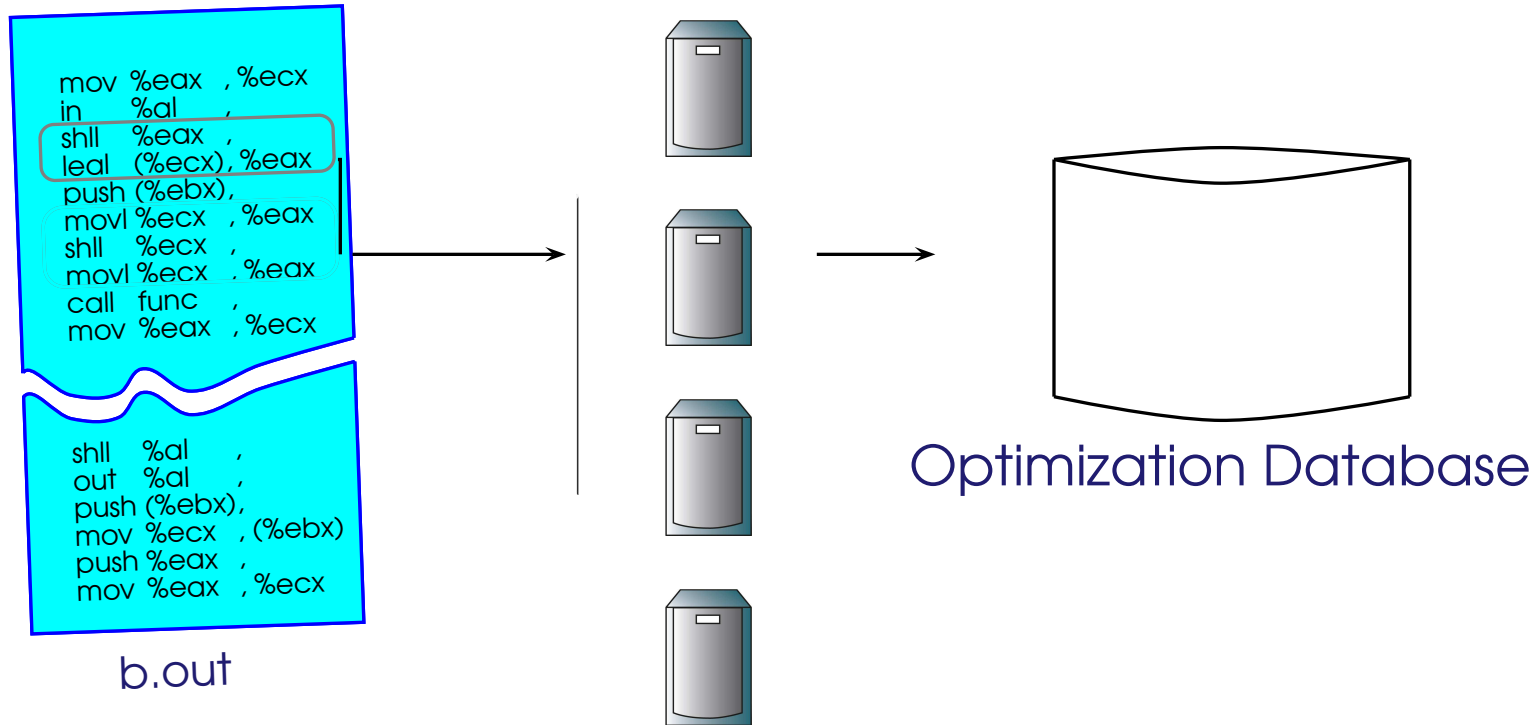


SUPEROPTIMIZATION



- To what length of instruction sequences can we scale?

SUPEROPTIMIZATION



- To what length of instruction sequences can we scale?
- Can the database be made large enough to capture most “traditional” optimizations?

Traversing a Link-List

```
struct node
{
    int          val;
    struct node *next;
};

void traverse (struct node *head)
{
    while (head)
    {
        head->value *= 2;
        head = head->next;
    }
}
```

Traversing a Link-List

```
struct node
{
    int          val;
    struct node *next;
};

void traverse (struct node *head)
{
    while (head)
    {
        head->value *= 2;
        head = head->next;
    }
}
```

Traversing a Link List

```
1:  movl  head,  %edx
2:  movl  head,  %eax
3:  movl  %(eax), %eax
4:  sall  %eax
5:  movl  %eax,  %(edx)
6:  movl  head,  %eax
7:  movl  4(%eax), %eax
8:  movl  %eax,  head
9:  cmpl  $0,    head
```

Naive assembly code
generated by gcc

Traversing a Link List

```
1:  movl  head,  %edx
2:  movl  head,  %eax
3:  movl  %(eax), %eax
4:  sall  %eax
5:  movl  %eax,  %(edx)
6:  movl  head,  %eax
7:  movl  4(%eax), %eax
8:  movl  %eax,  head
9:  cmpl  $0,    head
```

Naive assembly code
generated by gcc

Traversing a Link List

```
1:  movl  head,  %edx
2:  movl  head,  %eax
3:  movl  %(eax), %eax
4:  sall  %eax
5:  movl  %eax,  %(edx)
6:  movl  head,  %eax
7:  movl  4(%eax), %eax
8:  movl  %eax,  head
9:  cmpl  $0,    head
```

Naive assembly code
generated by gcc

```
1:  movl  head,  %edx
2:  movl  %edx,  %eax
3:  movl  %(eax), %eax
4:  sall  %eax
5:  movl  %eax,  %(edx)
6:  movl  head,  %eax
7:  movl  4(%eax), %eax
8:  movl  %eax,  head
9:  cmpl  $0,    head
```

Step One

Automatic Removal of Redundant Load

Traversing a Link List

```
1:  movl  head,  %edx
2:  movl  %edx,  %eax
3:  movl  %(eax), %eax
4:  sall  %eax
5:  movl  %eax,  %(edx)
6:  movl  head,  %eax
7:  movl  4(%eax), %eax
8:  movl  %eax,  head
9:  cmpl  $0,    head
```

Step One

Traversing a Link List

```
1:  movl  head,  %edx
2:  movl  %edx,  %eax
3:  movl  %(eax), %eax
4:  sall  %eax
5:  movl  %eax,  %(edx)
6:  movl  head,  %eax
7:  movl  4(%eax), %eax
8:  movl  %eax,  head
9:  cmpl  $0,   head
```

Step One

Traversing a Link List

```
1:  movl  head,  %edx
2:  movl  %edx,  %eax
3:  movl  %(eax), %eax
4:  sall  %eax
5:  movl  %eax,  %(edx)
6:  movl  head,  %eax
7:  movl  4(%eax), %eax
8:  movl  %eax,  head
9:  cmpl  $0,   head
```

Step One



```
1:  movl  head,  %edx
2:  movl  %edx,  %eax
3:  movl  %(eax), %eax
4:  sall  %eax
5:  movl  %eax,  %(edx)
6:  movl  head,  %eax
7:  movl  4(%eax), %eax
8:  movl  %eax,  head
9:  cmpl  $0,   %eax
```

Step Two

Automatic Elimination of Memory Access

Traversing a Link List

```
1:  movl  head,  %edx
2:  movl  %edx,  %eax
3:  movl  %(eax), %eax
4:  sall  %eax
5:  movl  %eax,  %(edx)
6:  movl  head,  %eax
7:  movl  4(%eax), %eax
8:  movl  %eax,  head
9:  cmpl  $0,    %eax
```

Step Two



```
1:  movl  head,  %edx
2:
3:
4:  sall  (%edx)
5:
6:  movl  head,  %eax
7:  movl  4(%eax), %eax
8:  movl  %eax,  head
9:  cmpl  $0,    %eax
```

Step Three

Automatic Instruction Selection

Traversing a Link List

```
1:  movl  head,  %edx
2:  sall  (%edx)
3:  movl  head,  %eax
4:  movl  4(%eax), %eax
5:  movl  %eax,  head
6:  cmpl  $0,   %eax
7:
8:
9:
```

Step Three



```
1:  movl  head,  %edx
2:  sall  (%edx)
3:  movl  %edx,  %eax
4:  movl  4(%eax), %eax
5:  movl  %eax,  head
6:  cmpl  $0,   %eax
7:
8:
9:
```

Step Four

Automatic Removal of Redundant Load

Traversing a Link List

```
1:  movl  head,  %edx
2:  sall  (%edx)
3:  movl  %edx,  %eax
4:  movl  4(%eax), %eax
5:  movl  %eax,  head
6:  cmpl  $0,   %eax
7:
8:
9:
```

Step Four



```
1:  movl  head,  %edx
2:  sall  (%edx)
3:
4:  movl  4(%edx), %eax
5:  movl  %eax,  head
6:  cmpl  $0,   %eax
7:
8:
9:
```

Step Five

Automatic Copy Propagation

Traversing a Link List

```
1:  movl  head,  %edx
2:  sall  (%edx)
3:  movl  4(%edx), %eax
4:  movl  %eax,  head
5:  cmpl  $0,    %eax
6:
7:
8:
9:
```

Step Five



```
1:  movl  head,  %edx
2:  sall  (%edx)
3:  movl  4(%edx), %edx
4:  movl  %edx,  head
5:  cmpl  $0,    %edx
6:
7:
8:
9:
```

Step Six

Automatic Register Usage Optimization

Traversing a Link List

```
1:  movl  head,  %edx
2:  sall  (%edx)
3:  movl  4(%edx), %edx
4:  movl  %edx,  head
5:  cmpl  $0,    %edx
6:
7:
8:
9:
```

Our optimizer

```
1:  sall  (%edx)
2:  movl  4(%edx), %edx
3:  cmpl  $0,    %edx
4:
5:
6:
7:
8:
9:
```

gcc-optimized

Traversing a Link List

```
1:  movl  head,  %edx
2:  sall  (%edx)
3:  movl  4(%edx), %edx
4:  movl  %edx,  head
5:  cmpl  $0,    %edx
6:
7:
8:
9:
```

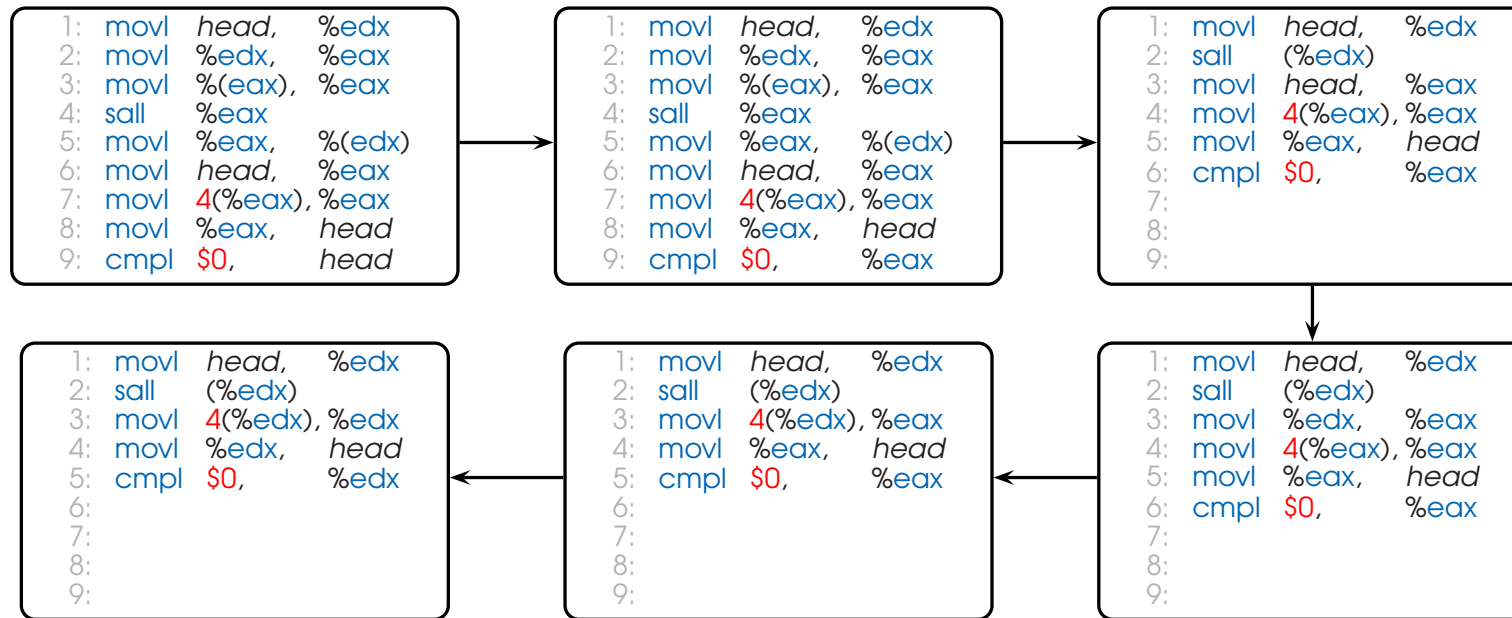
Our optimizer

```
1:  sall  (%edx)
2:  movl  4(%edx), %edx
3:  cmpl  $0,    %edx
4:
5:
6:
7:
8:
9:
```

gcc-optimized

Global Optimizations involving loop carried dependencies cannot be handled by peephole optimizations

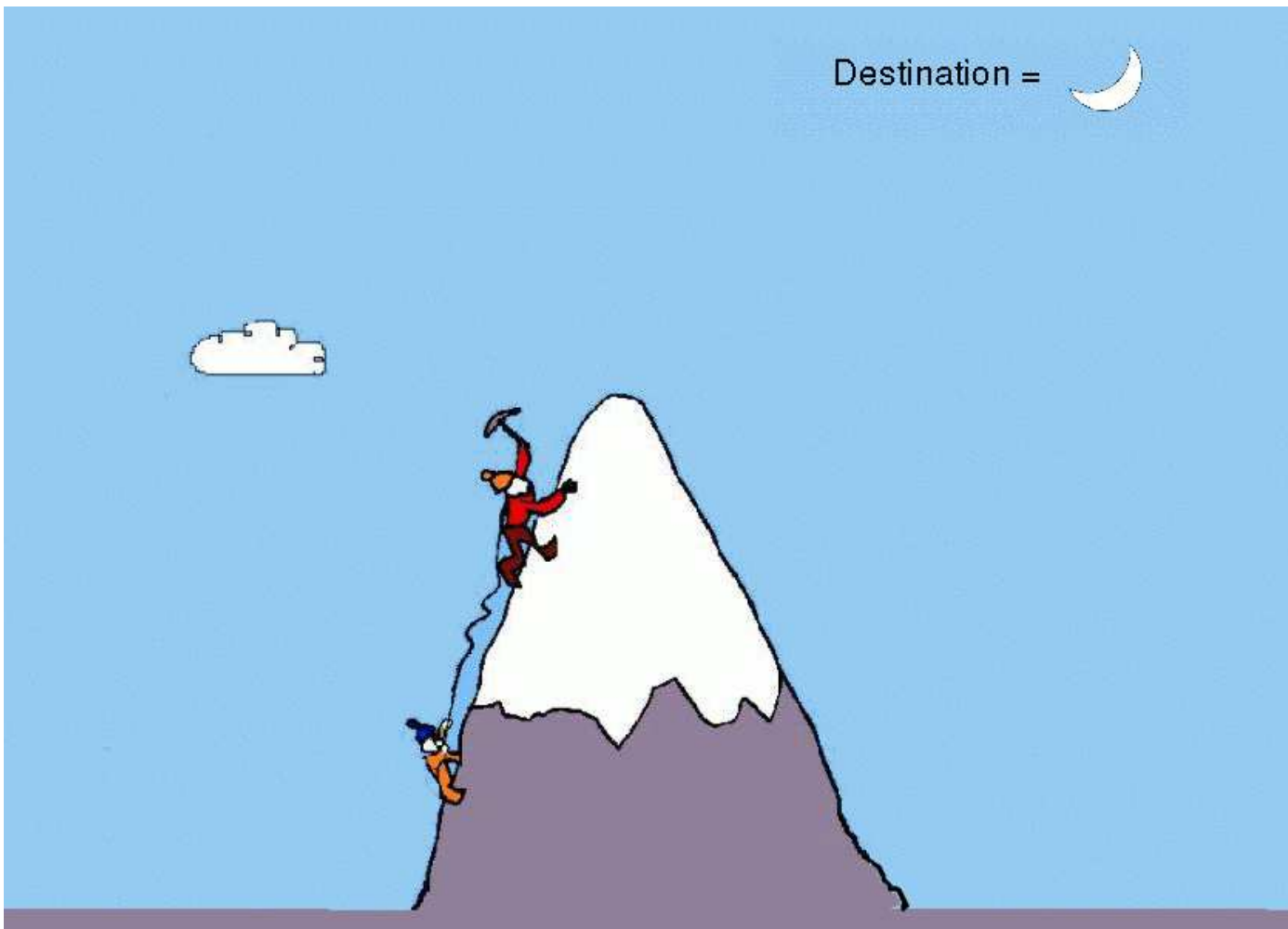
Traversing a Link-List



An automatically generated “peephole” optimizer can capture many traditional optimizations

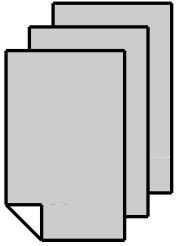
- Removal of Redundant Loads
- Instruction Selection
- Copy Propagation
- Reducing Register Usage





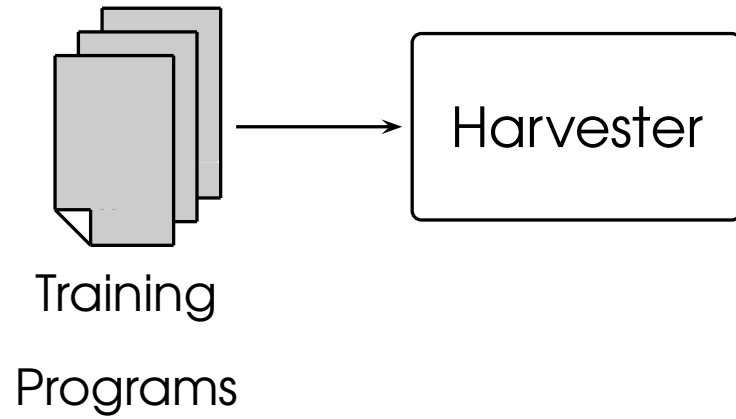
SYSTEM ARCHITECTURE

SYSTEM ARCHITECTURE

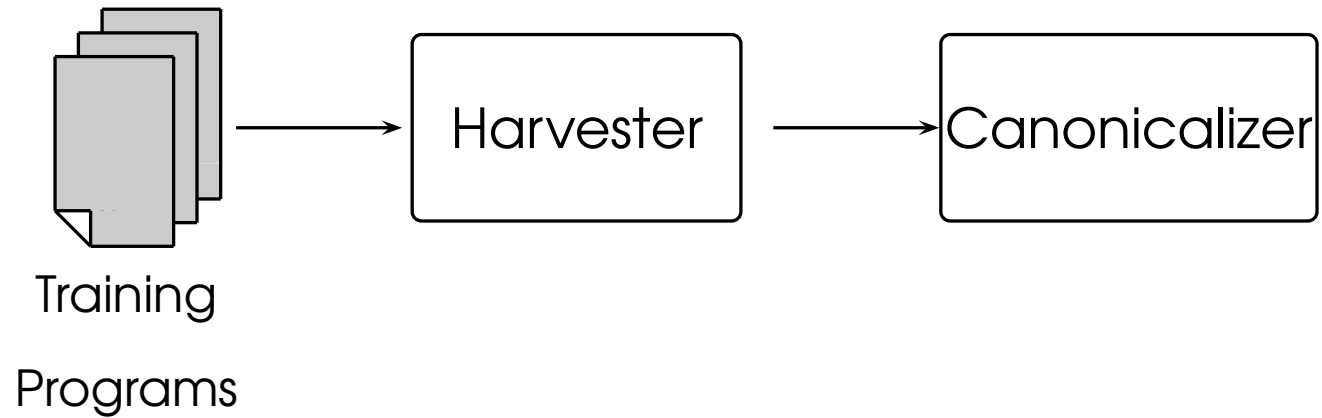


Training
Programs

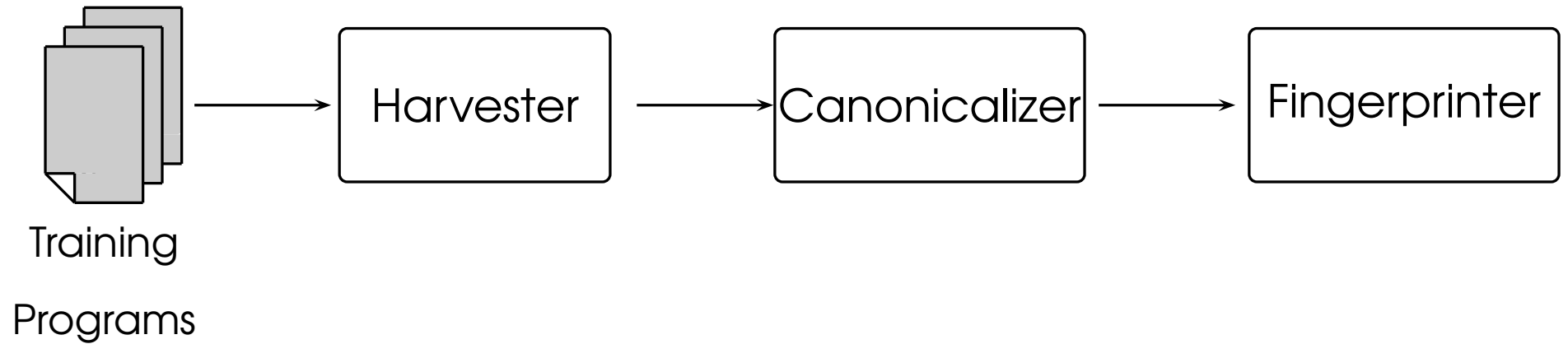
SYSTEM ARCHITECTURE



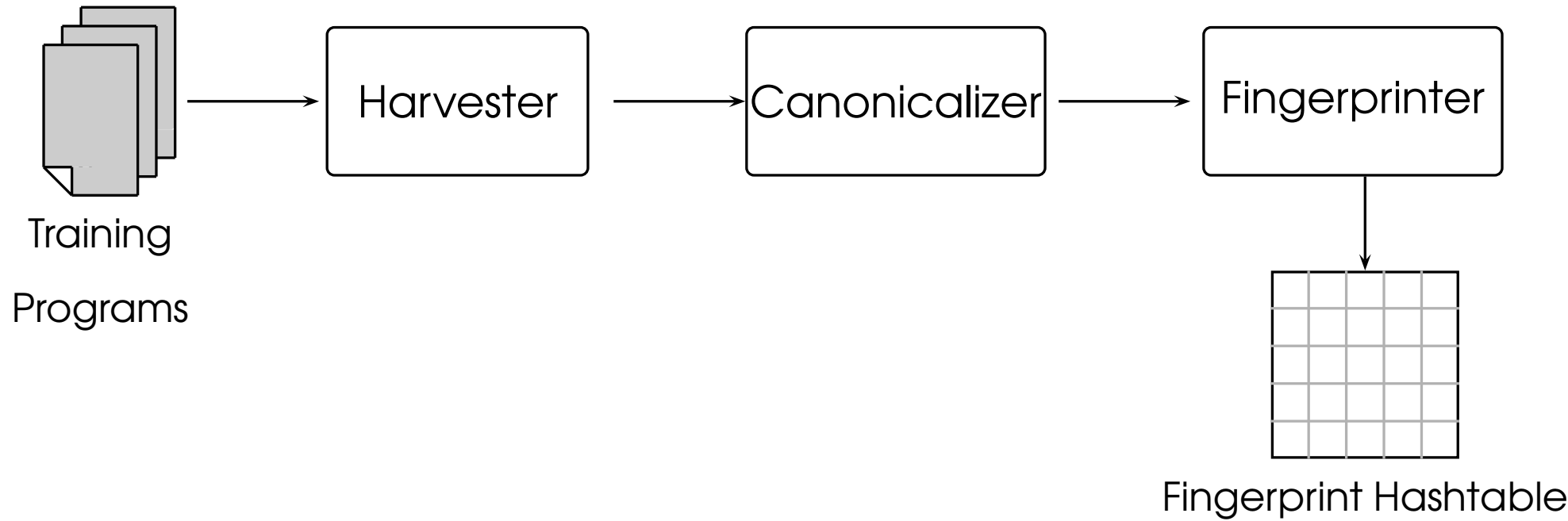
SYSTEM ARCHITECTURE



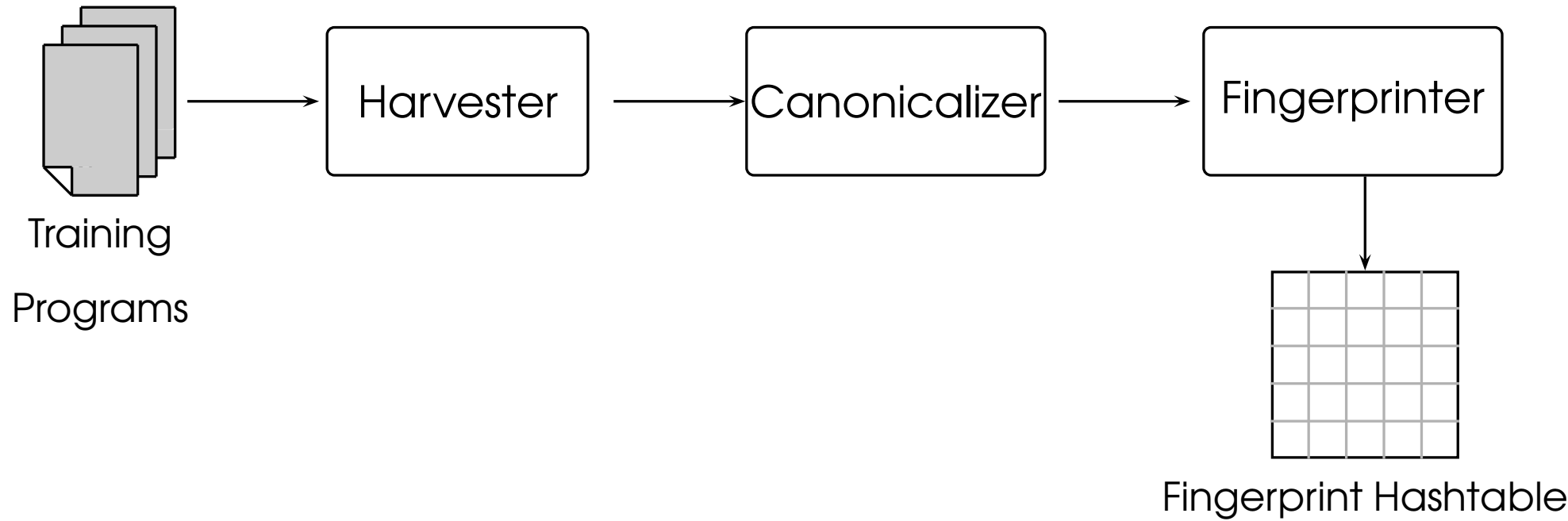
SYSTEM ARCHITECTURE



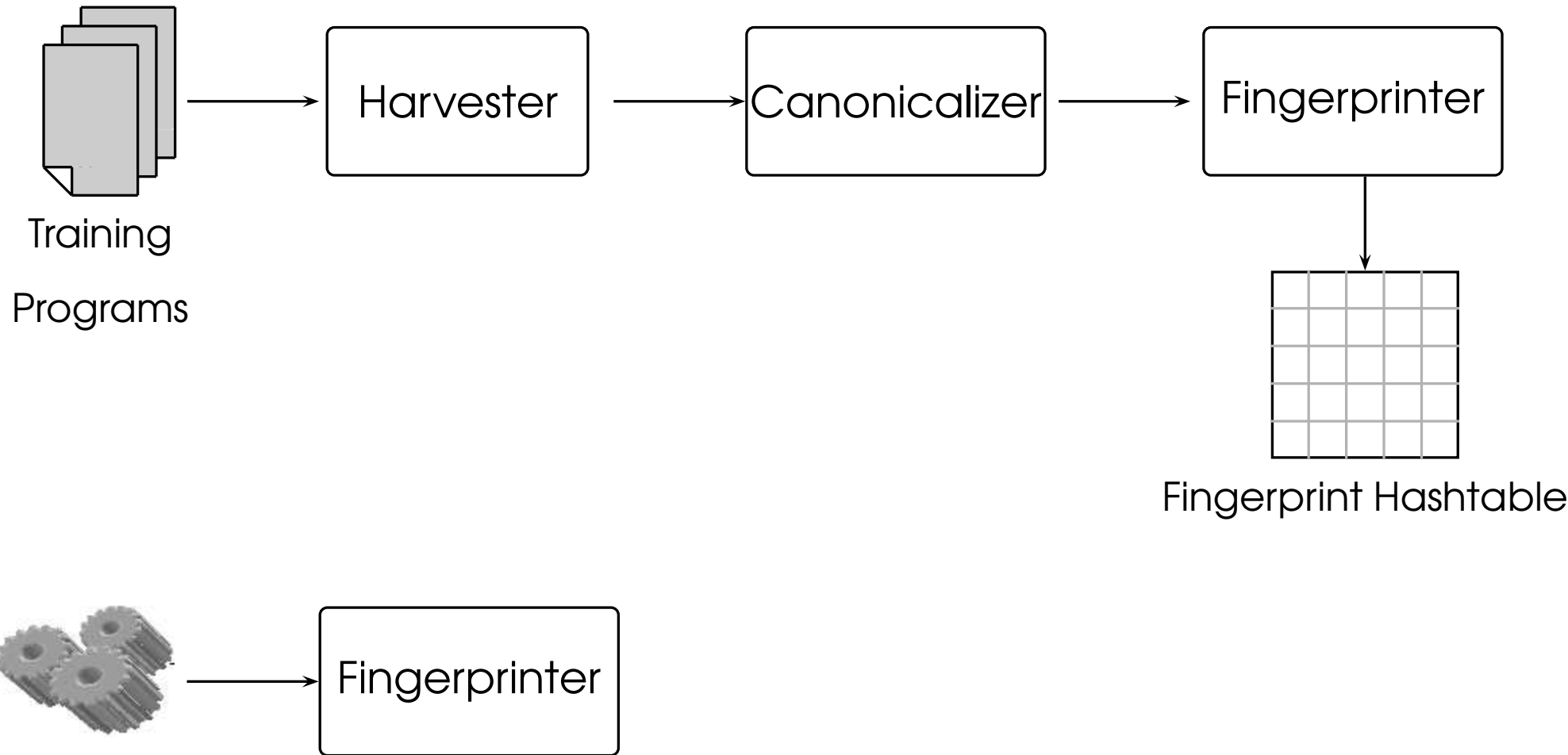
SYSTEM ARCHITECTURE



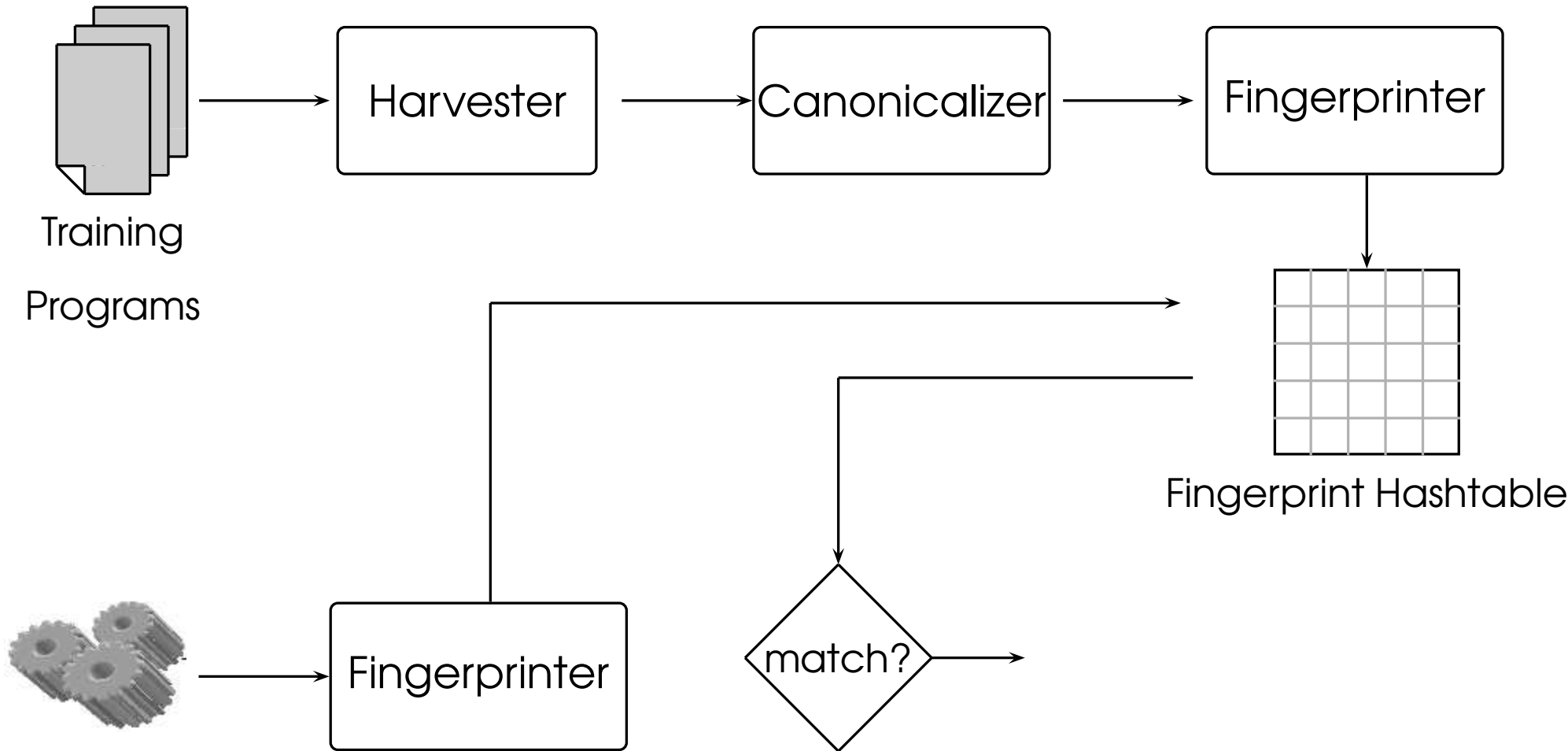
SYSTEM ARCHITECTURE



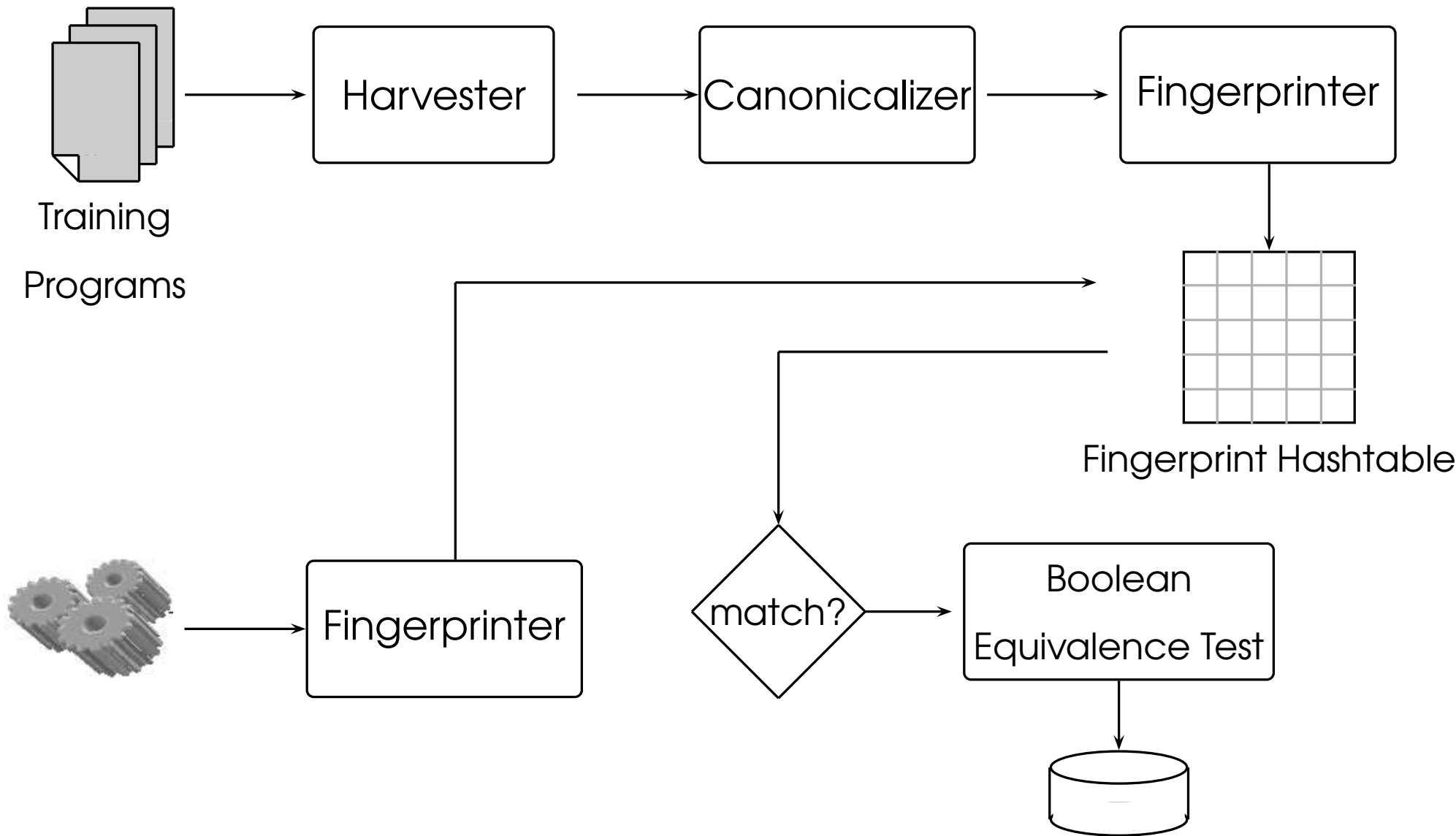
SYSTEM ARCHITECTURE



SYSTEM ARCHITECTURE



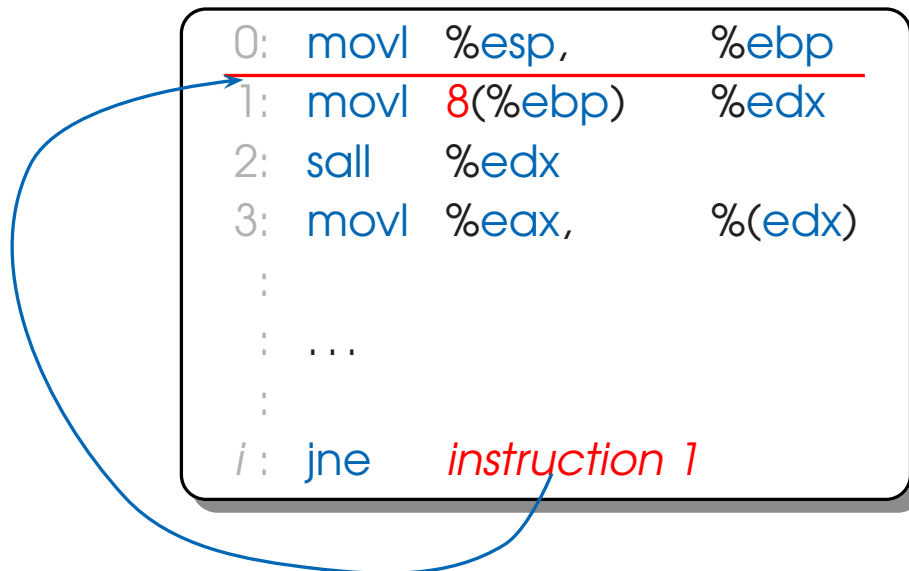
SYSTEM ARCHITECTURE



HARVESTER

Harvests *only* loop-free instruction sequences that have a single entry point

- no middle instruction is a jump target



HARVESTER

Harvests *only* loop-free instruction sequences that have a single entry point

- no middle instruction is a jump target



CANONICALIZER

```
movl %esp, %ebp  
movl %ebp, %esp
```



```
movl %esp, %ebp
```

CANONICALIZER

```
movl %esp, %ebp  
movl %ebp, %esp
```



```
movl %esp, %ebp
```

```
movl %eax, %ebp  
movl %ebp, %eax
```



```
movl %eax, %ebp
```

CANONICALIZER

```
movl %esp, %ebp  
movl %ebp, %esp
```

```
movl %esp, %ebp
```

```
movl %eax, %ebp  
movl %ebp, %eax
```

```
movl %eax, %ebp
```

```
movl %esp, %esi  
movl %esi, %esp
```

```
movl %esp, %esi
```

CANONICALIZER



There are 56 variants of this rule

CANONICALIZER



There are 56 variants of this rule



Reduce it to one rule

CANONICALIZER

```
addl %1234, %ebp  
addl %5678, %ebp
```



```
addl %7912, %ebp
```

CANONICALIZER

```
addl %1234, %ebp  
addl %5678, %ebp
```

```
addl %7912, %ebp
```

Use symbolic constants to deal with immediates

```
addl cons0, %ebp  
addl cons1, %ebp
```

```
addl cons0+cons1, %ebp
```

CANONICALIZER

Rename the target sequence to get it in its *canonical* form:

Registers

- First register used is renamed to `r0`
- Second *distinct* register used is renamed to `r1`
- and so on...

Constants

- First constant used is renamed to `c0` (a pre-defined constant)
- Second *distinct* constant is renamed to `c1`

Memory Addresses

- Rename a memory access in `s-i-b` form to a dereference of the canonical name of its base (and index) register

CANONICALIZER

Only instruction sequences in canonical form are enumerated

CANONICALIZER

Only instruction sequences in canonical form are enumerated

```
movl r3, r4  
movl r5, r3
```



CANONICALIZER

Only instruction sequences in canonical form are enumerated

```
movl r3, r4  
movl r5, r3
```




```
movl r0, r1  
movl r2, r0
```




CANONICALIZER

Only instruction sequences in canonical form are enumerated

```
movl r3, r4  
movl r5, r3
```



```
movl r0, r1  
movl r2, r0
```




Special Constants are enumerated for immediate operands

- 0 ; 1
- c0 ; c1
- c0 + 1 ; c0 - 1
- c0 + c1 ; c0 - c1


CANONICALIZER

Only instruction sequences in canonical form are enumerated

```
movl r3, r4  
movl r5, r3
```



```
movl r0, r1  
movl r2, r0
```

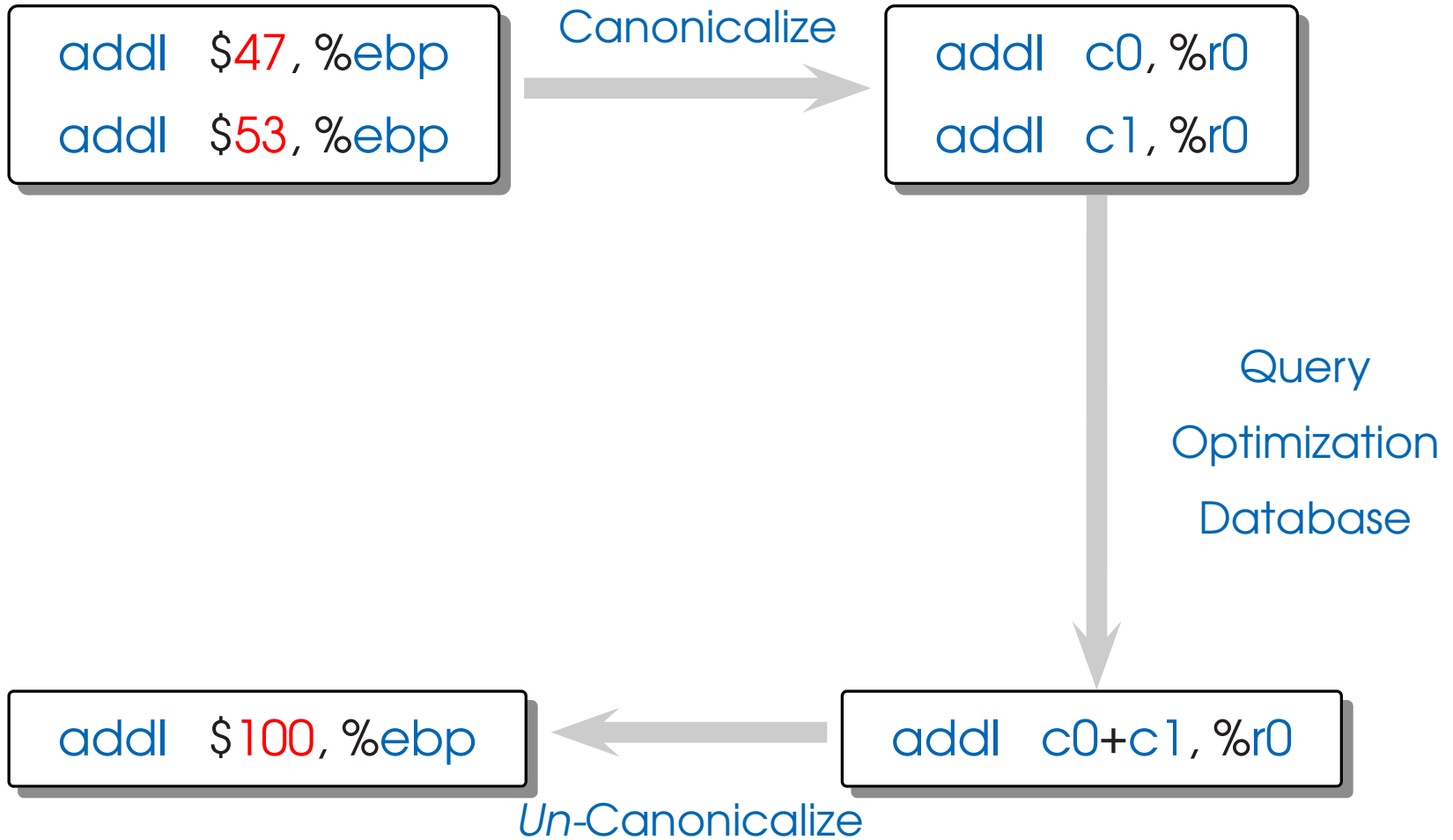


Special Constants are enumerated for immediate operands

- 0 ; 1
- c0 ; c1
- c0 + 1 ; c0 - 1
- c0 + c1 ; c0 - c1

LHS of an optimization rule is always in canonical form

CANONICALIZER



FINGERPRINTER

Execute the instruction sequences on random-bit states

Use hash of result to compute a *fingerprint*

FINGERPRINTER

Execute the instruction sequences on random-bit states

Use hash of result to compute a *fingerprint*

Memory is approximated by a 256 byte array

- Two equivalent accesses guaranteed to access the same location in array

FINGERPRINTER

Execute the instruction sequences on random-bit states

Use hash of result to compute a *fingerprint*

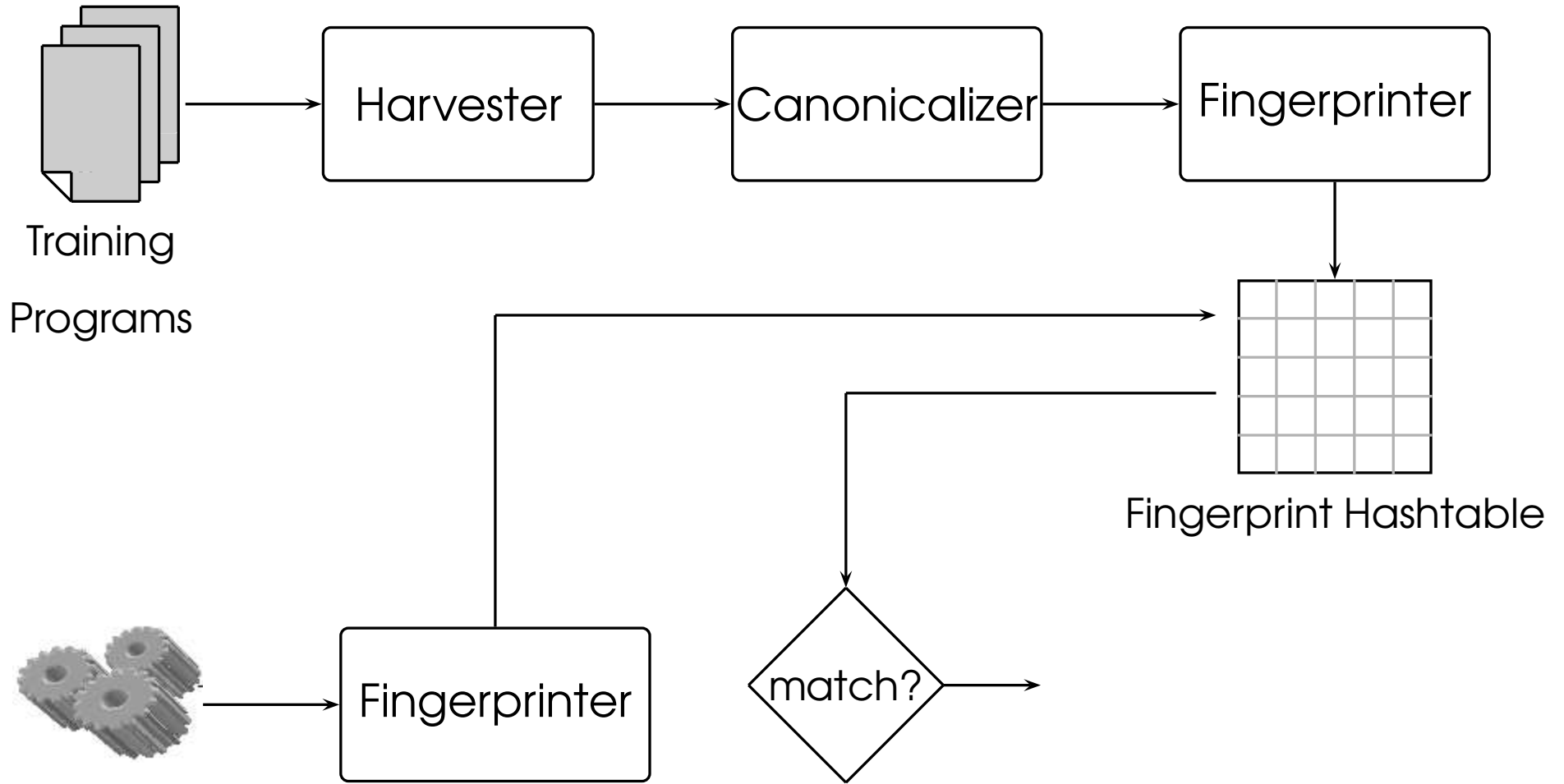
Memory is approximated by a 256 byte array

- Two equivalent accesses guaranteed to access the same location in array

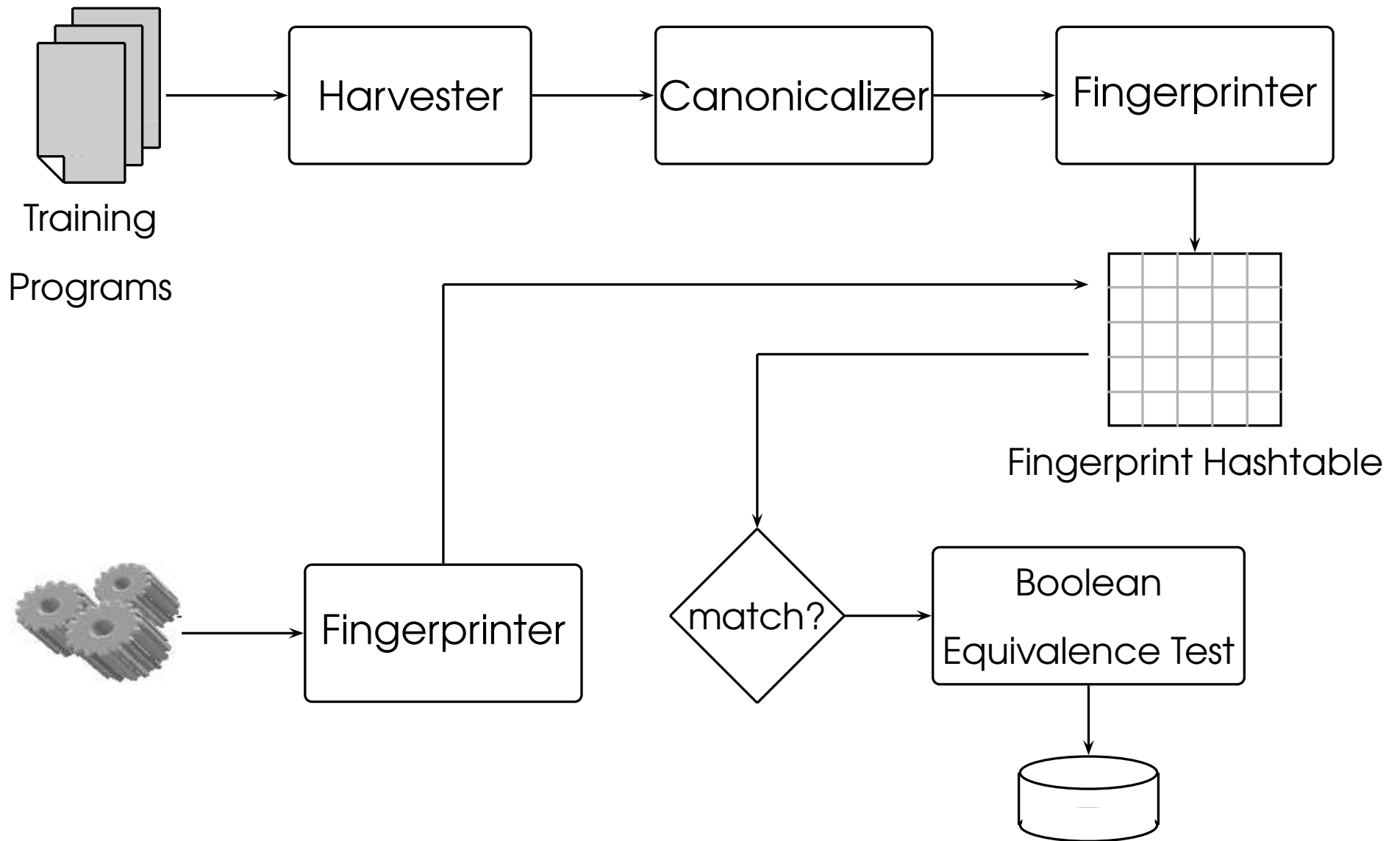
Fingerprint computed by direct execution on hardware

- Fingerprint of a sequence computed in $<3\mu s$

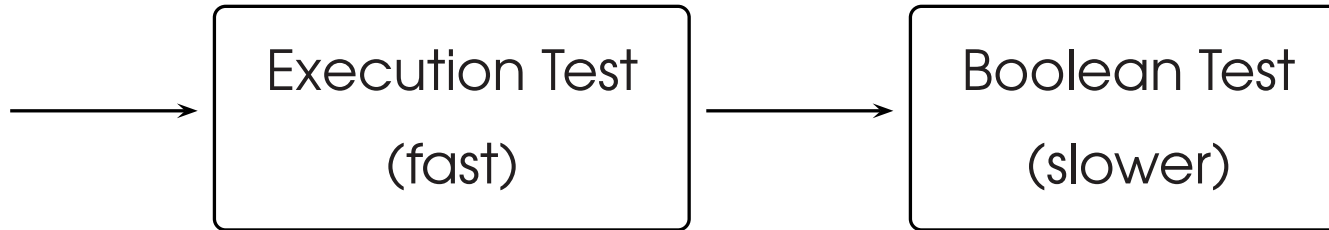
SYSTEM ARCHITECTURE



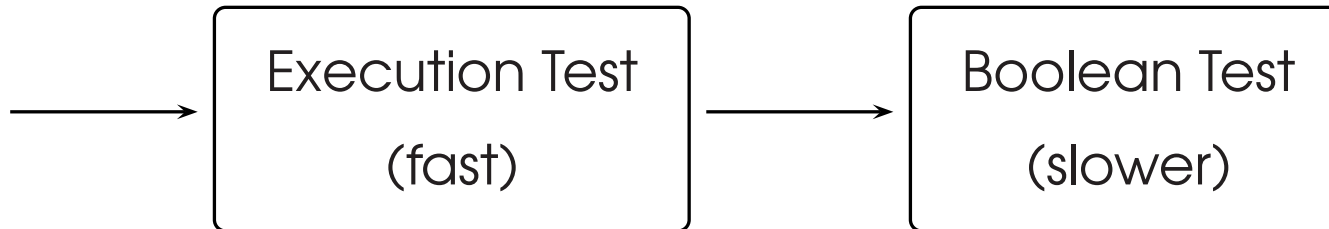
SYSTEM ARCHITECTURE



EQUIVALENCE TEST



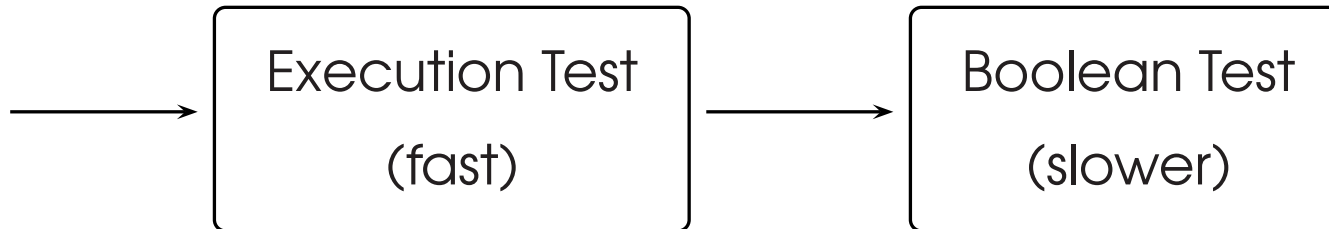
EQUIVALENCE TEST



Execution Test

Fingerprint on many different states

EQUIVALENCE TEST



Execution Test

Fingerprint on many different states

Boolean Test

Use SAT Solver

BOOLEAN TEST

Machine state represented by a set of registers and a model of memory

BOOLEAN TEST

Machine state represented by a set of registers and a model of memory

Memory model

- Map from address expressions to data expressions
- Aliasing handled by comparing addresses of multiple accesses

BOOLEAN TEST

Machine state represented by a set of registers and a model of memory

Memory model

- Map from address expressions to data expressions
- Aliasing handled by comparing addresses of multiple accesses

Instructions encoded as boolean circuits

input state → *output state*

BOOLEAN TEST

Machine state represented by a set of registers and a model of memory

Memory model

- Map from address expressions to data expressions
- Aliasing handled by comparing addresses of multiple accesses

Instructions encoded as boolean circuits

input state → *output state*

Use a SAT Solver to compute equivalence

CONTEXT SENSITIVITY

Equivalence of two instruction sequences is defined under the set of registers live beyond the sequence itself

EXPERIMENTAL RESULTS

Length	Original Search Space	After Canonicalize and Prune
1	5,606	654
2	31 m	1.09 m
3	176 b	2.8 b

Search Space

Exhaustively enumerate sequences up to length 3

EXPERIMENTAL RESULTS

Integer addition

```
for (i = 0; i < 64; i++)  
    sum += a[i]
```

EXPERIMENTAL RESULTS

Integer addition

```
for (i = 0; i < 64; i++)  
    sum += a[i]
```

psadbw: sum of absolute differences of 8 consecutive integers

```
psubb    %mm0, %mm0  
psadbw   &a[i], %mm0  
movd     %mm0, sum
```

3x faster

EXPERIMENTAL RESULTS

Comparison

```
c[i] = (a[i] < b[i]) ? c0 : c1
```

7x

Minimum

```
c[i] = (a[i] < b[i]) ? a[i] : b[i]
```

8x

Pixel-difference

```
sum += abs(a[i] - b[i])
```

10x

XOR

```
c[i] = a[i] ⊕ b[i]
```

2x

Sprite Copy

```
c[i] = (a[i] == 0) ? b[i] : a[i]
```

2x

MMX and conditional-move instructions are still under-used

EXPERIMENTAL RESULTS

We evaluate our optimizer on SPEC benchmarks compiled using `gcc`

Use two cost functions

- Codesize
- Runtime
 - Number of memory accesses
 - Number of jump instructions
 - Instruction costs

EXPERIMENTAL RESULTS

Program	Codesize Improvement
gzip	3.95%
mcf	5.86%
crafty	1.71%
bzip2	4.58%
gcc	1.12%
twolf	1.47%
parser	3.06%

Codesize improvement on executables
already optimized for size (-Os)

Codesize Improvement: 1 – 6%

EXPERIMENTAL RESULTS

Program	Instruction Count Improvement
gzip	4.16%
mcf	3.73%
crafty	2.19%
bzip2	4.11%
gcc	2.44%
twolf	2.17%
parser	3.84%

Instruction count improvement on
optimized executables (-O2)

Instruction Count Improvement: 2 – 4%

Speedup: 1 – 5%

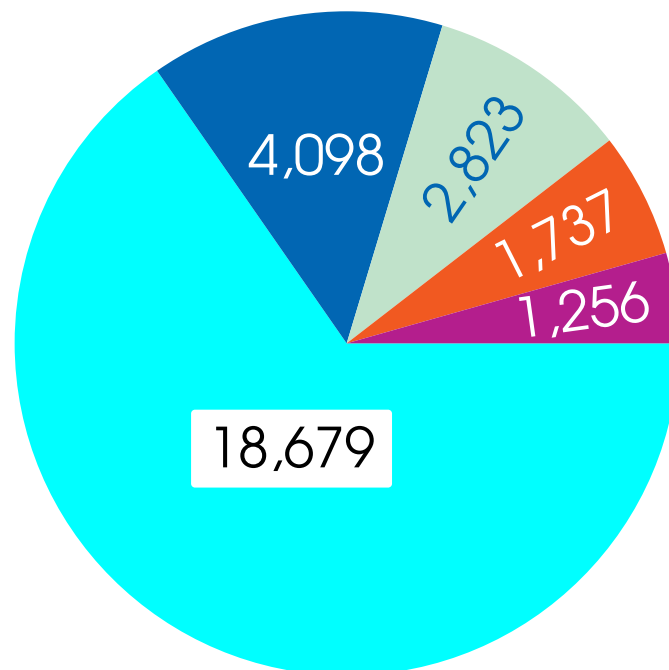
EXPERIMENTAL RESULTS

Number of Distinct Optimization Rules Learnt

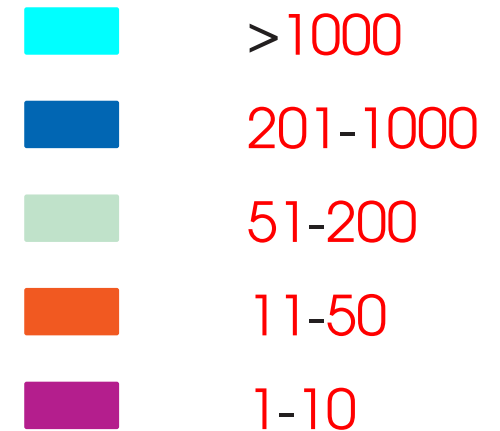
Codesize 3000

Runtime 2100

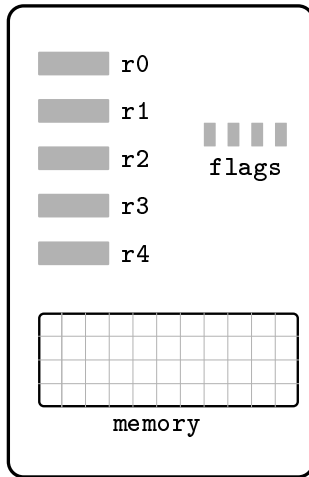
Re-use of Optimization Rules



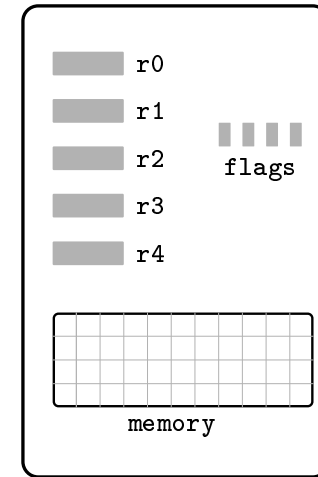
Frequency of Use



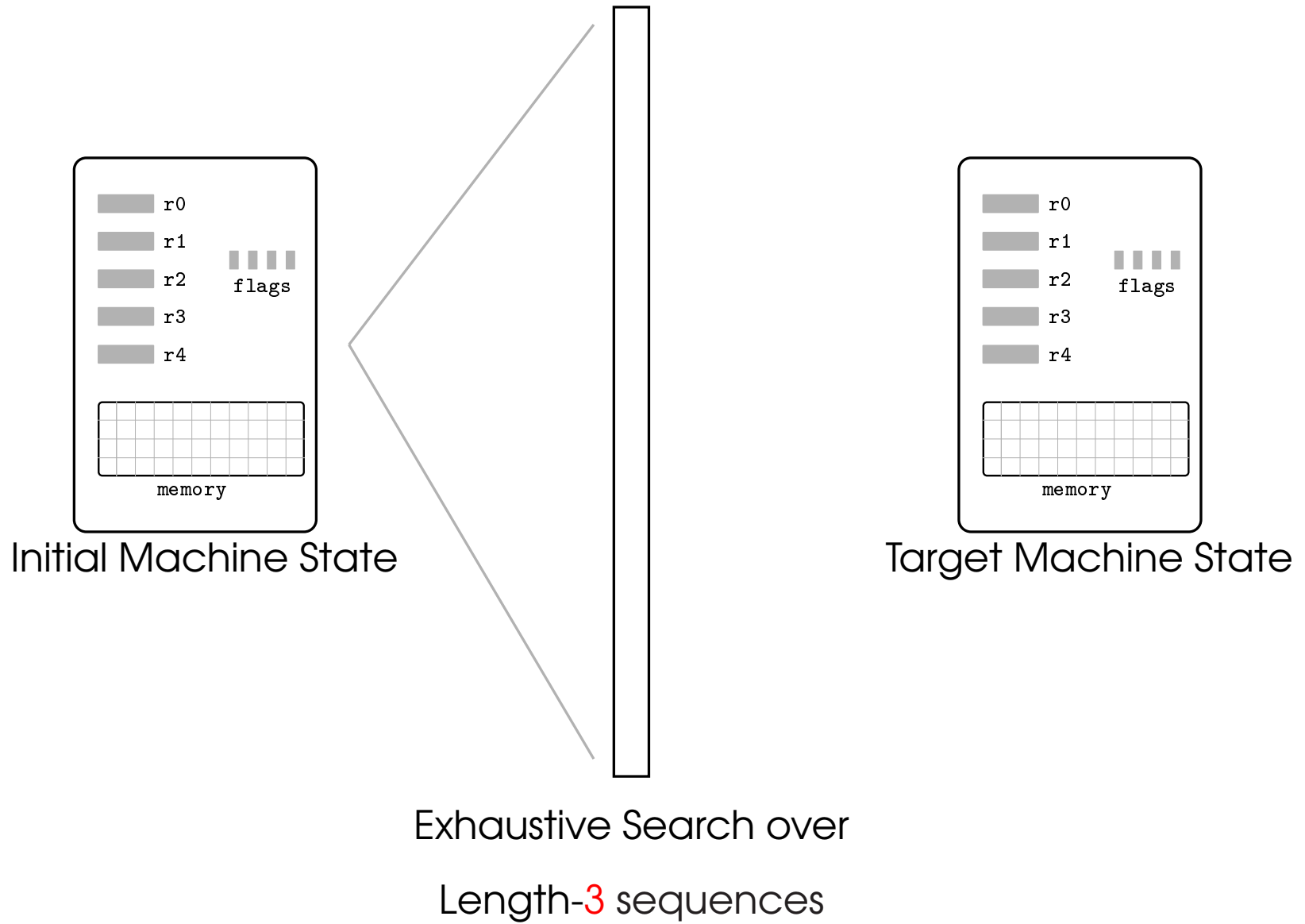
Total: 28,593 optimizations

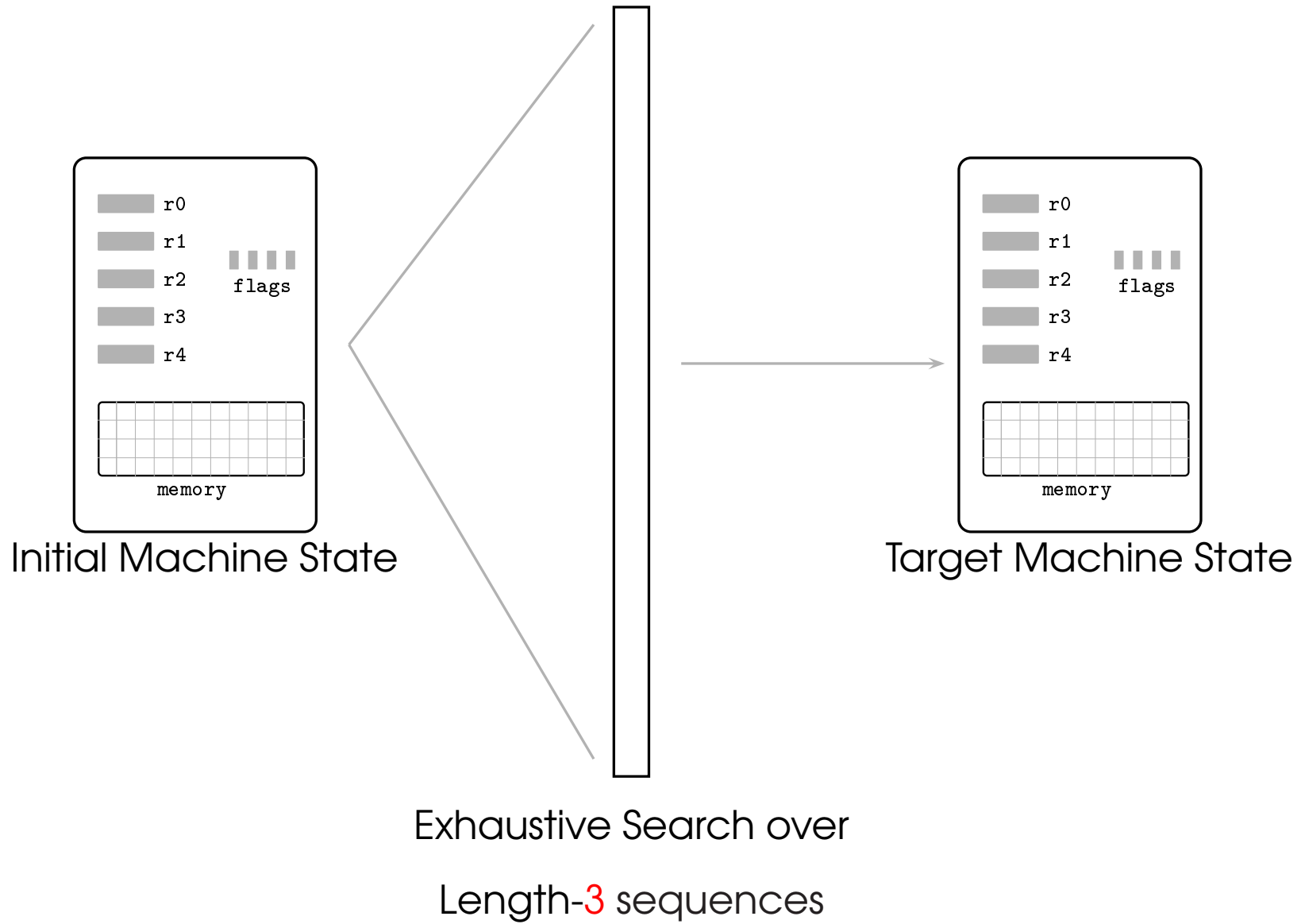


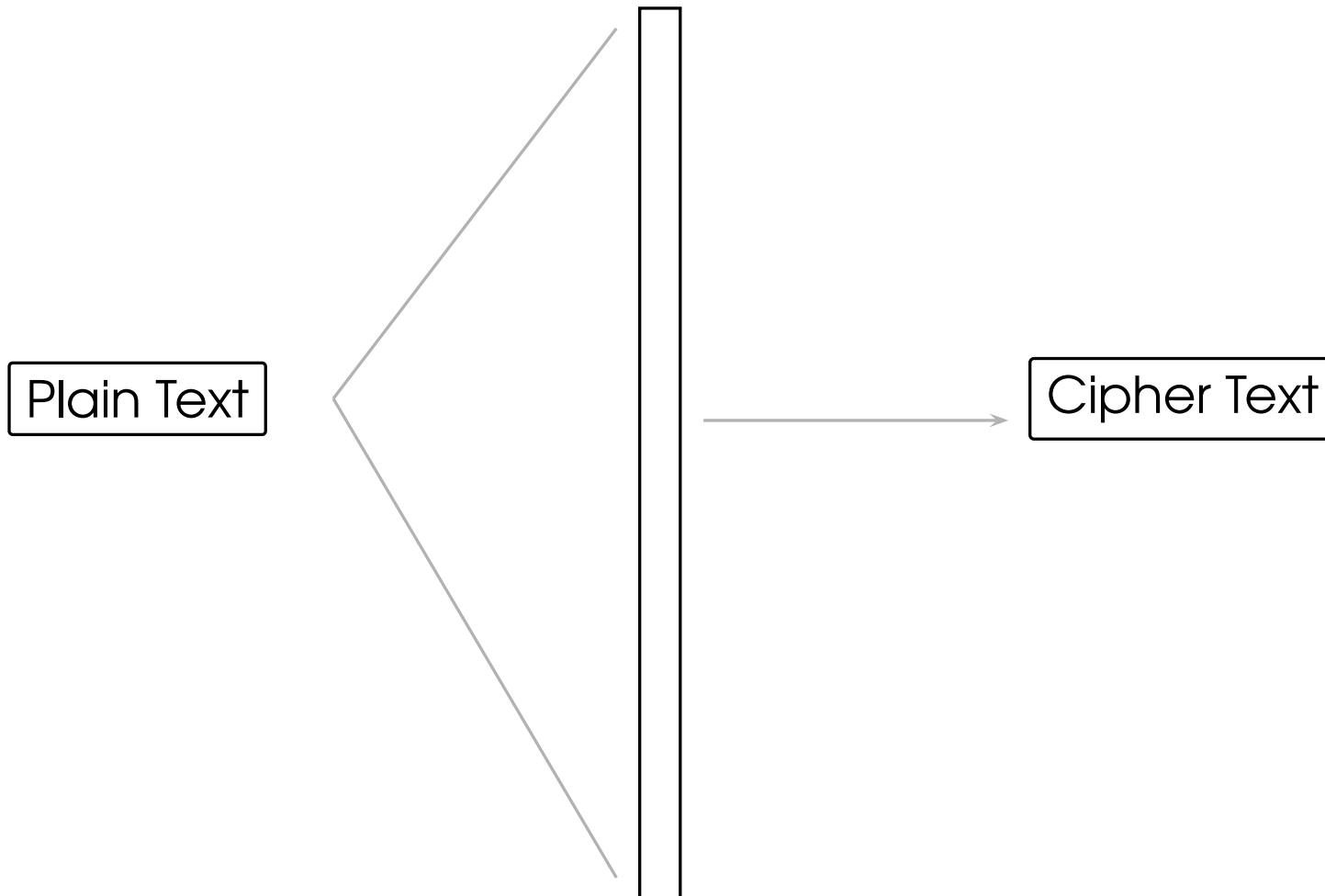
Initial Machine State



Target Machine State







Exhaustive Search over
Length ~~3~~ sequences
56-bit keys

$$CT = \text{Encrypt}_{56}(PT)$$

$$CT = \text{Encrypt}_{56}(PT)$$

$$\text{If, } Key_{56} = Key_{28} \circ Key_{28}$$

$$\text{s.t., } CT = \text{Encrypt}_{28}^1(\text{Encrypt}_{28}^2(PT))$$

$$CT = \text{Encrypt}_{56}(PT)$$

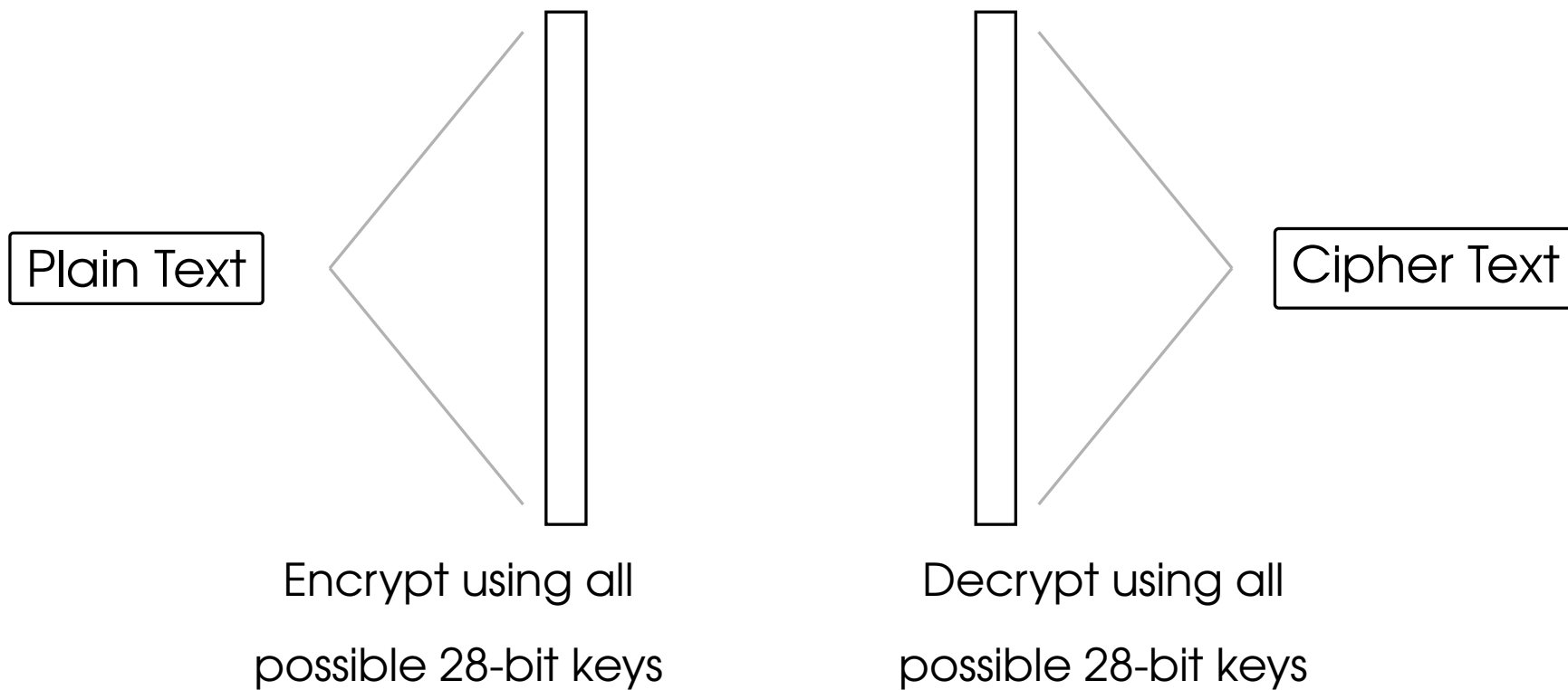
$$\text{If, } Key_{56} = Key_{28} \circ Key_{28}$$

$$\text{s.t., } CT = \text{Encrypt}_{28}^1(\text{Encrypt}_{28}^2(PT))$$

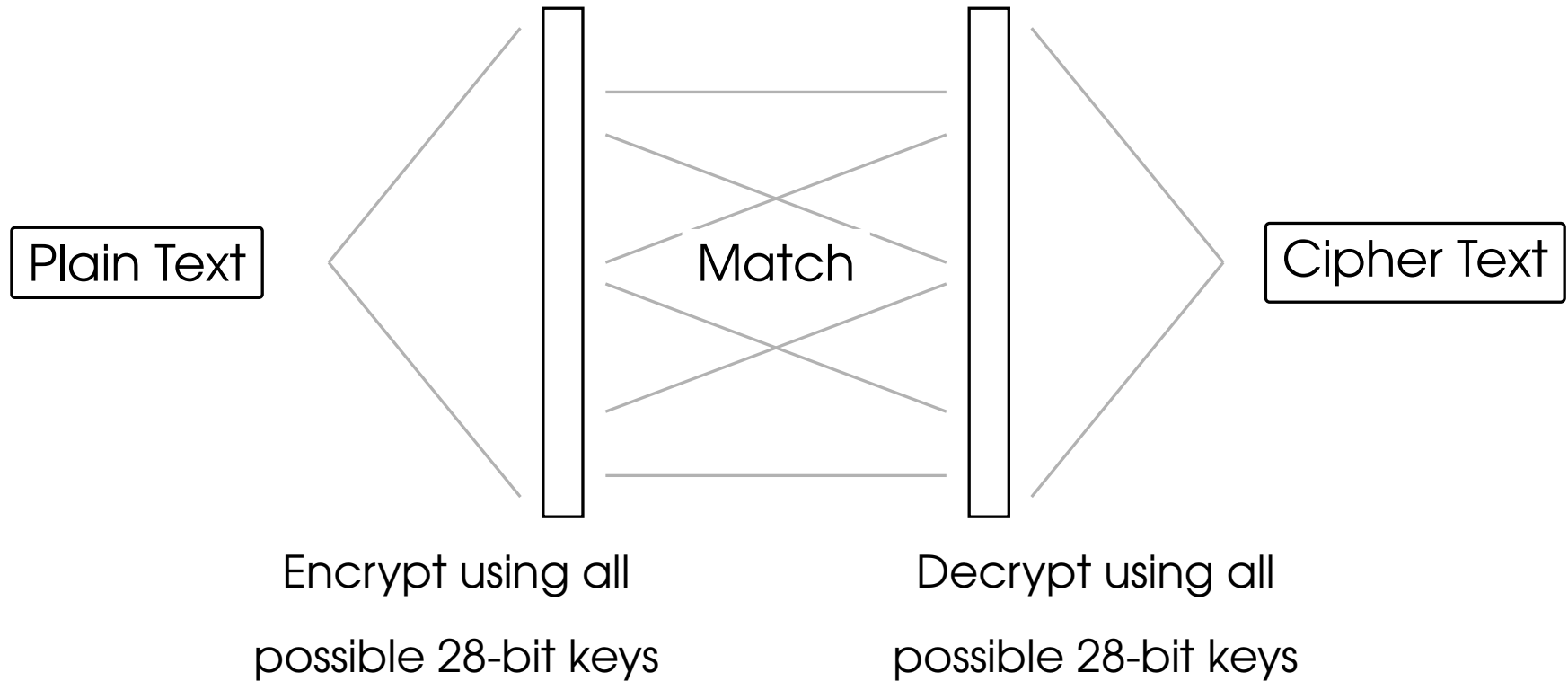
Then,

$$\text{Decrypt}_{28}^1(CT) = \text{Encrypt}_{28}^2(PT)$$

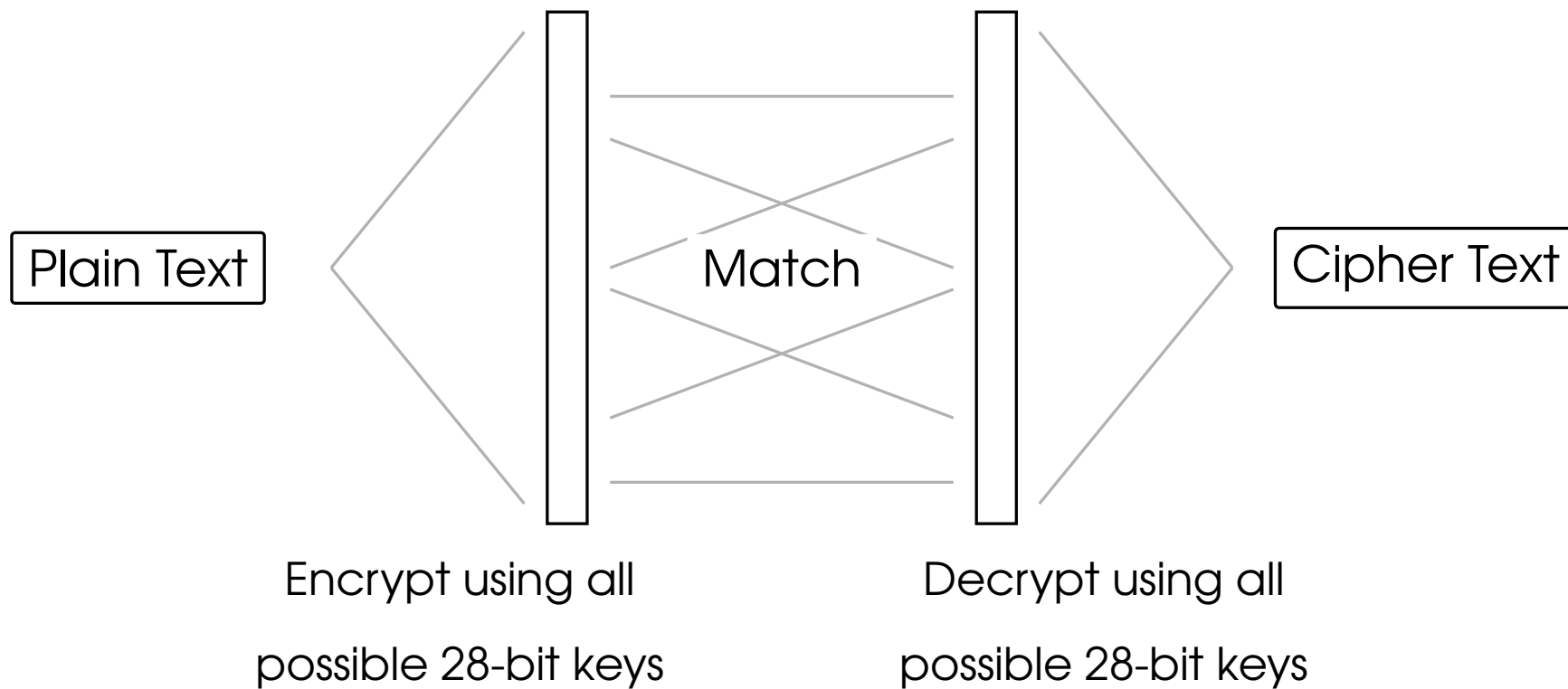
MEET-IN-MIDDLE



MEET-IN-MIDDLE

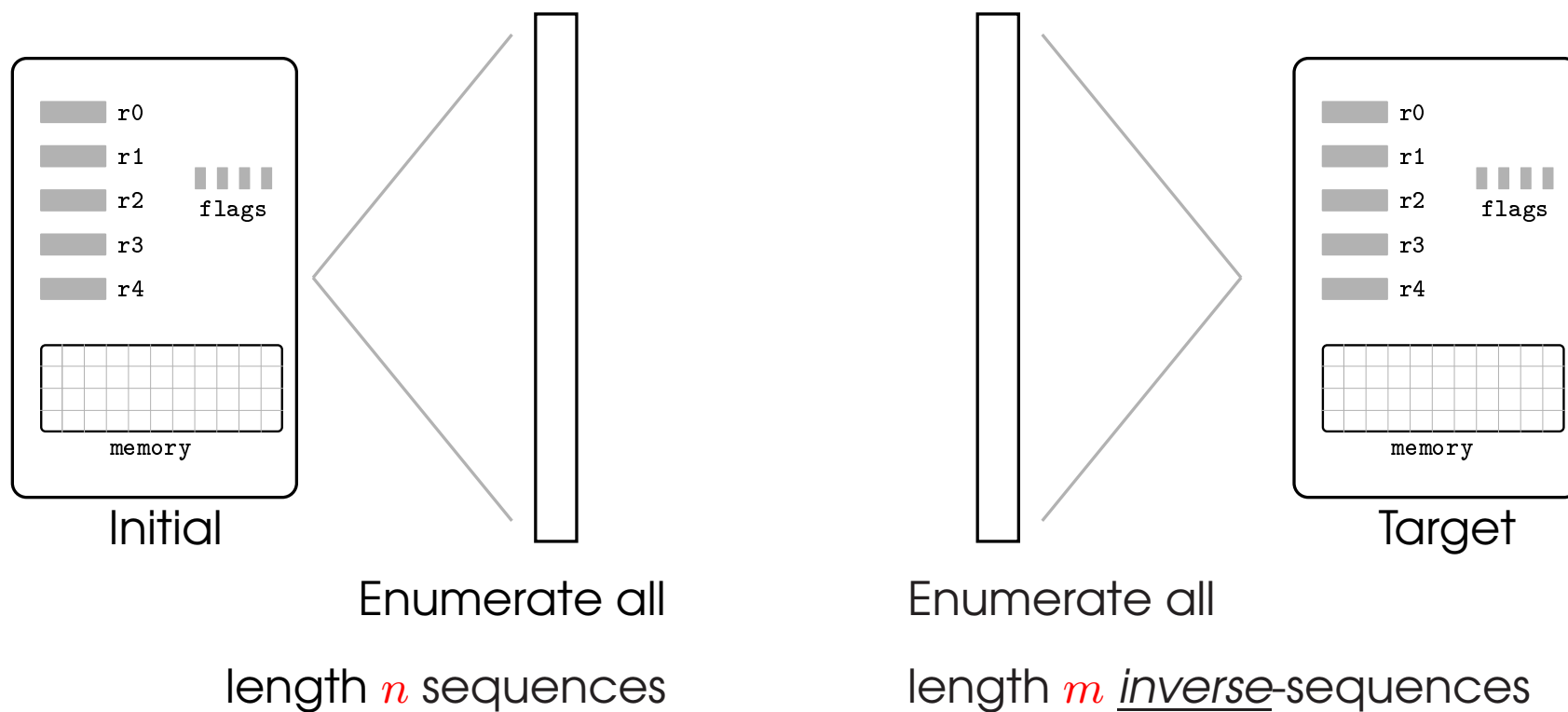


MEET-IN-MIDDLE

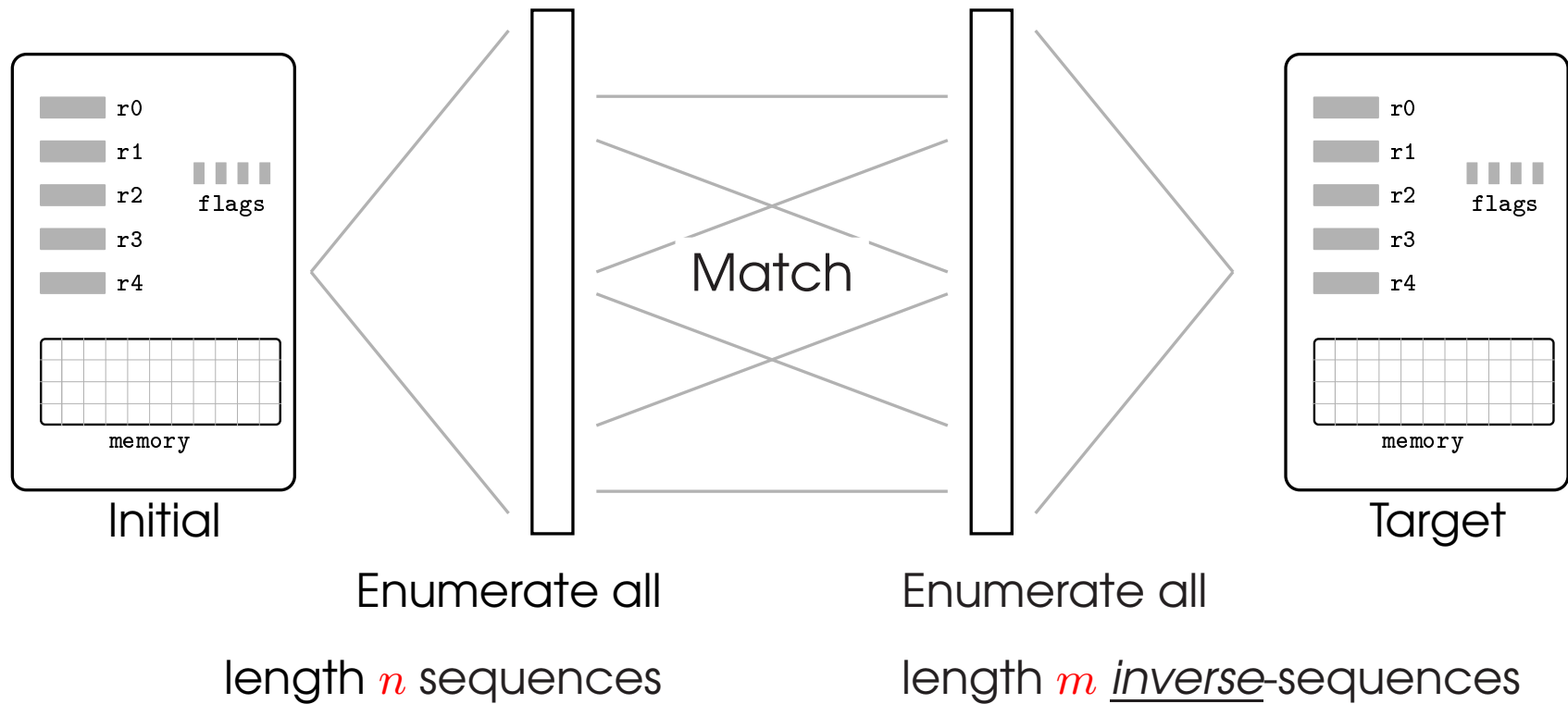


$$O(2^{28})$$

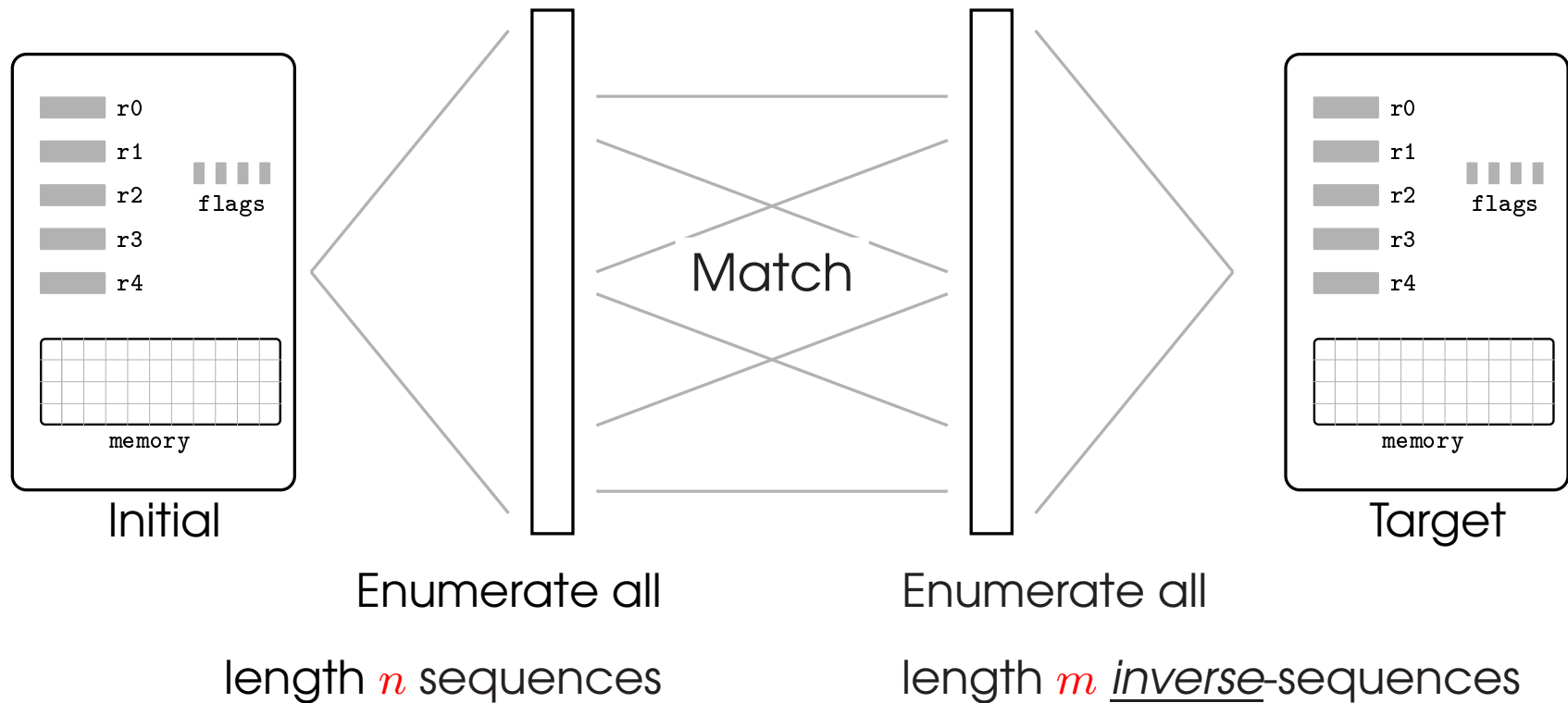
MEET-IN-MIDDLE



MEET-IN-MIDDLE



MEET-IN-MIDDLE



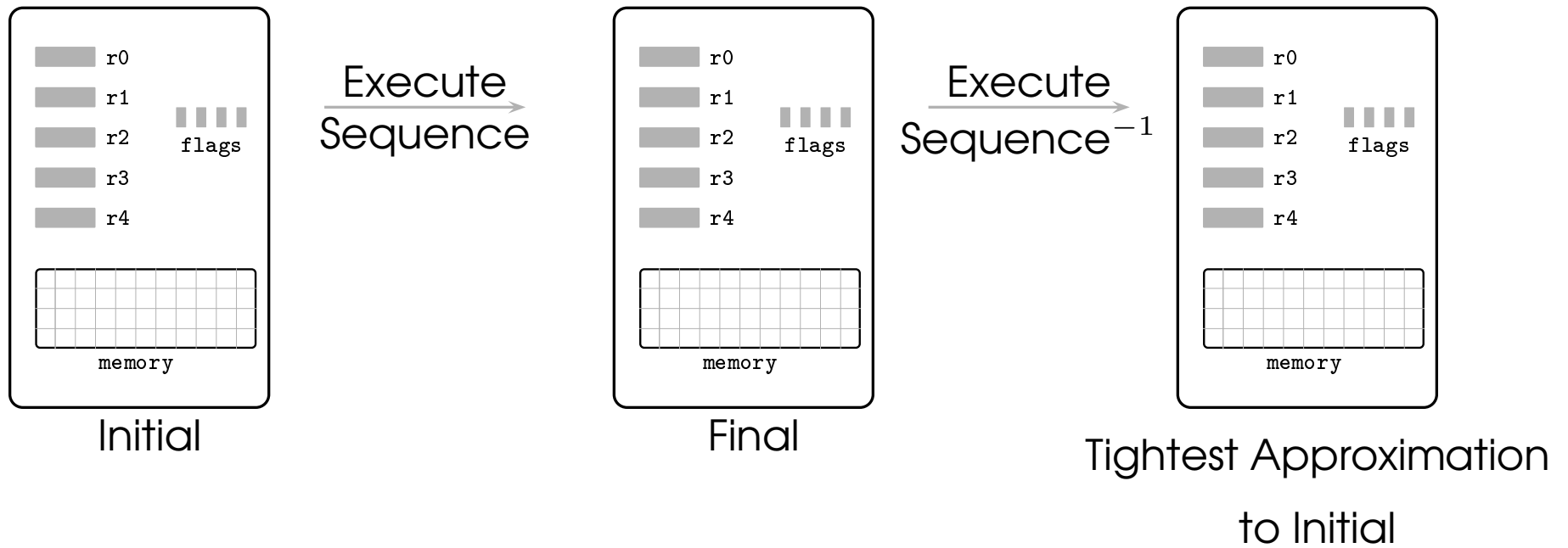
Effective Enumerated Length: $m + n$

MEET-IN-MIDDLE

What is the inverse of an instruction sequence?

How fast can we *match*?

INVERSE OF AN INSTRUCTION SEQUENCE



Approximate using *don't-know* bits

INSTRUCTION INVERSE

Instruction	Inverse
<code>add r0, r1</code>	<code>sub r0,r1</code> flags ← d

INSTRUCTION INVERSE

Instruction	Inverse	
<code>add r0, r1</code>	<code>sub r0,r1</code>	flags ← d
<code>add r0, r0</code>	<code>shr r0; r0[31] ← d</code>	flags ← d
<code>mov r0, r1</code>	<code>r0 ← d</code>	
<code>and r0, \$010110</code>	<code>r0 ← r0 & d1d11d</code>	flags ← d

INSTRUCTION INVERSE

Instruction	Inverse	
<code>add r0, r1</code>	<code>sub r0,r1</code>	flags ← d
<code>add r0, r0</code>	<code>shr r0; r0[31] ← d</code>	flags ← d
<code>mov r0, r1</code>	<code>r0 ← d</code>	
<code>and r0, \$010110</code>	<code>r0 ← r0 & d1d11d</code>	flags ← d
<code>or r0, \$010110</code>	<code>r0 ← r0 0d0dd0</code>	flags ← d

INSTRUCTION INVERSE

Instruction	Inverse	
<code>add r0, r1</code>	<code>sub r0,r1</code>	flags ← d
<code>add r0, r0</code>	<code>shr r0; r0[31] ← d</code>	flags ← d
<code>mov r0, r1</code>	<code>r0 ← d</code>	
<code>and r0, \$010110</code>	<code>r0 ← r0 & d1d11d</code>	flags ← d
<code>or r0, \$010110</code>	<code>r0 ← r0 0d0dd0</code>	flags ← d
<code>xor r0, r1</code>	<code>xor r0, r1</code>	flags ← d

INSTRUCTION INVERSE

Instruction	Inverse	
<code>add r0, r1</code>	<code>sub r0,r1</code>	flags ← d
<code>add r0, r0</code>	<code>shr r0; r0[31] ← d</code>	flags ← d
<code>mov r0, r1</code>	<code>r0 ← d</code>	
<code>and r0, \$010110</code>	<code>r0 ← r0 & d1d11d</code>	flags ← d
<code>or r0, \$010110</code>	<code>r0 ← r0 0d0dd0</code>	flags ← d
<code>xor r0, r1</code>	<code>xor r0, r1</code>	flags ← d
<code>xchg r0, r1</code>	<code>xchg r0, r1</code>	

INSTRUCTION INVERSE

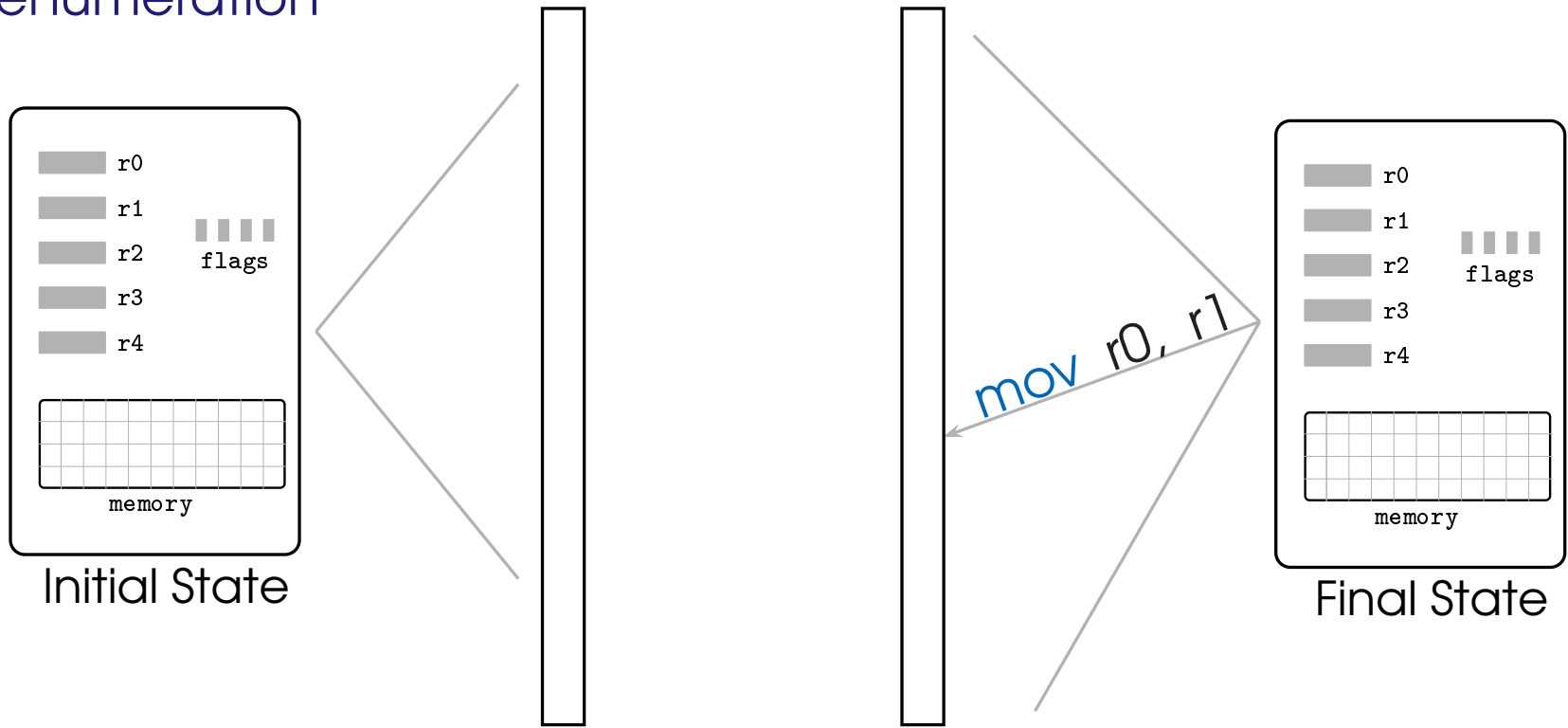
Instruction	Inverse
<code>add r0, r1</code>	<code>sub r0,r1</code> flags ← d
<code>add r0, r0</code>	<code>shr r0; r0[31] ← d</code> flags ← d
<code>mov r0, r1</code>	<code>r0 ← d</code>
<code>and r0, \$010110</code>	<code>r0 ← r0 & d1d11d</code> flags ← d
<code>or r0, \$010110</code>	<code>r0 ← r0 0d0dd0</code> flags ← d
<code>xor r0, r1</code>	<code>xor r0, r1</code> flags ← d
<code>xchg r0, r1</code>	<code>xchg r0, r1</code>
<code>shr r0, \$3</code>	<code>shl r0, \$3; r0[0..2] ← d</code> flags ← d

INSTRUCTION INVERSE

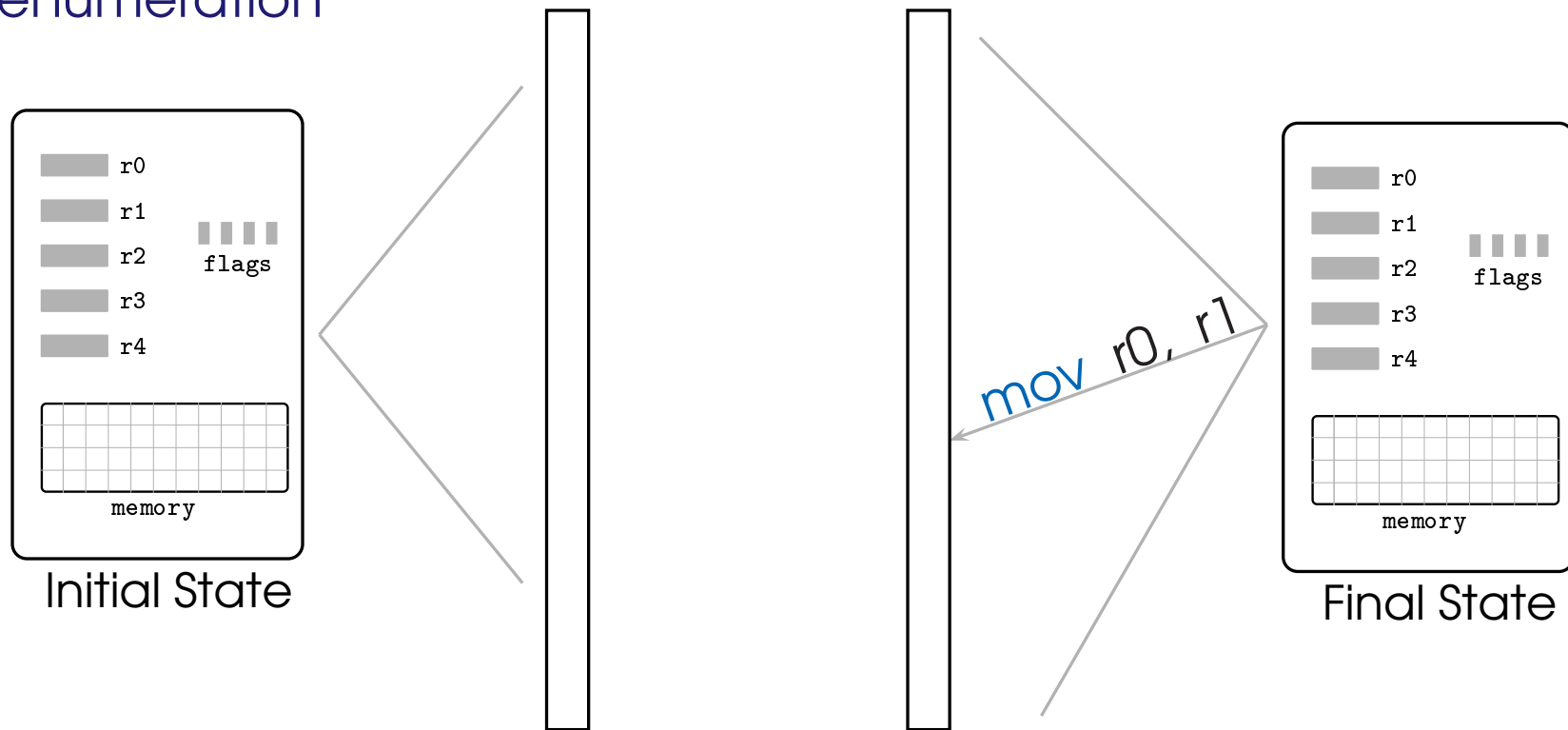
Instruction	Inverse
<code>add r0, r1</code>	<code>sub r0,r1</code> flags ← d
<code>add r0, r0</code>	<code>shr r0; r0[31] ← d</code> flags ← d
<code>mov r0, r1</code>	<code>r0 ← d</code>
<code>and r0, \$010110</code>	<code>r0 ← r0 & d1d11d</code> flags ← d
<code>or r0, \$010110</code>	<code>r0 ← r0 0d0dd0</code> flags ← d
<code>xor r0, r1</code>	<code>xor r0, r1</code> flags ← d
<code>xchg r0, r1</code>	<code>xchg r0, r1</code>
<code>shr r0, \$3</code>	<code>shl r0, \$3; r0[0..2] ← d</code> flags ← d
<code>ror r0, \$3</code>	<code>rol r0, \$3</code>
<code>inc r0</code>	<code>dec r0</code> flags ← d
<code>neg r0</code>	<code>neg r0</code> flags ← d
<code>not r0</code>	<code>not r0</code>

Some instructions need not be enumerated during inverse enumeration

Some instructions need not be enumerated during inverse enumeration

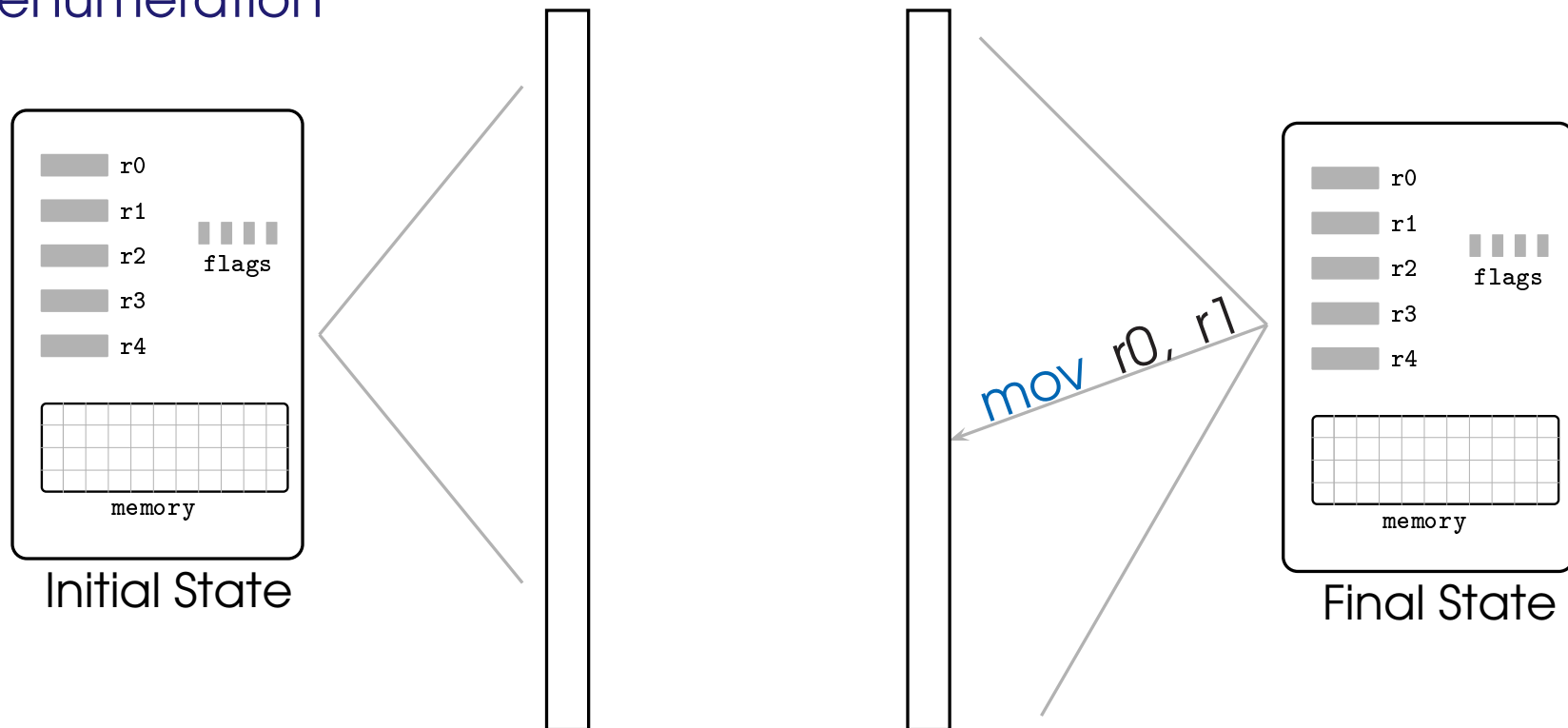


Some instructions need not be enumerated during inverse enumeration



{`mov r0, r1`} can be the last instruction of the optimal sequence only if $r0 = r1$ in the final state.

Some instructions need not be enumerated during inverse enumeration



`{mov r0, r1}` can be the last instruction of the optimal sequence only if $r0 = r1$ in the final state.

If $r0 \neq r1$, `{mov r0, r1}` need not be enumerated

The final state must be constrained w.r.t to the current instruction being enumerated

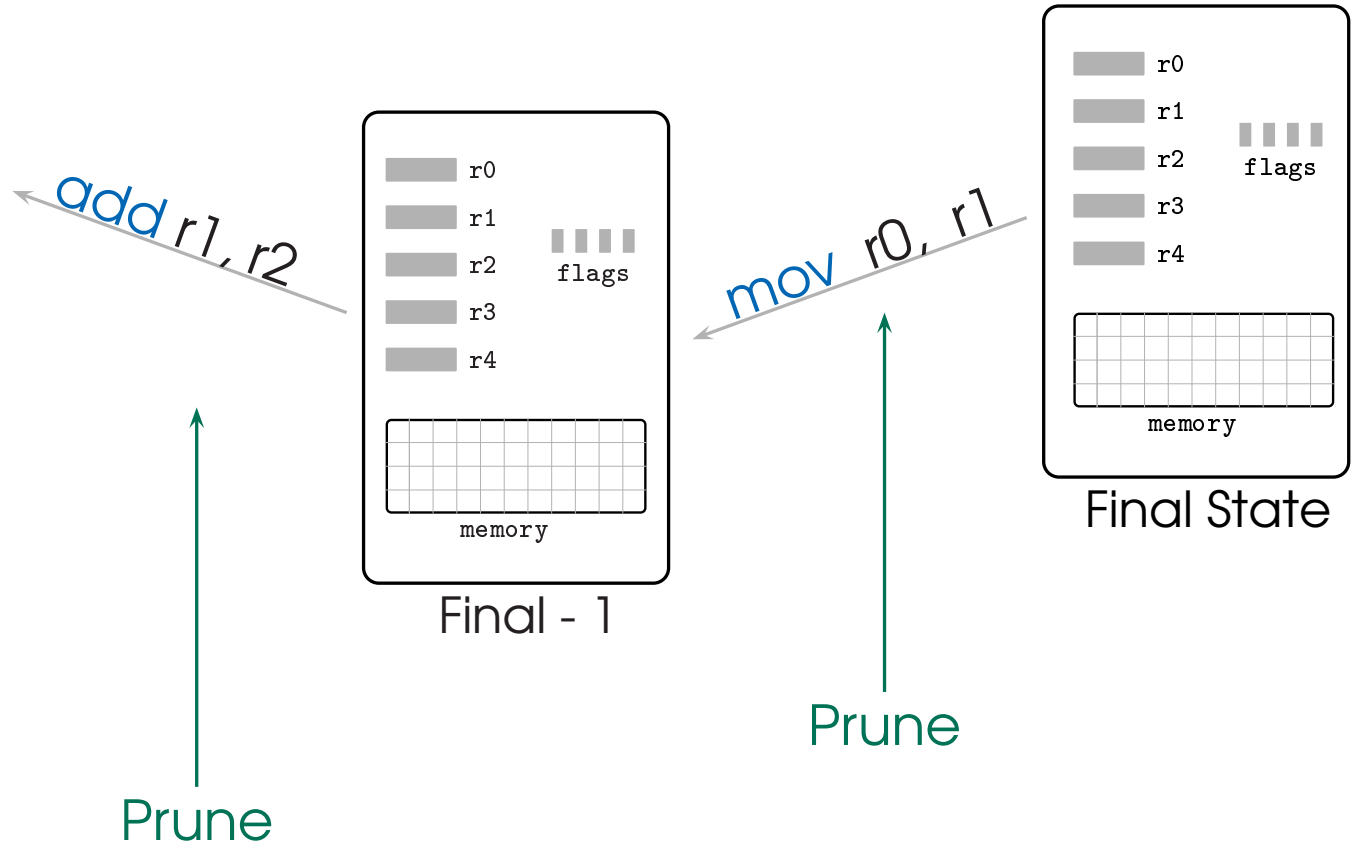
Instruction	Constraint
<code>mov r0, r1</code>	$r0 = r1$
<code>add r0, r1</code>	flags must agree with r0, r1
<code>and r0, \$010110</code>	bits 0, 3 and 5 must be 0
<code>or r0, \$010110</code>	bits 1, 2 and 4 must be 1
<code>shr r0, \$3</code>	Three MSBs must be zero
<code>xor r0, r1</code>	flags must agree with r0, r1
<code>inc r0</code>	flags must agree with r0
<code>dec r0</code>	flags must agree with r0

The final state must be constrained w.r.t to the current instruction being enumerated

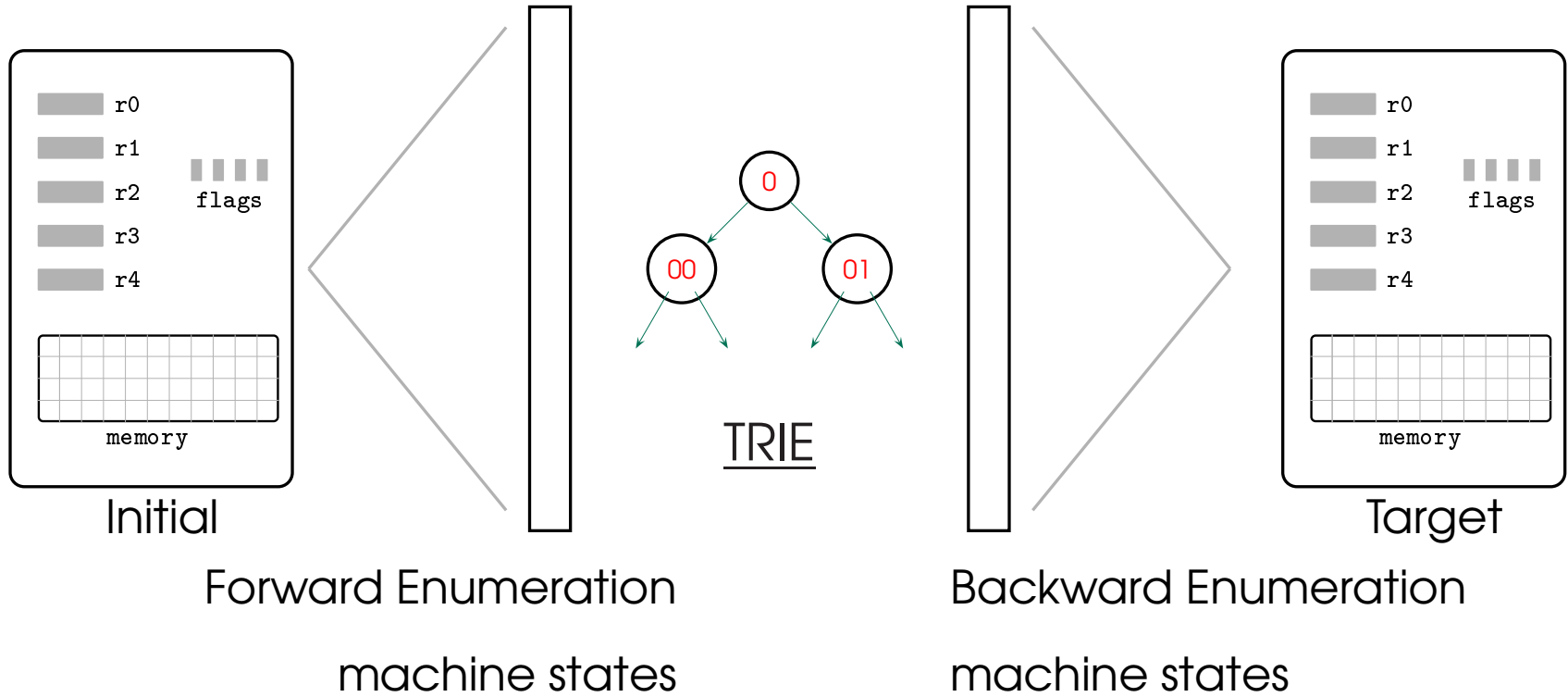
Instruction	Constraint
<code>mov r0, r1</code>	$r0 = r1$
<code>add r0, r1</code>	flags must agree with r0, r1
<code>and r0, \$010110</code>	bits 0, 3 and 5 must be 0
<code>or r0, \$010110</code>	bits 1, 2 and 4 must be 1
<code>shr r0, \$3</code>	Three MSBs must be zero
<code>xor r0, r1</code>	flags must agree with r0, r1
<code>inc r0</code>	flags must agree with r0
<code>dec r0</code>	flags must agree with r0

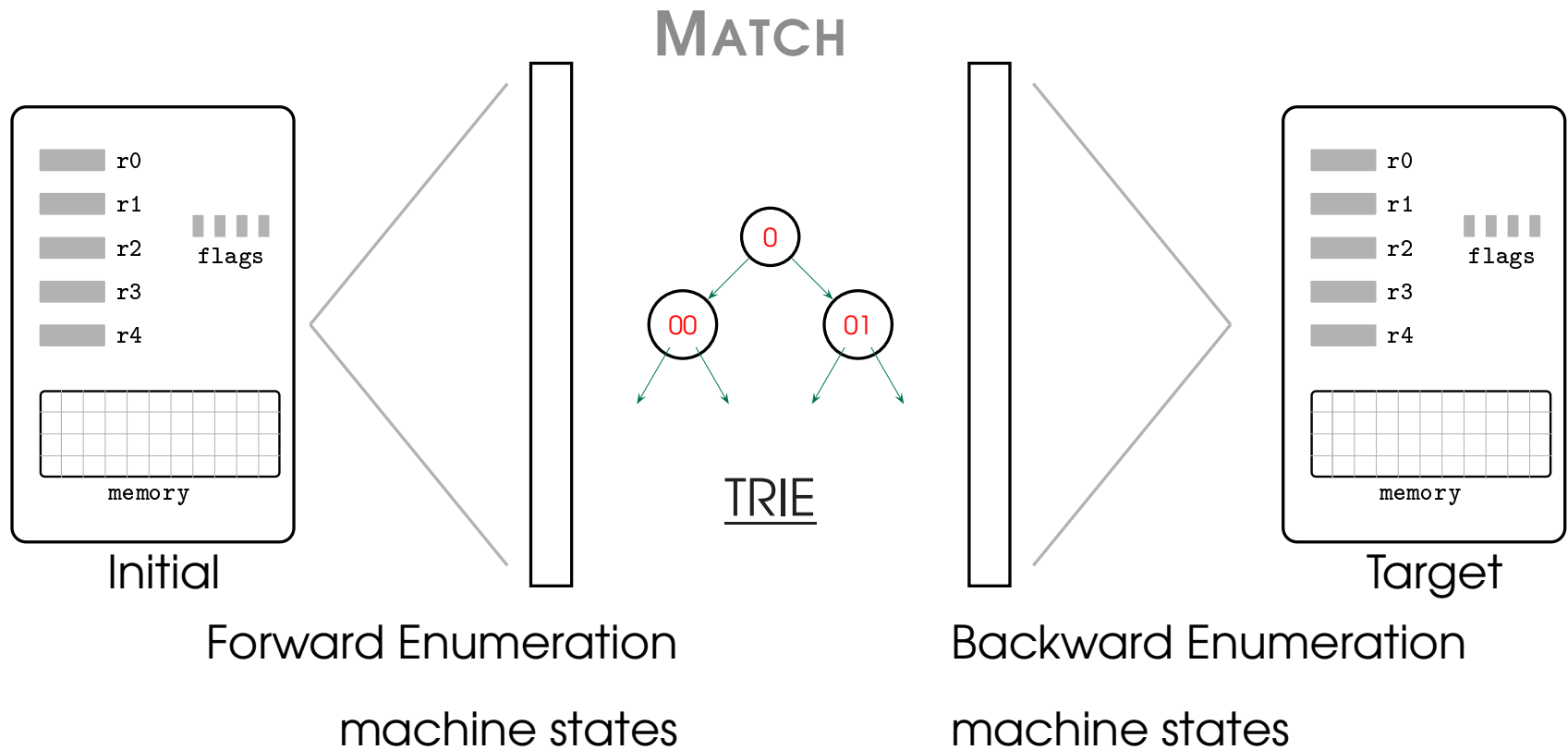
Intuitively: Goal-Directed Search can be pruned much better

Prune at every step

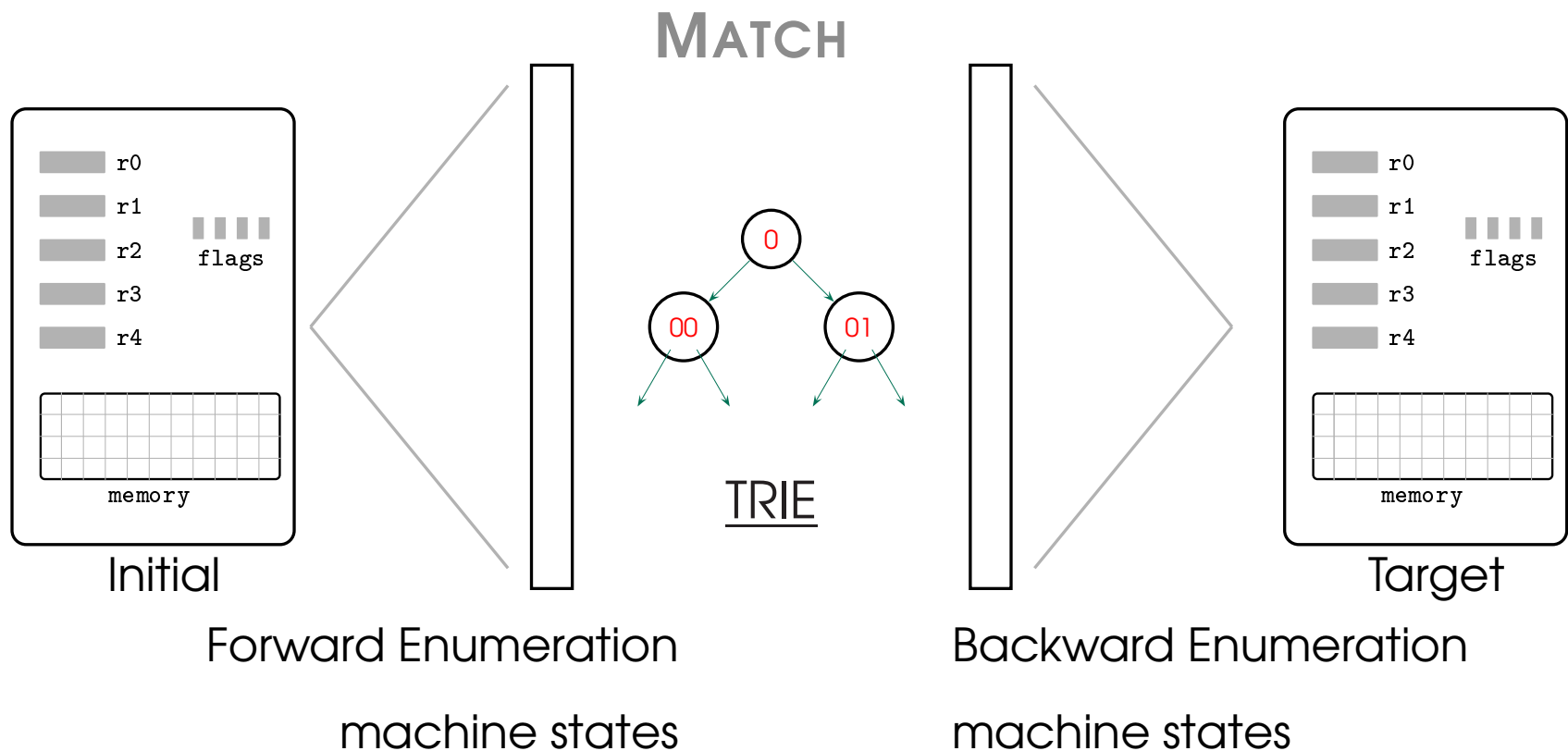


MATCH





- A Trie is constructed on the forward-enumeration machine states



- A Trie is constructed on the forward-enumeration machine states
- Each backward-enumeration machine state is searched on the constructed trie

COMPLEXITY

Assuming size of instruction set = I (approx 3000)

For a length $m + n$ instruction sequence:

- Original: $O(I^{m+n})$
- New Best Case: $O(I^n + I^m \cdot \log_2 I^n)$
- New Worst Case: $O(I^{m+n})$
- If average number of don't care bits are $f\%$: $O(f \cdot I^{m+n})$.
In practice, much less due to pruning.

PRELIMINARY RESULTS

477/5500 instructions are perfectly invertible (i.e., no don't knows)

(invertibility depends on both opcode and operands)

PRELIMINARY RESULTS

- $3 = 2 + 1$
 - Can enumerate length-3 in few minutes as opposed to 2 days

PRELIMINARY RESULTS

- $3 = 2 + 1$
 - Can enumerate length- 3 in few minutes as opposed to 2 days
- $4 = 2 + 2$
 - For some sequences, completes in few minutes
 - For some, it takes a day

PRELIMINARY RESULTS

- $3 = 2 + 1$
 - Can enumerate length-3 in few minutes as opposed to 2 days
- $4 = 2 + 2$
 - For some sequences, completes in few minutes
 - For some, it takes a day
- $4 = 3 + 1$
 - Currently Implementing. Likely to be much faster than 2+2.

PRELIMINARY RESULTS

- $3 = 2 + 1$
 - Can enumerate length-3 in few minutes as opposed to 2 days
- $4 = 2 + 2$
 - For some sequences, completes in few minutes
 - For some, it takes a day
- $4 = 3 + 1$
 - Currently Implementing. Likely to be much faster than 2+2.
- $5 = 3 + 2$
 - Perhaps, possible

CONCLUSIONS

- We have demonstrated the construction of an optimizer using only exhaustive search
- Many (sometimes huge) performance opportunities are still unexploited by compilers
- Scaling to longer sequence lengths is the key
- <http://cs.stanford.edu/~sbansal/superoptimizer>