Automatic Generation of Peephole Superoptimizers

Sorav Bansal and Alex Aiken
Stanford University

{sbansal, aiken}@cs.stanford.edu

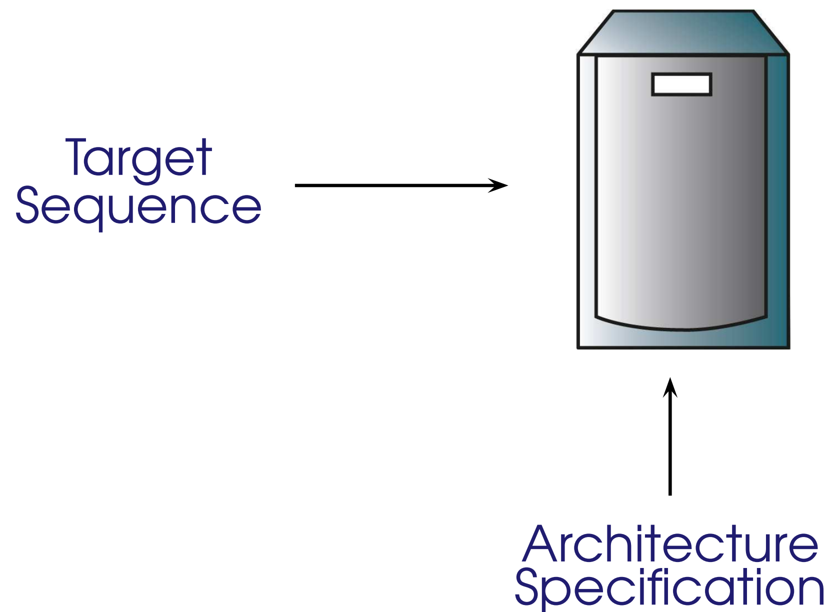
Compiler Folks

Architecture Folks

SUPEROPTIMIZATION

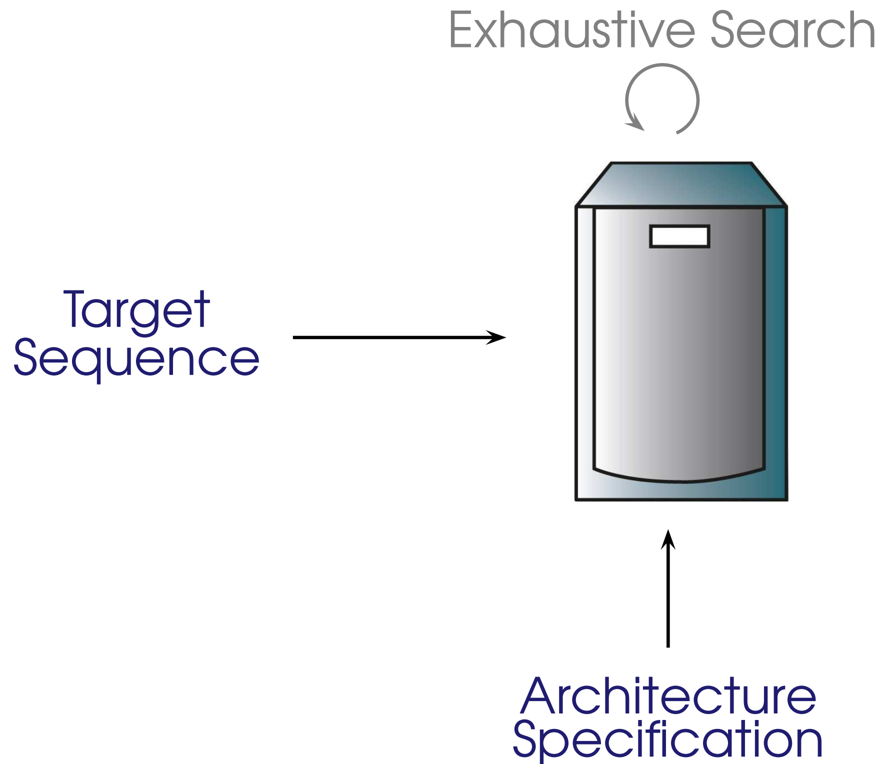
SUPEROPTIMIZATION

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



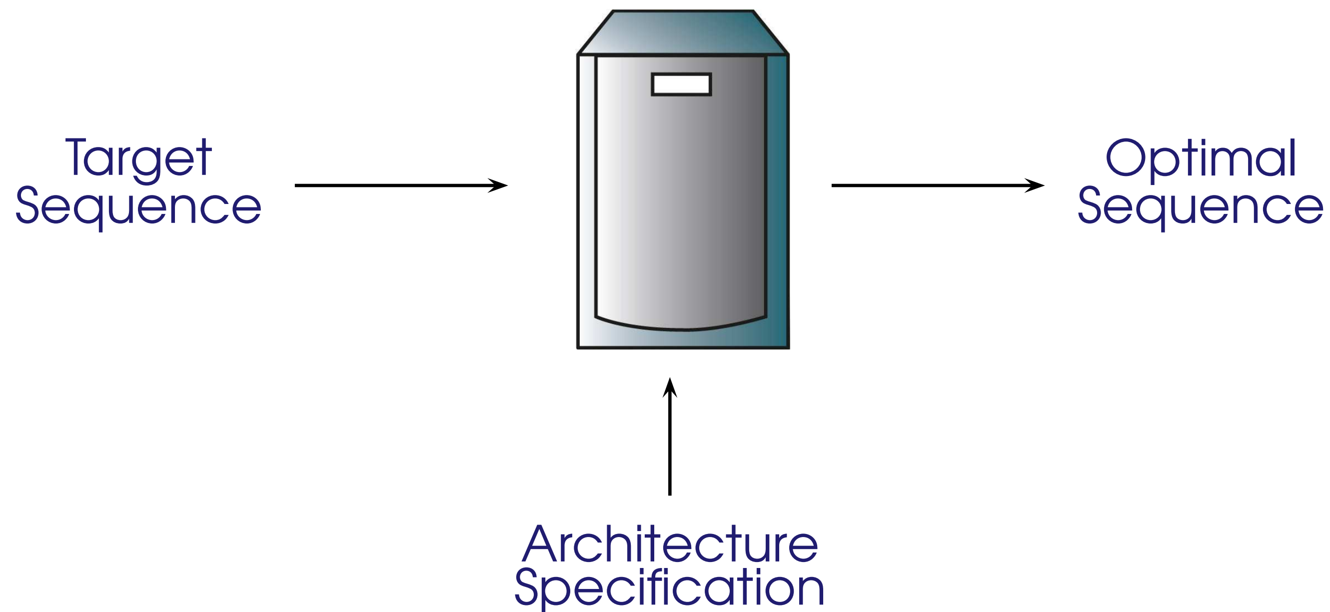
SUPEROPTIMIZATION

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

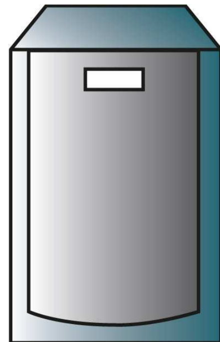
- ➜ Superoptimization is Sloowww..
- ➜ Superoptimization is useful only for occasional optimization of the critical inner loop

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

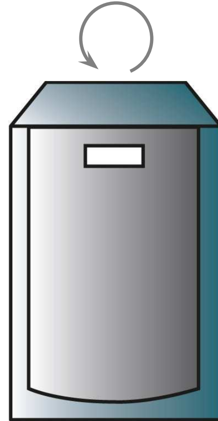


SUPEROPTIMIZATION

```
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

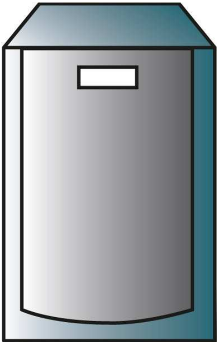


SUPEROPTIMIZATION

```
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

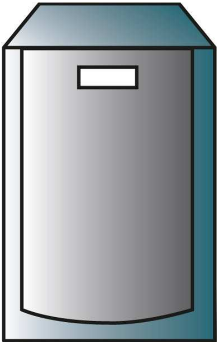
	target			optimal		
1.	movl	%ecx	(%ebx)	leal	9(%ecx)	%ebx
	shll	(%ebx)		movl	%ebx	(%ebx)
	addl	$9	(%ebx)			

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

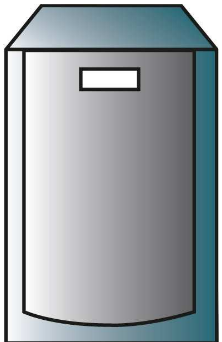
	target			optimal		
1.	movl	%ecx	(%ebx)	leal	9(%ecx)	%ebx
	shll	(%ebx)		movl	%ebx	(%ebx)
	addl	$9	(%ebx)			
2.	shll	%eax		movl	%ecx	%eax
	leal	(%ecx)	(%eax)			

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

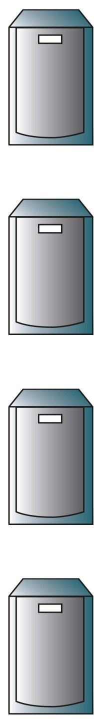
	target	optimal
1.	movl %ecx (%ebx) shll (%ebx) addl $9 (%ebx)	leal 9(%ecx) %ebx movl %ebx (%ebx)
2.	shll %eax leal (%ecx) (%eax)	movl %ecx %eax
3.	movl %ecx %eax shll %ecx movl %ecx %eax	shll %ecx movl %ecx %eax

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
```

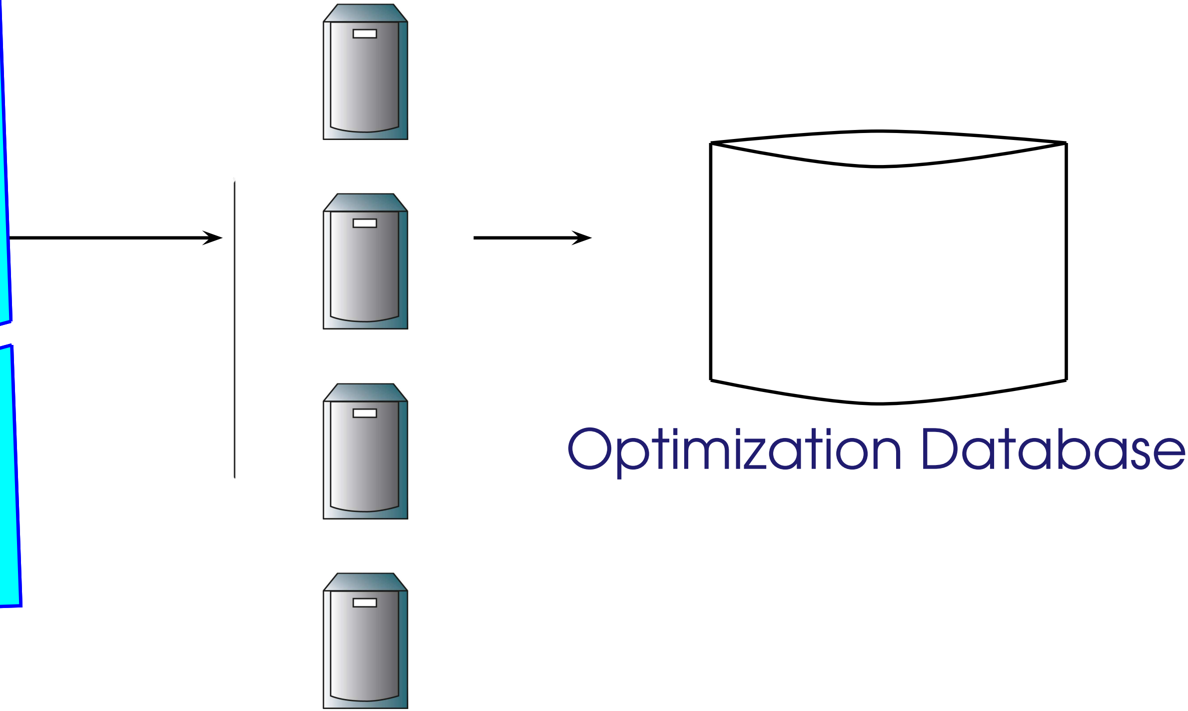
b.out



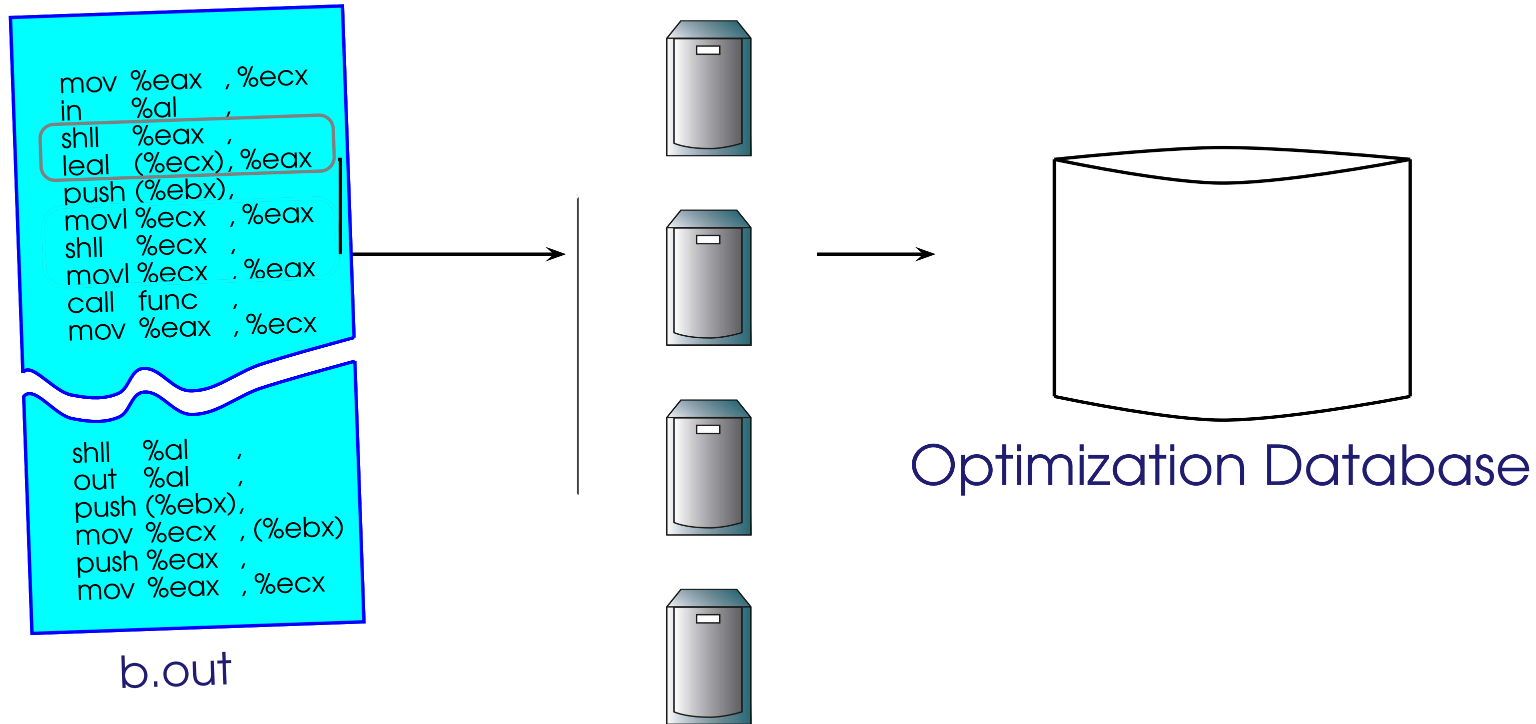
Optimization Table

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

SUPEROPTIMIZATION

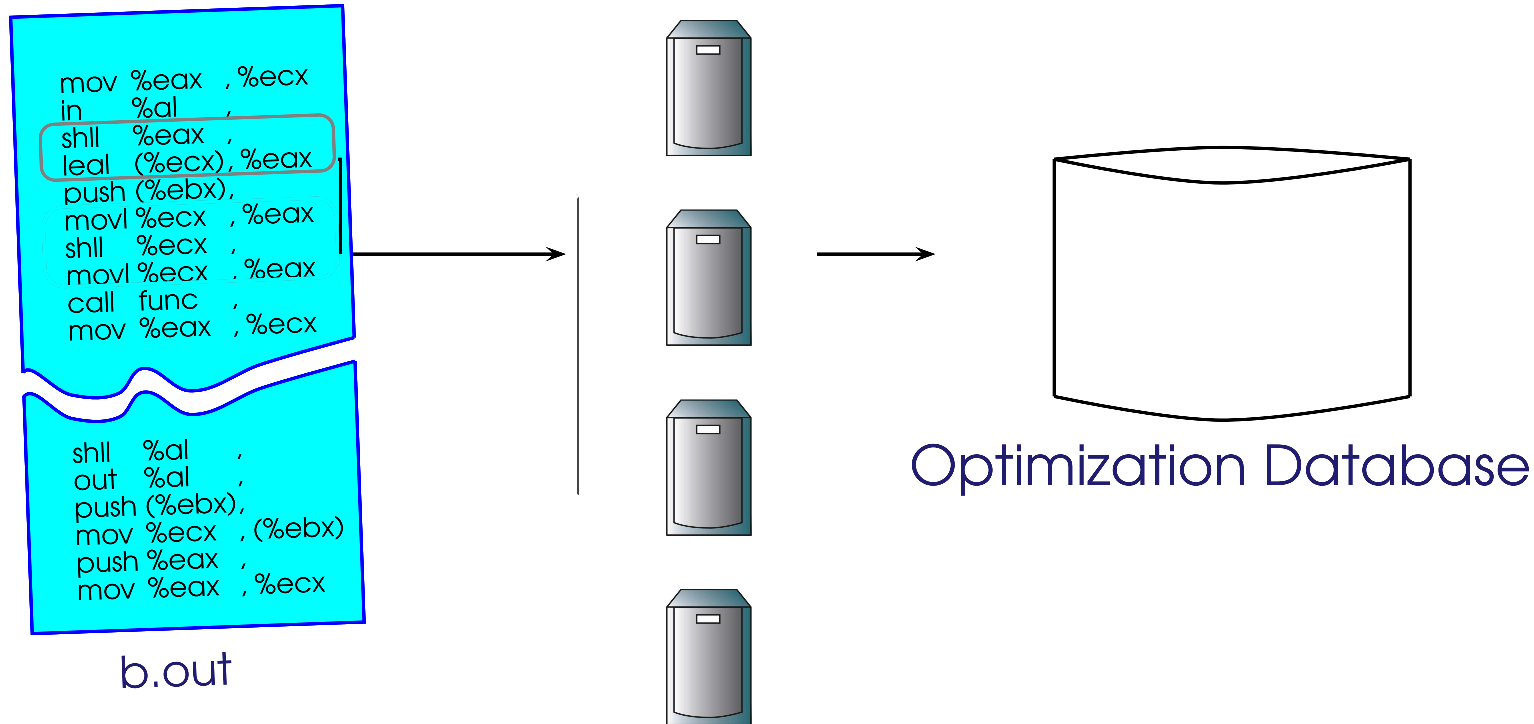


SUPEROPTIMIZATION



- To what length of instruction sequences can we scale?

Superoptimization



- To what length of instruction sequences can we scale?
- How large can we make the optimization database?

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:
```

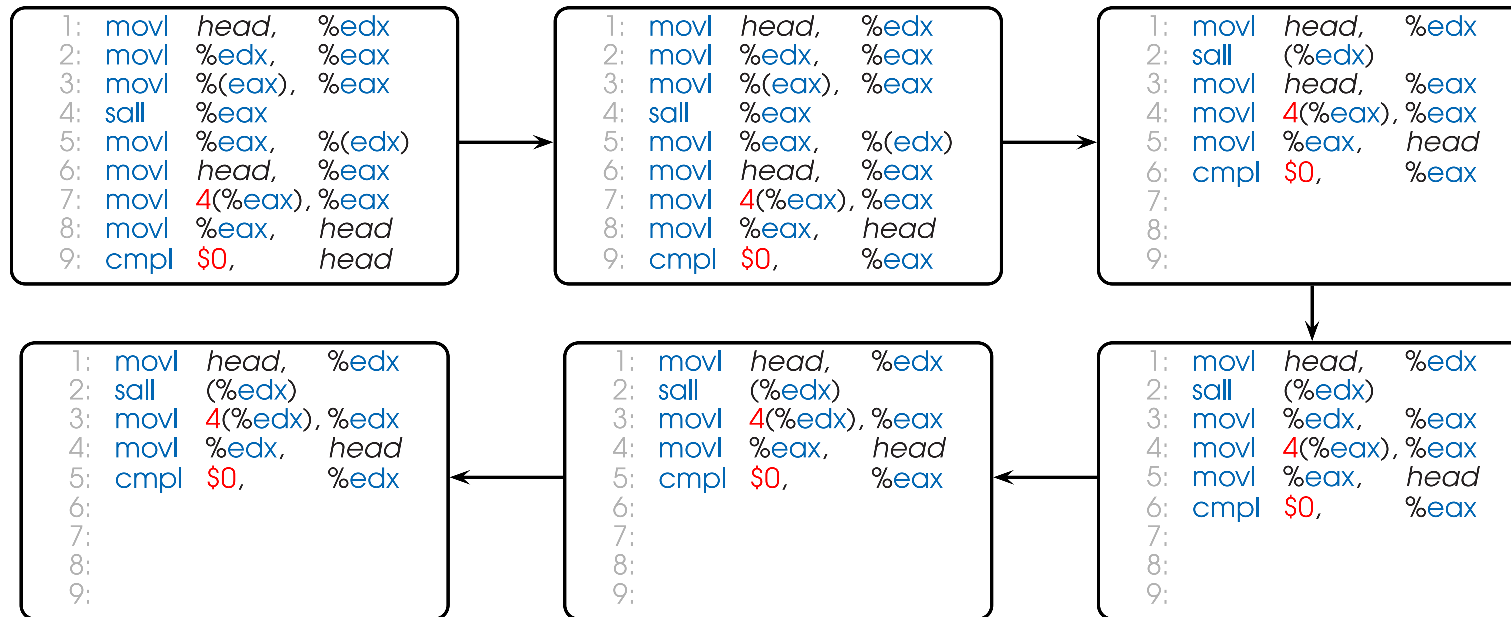
Our optimizer

```
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

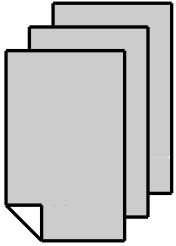


An automatically generated “peephole” optimizer can capture many traditional optimizations

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

System Architecture

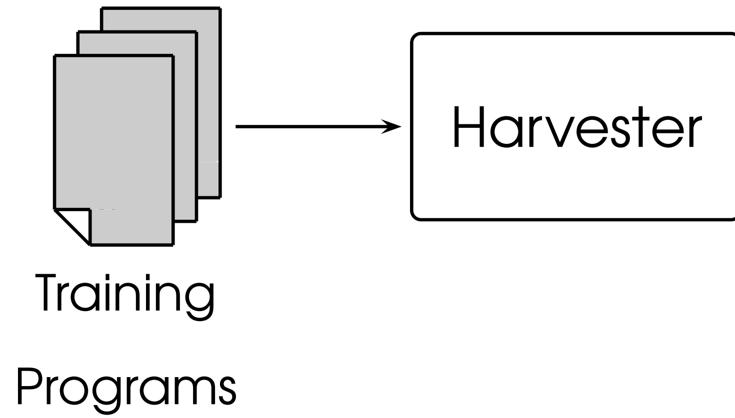
System Architecture



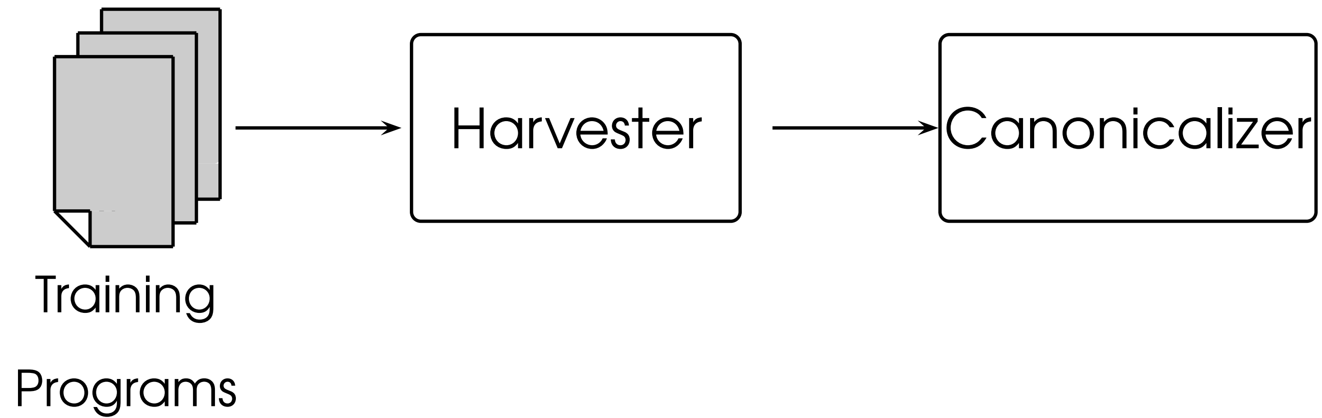
Training

Programs

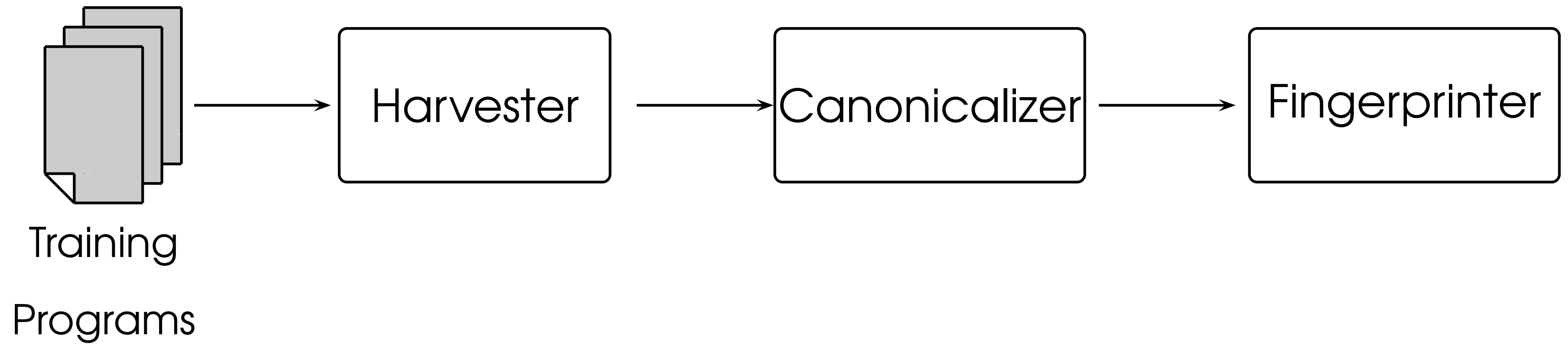
System Architecture



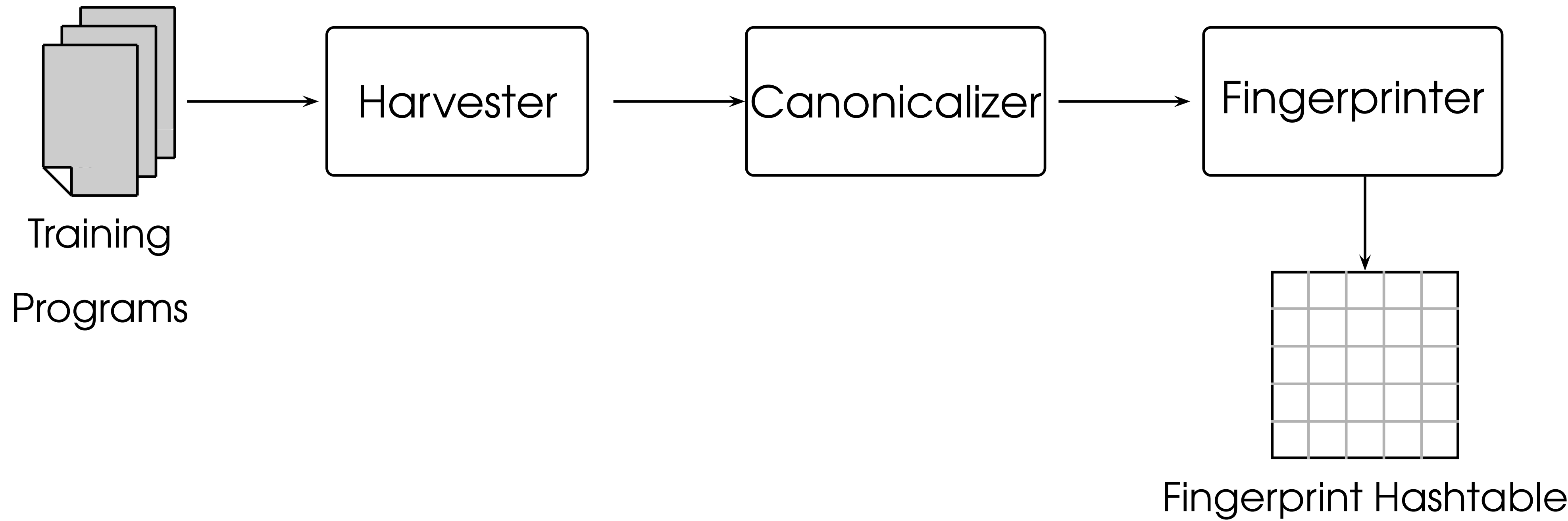
System Architecture



System Architecture

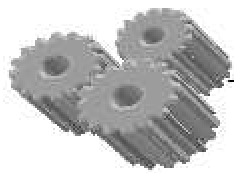


System Architecture

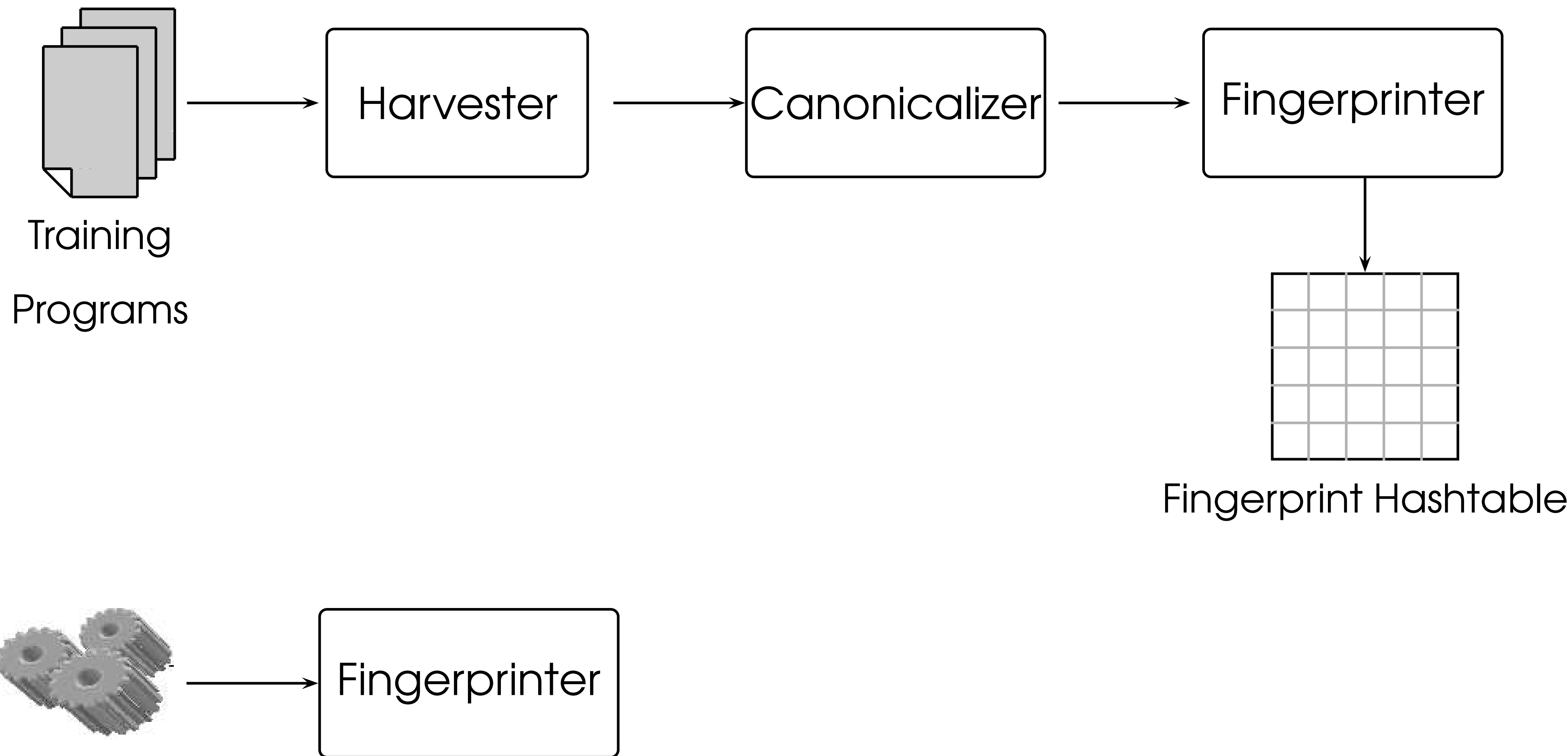


SYSTEM ARCHITECTURE

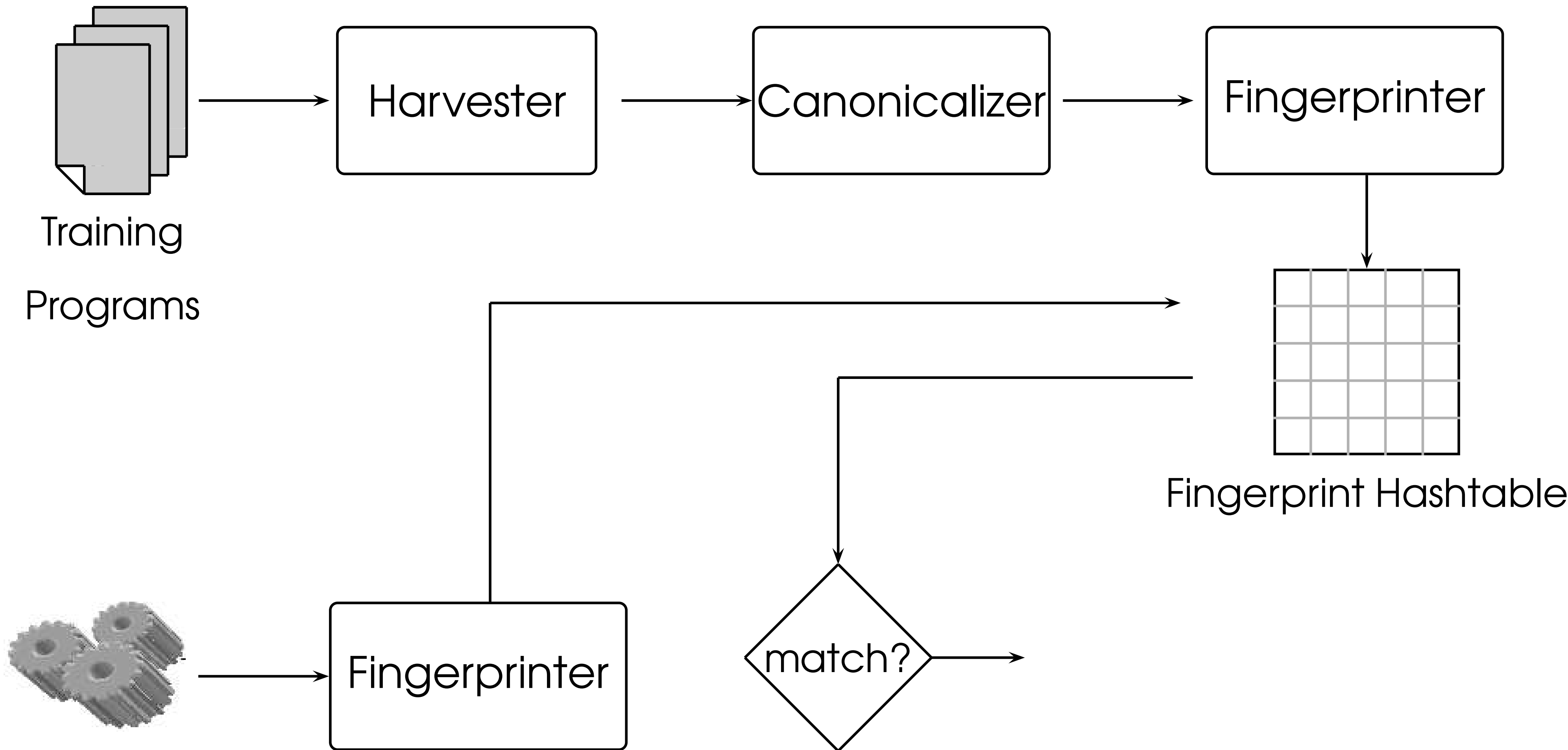
Training Programs → Harvester → Canonicalizer → Fingerprinter → Fingerprint Hashtable



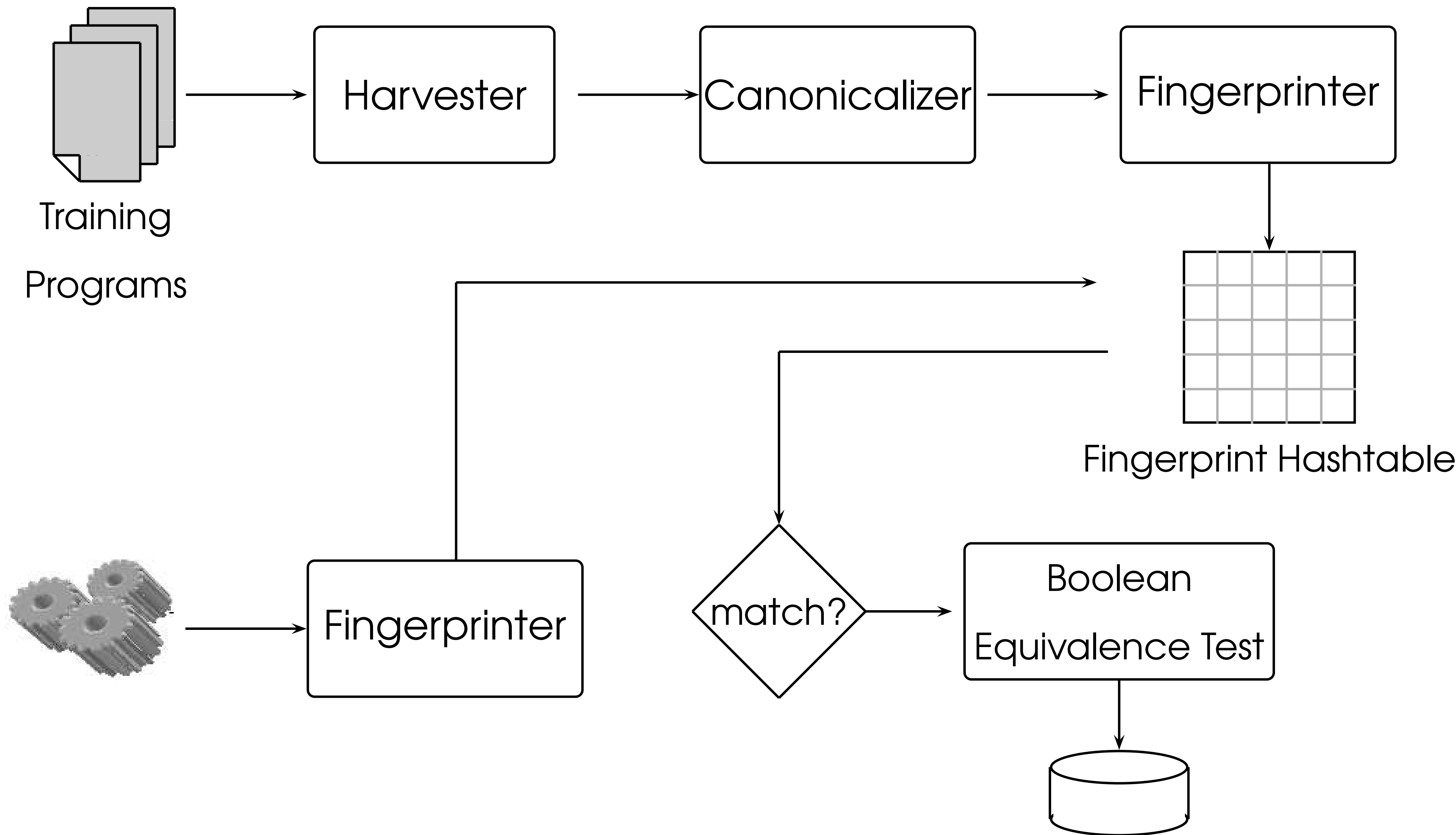
System Architecture



System Architecture



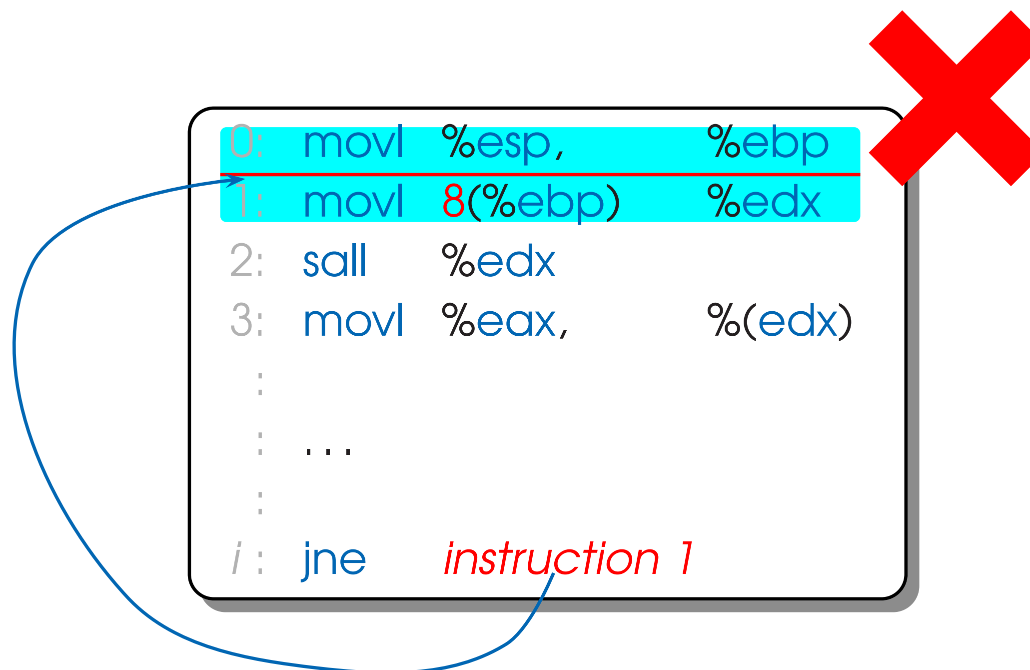
System Architecture



Harvester

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

- no middle instruction is a jump target



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
```

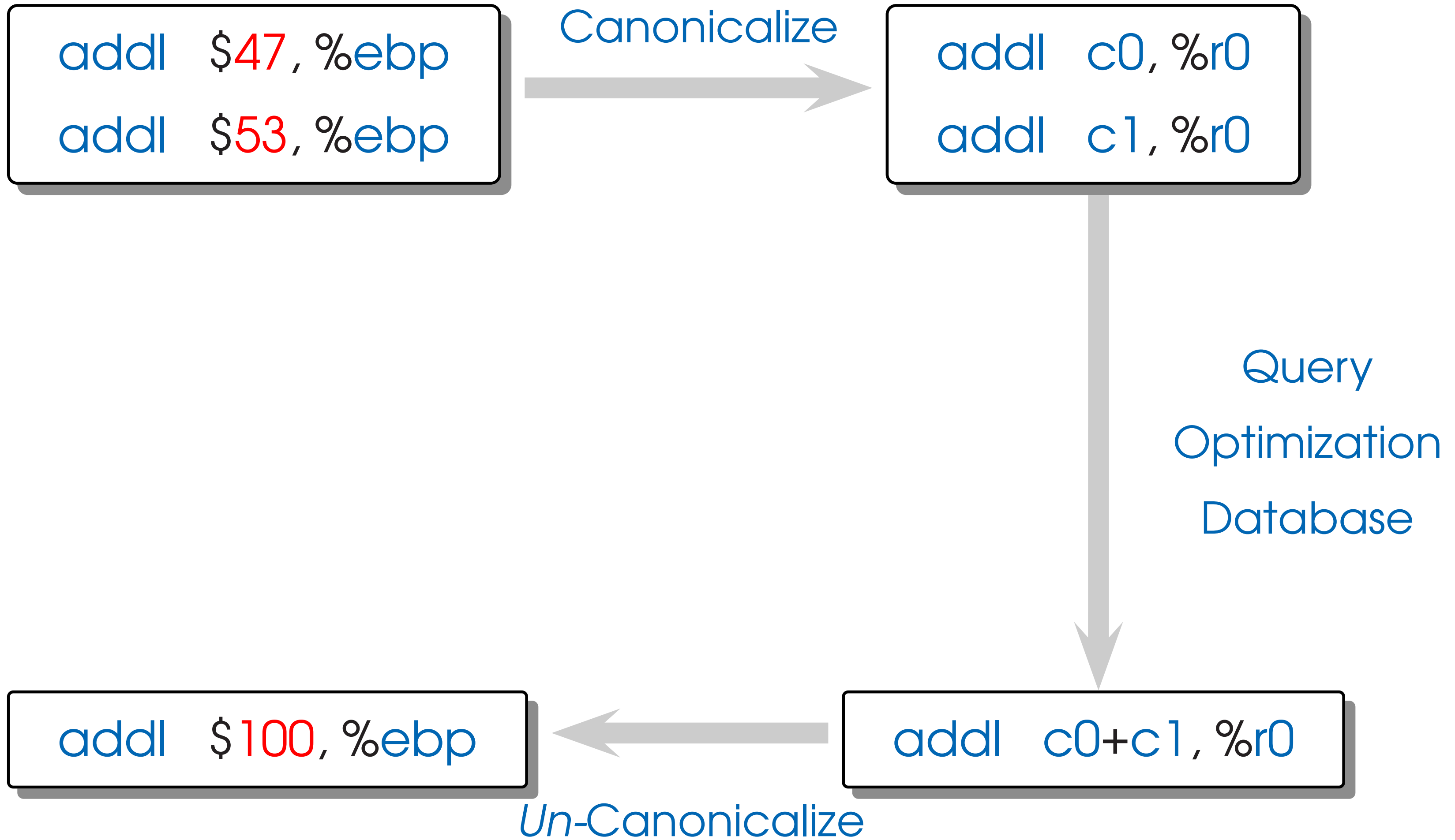
Use symbolic constants to deal with immediates

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

→

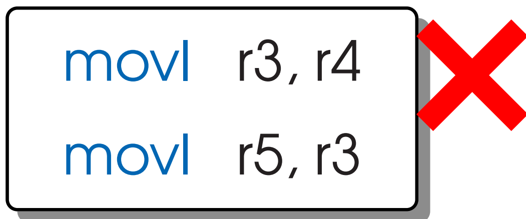
```
addl   cons0+cons1, %ebp
```

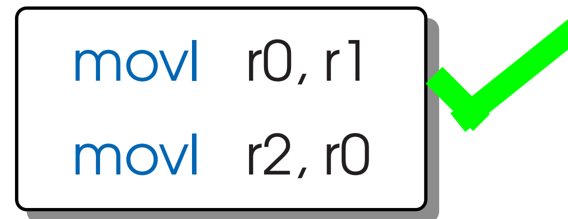
Canonicalizer



Canonicalizer

Only instruction sequences in canonical form are enumerated





Special Constants are enumerated for immediate operands

- ➜ `0` ; `1`
- ➜ `c0` ; `c1`
- ➜ `c0 + 1` ; `c0 - 1`
- ➜ `c0 + c1` ; `c0 - c1`

Fingerprinter

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*

Approximate memory by a small array (256 bytes)

➜ 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*

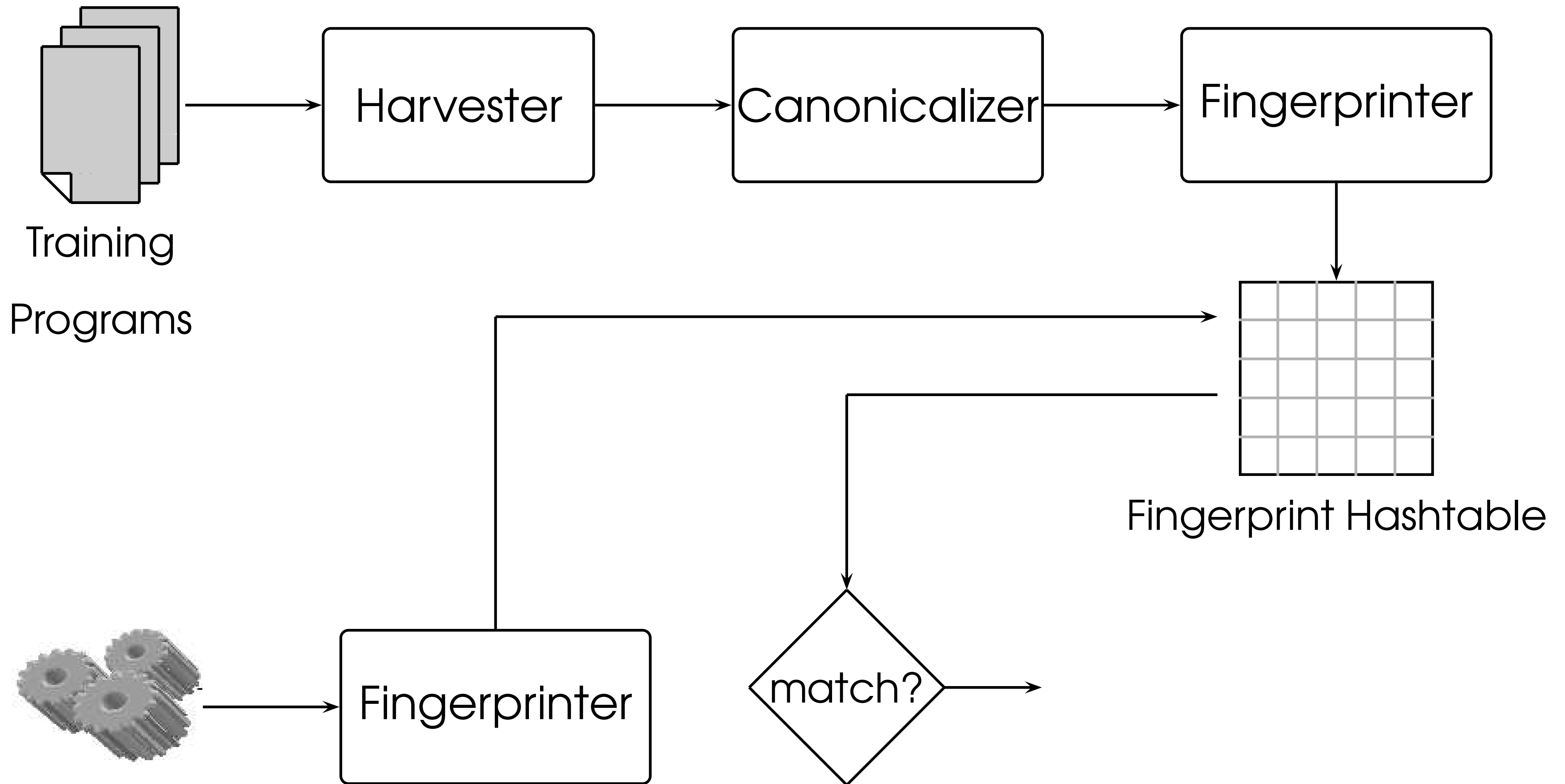
Approximate memory by a small array (256 bytes)

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

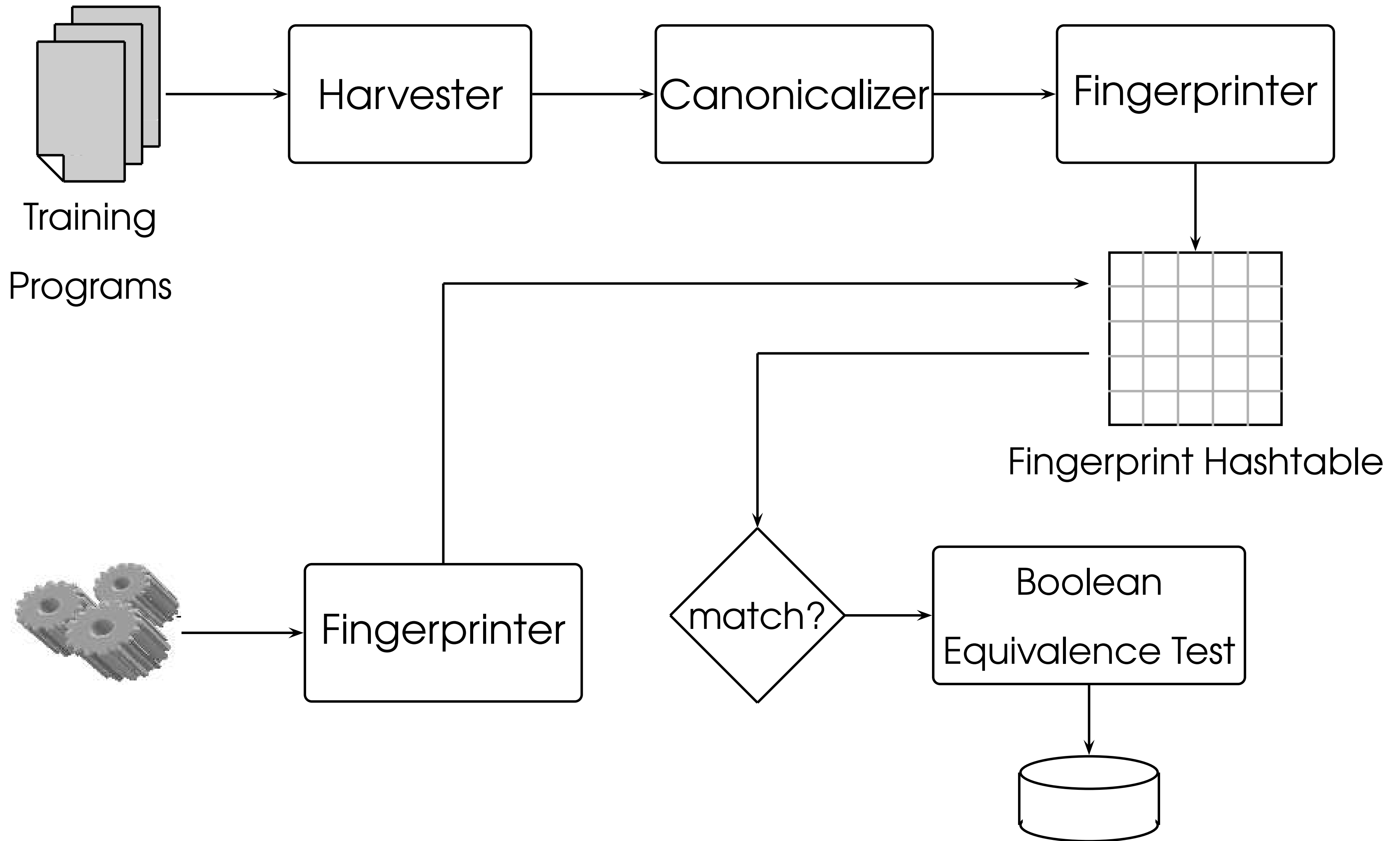
Compute fingerprint by executing directly on hardware

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

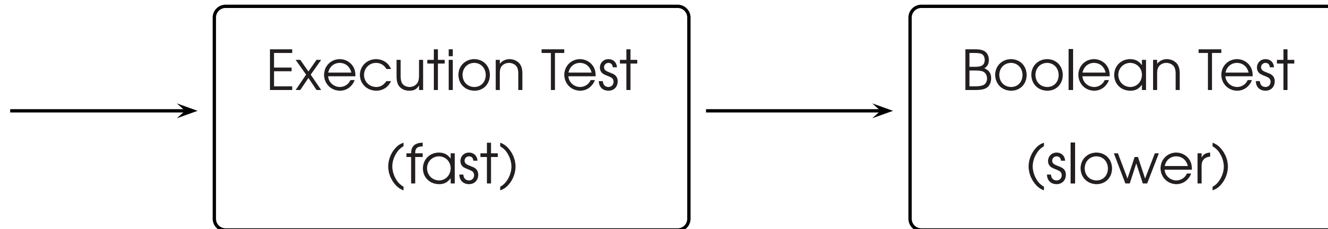
System Architecture



System Architecture



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 $\longrightarrow$ *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 $\longrightarrow$ *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] $\oplus$ b[i]	2x
Sprite Copy	c[i] = (a[i] $== 0$)? b[i] : a[i]	2x

MMX and conditional-move instructions are not well-exploited

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%

Time taken to optimize: <3 secs

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%

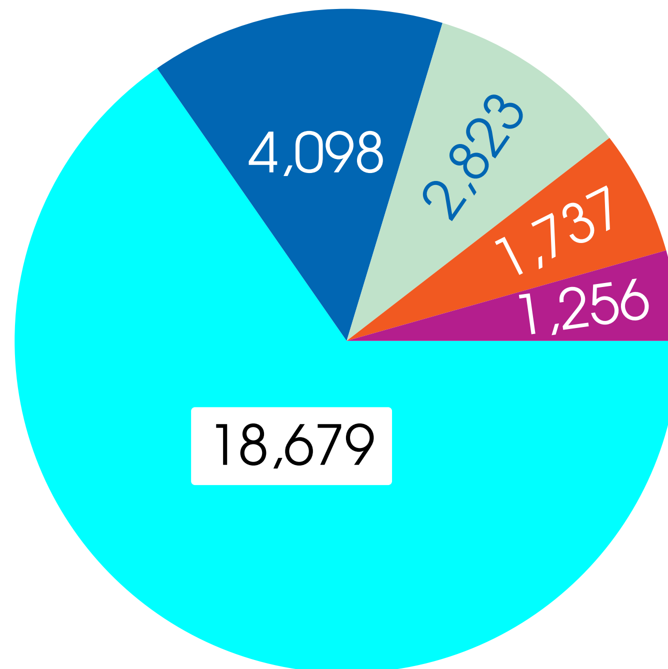
Time taken to optimize: <3 secs

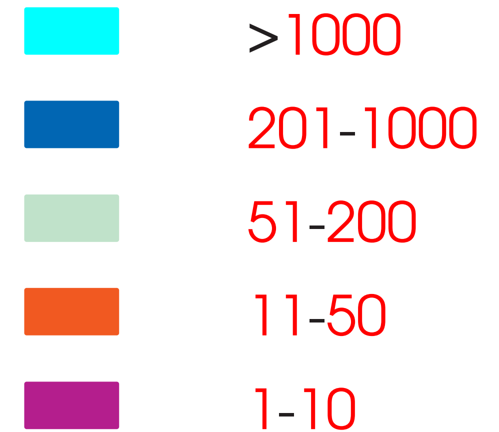
Experimental Results

Number of Distinct Optimization Rules Learnt

Codesize	3000
Runtime	2100

Re-use of Optimization Rules





Total: 28,593 optimizations

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