The optimizer focuses on:

- reducing the execution time; or
- reducing the code size; or
- reducing the power consumption (new).

These goals often conflict, since a larger program may in fact be faster.

The best optimizations achieve both goals.

Optimizations for space:

- were historically very important, because memory was small and expensive;
- when memory became large and cheap, optimizing compilers traded space for speed; but
- then Internet bandwidth is small and expensive, so Java compilers optimize for space,
- today Internet bandwidth is larger and cheaper, so we optimize for speed again.

⇒ Optimizations driven by economy!
Optimizations may take place:
- at the source code level;
- in an intermediate representation;
- at the binary machine code level; or
- at run-time (e.g. JIT compilers).

An aggressive optimization requires many small contributions from all levels.

Should you program in “Optimized C”?

If you want a fast C program, should you use LOOP #1 or LOOP #2?

```c
/* LOOP #1 */
for (i = 0; i < N; i++) {
    a[i] = a[i] * 2000;
    a[i] = a[i] / 10000;
}

/* LOOP #2 */
b = a;
for (i = 0; i < N; i++) {
    *b = *b * 2000;
    *b = *b / 10000;
    b++;
}
```

What would the expert programmer do?

If you said LOOP #2 . . . you were wrong!

<table>
<thead>
<tr>
<th>LOOP</th>
<th>opt. level</th>
<th>SPARC</th>
<th>MIPS</th>
<th>Alpha</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1 (array)</td>
<td>no opt</td>
<td>20.5</td>
<td>21.6</td>
<td>7.85</td>
</tr>
<tr>
<td>#1 (array)</td>
<td>opt</td>
<td>8.8</td>
<td>12.3</td>
<td>3.26</td>
</tr>
<tr>
<td>#1 (array)</td>
<td>super</td>
<td>7.9</td>
<td>11.2</td>
<td>2.96</td>
</tr>
<tr>
<td>#2 (ptr)</td>
<td>no opt</td>
<td>19.5</td>
<td>17.6</td>
<td>7.55</td>
</tr>
<tr>
<td>#2 (ptr)</td>
<td>opt</td>
<td>12.4</td>
<td>15.4</td>
<td>4.09</td>
</tr>
<tr>
<td>#2 (ptr)</td>
<td>super</td>
<td>10.7</td>
<td>12.9</td>
<td>3.94</td>
</tr>
</tbody>
</table>

- Pointers confuse most C compilers; don’t use pointers instead of array references.
- Compilers do a good job of register allocation; don’t try to allocate registers in your C program.
- In general, write clear C code; it is easier for both the programmer and the compiler to understand.

Optimization in JOOS:

```c
    c = a*b+c;
    if (c<a) a=a+b*113;
    while (b>0) {
        a=a*c;
        b=b-1;
    }
```
Smaller and faster code:
- remove unnecessary operations;
- simplify control structures; and
- replace complex operations by simpler ones (strength reduction).

This is what the JOOS optimizer does.

Later, we shall look at:
- JIT compilers; and
- more powerful optimizations based on static analysis.

Larger, but faster code: tabulation.

The sine function may be computed as:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

or looked up in a table:

<table>
<thead>
<tr>
<th>$x$</th>
<th>$\sin(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.0000</td>
</tr>
<tr>
<td>0.10</td>
<td>0.0998</td>
</tr>
<tr>
<td>0.20</td>
<td>0.1987</td>
</tr>
<tr>
<td>0.30</td>
<td>0.2955</td>
</tr>
<tr>
<td>0.40</td>
<td>0.3894</td>
</tr>
<tr>
<td>0.50</td>
<td>0.4794</td>
</tr>
<tr>
<td>0.60</td>
<td>0.5647</td>
</tr>
<tr>
<td>0.70</td>
<td>0.6442</td>
</tr>
</tbody>
</table>

Larger, but faster code: loop unrolling.

The loop:

```java
for (i=0; i<2*N; i++) {
    a[i] = a[i] + b[i];
}
```

is changed into:

```java
for (i=0; i<2*N; i+=2) {
    j = i+1;
    a[i] = a[i] + b[i];
    a[j] = a[j] + b[j];
}
```

which reduces the overhead and may give a 10–20% speedup.
The optimizer must undo fancy language abstractions:

- variables abstract away from registers, so the optimizer must find an efficient mapping;
- control structures abstract away from gotos, so the optimizer must construct and simplify a goto graph;
- data structures abstract away from memory, so the optimizer must find an efficient layout;
- method lookups abstract away from procedure calls, so the optimizer must efficiently determine the intended implementations.

Continuing: the OO language BETA unifies as patterns the concepts:

- abstract class;
- concrete class;
- method; and
- function.

A (hypothetical) optimizing BETA compiler must attempt to classify the patterns to recover that information.

Example: all patterns are allocated on the heap, but 50% of the patterns are methods that could be allocated on the stack.

Difficult compromises:

- a high abstraction level makes the development time cheaper, but the run-time more expensive; however
- high-level abstractions are also easier to analyze, which gives optimization potential.

Also:

- an optimizing compiler makes run-time more efficient, but compile-time less efficient;
- optimizations for speed and size may conflict; and
- different applications may require different optimizations.

The JOOS peephole optimizer:

- works at the bytecode level;
- looks only at peepholes, which are sliding windows on the code sequence;
- uses patterns to identify and replace inefficient constructions;
- continues until a global fixed point is reached; and
- optimizes both speed and space.
The optimizer considers the goto graph:

```c
while (a>0) {
    if (b==c) a=a-1; else c=c+1;
}
```

To capture the goto graph, the labels for a given code sequence are represented as an array of:

```c
typedef struct LABEL {
    char *name;
    int sources;
    struct CODE *position;
} LABEL;
```

where:
- the array index is the label’s number;
- the field `name` is the textual part of the label;
- the field `sources` indicates the in-degree of the label; and
- the field `position` points to the location of the label in the code sequence.

Operations on the goto graph:
- inspect a given bytecode;
- find the next bytecode in the sequence;
- find the destination of a label;
- create a new reference to a label;
- drop a reference to a label;
- ask if a label is dead (in-degree 0);
- ask if a label is unique (in-degree 1); and
- replace a sequence of bytecodes by another.

Inspect a given bytecode:

```c
int is_istore(CODE *c, int *arg)
{
    if (c==NULL) return 0;
    if (c->kind == istoreCK) {
        (*arg) = c->val.istoreC;
        return 1;
    } else {
        return 0;
    }
}
```

Find the next bytecode in the sequence:

```c
CODE *next(CODE *c)
{
    if (c==NULL) return NULL;
    return c->next;
}
```

Find the destination of a label:

```c
CODE *destination(int label)
{
    return currentlabels[label].position;
}
```

Create a new reference to a label:

```c
int copylabel(int label)
{
    currentlabels[label].sources++;
    return label;
}
```
Drop a reference to a label:

```c
void droplabel(int label)
{ currentlabels[label].sources--; }
```

Ask if a label is dead (in-degree 0):

```c
int deadlabel(int label)
{ return currentlabels[label].sources==0; }
```

Ask if a label is unique (in-degree 1):

```c
int uniquelabel(int label)
{ return currentlabels[label].sources==1; }
```

Replace a sequence of bytecodes by another:

```c
int replace(CODE **c, int k, CODE *r)
{ CODE *p;
  int i;
  p = *c;
  for (i=0; i<k; i++) p=p->next;
  if (r==NULL) {
    *c = p;
  } else {
    *c = r;
    while (r->next!=NULL) r=r->next;
    r->next = p;
  }
  return 1;
}
```

The expression:

```plaintext
x = x + k
```

may be simplified to an increment operation, if

```plaintext
0 ≤ k ≤ 127.
```

Corresponding JOOS peephole pattern:

```c
int positive_increment(CODE **c)
{ int x,y,k;
  if (is_iload(*c,&x) &&
      is_ldc_int(next(*c),&k) &&
      is_iadd(next(next(*c))) &&
      is_istore(next(next(next(*c))),&y) &&
      x==y && 0<k && k<127) {
    return replace(c,4,makeCODEiinc(x,k,NULL));
  }
  return 0;
}
```

We may attempt to apply this pattern anywhere in the code sequence.

The algebraic rules:

```plaintext
x * 0 = 0
x * 1 = x
x * 2 = x + x
```

may be used to simplify some operations.

Corresponding JOOS peephole pattern:

```c
int simplify_multiplication_right(CODE **c)
{ int x,k;
  if (is_iload(*c,&x) &&
      is_ldc_int(next(*c),&k) &&
      is_imul(next(next(*c)))) {
    if (k==0)
      return replace(c,3,makeCODEldc_int(0,NULL));
    else if (k==1)
      return replace(c,3,makeCODEiload(x,NULL));
    else if (k==2)
      return replace(c,3,makeCODEiload(x,
        makeCODEdup(
        makeCODEiadd(NULL))));
    return 0;
  }
  return 0;
}
```

A part of the goto graph:

```
goto L1
  L1: goto L2
  L2: goto L1
```

may be simplified by short-circuiting the jump to L1.

Corresponding JOOS peephole pattern:

```c
int simplify_goto_goto(CODE **c)
{ int l1,l2;
  if (is_goto(*c,&l1) &&
      is_goto(next(destination(l1)),&l2) && l1>l2) {
    droplabel(l1);
    copylabel(l2);
    return replace(c,1,makeCODEgoto(l2,NULL));
  }
  return 0;
}
```
The JOOS peephole pattern:

```c
int simplify_astore(CODE **c)
{
    int x;
    if (is_dup(*c) &&
        is astore(next(*c), &x) &&
        is pop(next(next(*c)))) {
        return replace(c, 3, makeCODEastore(x, NULL));
    }
    return 0;
}
```

is clearly sound, but will it ever be useful?

Yes, the assignment statement:

```c
da = b;
```

generates the code:

```c
aload_2
dup
astore_1
pop
return
```
because of our simpleminded strategy.

Caching assignments:

```c
void codeEXP(EXP *e) {
    case assignK:
        codeEXP(e->val.assignE.right);
        code_dup();
        ...
    case formalSym:
        if (e->val.assignE.leftsym->val.formalS->type->kind==refK) {
            code astore(e->val.assignE.leftsym->val.formalS->offset);
        } else {
            code istore(e->val.assignE.leftsym->val.formalS->offset);
        }
        break;
    }
```

to decide whether to pop or not. A peephole pattern is simpler and more modular.

Any collection of peephole patterns:

```c
typedef int(*OPTI)(CODE **);
#define MAX_PATTERNS 100

int init_patterns() {
    ADD_PATTERN(simplify_multiplication_right);
    ADD_PATTERN(simplify astore);
    ADD_PATTERN(positive_increment);
    ADD_PATTERN(simplify goto goto);
    return 1;
}
```
can be applied to a goto graph in a fixed point process:

```c
repeat
    for each bytecode in succession do
        for each peephole pattern in succession do
            repeat
                apply the peephole pattern
                to the bytecode
                until the goto graph didn’t change
            end
        end
    until the goto graph didn’t change
```

JOOS code for the fixed point driver:

```c
int optiCHANGE;

void optiCODEtraverse(CODE **c)
{
    int i, change;
    change = 1;
    if (*c!=NULL) {
        while (change) {
            change = 0;
            for (i=0; i<OPTS; i++) {
                change = change | optimization[i](c);
            }
            optiCHANGE = optiCHANGE || change;
        }
        if (*c!=NULL) optiCODEtraverse(&((*c)->next));
    }
}
```

```c
void optiCODE(CODE **c) {
    optiCHANGE = 1;
    while (optiCHANGE) {
        optiCHANGE = 0;
        optiCODEtraverse(c);
    }
}
Why does this process terminate?
Each peephole pattern does not necessarily make the code smaller.
To demonstrate termination for our examples, we use the lexicographically ordered measure:

\[
< \#\text{bytecodes}, \#\text{imul}, \sum_{L} |\text{gotochain}(L)| >
\]

which can be seen to become strictly smaller after each application of a peephole pattern.

The goto graph obtained as a fixed point is not unique.
It depends on the sequence in which the peephole patterns are applied.
That does not happen for the four examples given, but consider the two peephole patterns:

They clearly do not commute:

The word “optimizer” is somewhat misleading, since the code is not optimal but merely better.
Suppose \( OPM(G) \) is the shortest goto graph equivalent to \( G \).
Clearly, the shortest diverging goto graph is:

\[
D_{\text{min}} = L: \text{goto } L
\]

We can then decide the Halting problem on an arbitrary goto graph \( G \) as:

\[
OPM(G) = D_{\text{min}}
\]

Hence, the program \( OPM \) cannot exist.
The testing strategy for the optimizer has three phases.

First a careful argumentation that each peephole pattern is sound.

Second a demonstration that each peephole pattern is realized correctly.

Third a statistical analysis showing that the optimizer improves the generated programs.