Compiler Optimizations

CS 498: Compiler Optimizations
Fall 2007

University of Illinois at Urbana-Champaign

A note about the name of optimization

- It is a misnomer since there is no guarantee of optimality.
- We could call the operation *code improvement*, but this is not quite true either since compiler transformations are not guaranteed to improve the performance of the generated code.

Outline

Assignment Statement Optimizations
Loop Body Optimizations
Procedure Optimizations
Register allocation
Instruction Scheduling
Control Flow Optimizations
Cache Optimizations
Vectorization and Parallelization

Advanced Compiler Design Implementation. Steven S. Muchnick, Morgan and Kaufmann Publishers, 1997. Chapters 12 - 19

Historical origins of compiler optimization

"It was our belief that if FORTRAN, during its first months, were to translate any reasonable "scientific" source program into an object program only half as fast as its hand coded counterpart, then acceptance of our system would be in serious danger. This belief caused us to regard the design of the translator as the real challenge, not the simple task of designing the language."... "To this day I believe that our emphasis on object program efficiency rather than on language design was basically correct. I believe that has we failed to produce efficient programs, the widespread use of language like FORTRAN would have been seriously delayed.

John Backus
FORTRAN I, II, and III
Annals of the History of Computing
Vol. 1, No 1, July 1979

Compilers are complex systems

"Like most of the early hardware and software systems, Fortran was late in delivery, and didn't really work when it was delivered. At first people thought it would never be done. Then when it was in field test, with many bugs, and with some of the most important parts unfinished, many thought it would never work. It gradually got to the point where a program in Fortran had a reasonable expectancy of compiling all the way through and maybe even running. This gradual change of status from an experiment to a working system was true of most compilers. It is stressed here in the case of Fortran only because Fortran is now almost taken for granted, as it were built into the computer hardware."

Saul Rosen
Programming Languages and Systems
McGraw Hill 1967

Classifications of compiler optimizations

- By the scope
  - **Peephole optimizations**. A local inspection of the code to identify and modify inefficient sequence of instructions.
  - **Intraprocedural**. Transform the body of a procedure or method using information from the procedure itself.
  - **Interprocedural**. Uses information from several procedures to transform the program. Because of separate compilation this type of optimization is infrequently applied to complete programs.

Classifications of compiler optimizations

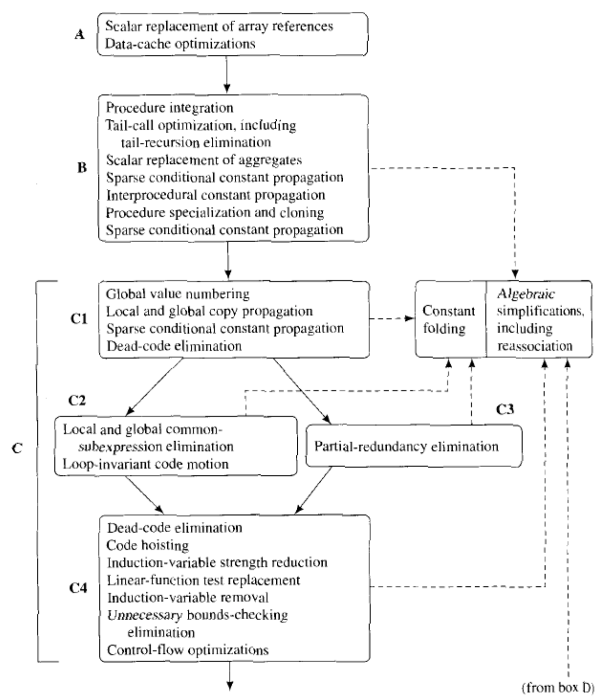
- By the time of application
  - Static. At compile-time
    - Source-to-source
    - Low-level optimizations
  - Dynamic. At execution time.
- By the source of the information
  - Code only
  - Code plus user assertions
  - Code plus profile information.

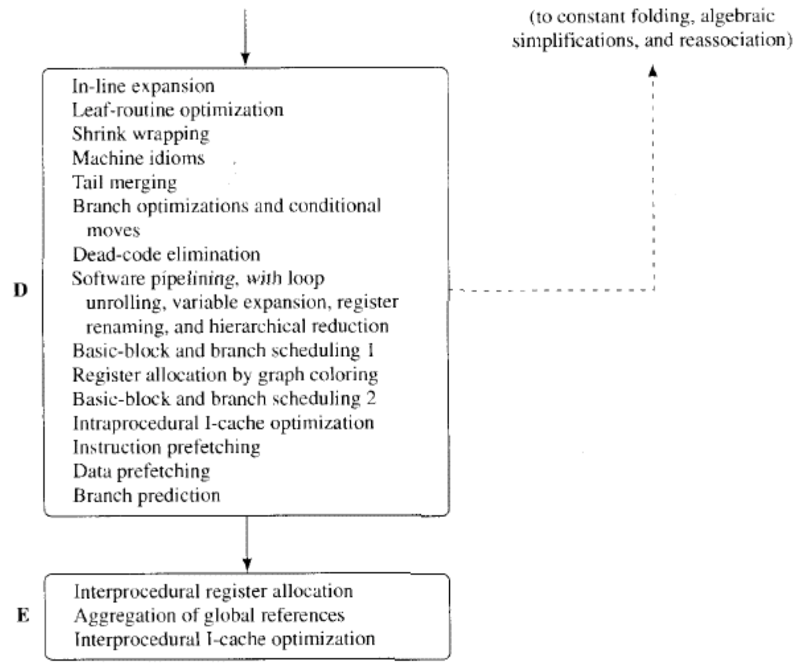
Optimizations included in a compiler

- The optimizations must be effective across the broad range of programs typically encountered.
- Also important is the time it takes to apply the optimization. A slow compiler is not desirable (and for some transformations it can become very slow).
- These factors limit the power of compilers. They are part of the reason why manual optimizations are needed.

Order and repetition of optimizations

- A possible order of optimizations appear in the figure below taken from S. Muchnick's book "Advanced compiler design implementation".
- Two quotes from that book:
  - "One can easily invent examples to show that no order can be optimal for all programs."
  - "It is easy to invent programs that will benefit from any number of repetitions of a sequence of optimizing transformations. While such examples can be constructed, it is important to note that they occur very rarely in practice. It is usually sufficient to apply the transformations that make up an optimizer once, or at most twice to get all or almost all the benefit one is likely to derive from them."
- The second phrase and the statements in the previous slide are the reasons why compilers are implemented the way they are.





I. Assignment statement optimizations

1. Constant folding
2. Scalar replacement of aggregates
3. Algebraic simplification and Reassociation
4. Common subexpression elimination
5. Copy propagation

Constant folding

- Constant-expressions evaluation or constant folding, refers to the evaluation at compile time of expressions whose operands are known to be constant.

- Example

```
i = 320 * 200 * 32

Most compilers will substitute the
computed value at compile time
```

Constant folding and propagation

Example	***Constant Propagation***	***Constant Propagation***	***Dead code elimination***

int a = 5
int b= a - 12/5;
int c;
 	
c = b*4;
if (c > 10) {
    c = c -10;
}
return c * (60/a)
	int a = 5
	int b= 3;
	int c;
	
	c = b*4;
	if (c > 10) {
	    c = c -10;
	}
	return c * 2;
		int c;
		
		c = 12;
		if (12 > 10) {
		    c = c -10;
		}
		return c * 2;
			return 4;

Constant folding and procedures

- Interprocedural constant propagation is particularly important when procedures or macros are passed constant parameters.
- Compilers do not do a perfect job at recognizing all constant expressions as can be seen in the next three examples from the c Sparc compiler (Workshop Compiler, version 5.0).
- In fact, constant propagation is undecidable.

Constant folding: Example-I

pp.c

```
#include <stdio.h>
int pp( )
{
  int ia =1;
  int ib =2;
  int result;

  result = ia +ib;
  return result;
}
```

cc -O3 -S pp.c

```
•   .global pp
                     pp:
/* 000000          */       retl    ! Result =  %o0
/* 0x0004          */       or      %g0,3,%o0
/* 0x0008        0 */       .type   pp,2
/* 0x0008          */       .size   pp,(.-pp)
```

Constant folding: Example-I

pp.c

```
#include <stdio.h>
int pp( )
{
  int ia =1;
  int ib =2;
  int result;

  result = ia +ib;
  return result;
}
```

cc -O3 -S pp.c

```
•   .global pp
                        pp:
/* 000000          */      retl    ! Result =  %o0
/* 0x0004          */      or      %g0,3,%o0
/* 0x0008        0 */       .type   pp,2
/* 0x0008          */      .size   pp,(.-pp)
```

Constant folding: Example-II

pp1.c

```
int pp(int id){
  int ic, ia, ib;
  if (id == 1) {
    ia =1;
    ib =2; }
  else {
    ia =2;
    ib =1;}
  ic = ia + ib;
  return ic;
}
```

cc -O3 -S pp1.c

```
!   3           !int pp(int id){
!   4           ! int ic, ia, ib;
!   5           ! if (id == 1) {
/* 000000     5 */     cmp     %o0,1
/* 0x0004       */    bne     .L77000003
/* 0x0008       */    or      %g0,1,%g1
                .L77000002:
!   6           !   ia =1;
!   7           !   ib =2; }
/* 0x000c     7 */     or      %g0,2,%g2
/* 0x0010       */    retl    ! Result =  %o0
/* 0x0014       */    add     %g1,%g2,%o0
                .L77000003:
!   8           ! else {
!   9           !   ia =2;
/* 0x0018     9 */     or      %g0,2,%g1
!  10            !   ib =1;}
/* 0x001c    10 */     or      %g0,1,%g2
/* 0x0020       */    retl    ! Result =  %o0
/* 0x0024       */    add     %g1,%g2,%o0
/* 0x0028     0 */     .type   pp,2
/* 0x0028       */    .size   pp,(.-pp)
```

Constant Folding: Example-III

pp2.c

```
int pp() {
  int ic, ia, ib;
  int id =1;
  if (id == 1) {
    ia =1;
    ib =2; }
  else {
    ia =2;
    ib =1;}
  ic = ia + ib;
  return ic;
}
```

cc -O3 -S pp1.c

```
 .global pp
                pp:
/* 000000      */    retl   ! Result =  %o0
/* 0x0004      */    or     %g0,3,%o0
/* 0x0008    0 */     .type  pp,2
/* 0x0008      */    .size  pp,(.-pp)
```

Scalar replacement of aggregates

- Replaces aggregates such as structures and arrays with scalars.
- Scalar replacement facilitates other optimizations such as register allocation, constant and copy propagation.

Scalar replacement of aggregates

```
typedef enum { APPLE, BANANA, ORANGE } VARIETY;
typedef enum { LONG, ROUND } SHAPE;
typedef struct fruit {
        VARIETY variety;
        SHAPE shape; } FRUIT;
char* Red = "red";
char* Yellow = "yellow";
char* Orange = "orange";

char*
color(CurrentFruit)
    FRUIT *CurrentFruit;
{   switch (CurrentFruit->variety) {
        case APPLE:   return Red;
                      break;
        case BANANA:  return Yellow;
                      break;
        case ORANGE:  return Orange;
    }
}

main( )
{   FRUIT snack;
    snack.variety = APPLE;
    snack.shape = ROUND;
    printf("%s\n",color(&snack));
}
```

Scalar replacement of aggregates

```
char* Red = "red";
char* Yellow = "yellow";
char* Orange = "orange";

main( )
{   FRUIT snack;
    VARIETY t1;
    SHAPE t2;
    COLOR t3;
    t1 = APPLE;
    t2 = ROUND;
    switch (t1) {
        case APPLE:   t3 = Red;
                      break;
        case BANANA:  t3 = Yellow;
                      break;
        case ORANGE:  t3 = Orange;
    }
    printf("%s\n",t3);
}
```

Scalar replacement of aggregates

- After dead code elimination and constant propagation, the result is:

```
main()
{
  printf ("%s \n", "red");
}
```

Scalar replacement of aggregates

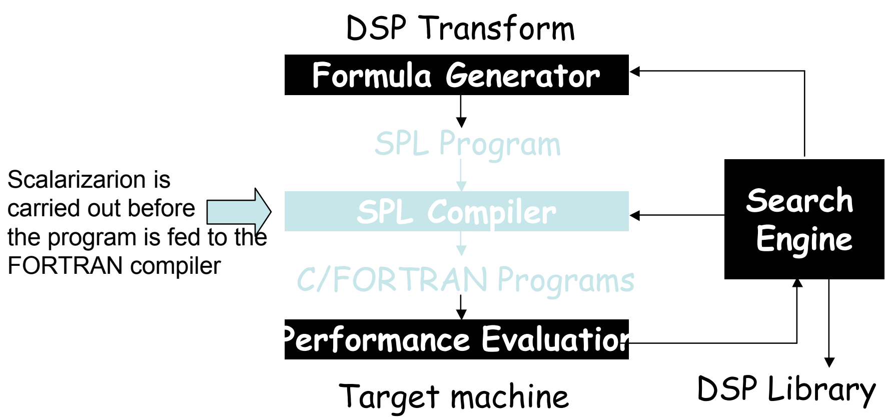
```
DO I = 1, N
   DO J = 1, M
        A(I) = A(I) + B(J)
   ENDDO
ENDDO
```

```
DO I = 1, N
   T = A(I)
   DO J = 1, M
        T = T + B(J)
   ENDDO
   A(I) = T
ENDDO
```

- A(I) can be left in a register throughout the inner loop
- Register allocation fails to recognize this

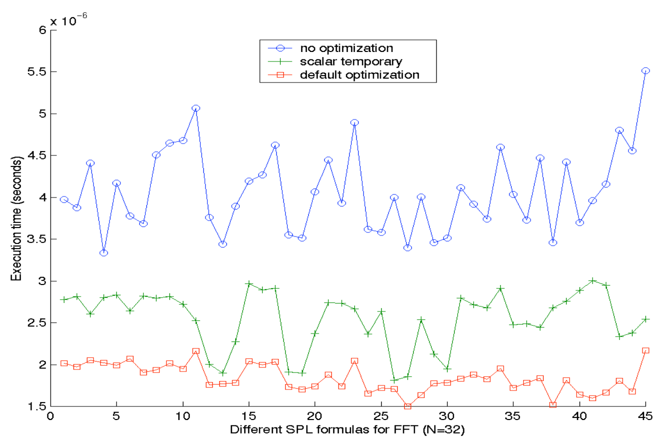
- All loads and stores to A in the inner loop have been saved
- High chance of T being allocated a register by the coloring algorithm

The effect of scalarization. An example from the SPIRAL project

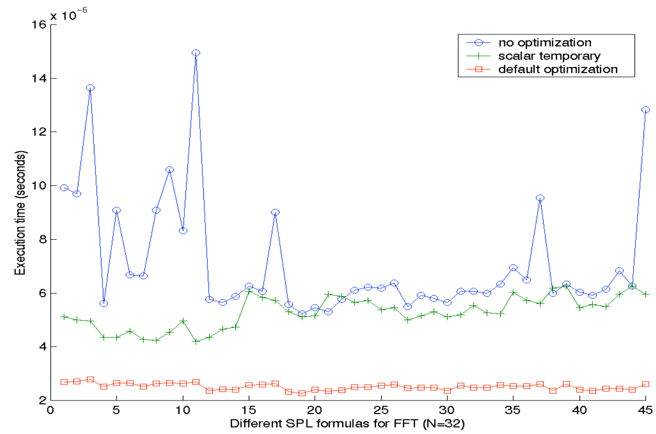


The SPIRAL System

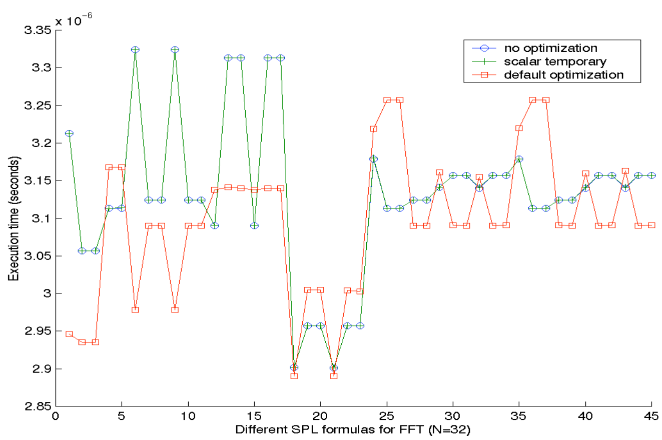
Basic Optimizations
(FFT, N=2^5, SPARC, f77 -fast -O5)



Basic Optimizations
(FFT, $N=2^5$, PII, g77 -O6 -malign-double)



Basic Optimizations
(FFT, $N=2^5$, MIPS, f77 -O3)



Algebraic simplification and Reassociation

- Algebraic simplification uses algebraic properties of operators or particular operand combinations to simplify expressions.

- Reassociation refers to using associativity, commutativity, and distributivity to divide an expressions into parts that are constant, loop invariant and variable.

Algebraic simplification and Reassociation

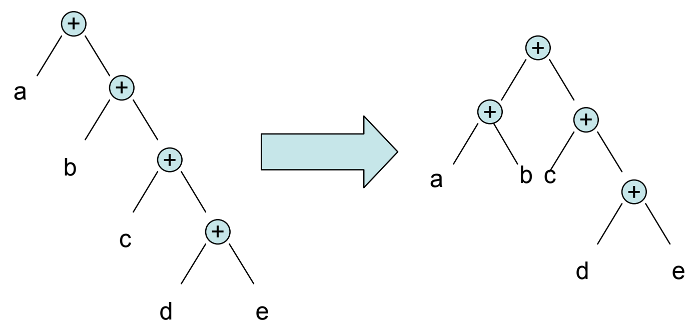
- For integers:
  - Expressions simplification
    - *i +0 = 0 + i = i - 0 = i*
    - *i ^ 2 = i * i (also strenght reduction)*
    - *i*5 can be done by t := i shl 3; t=t-i*
  - Associativity and distributivity can be applied to improve parallelism (reduce the height of expression trees).
- Algebraic simplifications for floating point operations are seldom applied.
  - The reason is that floating point numbers do not have the same algebraic properties as real numbers. For example, the in the code

```
eps:=1.0
while eps+1.0>1.0
        oldeps := eps
        eps:=0.5 * eps
```

  - Replacing eps+1.0 > 1.0 with eps >0.0 would change the result significantly. The original form computes the smallest number such that *1+ x = x* while the optimized form computes the maximal *x* such that *x/2* rounds to *0*.

Tree-height reduction

The goal is to reduce height of expression tree to reduce execution time in a parallel environment



Common subexpression elimination

- Transform the program so that the value of a (usually scalar) expression is saved to avoid having to compute the same expression later in the program.
- For example:

```
x = e^3+1
…
y= e^3
```

Is replaced (assuming that e is not reassigned in …) with

```
t=e^3
x = t+1
…
y=t
```

- There are local (to the basic block), global, and interprocedural versions of cse.

Copy propagation

- Eliminates unnecessary copy operations.
- For example:

```
x = y
<other instructions>
t = x + 1
```

Is replaced (assuming that neither x nor y are reassigned in …) with

```
<other instructions>
t = y + 1
```

- Copy propagation is useful *after* common subexpression elimination. For example.

```
x = a+b
…
y = a+b
```

- Is replaced by CSE into the following code

```
t = a+b
x = t
…
z = x
y = a+b
```

- Here `x=t` can be eliminated by copy propagation.

II. Loop body optimizations

6. Loop invariant code motion
7. Induction variable detection
8. Strength reduction

University of Illinois at Urbana-Champaign

Loop invariant code motion

- Recognizes computations in loops that produce the same value on every iteration of the loop and moves them out of the loop.

- An important application is in the computation of subscript expressions:

```
do i=1,n
 do j=1,n
      …a(j,i)….
```

- Here `a(j,i)` must be transformed into something like `a((i-1)*M+j-1)` where `(i-1)*M` is a loop invariant expression that could be computed outside the `j` loop.

Example

```
pp1()
{
float a[100][100];
int i,j;
for (i=1;i++;i<=50)
for (j=1;j++;j<=50)
a[i][j]=0;
}
```

unoptimized →

```
.L95:
! 8 a[i][j]=0;
sethi%hi(.L_cseg0),%o0
ld[%o0+%lo(.L_cseg0)],%f2
sethi39,%o0
xor%o0,-68,%o0
add%fp,%o0,%o3
sethi39,%o0
xor%o0,-72,%o0
ld[%fp+%o0],%o2
sll%o2,4,%o1
sll%o2,7,%o0
add%o1,%o0,%o1
sll%o2,8,%o0
add%o1,%o0,%o0
add%o3,%o0,%o1
sethi39,%o0
xor%o0,-76,%o0
ld[%fp+%o0],%o0
sll%o0,2,%o0
st%f2,[%o1+%o0]
sethi39,%o0
xor%o0,-76,%o0
ld[%fp+%o0],%o0
mov%o0,%o2
sethi39,%o0
xor%o0,-76,%o0
ld[%fp+%o0],%o0
add%o0,1,%o1
sethi39,%o0
xor%o0,-76,%o0
cmp%o2,%g0
bne.L95
st%o1,[%fp+%o0]
```

↓ optimized
cc -O3 -S pp1.c

```
a[i][j]=0;
...
.L900000109:
or%g0,%o0,%g2
add%o3,4,%o3
add%o0,1,%o0
cmp%g2,0
bne,a.L900000109
st%f0,[%o3] ! volatile
```

Induction variable detection

- Induction variables are variables whose successive values form an arithmetic progression over some part of the program.

- Their identification can be used for several purposes:
  - Strength reduction (see next).
  - Elimination of redundant counters.
  - Elimination of interactions between iterations to enable parallelization.

Elimination of redundant counters

```
integer a(100)
t1=202
do i=1,100
  t1= t1-2
  a(i)=t1
```

```
     t1 ← 202
     i ← 1
L1:  t2 ← i > 100
     if t2 goto L2
     t1 ← t1 - 2
     t3 ← addr a
     t4 ← t3 - 4
     t5 ← 4 * i
     t6 ← t4 + t5
     *t6 ← t1
     i ← i + 1
     goto L1
L2:
```

(a)

```
     t1 ← 202
     t3 ← addr a
     t4 ← t3 - 4
     t5 ← 4
     t6 ← t4
     t7 ← t3 + 396
L1:  t2 ← t6 > t7
     if t2 goto L2
     t1 ← t1 - 2
     t6 ← t4 + t5
     *t6 ← t1
     t5 ← t5 + 4
     goto L1
L2:
```

(b)

Bernstein's conditions and induction variables

- The following loop cannot be transform as is into parallel form

```
do i=1,n
   k=k+3
   A(k) = B(k)+1
end do
```

- The reason is that induction variable k is both read and written on all iterations. However, the collision can be easily removed as follows

```
do i=1,n
   A(3*i) = B(3*i) +1
end do
```

- Notice that removing induction variables usually has the opposite effect of strength reduction.

Strength reduction

- From Allen, Cocke, and Kennedy "Reduction of Operator Strength" in Muchnick and Jones "Program Flow Analysis" AW 1981.
- In real compiler probably multiplication to addition is the only optimization performed.
- Candidates for strength reduction

1.Multiplication by a constant

```
loop
   n=i*a
   ...
   i=i+b
```

– after strength reduction

```
loop
   n=t1
   ...
   i=i+b
   t1=t1+a*b
```

– after loop invariant removal

```
c = a * b
t1 = i*a
loop
   n=t1
   ...
   i=i+b
   t1=t1+c
```

Strength reduction

2.Multiplication by a constant plus a term

```
loop
  n=i*a+c
  ...
  i=i+b
```

– after strength reduction

```
loop
  n=t1
  ...
  i=i+b
  t1=t1+a*b
```

– Notice that the update to t1 does not change by the addition of the constant. However, the initialization assignment before the loop should change.

Strength reduction

3.Two induction variables multiplied by a constant and added

```
loop
  n=i*a+j*b
  ...
  i=i+c
  ...
  j=j+d
```

– after strength reduction

```
loop
   n=t1
   ...
   i=i+c
   t1=t1+a*c
   j=j+d
   t1=t1+b*d
```

Strength reduction

4.Multiplication of one induction variable by another

```
loop
    n=i*j
    ...
    i=i+a
    ...
    j=j+b
```

– After strength reduction of i*j

```
loop
    n=t1
    ...
                    --------- t1=i*j
    i=i+a
                    --------- new t1 should be (i+a)*j=t1+a*j
    t1=t1+a*j
    ...
    j=j+b
                    -------- new t1 should be i*(j+b)=t1+b*i
    t1=t1+b*i
```

Strength reduction

- After strength reduction of a*j

```
loop
    n=t1
    ...
    i=i+a
    t1=t1+t2
    ...
    j=j+b
    t1=t1+b*i
    t2=t2+a*b
```

- b*i is handled similarly.

Strength reduction

5.Multiplication of an induction variable by itself

```
loop
    n=i*i
    ...
    i=i+a
```

– After strength reduction

```
loop
    n=t1
    ...
    i=i+a
    -------- new t1 should be (i+a)*(i+a)=t1+2*a*i+a*a
    t1=t1+2*a*i+a*a
```

– Now strength reduce 2*a*i+a*a

```
loop
    n=t1
    ...
    i=i+a
    t1=t1+t2
                -------- new t2 should be 2*a*(i+a)+a*a=t2+2*a*a
    t2=t2+2*a*a
```

Strength reduction

6. Integer division

```
loop
    n=i/a
    ...
    i=i+b
```

– After strength reduction

```
loop
    n=t1
    ...
    i=i+b
    t2=t2+(b mod a)
    if t2 >= a then
            t1++
            t2=t2-a
    t1=t1+b/a
```

Strength reduction

7. Integer modulo function

```
loop
    n=i mod a
    ...
    i=i+b
```

– After strength reduction

```
loop
    n=t1
    ...
    i=i+b
    t1=t1+(b mod a)
    if t1 >= a then
           t1 = t1 -a
```

Strength reduction

8. Exponentiation

```
loop
    x=a^i
    ...
    i=i+b
```

– After strength reduction

```
loop
    x=t1
    ...
    i=i+b
    t1=t1*(a^b)
```

Strength reduction

9. Trigonometric functions

```
loop
    y=sin(x)
    ...
    x=x+Δx
```

– After strength reduction

```
loop
    y=sin(x)
    ...
    x=x+Δx
    tsinx=tsinx*tcosΔx+tcosx*tsinΔx
    tcosx=tsinx*tsinΔx+tcosx*tcosΔx
```

Unnecessary bounds checking elimination

- By propagating assertions it is possible to avoid unnecessary bound checks
- For example, bound checks are not needed in:

```
real a(1000)
do i=1,100
  … a(i)…
end do
```

- And they are not needed either in

```
if i > 1 and i < 1000 then
  … a(i)…
end if
```

- A related transformation is predicting the maximum value subscripts will take in a region to do pre-allocation for languages (like MATLAB) where arrays grow dynamically.

III. Procedure optimizations

10. Tail recursion elimination
11. Procedure integration
12. Leaf routine optimization

University of Illinois at Urbana-Champaign

Tail call and tail recursion elimination

- A call from procedure `f()` to procedure `g()` is a tail call if the only thing `f()` does, after `g()` returns to it, is itself return.
- Converts tail recursive procedures into iterative form (See example in next slide)

Tail recursion elimination

```
void make_node(p,n)
   struct node *p;
   int n;
{  struct node *q;
   q = malloc(sizeof(struct node));
   q->next = nil;
   q->value = n;
   p->next = q;
}

void insert_node(n,l)
   int n;
   struct node *l;
{  if (n > l->value)
      if (l->next == nil) make_node(l
      else insert_node(n,l->next);
}
```

```
void insert_node(n,l)
   int n;
   struct node *l;
{loop:
   if (n > l->value)
      if (l->next == nil) make_node(l,n);
      else
      {  l := l->next;
         goto loop;
      }
}
```

Procedure integration

- Inline the procedure.
- This gives the compiler more flexibility in the type of optimizations it can apply.
- Can simplify the body of the routine using parameter constant values.
- Procedure cloning can be used for this last purpose also.
- If done blindly, it can lead to long source files and incredibly long compilation times

```
      subroutine sgefa(a,lda,n,ipvt,info)
      integer lda,n,ipvt(1),info
      real a(lda,1)
      real t
      integer isamax,j,k,kp1,l,nm1
            . . .
            do 30 j = kp1, n
               t = a(l,j)
               if (l .eq. k) go to 20
                  a(l,j) = a(k,j)
                  a(k,j) = t
   20          continue
               call saxpy(n-k,t,a(k+1,k),1,a(k+1,j),1)
   30       continue
            . . .
      subroutine saxpy(n,da,dx,incx,dy,incy)
      real dx(1),dy(1),da
      integer i,incx,incy,ix,iy,m,mp1,n
      if (n .le. 0) return
      if (da .eq. ZERO) return
      if (incx .eq. 1 .and. incy .eq. 1) go to 20
      ix = 1
      iy = 1
      if (incx .lt. 0) ix = (-n+1)*incx + 1
      if (incy .lt. 0) iy = (-n+1)*incy + 1
      do 10 i = 1,n
          dy(iy) = dy(iy) + da*dx(ix)
          ix = ix + incx
          iy = iy + incy
   10 continue
      return
   20 continue
      do 30 i = 1,n
          dy(i) = dy(i) + da*dx(i)
   30 continue
      return
      end
```

```
      subroutine sgefa(a,lda,n,ipvt,info)
      integer lda,n,ipvt(1),info
      real a(lda,1)
      real t
      integer isamax,j,k,kp1,l,nm1
            . . .
            do 30 j = kp1, n
               t = a(l,j)
               if (l .eq. k) go to 20
                  a(l,j) = a(k,j)
                  a(k,j) = t
   20          continue
               if (n-k .le. 0) goto 30
               if (t .eq. 0) goto 30
               do 40 i = 1,n-k
                  a(k+i,j) = a(k+i,j) + t*a(k+i,k)
   40          continue
   30       continue
            . . .
```

Leaf routine optimization

- A leaf routine is a procedure that is a leaf in the call graph of a program, i.e., it does not call other procedures.
  - Need to determine a routine is a leaf routine
  - Nead to determine how much storage (registers and stack, the routine requires)
- Can save instructions to save registers
- Can save the code that creates and reclaims a stack frame for the leaf routine is stack space is not necessary.

IV. Register allocation

University of Illinois at Urbana-Champaign

Register allocation

- Objective is to assign registers to scalar operands in order to minimize the number of memory transfers.
- An NP-complete problem for general programs. So need heuristics. Graph coloring-based algorithm has become the standard.
- Register allocation algorithms will be discussed in more detail later.

V. Instruction Scheduling

University of Illinois at Urbana-Champaign

Instruction scheduling

- Objective is to minimize execution time by reordering executions.
- Scheduling is an NP-complete problem.
- More about instruction scheduling when we discuss instruction-level parallelism.

	Issue latency	Result latency
L: ldf [r1],f0	1	1
fadds f0,f1,f2	1	7
stf f2,[r1]	6	3
sub r1,4,r1	1	1
cmp r1,0	1	1
bg L	1	2
nop	1	1

FIG. 17.15 A simple SPARC loop with assumed issue and result latencies.

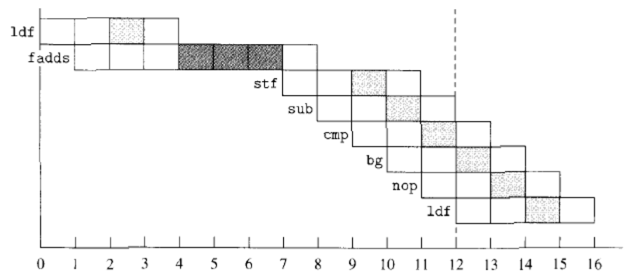


FIG. 17.16 Pipeline diagram for the loop body in Figure 17.15.

	Issue latency	Result latency
ldf [r1],f0		
fadds f0,f1,f2		
ldf [r1-4],f0		
L: stf f2,[r1]	1	3
fadds f0,f1,f2	1	7
ldf [r1-8],f0	1	1
cmp r1,8	1	1
bg L	1	2
sub r1,4,r1	1	1
stf f2,[r1]		
sub r1,4,r1		
fadds f0,f1,f2		
stf f2,[r1]		

FIG. 17.17 The result of software pipelining the loop in Figure 17.15, with issue and result latencies.

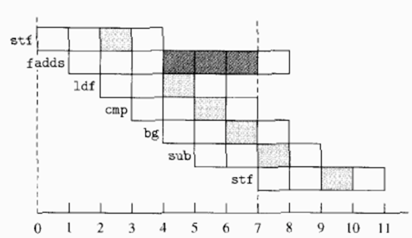


FIG. 17.18 Pipeline diagram for the loop body in Figure 17.17.

VI. Control-flow optimizations

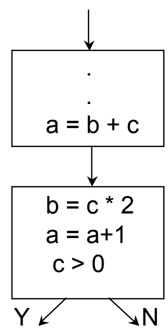
University of Illinois at Urbana-Champaign

Unreachable code elimination

- Unreachable code is code that cannot be executed regardless of the input.
- Eliminating it saves space.
- Unreacheable code elimination is different from dead code elimination.
  - Dead code is code that can be executed but has no effect in the result of the computation being performed.

Straightening

- It applies to pairs of basic blocks so that the first has no successors other than the second and the second has no predecessors other than the first.



Straightening

```
L1: . . .                        L1: . . .
    a ← b + c                        a ← b + c
    goto L2                          b ← c * 2
                                     a ← a + 1
                                     if (c > 0) goto L3
                                     goto L5
L6: . . .                        L6: . . .
    goto L4                          goto L4
L2: b ← c * 2
    a ← a + 1
    if (c > 0) goto L3
L5: . . .                        L5: . . .
(a)                              (b)
```

FIG. 18.4 MIR code versions of the example code in Figure 18.3.

If simplification

- Applies to conditional constructs one or both of whose arms are empty

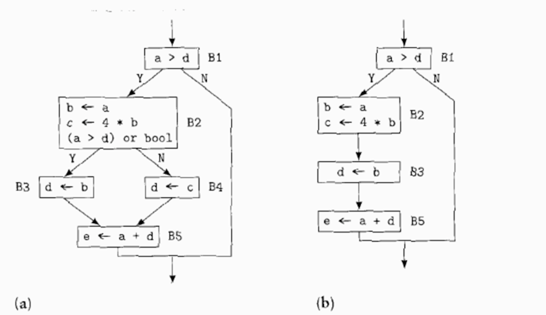


FIG. 18.7 (a) An example with a condition that is a common subexpression, and (b) the result of removing the dead arm of the if.

Loop inversion

- Transforms a while loop into a repeat loop.
  - Repeat loops have only a conditional branch at the end.
  - While loops have a conditional branch at the beginning and an uncoditional branch at the end
  - For the conversion the compiler needs to prove that the loop is entered at least once.

```
for (i = 0; i < 100; i++)
{  a[i] = i + 1;

}
```
(a)

```
i = 0;
while (i < 100)
{   a[i] = i + 1;
    i++;
}
```
(b)

```
i = 0;
repeat
{   a[i] = i + 1;
    i++;
} until (i >= 100)
```
(c)

FIG. 18.9 An example of loop inversion in C.

Unswitching

- Moves loop-invariant conditional branches out of loops

```
do i = 1,100
   if (k.eq.2) then
      a(i) = a(i) + 1
   else
      a(i) = a(i) - 1
   endif
enddo
```

(a)

```
if (k.eq.2) then
   do i = 1,100
      a(i) = a(i) + 1
   enddo
else
   do i = 1,100
      a(i) = a(i) - 1
   enddo
endif
```

(b)

FIG. 18.12 (a) Fortran 77 code with a loop-invariant predicate nested in a loop, and (b) the result of unswitching it.

Dead code elimination

- Eliminates code that do not affect the outcome of the program.

VI. Cache optimizations

University of Illinois at Urbana-Champaign

Cache Optimizations

- Usually required “deep” transformations to the program.
- Most studied are those transformations related to loops:
  - Loop tiling (blocking)
  - Loop fission/fusion
  - Loop interchange
- Although these are well understood, they are seldom implemented in real compilers
- Other transformations that have been studied include the change of data structures to increase locality.
- More will be said about this when we discuss locality optimizations.

VII. Vectorization and parallelization

University of Illinois at Urbana-Champaign

Vectorization and Parallelization

- Transformations for vectorization and parallelization rely on dependence analysis.
- Most compilers (try to) do some form of parallelization.
- Vectorization was crucial for the early supercomputers, mainly because there was no notation for vector expressions in Fortran 77 (the language of choice for supercomputing in those days).
- Fortran 90 solved that problem, but most supercomputers today are not vector machines and Fortran is no longer the language of choice.
- Intel implemented a vectorizer to automatically take advantage of their SSE devices.

How well compilers work?

- Evidence accumulated for many years show that compilers today do not meet their original goal.

- Problems at all levels:

-- Detection of parallelism

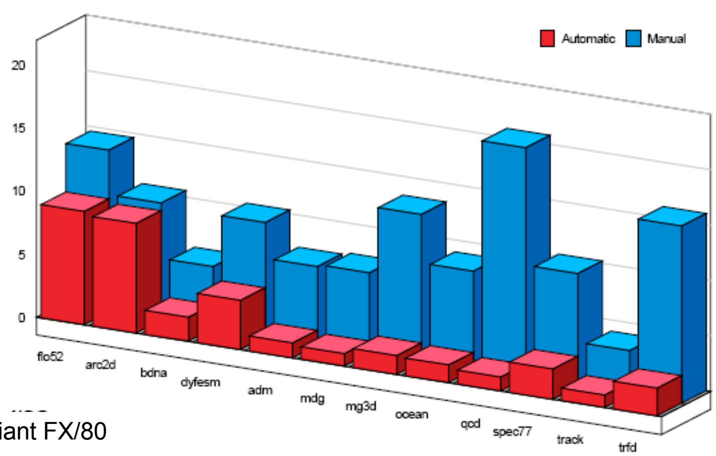
-- Vectorization

-- Locality enhancement

-- Traditional compilation

- Results from our research group.

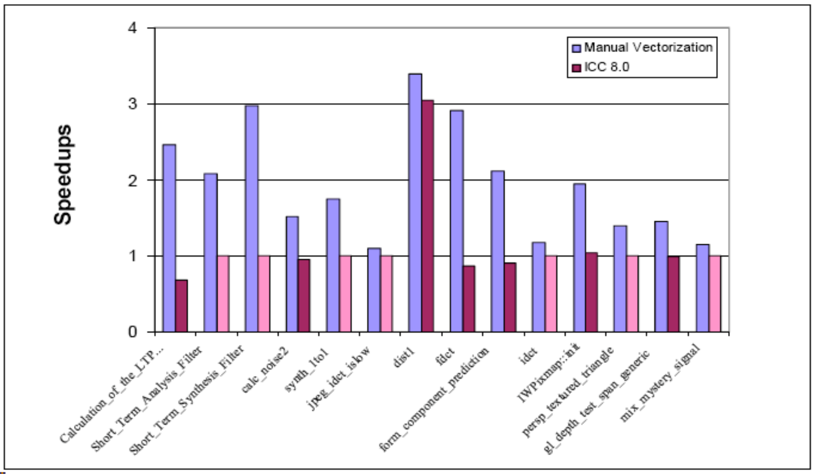
Automatic Detection of Parallelism



Alliant FX/80

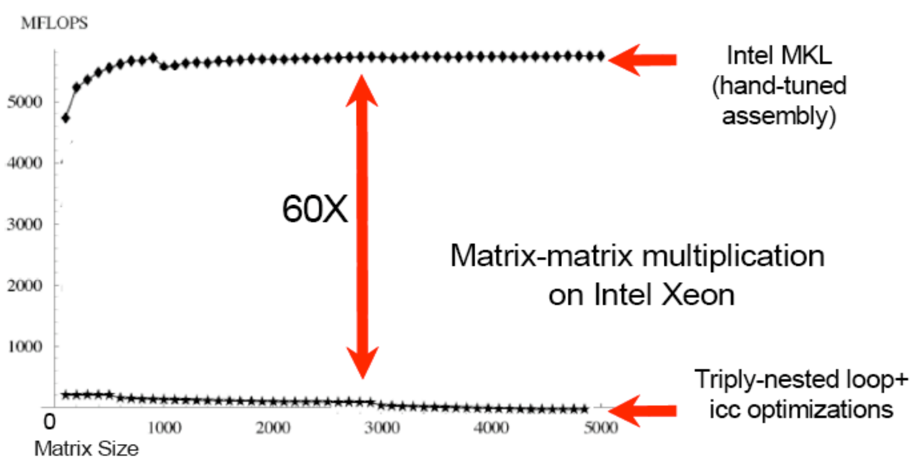
See. R. Eigenmann, J. Hoeflinger, D. Padua On the Automatic Parallelization of the Perfect Benchmarks. IEEE TPDS, Jan. 1998.

Vectorization



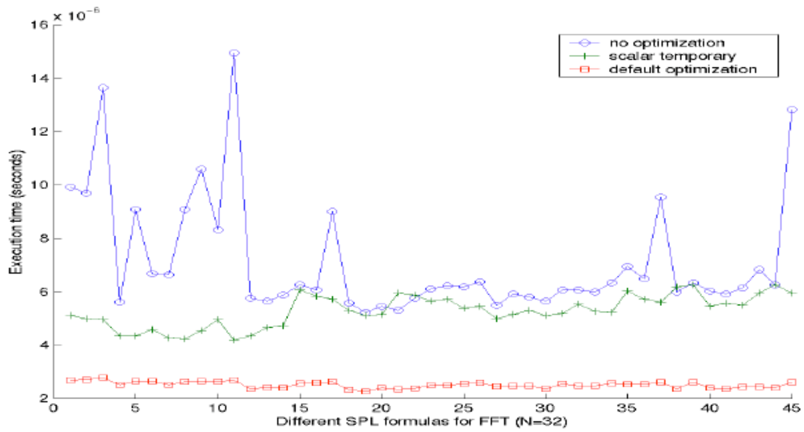
G. Ren, P. Wu, and D. Padua: An Empirical Study on the Vectorization of Multimedia Applications for Multimedia Extensions. IPDPS 2005

Locality Enhancement



See. K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, P. Stodghill. Is Search Really Necessary to Generate High-Performance BLAS? Proceedings of the IEEE. February 2005.

Scalar Optimizations



See. J. Xiong, J. Johnson, and D Padua. *SPL: A Language and Compiler for DSP Algorithms.* PLDI 2001

Empirical Search to Compiler Switch Selection

- Empirical search has been used to identify the best compiler switches.
- Compilers have numerous switches. Here is a partial list of gcc switches:

-fdefer-pop
-fdelayed-branch
-fguess-branch-probability
-fcprop-registers
-floop-optimize
-fif-conversion
-fif-conversion2
-ftree-ccp -ftree-dce
-ftree-dominator-opts
-ftree-dse
-ftree-ter -ftree-lrs
-fcaller-saves
-fpeephole2
-fschedule-insns

-ftree-sra -ftree-copyrename
-ftree-fre -ftree-ch
-fmerge-constants
-fthread-jumps
-fcrossjumping
-foptimize-sibling-calls
-fcse-follow-jumps
-fcse-skip-blocks
-fgcse
-fgcse-lm
-fexpensive-optimizations
-fstrength-reduce
-frerun-cse-after-loop
-frerun-loop-opt

-fschedule-insns2
-fsched-interblock
-fsched-spec
-fregmove
-fstrict-aliasing
-fdelete-null-pointer-checks
-freorder-blocks
-freorder-functions
-funit-at-a-time
-falign-functions
-falign-jumps
-falign-loops
-falign-labels
-ftree-vrp
-ftree-pre

Empirical Search to Compiler Switch Selection

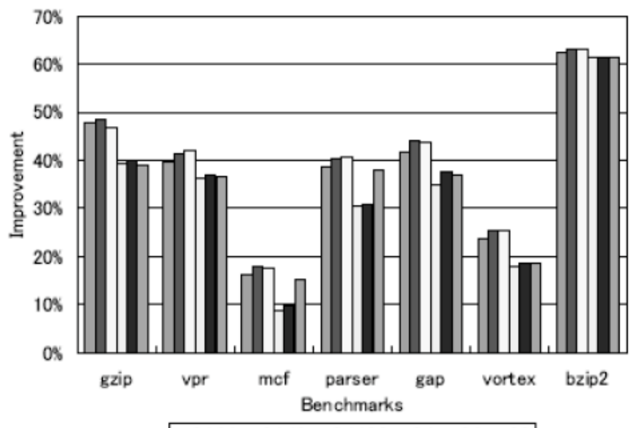
- These switches can be set by groups using -O1, -O2 (all the previous switches) and, -O3 (all of the previous switches plus a few others). See:http://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Optimize-Options.html#Optimize-Options
- Most compilers can be controlled by numerous switches. Some are not publicly known. See for example the *Intel Compiler Black Belt Guide to Undocumented Switches*.
- For all compilers, documented or not, it is not clear which subset of switches should be set to maximize performance.

Empirical Search to Compiler Switch Selection

•Two different projects have studied the problem of searching for the best set of compiler switches. Both focused on gcc. Since the number of combinations is astronomical, these projects use heuristics.

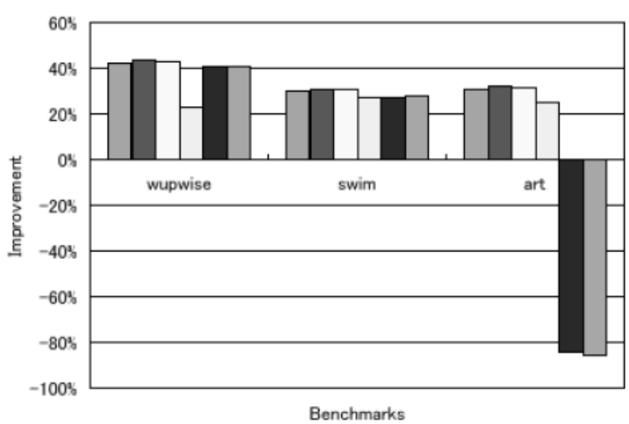
• The following figures are from: *M. Haneda, P.M.W. Knijnenburg, H.A.G. Wijshoff. Automatic selection of compiler options using non-parametric inferential statistics. PACT'05.*

Empirical Search to Compiler Switch Selection



(a) SPECint

Empirical Search to Compiler Switch Selection



(b) SPECfp

Empirical Search to Compiler Switch Selection

- See also: *Zhelong Pan and Rudolf Eigenmann, Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning, The 4th Annual International Symposium on Code Generation and Optimization (CGO),March 2006.*

- *L. Almagor, K.D. Cooper, A. Grosul, T.J. Harvey, S.W. Reeves, D.Subramanian, L. Torczon, and T. Waterman "Finding Effective Compilation Sequences." Proceedings of the 2004 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES) (June 2004): 231-230.*

and

- *http://www.coyotegulch.com/products/acovea/index.html*

Empirical Search to Compiler Switch Selection

- Orchestration algorithms
  - Batch Elimination (BE)
    - Identify the optimizations with negative effects and turn them off at once.

      1. Compile the application under the baseline $B = \{F_1 = 1, F_2 = 1, ..., F_n = 1\}$. Execute the generated code version to get the baseline execution time T_B.
      2. For each optimization F_i, switch it off from B and compile the application. Execute the generated version to get $T(F_i = 0)$, and compute the $RIP_B(F_i = 0)$ according to Equation 3.
      3. Disable all optimizations with negative RIPs to generate the final, tuned version.

    - Good when the optimizations do not interact with each other
    - O(n)

Empirical Search to Compiler Switch Selection

- Orchestration algorithms
  - Iterative Elimination (IE)

1. Let B be the option combination for measuring the baseline execution time, T_B. Let the set of S represent the optimization search space. Initialize $S = \{F_1, F_2, ..., F_n\}$ and $B = \{F_1 = 1, F_2 = 1, ..., F_n = 1\}$.
2. Compile and execute the application under the baseline setting to get the baseline execution time T_B.
3. For each optimization $F_i \epsilon S$, switch F_i off from B and compile the application, execute the generated code version to get $T(F_i = 0)$, and compute the RIP of F_i relative to the baseline B, $RIP_B(F_i = 0)$, according to Equation 3.
4. Find the optimization F_x with the most negative RIP. Remove F_x from S, and set F_x to 0 in B.
5. Repeat Steps 2, 3 and 4 until all options in S have non-negative RIPs. B represents the final option combination.

•Take into account the interaction of optimizations

•Switches off the one optimization with the most negative effect from the baseline.

•O(n*n)

Empirical Search to Compiler Switch Selection

- Orchestration algorithms
  - Combined Elimination (CE)

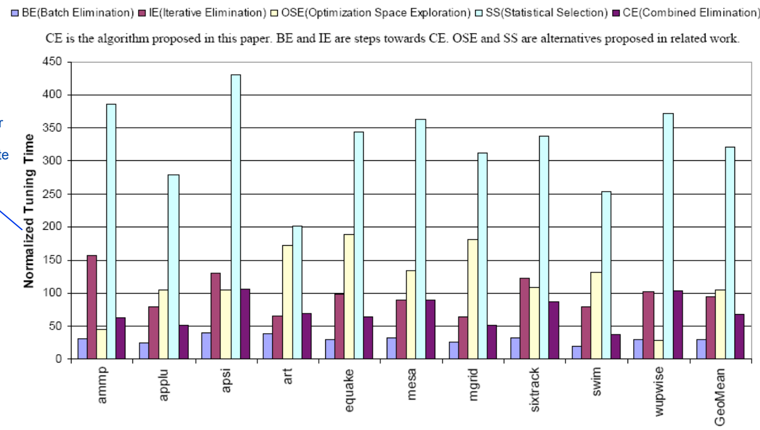
1. Let B be the baseline option combination. Let the set of S represent the optimization search space. Initialize $S = \{F_1, F_2, ..., F_n\}$ and $B = \{F_1 = 1, F_2 = 1, ..., F_n = 1\}$.
2. Compile and execute the application under the baseline setting to get the baseline execution time T_B. Measure the $RIP_B(F_i = 0)$ of each optimization option F_i in S relative to the baseline B.
3. Let $X = \{X_1, X_2, ..., X_l\}$ be the set of optimization options with negative RIPs. X is sorted in an increasing order, that is, the first element, X_1, has the most negative RIP. Remove X_1 from S and set X_1 to 0 in B. (B is changed in this step.) For i from 2 to l,
   - Measure the RIP of X_i relative to the baseline B.
   - If the RIP of X_i is negative, remove X_i from S and set X_i to 0 in B.
4. Repeat Steps 2 and 3 until all options in S have non-negative RIPs. B represents the final solution.

•Apply the idea of BE in each iteration after identifying the optimizations with negative effects.

•O(n*n)

Empirical Search to Compiler Switch Selection

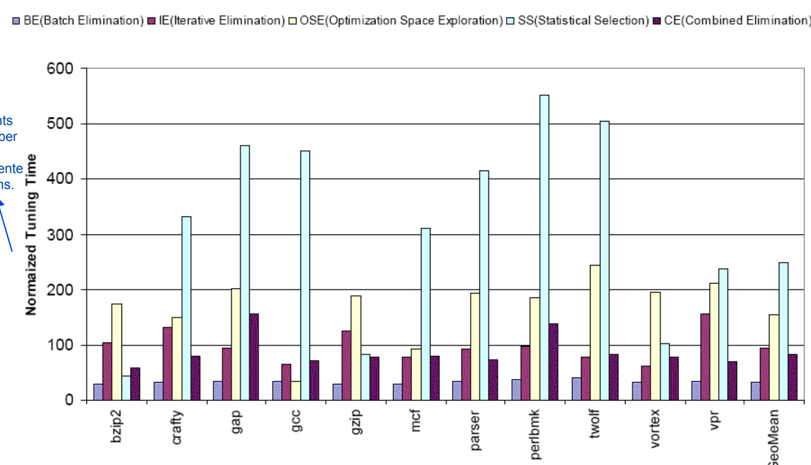
Use train input set to tune optimization switches. Use ref input to measure execution time.



Roughly represents the number of experimented versions.

(a) Normalized tuning time of the orchestration algorithms for SPEC CPU2000 FP benchmarks on Pentium IV

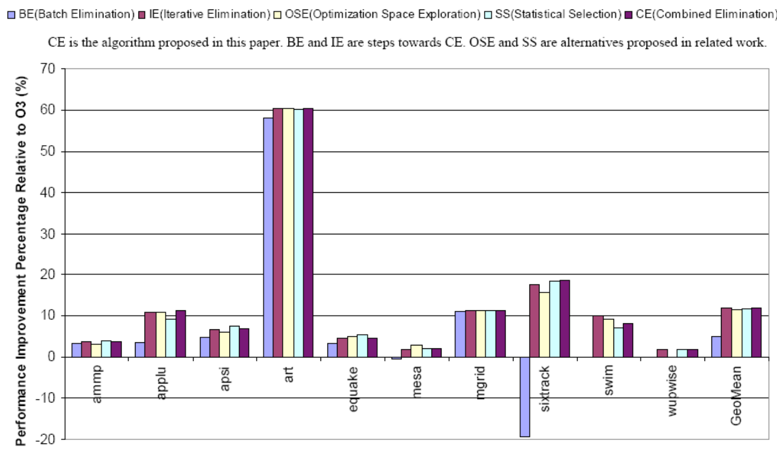
Empirical Search to Compiler Switch Selection



Roughly represents the number of experimented versions.

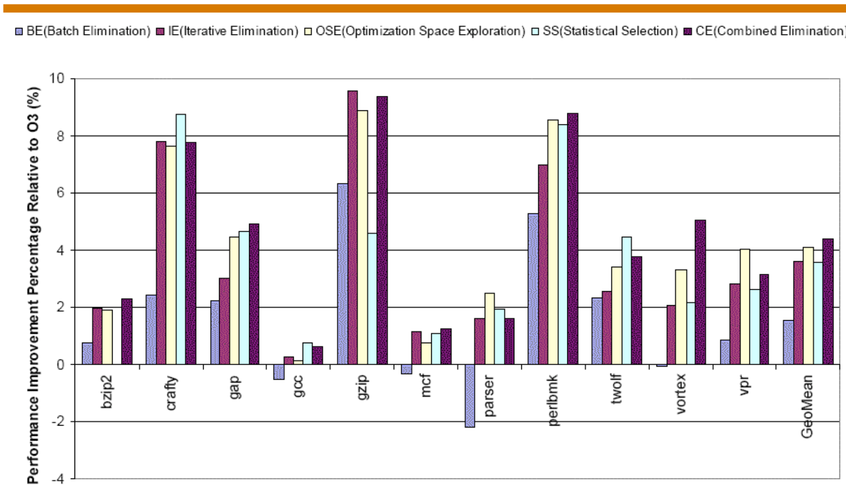
(b) Normalized tuning time of the orchestration algorithms for SPEC CPU2000 INT benchmarks on Pentium IV

Empirical Search to Compiler Switch Selection



(a) Program performance achieved by the orchestration algorithms relative to the baseline under the highest optimization level "O3" for the SPEC CPU2000 FP benchmarks on Pentium IV

Empirical Search to Compiler Switch Selection



(b) Program performance achieved by the orchestration algorithms relative to the baseline under the highest optimization level "O3" for the SPEC CPU2000 INT benchmarks on Pentium IV