Addendum to Chapter 2

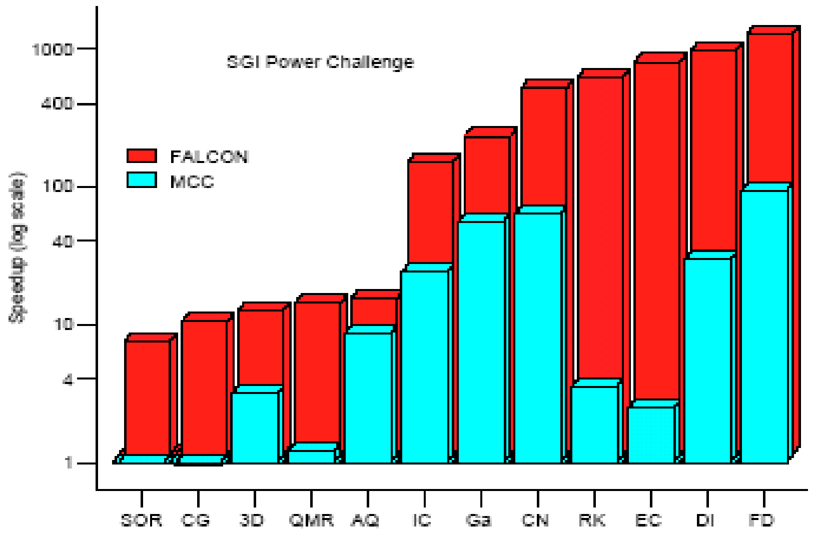
Basic optimization techniques for serial code

Languages

- Languages are the interface between programmers and machines.
- They affect
  - -- Productivity
  - -- Performance
- Typically, performance is sacrificed in the name of productivity (including portability). There is obviously a trade-off. Importance of performance should affect choice.
  - -- Assembly language is best from the point of view of performance (assuming unlimited human resources).

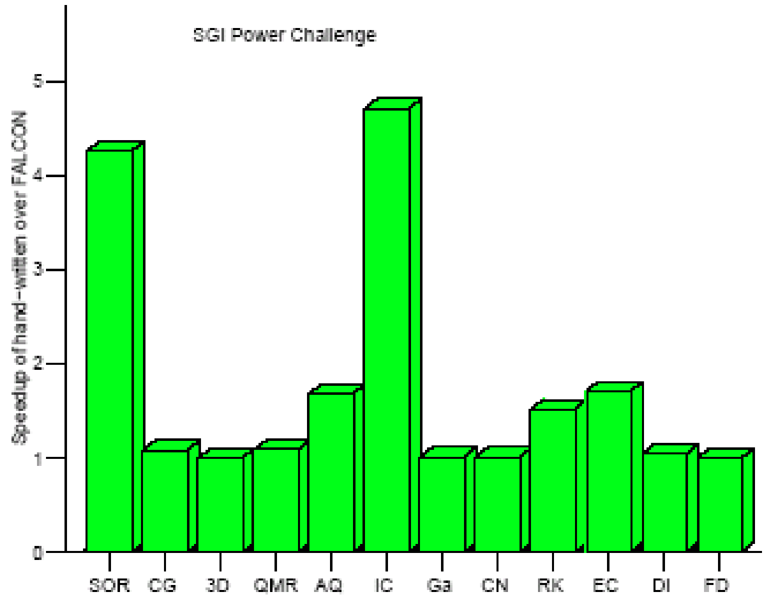
-- Fortran 77 and C are, among popular languages, second best (for performance)

-- Interpreted languages are slower. Sometimes they are dramatically slower. Examples include: MATLAB, Python, Java.



Speedup due to compilation of MATLAB

-- Languages originally designed for interpretation can be compiled, but performance tends to suffer because of language characteristics. Runtime specification of types is a problem for performance. For MATLAB compilation see L. De Rose, D. Padua, Techniques for the translation of MATLAB programs into Fortran90, ACM Trans. on Programming Languages and Systems 21 (2) (1999), pp. 286--323.



- -- Another example is provided by Java. Implementations are required to store arrays in ways that affect performance. See: José E. Moreira, Samuel P. Midkiff, Manish Gupta, Pedro V. Artigas, Peng Wu, George Almasi The Ninja Project. October 2001 Communications of the ACM, Volume 44 Issue 10.

- High-level notations/new languages should be studied. Much to be gained.(See *G. Bikshandi et al Programming for Parallelism and Locality with Hierarchically Tiled. Proc. of the International Symposium on Principles and Practice of Parallel Programming, March 2006.*)
  - -- But .. New languages will not significantly reduce the cost of the optimization process
  - -- Automatic optimization is needed.
  - -- Programming languages designed for performance should be **automatic optimization enablers**.
  - -- Need language/compiler co-design.

Compilers

- Program optimization was the objective of compilers from the outset.

"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."

John Backus
Fortran I, II and III
Annals of the History of Computing, July 1979.

- Still far from solving the problem. Problems in Computer Science seem much easier than they are.
- The objective of compilers is to bridge the gap between programmer's world and machine world. Between readable/easy to maintain code and unreadable high-performing code.

Compiler Optimizations

First, a note about the word *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.

A classification

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.

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*.

Which optimizations to include?

- 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 the compiler can become very slow).

Order and repetition of optimizations

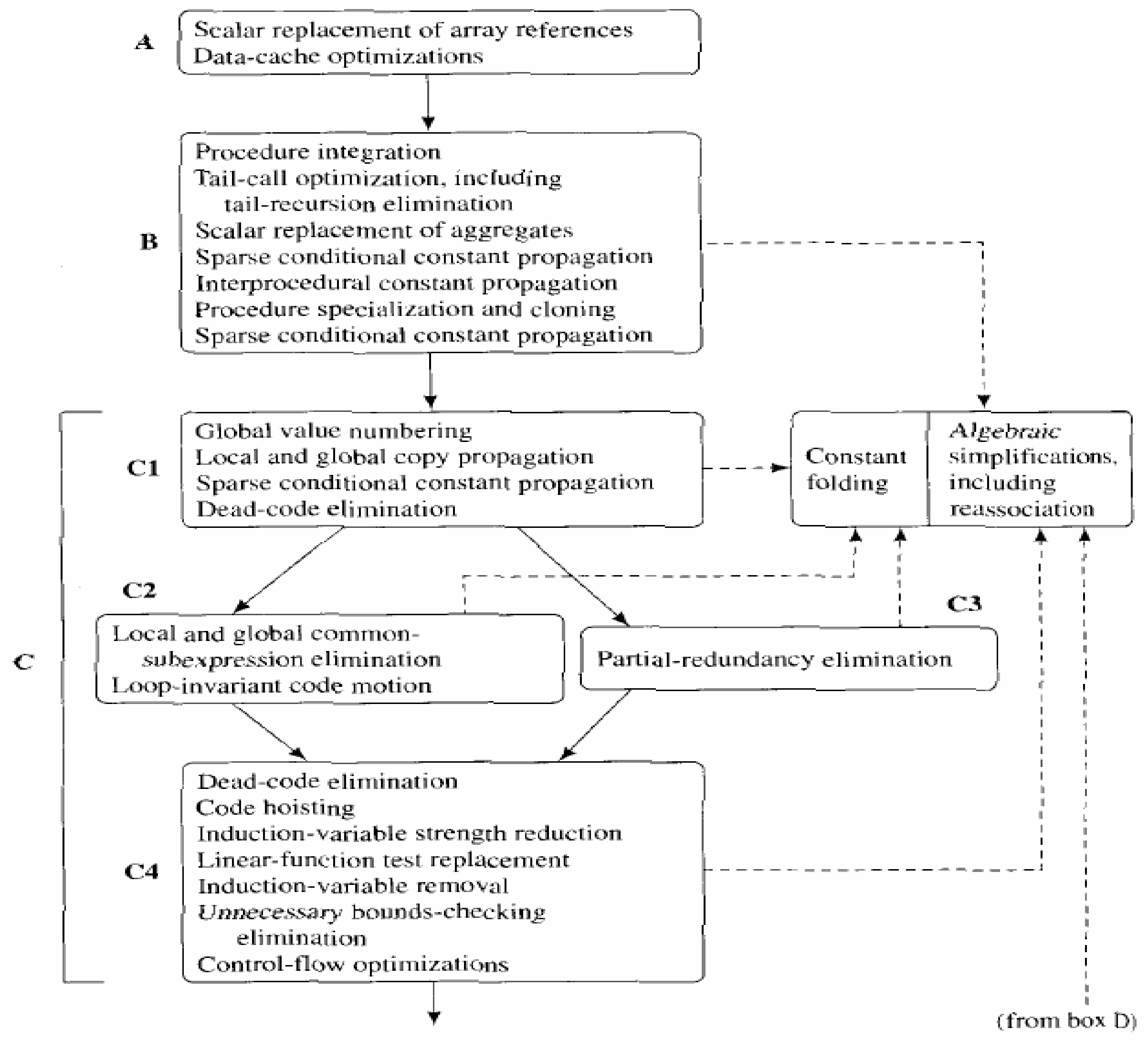
A possible order of optimizations, shown in the figure below, is from S. Muchnick's book "Advanced compiler design implementation".

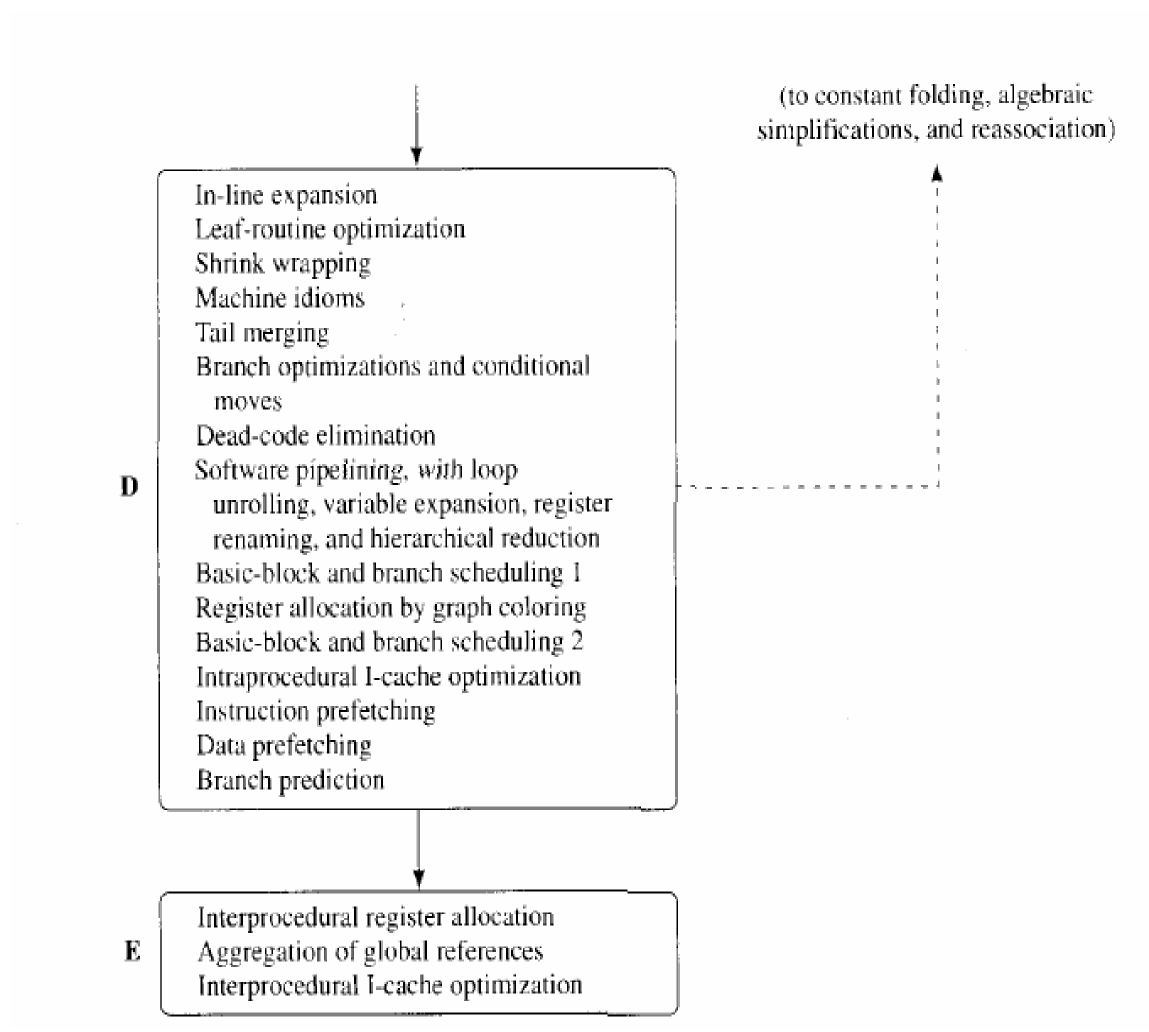
Two quotes from that book:

"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 illustrations of the limitations of compilers. They are part of the reason why manual optimizations are needed.


A
Scalar replacement of array references
Data-cache optimizations
B
Procedure integration
Tail-call optimization, including tail-recursion elimination
Scalar replacement of aggregates
Sparse conditional constant propagation
Interprocedural constant propagation
Procedure specialization and cloning
Sparse conditional constant propagation
C
C1
Global value numbering
Local and global copy propagation
Sparse conditional constant propagation
Dead-code elimination
Constant folding
Algebraic simplifications, including reassociation
C2
Local and global common-subexpression elimination
Loop-invariant code motion
C3
Partial-redundancy elimination
C4
Dead-code elimination
Code hoisting
Induction-variable strength reduction
Linear-function test replacement
Induction-variable removal
Unnecessary bounds-checking elimination
Control-flow optimizations
(from box D)


(to constant folding, algebraic simplifications, and reassociation)
D
In-line expansion
Leaf-routine optimization
Shrink wrapping
Machine idioms
Tail merging
Branch optimizations and conditional moves
Dead-code elimination
Software pipelining, with loop unrolling, variable expansion, register renaming, and hierarchical reduction
Basic-block and branch scheduling 1
Register allocation by graph coloring
Basic-block and branch scheduling 2
Intraprocedural I-cache optimization
Instruction prefetching
Data prefetching
Branch prediction
E
Interprocedural register allocation
Aggregation of global references
Interprocedural I-cache optimization

Assignment statement optimizations

Constant folding

Scalar replacement of aggregates

Algebraic simplification and Reassociation

Common subexpression elimination

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.
- Interprocedural constant propagation is particularly important when procedures or macros are passed constant parameters.
- Although it is apparently a relatively simple transformation, compilers do not do a perfect job at recognizing all constant expressions as can be seen in the next three examples from the Sparc Fortran compiler.
- In fact, constant propagation is undecidable.

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)
```

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)
```

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 to facilitate other optimizations such as register allocation, constant and copy propagation.

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

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

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

- 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

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.

For integers:

Expression simplification such as

```
i+0 -> i
i ^ 2  -> i * i
i*5 -> 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.

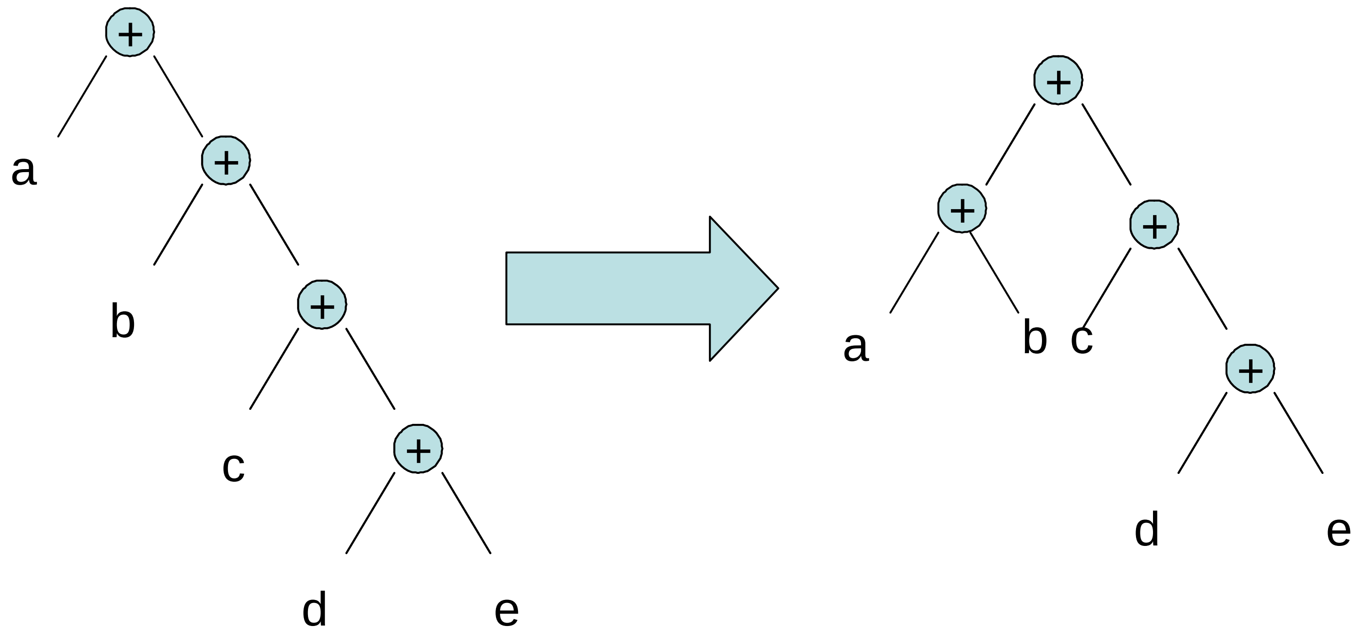
For example, in the code

```
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
<statement sequence>
y= e^3
```

- is replaced (assuming that `e` is not reassigned in `<statement sequence>`) with

```
t=e^3
x = t+1
<statement sequence>
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
<statement sequence>
t = x + 1
```

Is replaced (assuming that neither `x` nor `y` are reassigned in `<statement sequence>`) with

```
<statement sequence>
t = y + 1
```

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

```
x = a+b
<statement sequence>
z=x
y = a+b
```

Is replaced by CSE into the following code

```
t = a+b
x = t
<statement sequence>
z = x
y = t
```

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

Loop body optimizations

Loop invariant code motion

Induction variable detection

Strength reduction

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.

```
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 below).
- Elimination of redundant counters.

```
do  i=1,n
   j=j+2
   a(j)=
end
```

$\longrightarrow$

```
do  i=1,n
   a(j+2*i)=
end
```

- Elimination of interactions between iterations to enable parallelization.
  - -- 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 only multiplication to addition is the only optimization performed.

Candidates for strength reduction include:

1. Multiplication by a constant

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

after strength reduction

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

after loop invariant removal

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

2. 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
```

3. Trigonometric functions

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

After strength reduction

```
   loop
      ...
      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.

Procedure optimizations

Tail recursion elimination

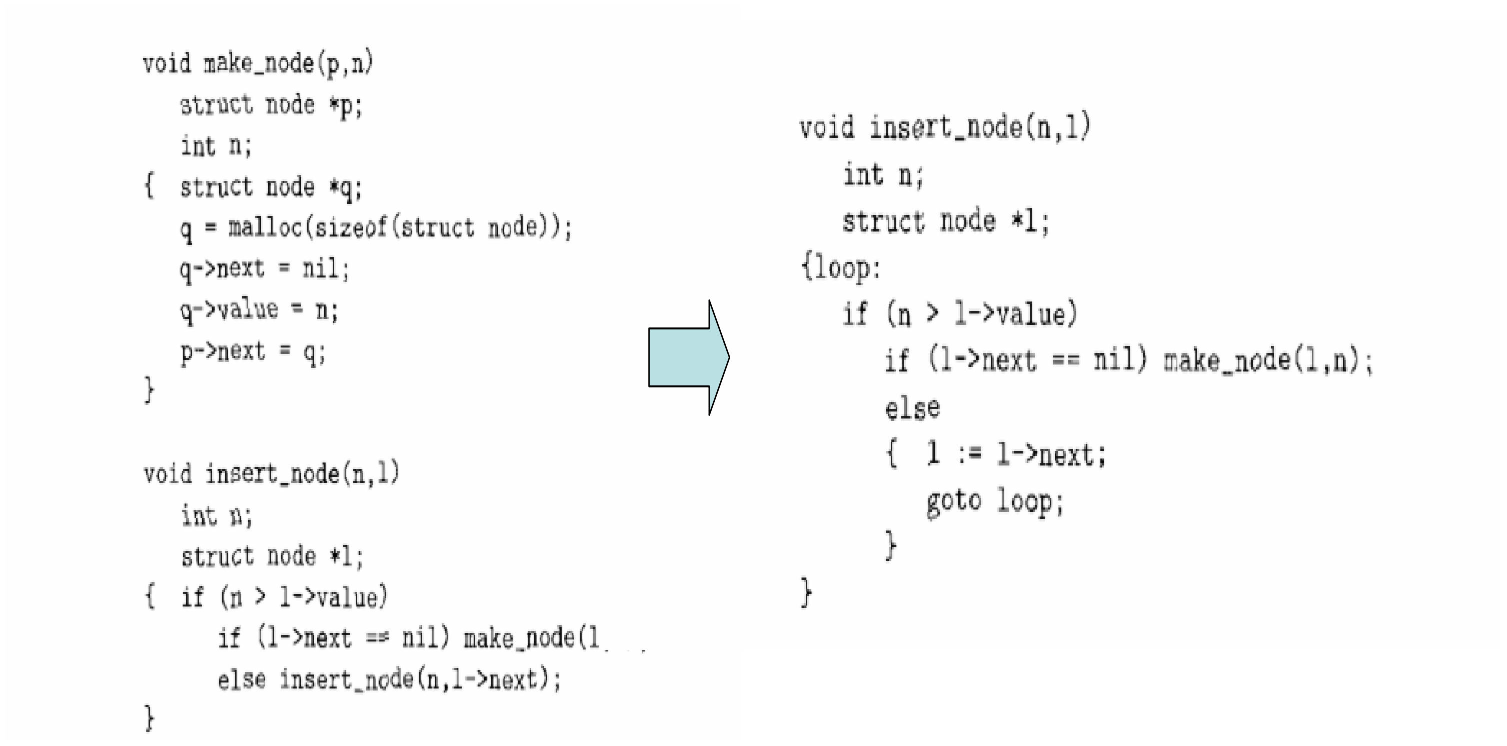
Procedure integration

Leaf routine optimization

Tail recursion elimination

Tail recursion elimination

- Converts tail recursive procedures into iterative form



```
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

- Expands 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

- Leaf routines tend to be the majority (leafs of binary trees are one more than the number of interior nodes).
- Save instructions that prepare for further calls (set up the stack/display registers, save/restore registers)

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.

Instruction scheduling

- Objective is to minimize execution time by reordering executions.
- Scheduling is an NP-complete problem.

	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.

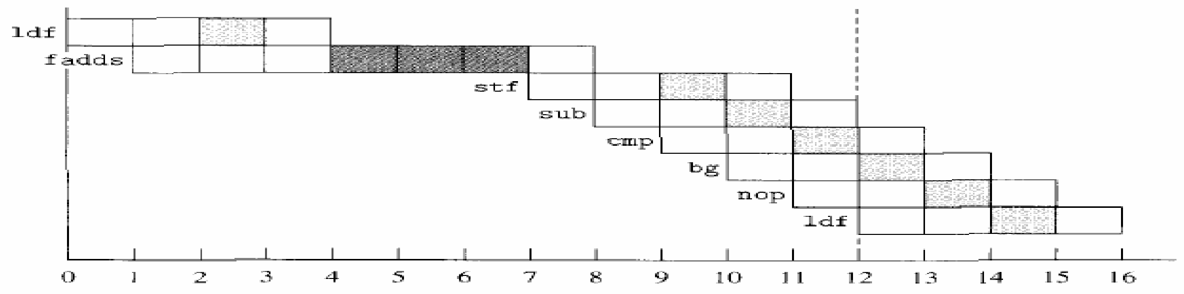


FIG. 17.16 Pipeline diagram for the loop body in Figure 17.15.

Control-flow optimizations

Unreachable code elimination

- Unreachable code is code that cannot be executed regardless of the input.
- Eliminating it save space

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.

If simplification

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

Loop inversion

- Transforms a while loop into a repeat loop.
- Fewer loop bookkeeping operations are needed

Unswitching

- Moves loop-invariant conditional branches out of loops

Dead code elimination

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

Stripmining

Transforms a loop into a doubly nested loop.

```
for (i=1;i<=100,i++)
   {...}
```

$\downarrow$

```
for (K=1;K<=100;k+=20)
   for (i=K;i<=K+19;i++)
      {...}
```

Loop fusion

Joins two loops into a single one.

Loop fission

Breaks a loop into two loops

```
for (i=1;i<=n;i++) {α; β; χ; δ; ε}
```

$$\downarrow$$

```
for (i=1;i<=n;i++) {α; β; χ}
for (i=1;i<=n;i++) {δ; ε}
```

Loop interchange

Changes the order of loop headers

```
for (i=1;i<=n;i++)
   for (j=1;j<=m;j++) {...}
```

$$\downarrow$$

```
for (j=1;j<=m;j++)
   for (i=1;i<=n;i++) {...}
```

Loop tiling

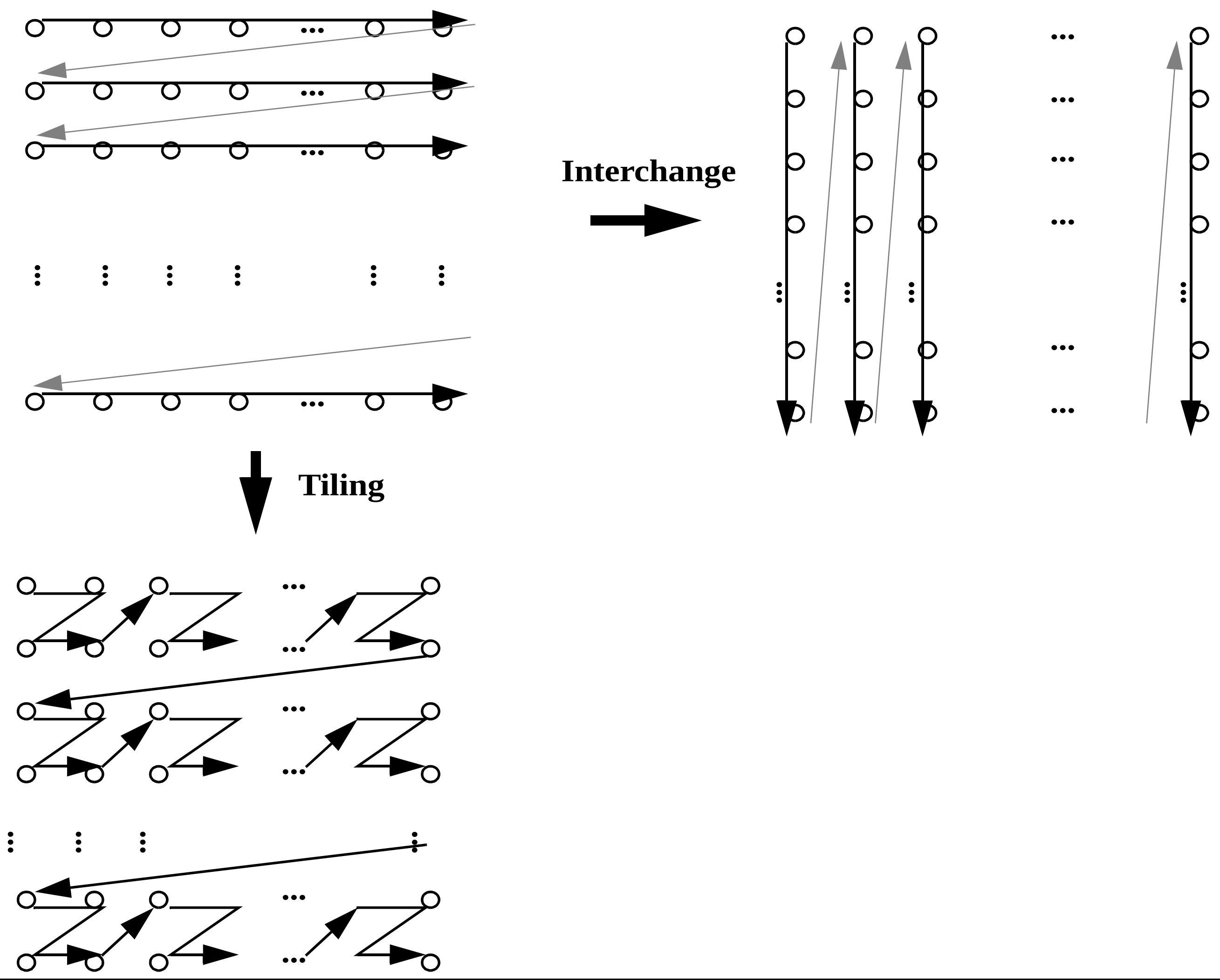
This is a combination of strip mining followed by interchange that changes traversal order of a multiply nested loop so that the iteration space is traversed on a tile-by tile basis.

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        c[i] = a[i,j]*b[i];
```

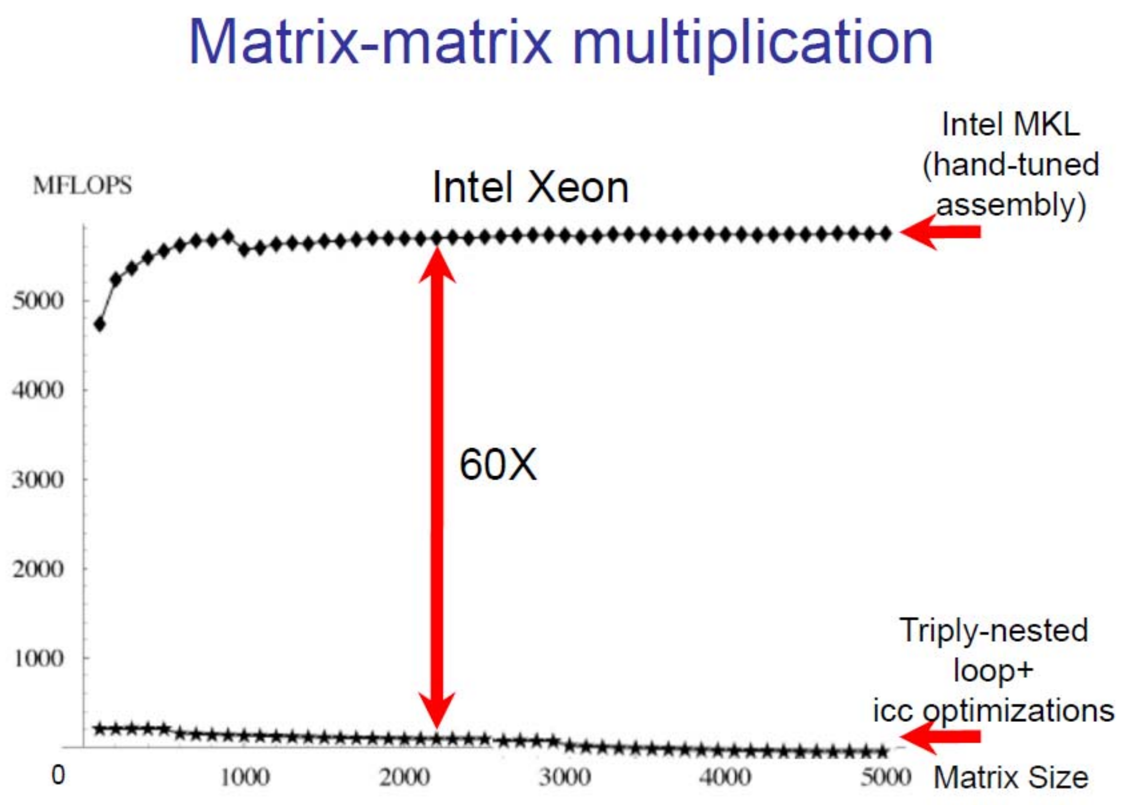
$$\downarrow$$

```
for (i=0; i<N; i+=2)
    for (j=0; j<N; j+=2)
        for (ii=i; ii<min(i+2,N); ii++)
           for (jj=j;jj<min(j+2,N); jj++)
                c[ii] = a[ii,jj]*b[ii];
```

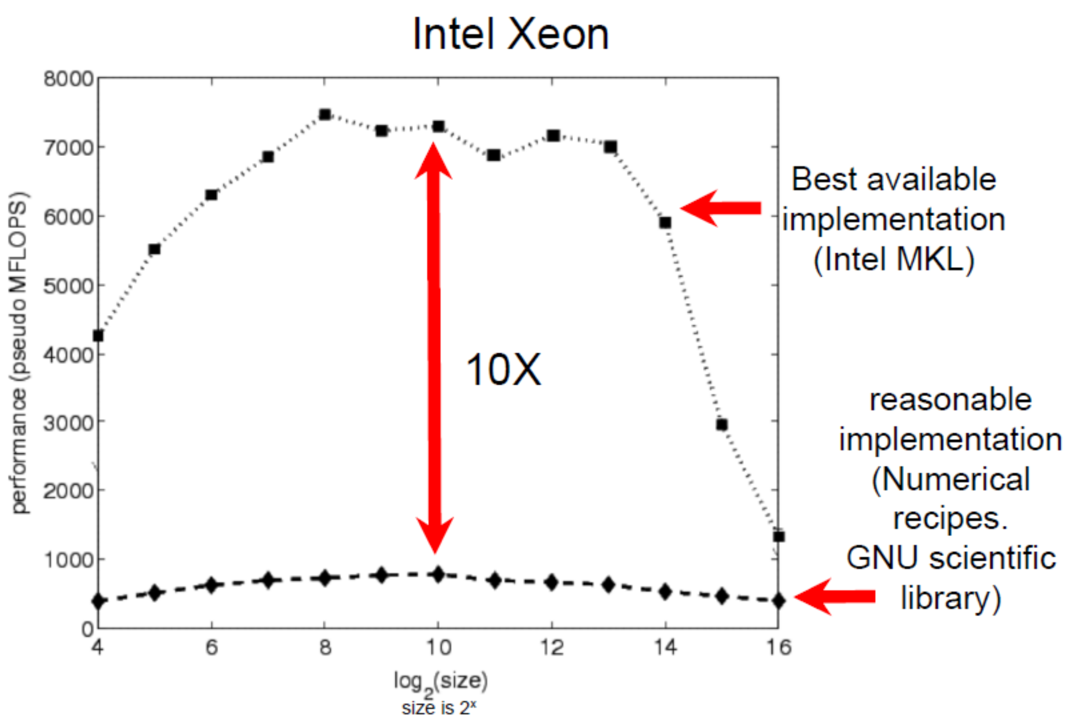
Iteration space and loop transformations



Compilers vs Manual Programming



FFT


Intel Xeon
8000
7000
6000
5000
4000
3000
2000
1000
0
performance (pseudo MFLOPS)
4
6
8
10
12
14
16
log2(size)
size is 2^x
10X
Best available implementation (Intel MKL)
reasonable implementation (Numerical recipes. GNU scientific library)