Performance Programming: Theory, Practice and Case Studies

Module II:
Optimizing Serial Programs

Part 2 - Source Code

Outline

- **Overview of Memory Hierarchy**
  - Memory Levels & Memory Organization of RISC based systems
- **Memory Hierarchy Optimizations**
  - Cache Blocking
  - Reducing Cache Conflicts
  - Reducing TLB Misses
- **Optimal Data Alignment**
- **Aliasing Optimizations**
- **Common Loop Optimization Techniques**
  - Unrolling and tiling; Loop interchange; Loop fusion, fission and peeling
  - Loops with conditionals; Strength reduction in loops
- **Summary**

Memory Levels

- Gap between memory speed and processor speed is increasing [eg. increased from 2 (US-I) to 5 (US-III)]
- Levels (fastest to slowest):
  - Registers: On-chip
  - On-chip caches
  - Off-chip caches (possibly multiple levels)
  - Physical memory
  - Virtual memory (physical memory and/or disk)
  - Magnetic disks
  - Other storage media (tape, CDROM drives etc.)

Memory Levels (contd.)

- Caches: Fast memory storage
  - Harvard caches: Data+Program in separate caches
  - Unified caches: Data+Program in same cache
  - Direct mapped caches: Addresses map to unique locations in cache
  - Set-associative and/or Fully-associative caches: Addresses map at multiple of any location in the cache
  - Write Through cache: Data updated in cache written into memory also
  - Write Back cache: Data updated in cache written only when replaced

Memory Levels (contd.)

- Physical and Virtual memory
  - Organized in pages (For e.g. default pagesize is 8KB in Solaris 9 on UltraSPARC systems)
  - Virtual-to-Physical address translations stored in Translation Lookaside Buffer (TLB)
- Memory performance benchmarks
  - Lmbench (to measure latency)
  - STREAM (to measure bandwidth)
  - Other benchmarks

Memory Levels (contd.)

- Typical memory access times (in CPU clock cycles)

Memory Level	Size	Access Time
CPU registers	1KB	Order of 1 cycle
On-chip cache	16KB – 512KB	Order of 3-4 cycles
Off-chip cache	256KB – 8MB	Order of 10 cycles
RAM memory	64MB – 512 GB	Order of 100 cycles
Magnetic Disk	Terabytes	Order of million cycles

Characteristics of Some Processors

- Processor pipeline features

Processor	*Number of Pipeline Stages*	*Intruction Issue Rate*	*Peak FP Issue Rate*	*No. Registers Int/FP (others)*
UltraSPARC-II	**10**	**4**	**2**	**32/32**
UltraSPARC-IIIcu	**14**	**4**	**2**	**32/32**
Intel Xeon	**12**	**3**	**1**	**8/8 (40 rename)**
Alpha 21264	**7**	**4**	**4**	**32/32**
Itanium2	**8**	**6**	**4**	**128/128**
Power-4	**14**	**5**	**4**	

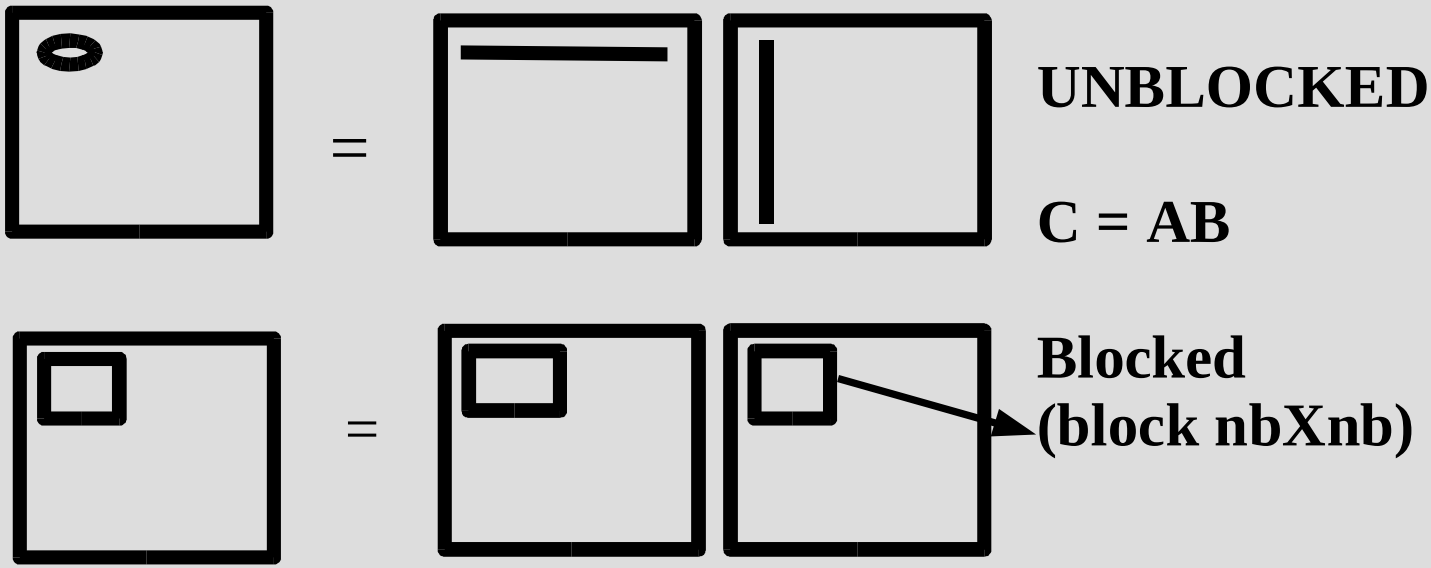
Characteristics of some Processors (contd.)

- Processor Cache features

System	L1-I cache size/line/ Associativity	L1-D cache size/line/ Associativity	L2 cache size/line/ Associativity	L3 cache size/line/ Associativity
UltraSPARC-II	16KB/32B/2	16KB/32B/1	8MB/64B/1	
UltraSPARC-III Cu	32KB/32B/4	64KB/32B/4 P$ 2KB/64B/4 W$ 2KB/64B/4	8MB/512B/2	
Intel Xeon	16KB/32B/4	16KB/32B/2	1MB/32B/4	
Alpha 21264	64KB/64B/2	64KB/64B/2	4MB/64B/1	
Itanium-2	16KB/64B/4	16KB/64B/4	256KB/128B/8	3MB/128B/12
Power-4	64KB/128B/1	32KB/128B/2	1.44MB/128B/8	128MB/512B/8 (on a MCM)

Cache Blocking

- Memory reference optimization that improves temporal and spatial locality of memory references
  - Decreases cache miss rate by increasing reuse of data
  - Computations with high ratio of computational to memory operations benefit most [for e.g. BLAS 3 operations have 0(N) ratio]
  - Canonical matrix multiplication example



Cache Blocking (contd.)

- Unblocked

```
for (i=0;i<n1;i++) {
  for(j=0;j<n3;j++) {
     sum = 0.0;
#pragma pipeloop(0)
     for (k=0;k<n2;k++) {
        sum = sum +
        a[i][k]*b[j][k];
     }
     c[i][j] = sum;
  }
}
```

- Blocked

```
for (ii=0;ii<n1;ii+=na) {
 for (jj=0;jj<n3;jj+=nb) {
  for (i=ii;i<min((ii+na),n1)
                          ;i++) {
    for (j=jj;j<min((jj+nb),n3)
                           ;j++) {
        sum = 0.0;
#pragma pipeloop(0)
        for (k=0;k<n2;k++) {
            sum = sum +
            a[i][k]*b[j][k];
        }
        c[i][j] = sum;
    }
   }
  }
}
```

Cache Blocking (contd.)

- **Results of C = AB^T on Ultra-60 (450 MHz UltraSPARC-II, 4 MB L2-cache)**

Size	*Unblocked L2 Cache Hit rate*	*Unblocked Mflops*	*Blocking Factor (nb)*	*Blocked L2 Cache Hit rate*	*Blocked Mflops*
480	*0.982*	*230*	*32*	*0.998*	*311*
640	*0.932*	*125*	*32*	*0.996*	*297*
800	*0.896*	*92*	*64*	*0.996*	*290*
960	*0.864*	*71*	*80*	*0.996*	*286*
1120	*0.849*	*63*	*80*	*0.996*	*282*

Cache Blocking (contd.)

- For matrix multiplication, use Vendor provided math library (eg. Intel Math Kernel Library, Sun Performance Library) implementation as numerous other factors impact performance (TLB sizes, prefetching, load/store flow in pipeline, modulo scheduling of inner loops etc.)
- Interpret performance metrics in context: a high cache hit-rate is not necessarily indicative of good performance (bottleneck may shift elsewhere and overall runtime may still remain poor).

Reducing Cache Conflicts

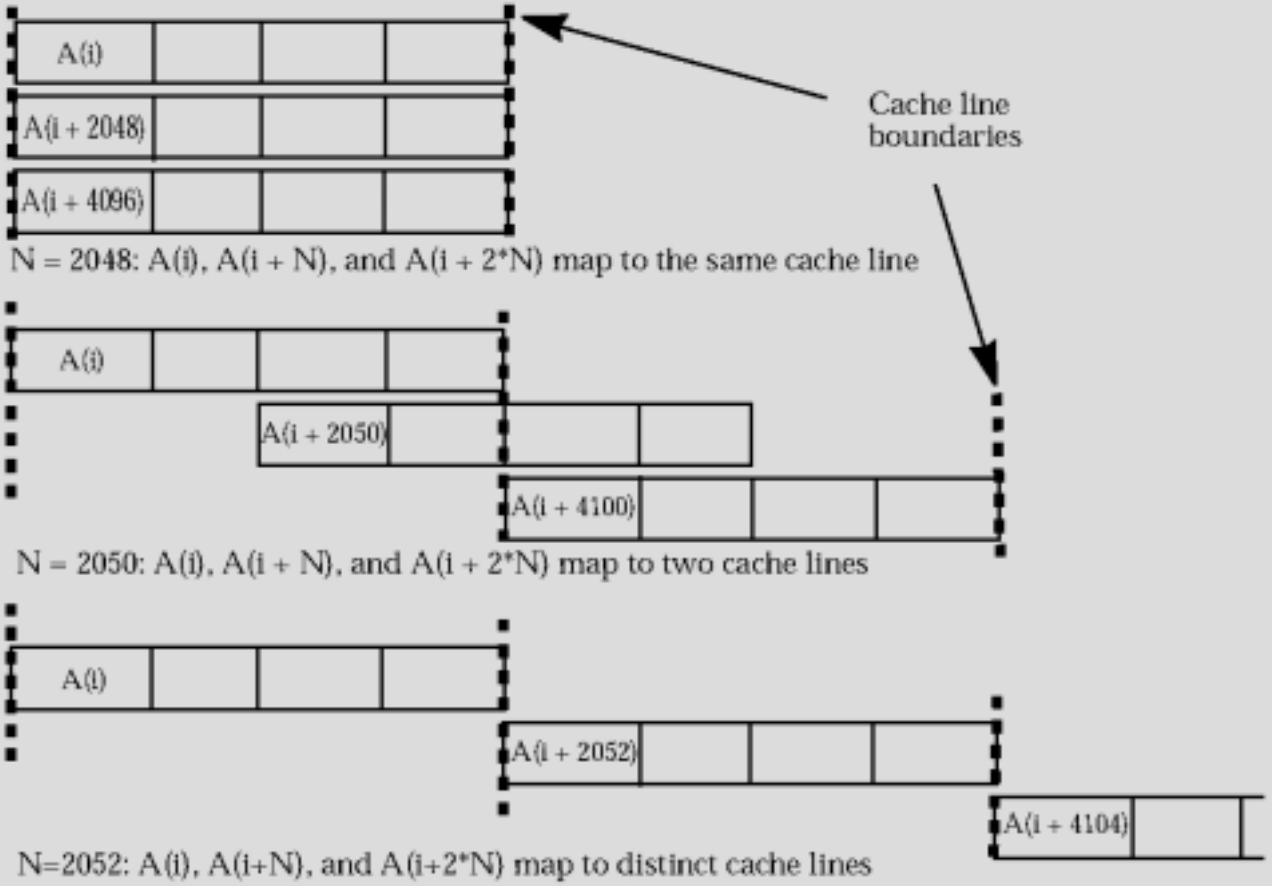
- Direct mapped caches: low cache-hit times but higher conflict and collision misses
- Conflict misses occur when multiple data items compete for same cache locations in non fully associative caches (e.g. power of 2 accesses as in FFT's, Strassen Matrix Multiplication etc.)
- Various approaches to reduce cache conflicts:
  - Compiler option in Fortran to pad local and common block variables to avoid powers of 2 (e.g. -xpad in Sun Fortran compiler)
  - Page coloring: affects mapping of free physical pages mapped to faulted virtual pages (e.g. reduces conflicts in level 2 cache on UltraSPARC-II, III)

Reducing Cache Conflicts (contd.)

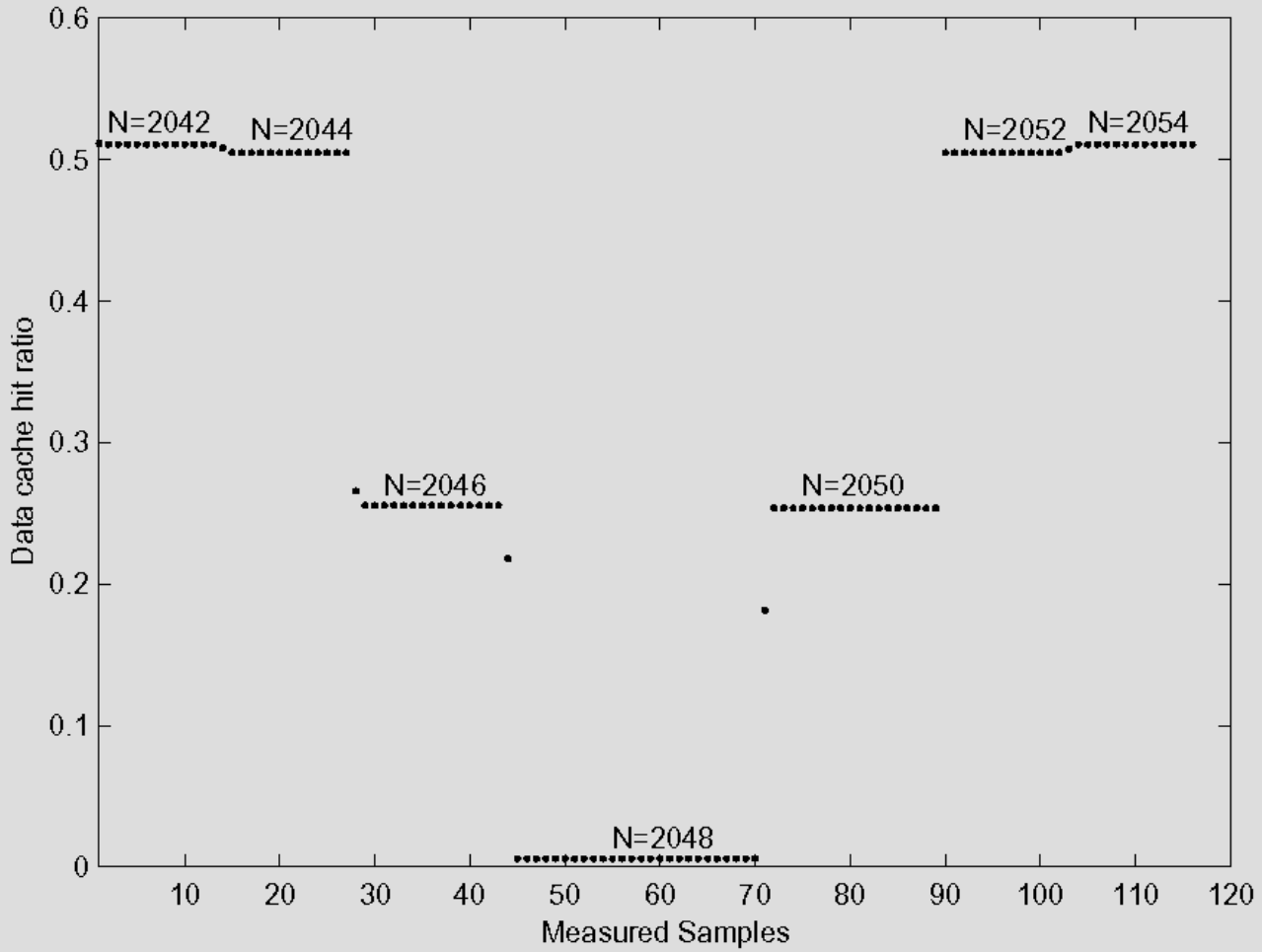
- Example: performance penalty of conflicts in Level-1 D-cache on UltraSPARC-II processor

  A(2*N+i)=A(i)+A(N+i)

  - For N=2048: pathological cache conflicts occur



Reducing Cache Conflicts (contd.)



- Data cache hits are close to zero for array stride of N=2048. The hit ratio improves once the stride doesn't cause cache conflicts
- The runtimes also are also worst with N=2048

Reducing TLB Misses

- TLB speeds up address translation by keeping translated addresses in buffer
- TLB misses are costly as trap into kernel is required to complete the translation
- No general recipe to avoid TLB misses except general guideline of designing application to operate on localized data
- Use multiple page-size support in the OS. Multiple page sizes supported on nearly all architectures and many OS's provide the support for applications to utilize pages of different sizes for program text, heap and stack space. E.g. HPUX, SGI IRIX, IBM AIX, Solaris 9

Reducing TLB Misses (contd.)

- Solaris 9

```
ppgsz -o heap=4M,stack=64k a.out
```

  - IRIX 6.5

```
dplace -data_pagesize 64k  \
       -stack_pagesize 64k a.out
```

  - AIX

```
LDR_CNTRL=LARGE_PAGE_DATA=Y envar, vmtune
command,
```

Optimal Data Alignment

- Maintaining preferred alignment restrictions for different data-types important for performance
  - Restructuring for better data alignment
  - Cache line Alignment

- Restructuring for better data alignment
  - Data should be placed on preferred alignment boundaries
  - C: structs, globals and static variables should be ordered from largest to smallest data-type
  - Fortran: In COMMON blocks variables should be ordered from largest to smallest data-type

```
integer*1 a,zs(10)
real*4 ys(21)
real*8 x(len),y(len),z(len)
common /blk1/a,x,zs,y,ys,z ! IMPROPERLY ALIGNED
common /blk1/x,y,z,ys,zs,a ! PREFERRED: PROPERLY ALIGNED
```

Optimal Data Alignment (contd.)

- Dynamically allocated data-items and pointer manipulation:
  - Care should be taken to keep the data properly aligned

```
char *x; double *y;
x = (char *) malloc(10*sizeof(char));
y = (double *) (x+2); /* y is misaligned */
```

  - `malloc` usually returns data aligned on at least 8-byte boundary (e.g. In Solaris `malloc` returns 8-byte aligned data in 32-bit and 16-byte aligned data in 64-bit)
  - Finer control on alignment can be obtained through `memalign, valloc` or by manually aligning the pointer via use of offset

```
int *a, *b, len; unsigned long temp;
a = (int *) malloc(len*sizeof(int)+32);
temp = ((unsigned long) a + 32) & 0xffffffe0;
b = (int *) temp;
    /* b is aligned on 32-byte boundary */
```

Aliasing Optimizations

- **Effects of aliasing on correctness and performance**
  - Aliasing in Fortran Programs
  - Aliasing in C Programs

Aliasing Optimizations (contd.)

- Aliasing in Fortran Programs
  - Language standard does not have restriction on locations of variables in memory
  - Compilers can optimize code under assumption that variables are independent and stored in nonoverlapping portions of memory
  - Problems can occur with `EQUIVALENCE` and `COMMON` variables
  - Problems can also occur if same variable is passed to the subroutine as different arguments (inducing memory aliasing when compiler might assume none exists)

Aliasing Optimizations (contd.)

- Example Program:

```
c example_alias.f
c f77 -xO1 example_alias.f -o example_alias
c f77 -xO3 example_alias.f -o example_alias: wrong results
c
integer n, isum
   do n=1,10
      call foo(isum,isum,n)
      print*,' n = ', n, ' isum = ', isum
   enddo
end
c
subroutine foo(i,isum,n)
integer n, isum, i
   do i=1,n-1
      isum = isum+i
   enddo
   isum=(isum+1)/2
return
end
```

  - Unconventional way to compute powers of 2: program abuses argument passing rules
  - At `-xO3` level in Sun Fortran compiler, the optimizer unrolls the loop with the assumption that i, isum are distinct leading to incorrect results

Aliasing Optimizations (contd.)

- Aliasing in C programs
  - In general a pointer can alias any other pointer reference or global variable. It can also be an alias to a local variable whose address is accessed via & operator
  - Aliasing can occur regardless of data-type except if program conforms to ANSI/ISO rules: pointers and variables of different basic data-types do not alias
  - Incorrect code can be generated if program violates ANSI/ISO aliasing rules but compiled assuming conformance (check program with lint)

Aliasing Optimizations (contd.)

- Aliasing in C programs

```
#include <stdlib.h>
int *p;
double *q;
void foo();
void foo() {
   int i;
   p = (int *) malloc(sizeof(int)*10);
   q = p; /* not allowed by ANSI C standard */
   for (i=0;i<5;i++)
       q[i] = i;
   }
   ....
```

Aliasing Optimizations (contd.)

- **Unmodified**

```
int key, *array;
void binsearch(int n,
               int *loc)
{
 int a=0, b=n;
 while (b-a > 8) {
   *loc = (a+b)>>1;
   if (array[*loc] > key)
       b = *loc;
   else
       a = *loc;
 }
 for (*loc=a; *loc<b;
           (*loc)++) {
   if (array[*loc] == key)
   break;
 }
}
```

- **Modified**

```
int key, *array;
void binsearchmod(int n,
                  int *loc)
{
 int a=0, b=n, c=*loc;
 while (b-a > 8) {
     c = (a+b)>>1;
 if (array[c] > key)
     b = c;
 else
     a = c;
 }
 for (c=a; c<b; c++) {
   if (array[c] == key){
      *loc = c;
      break;
   }
 }
}
```

Aliasing Optimizations (contd.)

- Aliasing in C programs
  - Minor code changes help improve performance in two situations: (a) possibility of aliasing in global variables/-pointers, local variables/pointers and function-call arguments (b) using pointer variable as loop index

Loop Optimization

- Loops are one of most commonly used constructs in HPC programs
- Variety of optimization techniques have been developed (many will be discussed here)
- Compiler performs many of loop optimization techniques automatically but in some cases source code modifications enhance optimizer's analysis
- Loop optimization related source code modifications should always be re-evaluated with the availability of newer compilers and architecture (as may or may not be required any longer. Can even hurt performance)

Loop Unrolling and Tiling

Untiled Loop Nest

```
do j=1,m
  do i=1,n
   do k=1,p
     c(i,j)=c(i,j) +
            a(k,i)*b(k,j)
   enddo
  enddo
enddo
```

Floating Pt. Ops ~ 2mnp
Memory Ops ~ 2mnp + 2mn

2x2 Tiled Loop Nest

```
do j=1,m,2
     do i=1,n,2
           f11=c(i,j)
           f21=c(i+1,j)
           f12=c(i,j+1)
           f22=c(i+1,j+1)
        do k=1,p
           f11=f11 + a(k,i) * b(k,j)
           f21=f21 + a(k,i+1) * b(k,j)
           f12=f12 + a(k,i) * b(k,j+1)
           f22=f22 + a(k,i+1) * b(k,j+1)
         enddo
           c(i,j) =f11
           c(i+1,j) =f21
           c(i,j+1) =f12
           c(i+1,j+1)=f22
    enddo
enddo
```

Floating Pt. Ops ~ 2mnp
Memory Ops ~ 2mnp + 2mn

- *Register Blocking*: Data partitioned to fit in on-chip registers (similar to *Cache Blocking*)
- *Loop Tiling*: technique of register blocking applied to loop nests

Loop Tiling & Unrolling (contd.)

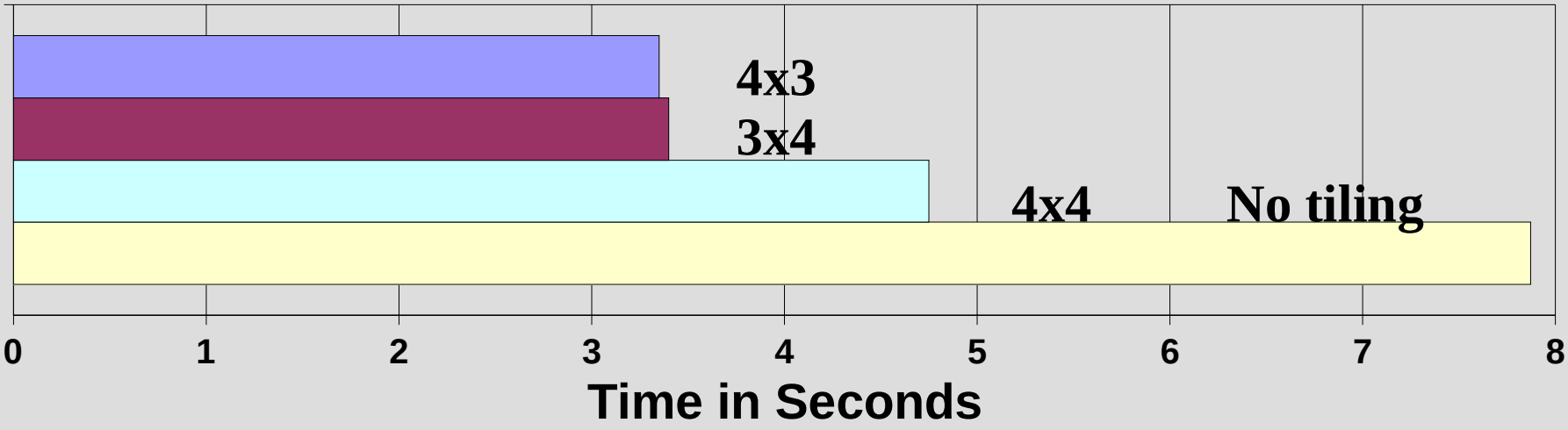
- General Guidelines for loop tiling:
  - In general, Manual unrolling of loops is discouraged. Compiler performs loop unrolling very efficiently. Unrolling of loops should only be done for complex loop-nests
  - The innermost loop in a loop-nest should not be unrolled. The compiler unrolls and software pipelines the innermost loop
  - Loop tiling should be used with cache-blocking as that increases effectiveness of software pipelining of innermost loop
  - Tiling should be applied to outer loops: it increases the spatial and temporal locality of computation

Loop Unrolling & Tiling (contd.)

- Tile-size determination:
  - A difficult theoretical problem and usually heuristics applied
  - Tile-size depends on: latency to cache, latency of floating point operations, ratio of flops to mem-ops, number of available registers on processor.
  - Tile should be selected to maximize ratio of flops to mem-ops without causing the compiler to generate *register spills*

Loop Unrolling & Tiling (contd.)

- Times on Compaq Alphaserver (500MHz)
  - 4x3 tiling performs best in above mat-mul example: flops/mem-ops=12/7 (12 adds + 12 muls, 7 loads)
  - Recommend to check if a particular tiling optimal when new compilers and chip become available change)



Loop Interchange

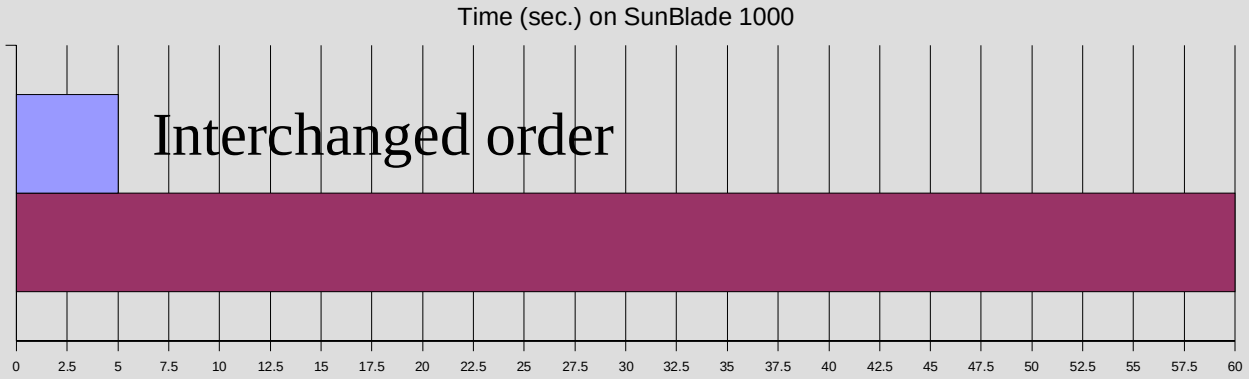
- Improves spatial locality and maximizes use of data brought into cache
- Loops are reordered to minimize stride and align access pattern in loop with pattern of data-storage in memory

No Interchange

```
do i=1,nele
   do j=1,nele
      y(i,j) = 2.0*x(i,j)
   enddo
enddo
```

Loop Interchange

```
do j=1,nele
   do i=1,nele
      y(i,j) = 2.0*x(i,j)
   enddo
enddo
```



Loop Fusion

- Adjacent or closely spaced loops fused together:
  - Decreased loop overhead & increase in computational density enables improvement in software pipelining
  - Increase in cache-locality of data structures
  - Compilers performs fusion but in some cases this optimization not performed and required to be done manually

- Fused loop below takes advantage of temporal locality and reuse in loaded values of `a[i]` and `c[i]`

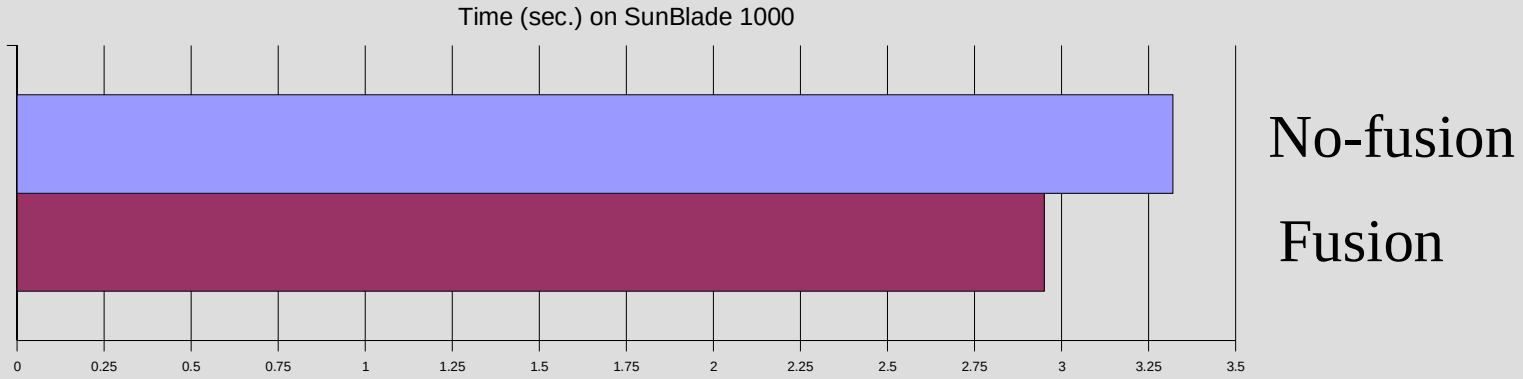
No Loop Fusion

```
for (i=0;i<nodes;i++) {
    A[i] = a[i]*small;
    C[i] = (a[i] + b[i])*relaxn;
}
for (i=1;i<nodes-1;i++) {
    D[i] = c[i] - a[i];
}
```

Loop Fusion

```
a[0] = a[0]*small;
c[0] = (a[0]+b[0])*relaxn;
a[nodes-1] = a[nodes-1]*small;
c[nodes-1] = c[nodes-1]*relaxn;
for (i=1;i<nodes-1;i++) {
     A[i] = a[i]*small;
     C[i] = (a[i] + b[i])*relaxn;
     D[i] = c[i] - a[i];
}
```

Loop Fusion (contd.)



- Should not be applied indiscriminately. If the loop becomes computationally “fat”, performance might degrade (as soft-ware pipelining efficiency decreases)

Loop Fission

- Split the loop into multiple loops:
  - Ensure that trip count is sufficiently large for index overhead to remain small
  - Can be used when loop has conditional: split into conditional-free and conditional-containing loops
  - Computationally “fat”: splitting may decrease register pressure
- Pitfall
  - Compiler may fuse split loops back (check assembly listing). Can peel first or last loop iteration (discussed next) or insert a call to dummy function

Loop Fission (contd.)

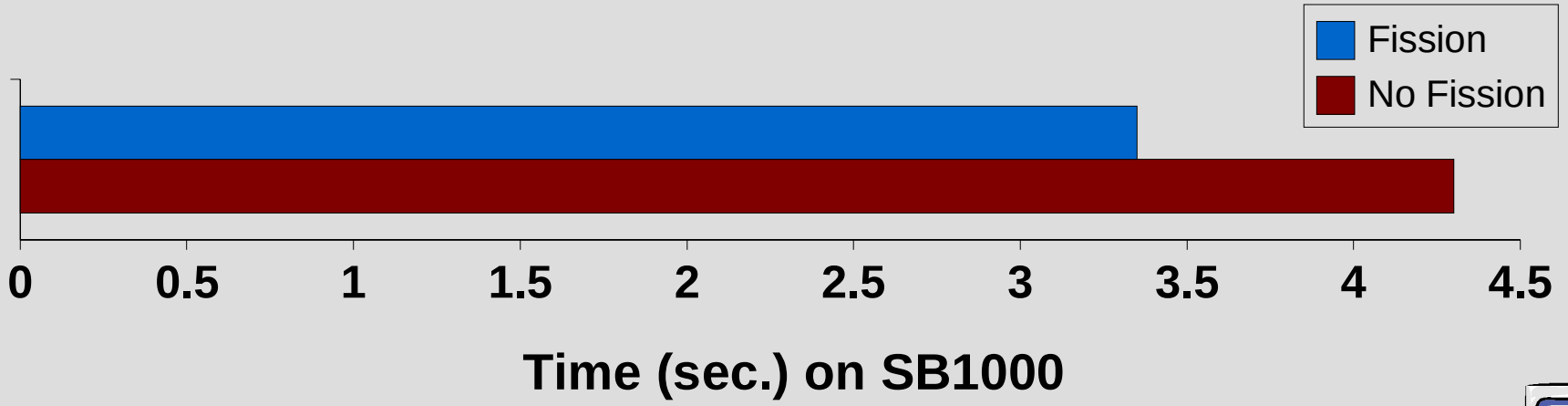
- Example

No Loop Fission

```
for (i=0;i<nodes;i++) {
a[i] = a[i]*small;
dtime = a[i] + b[i];
dtime = fabs(dtime*ratinpmt);
temp1[i] = dtime*relaxn;
if(temp1[i] > hgreat) {
*ierror = 4;
break;
}
```

Loop Fission

```
#pragma pipeloop(0)
for (i=0;i<nodes;i++) {
a[i] = a[i]*small;
dtime = a[i] + b[i];
dtime = fabs(dtime*ratinpmt);
temp1[i] = dtime*relaxn;
}
for (i=0;i<nodes;i++) {
if(temp1[i] > hgreat) {
*ierror = 4;
break;
}
```



Loop Peeling

- Peeling k iterations mean removing these from loop body and placing them ahead or after the loop-body
- Usually performed by compiler but may not happen in complex cases
- Example
  - Peeled loop has one load and store (unpeeled loop has extra load). Running on Sun Blade1000 results in ~1.5X speedup

No Peeling

```
do i=1,n
  Y(i,n)=
  (1.0 - x(i,1))*y(1,n)+x(i,1)*y(n,n)
enddo
```

Loop Peeling

```
t2 = y(n,n)
x(i,1)*y(n,n)
y(1,n)= (1.0-x(1,1))*y(1,n)+x(1,1)*t2
t1 = y(1,n)
do i=2,n-1
  Y(i,n)= (1.0-x(i,1))*t1 + x(i,1)*t2
enddo
y(n,n) = (1.0-x(n,1))*t1 + x(n,1)*t2
```

Loops with Conditionals

- Modern processors have deep pipelines to obtain high ILP and achieve high clock frequencies
  - Sun UltraSPARC-I,II: 10-stage; Sun UltraSPARC-III: 14-stage
  - IBM Power4: 14-stage; Intel Pentium4: 20-stage
  - Branch-misprediction penalty is high (pipeline is stalled and restarted at branch target address)
- Manual restructuring needed to alleviate impact of branches
  - For example: loop with if statements almost certain to be not pipelined
- Conditional statements in loops:
  - loop invariant conditionals
  - loop index dependent conditionals
  - independent loop conditionals
  - dependent loop conditionals
  - reductions
  - Conditionals that transfer control

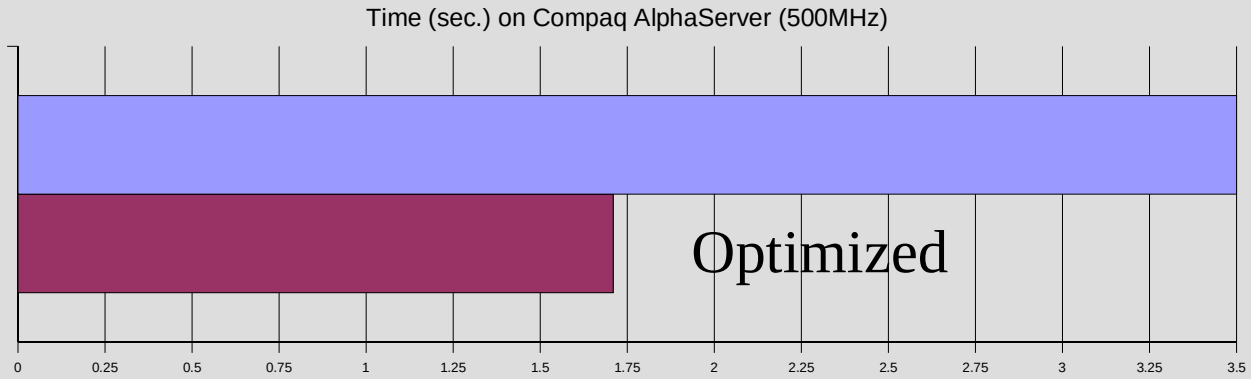
Loops with Conditionals (contd.)

Conditional

```
do i=1,len2
 Do j=1,len2
  If(j < i) then
    A2d(j,i) = a2d(j,i) + b2d(j,i)*con1
  Else
    A2d(j,i) = 1.0
  Endif
 Enddo
enddo
```

No Conditional

```
do i=1,len2
 Do j=1,i-1
   A2d(j,i) = a2d(j,i) + b2d(j,i)*con1
 Enddo
 Do j=i,len2
   A2d(j,i) = 1.0
 Enddo
enddo
```



Strength Reduction in Loops

- Strength Reduction: optimization of arithmetic expression by replacing computationally expensive operations with cheaper ones
  - When applied to loops it can magnify performance impact
  - Compiler usually performs this on simpler operations (eg. integer multiplication or division by powers of 2 replaced by shifts)
- We will consider 2 uncommon examples:
  - Division Replacement
  - Operations on complex and real operands

Strength Reduction in Loops (contd.)

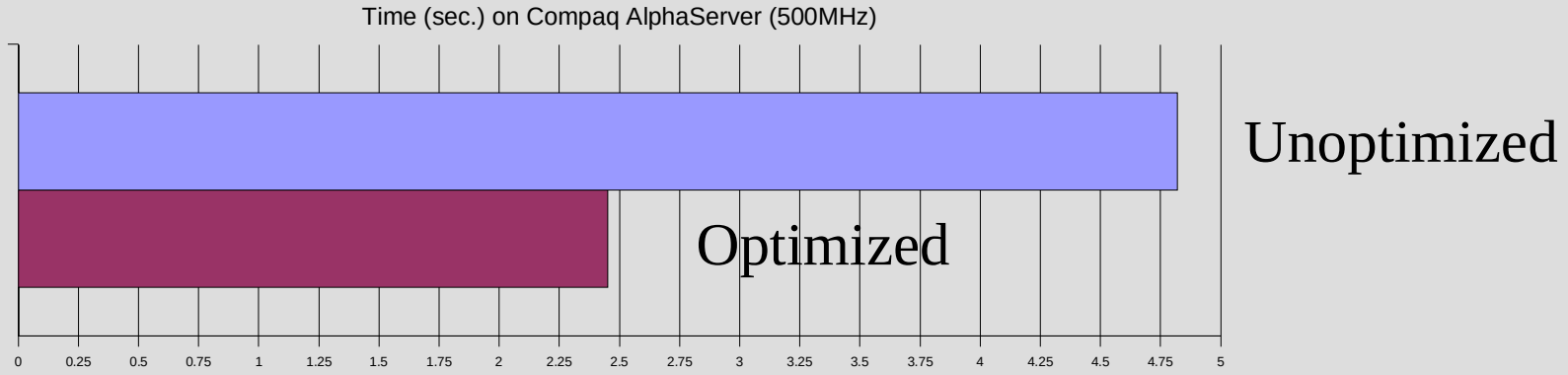
- Division Replacement

Unoptimized

```
z(i) = x(i)/y(i)
w(i) = u(i)/v(i)
```

Optimized

```
tmp = 1.0/(y(i)*v(i))
z(i) = x(i)*v(i)*tmp
w(i) = u(i)*y(i)*tmp
```

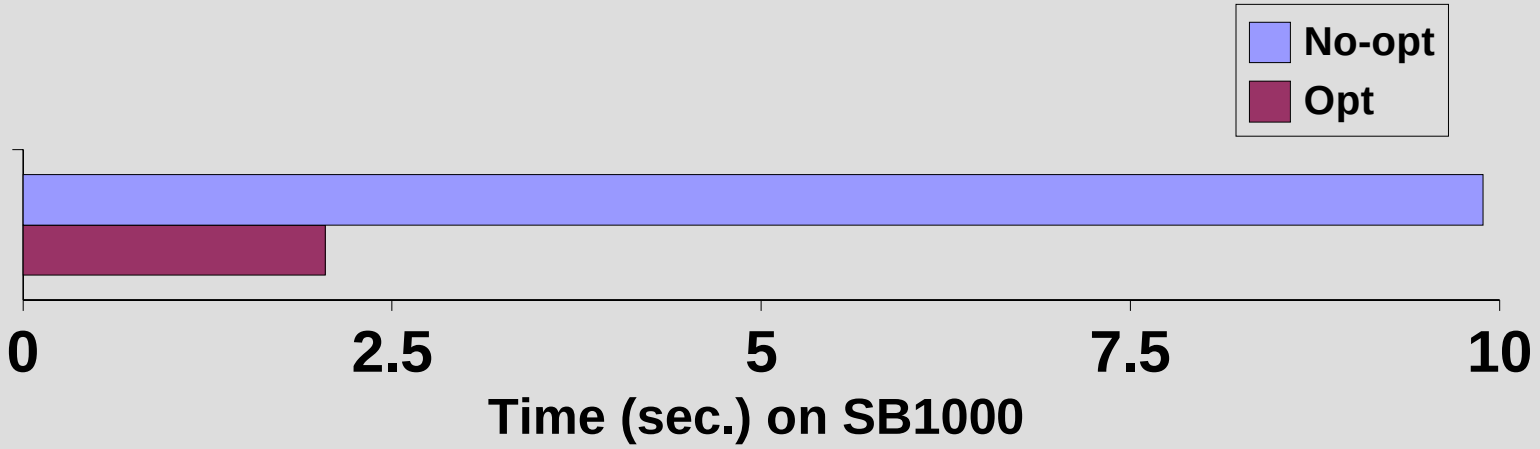


Strength Reduction in Loops (contd.)

- Operations on complex and real operands

```
Re(a*b) = Re(a)*Re(b)-Im(a)*Im(b)
Im(a*b) = Im(a)*Re(b)+Re(a)*Im(b)
Re(a/b) = (Re(a)*Re(b)+Im(a)*Im(b))/(Re(b)*Re(b)+Im(b)*Im(b))
Im(a/b) = (Im(a)*Re(b)-Re(a)*Im(b))/(Re(b)*Re(b)+Im(b)*Im(b))
```

- Can be implemented directly with real data types in numerically stable cases (for e.g. in division can give significant speedup)



Summary

- Memory Hierarchy Optimizations have the potential for most performance improvement:
  - Organize data structures and algorithms keeping cache sizes, associativities, line sizes, memory latency and cache levels in mind
  - Cache blocking increases spatial and temporal locality.
  - Avoid array strides with large powers of 2 to avoid cache mapping conflicts
  - Avoid strided access to decrease TLB miss penalties
  - Page coloring helps alleviate conflicts in direct mapped caches
  - Memory interleaving crucial to obtain good memory b/w performance
- Misaligned data can cause significant performance penalty
- Data aliasing in C and Fortran programs can also have big impact on performance and correctness

Summary (contd.)

- Loops usually account for large portion of runtime
  - Monitor compiler optimization of loops and perform manual loop restructuring as needed
  - Identify hot loops with performance analysis tools
  - Always evaluate efficiency of manually restructured loops with availability of new compilers and chips
- Common loop optimization techniques
  - Loop unrolling and tiling: improves pipelining and register usage
  - Loop fusion and fission: complementary techniques
  - Loop peeling
  - Loops with conditionals: difficult to optimize so try to separate out the conditional
  - Strength reduction optimizations applied to loops magnify performance gains