CAPS entreprise

Code Tuning for Superscalar Processors

François Bodin

INRIA

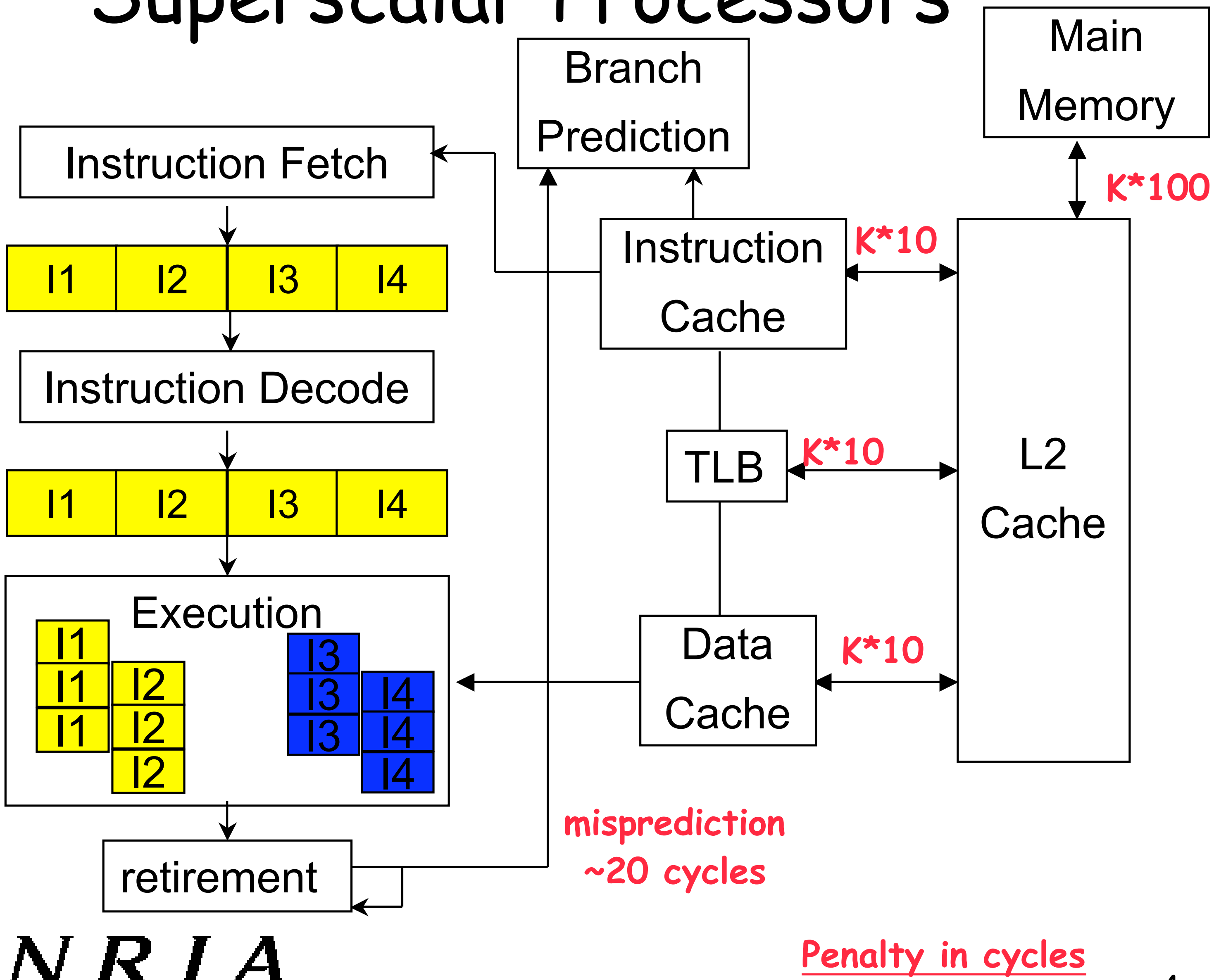
Overview

- Superscalar processors
- Code tuning
- Compilers and program transformations
- Examples of transformation

Superscalar Performance

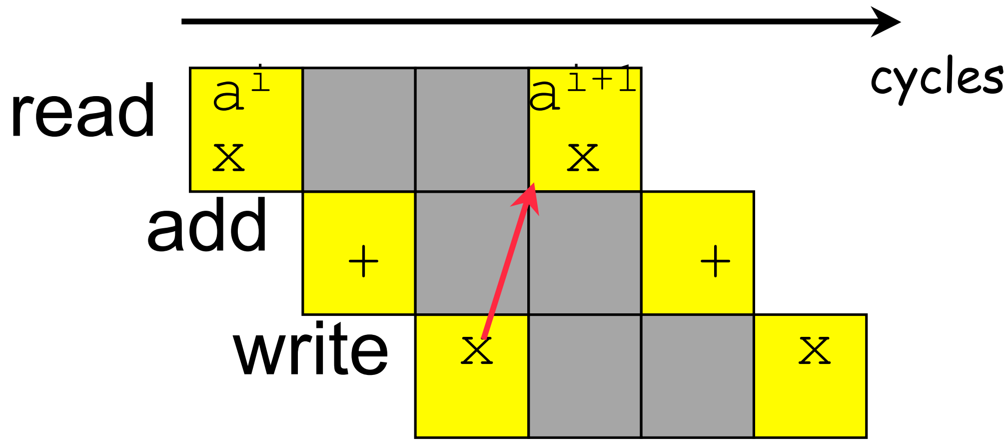
- Instruction level parallelism: pipeline, multiple functional unit and out-of-order execution
- Memory hierarchy
- Speculative execution
- Vector processing unit

Superscalar Processors

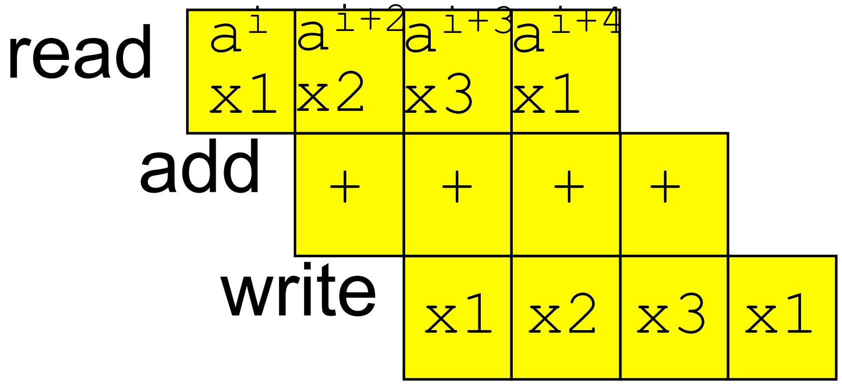


Pipeline Execution

```
do i=1,n
  x = x + a(i)
enddo
```




```
do i=1,n,3
  x1 = x1 + a(i)
  x2 = x2 + a(i+1)
  x3 = x3 + a(i+2)
enddo
x= x1+x2+x3
```

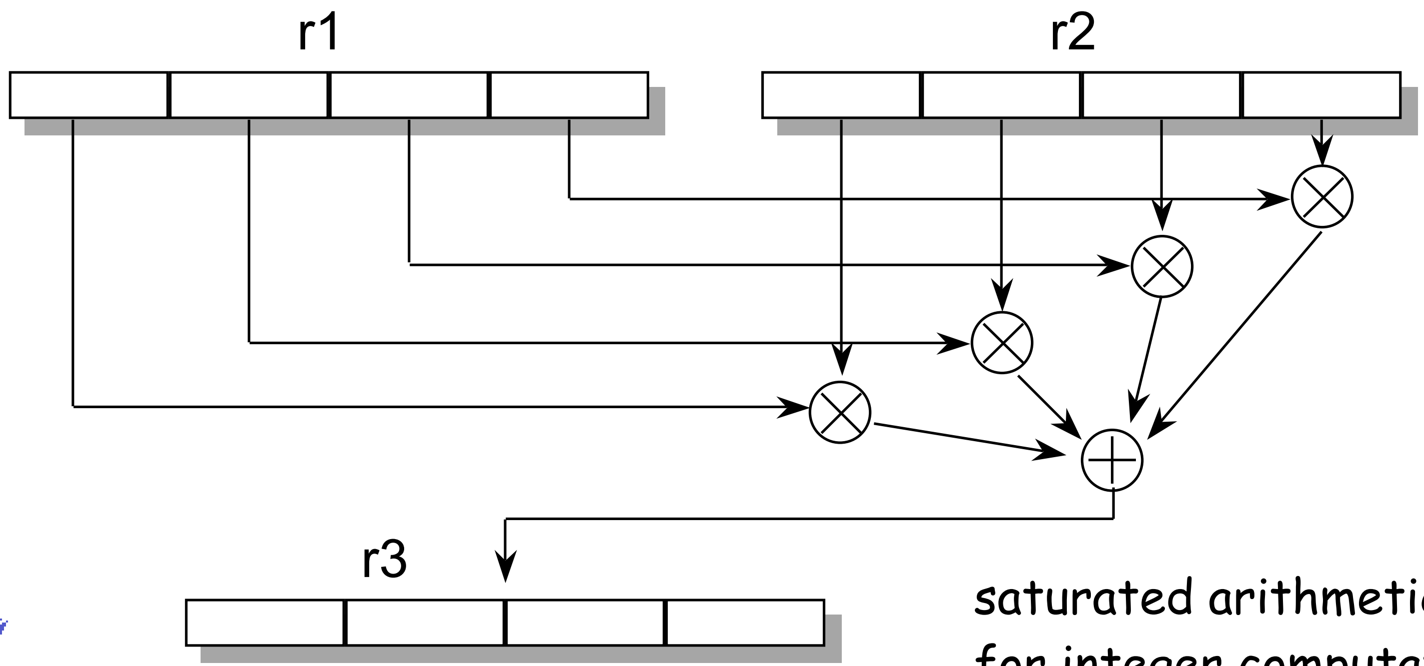


(Multimedia) Vector Instructions
SIMD within a Register

Example from Trimedia:

```
ifir8ui r1 r2-> r3
```

z=(x&15*y&15) +
((x>>8) &15*(y>>8)&15) +
((x>>16) &15*(y>>16)&15) +
((x>>24) &15*(y>>24)&15)



saturated arithmetic
for integer computations

Intel SSE Example

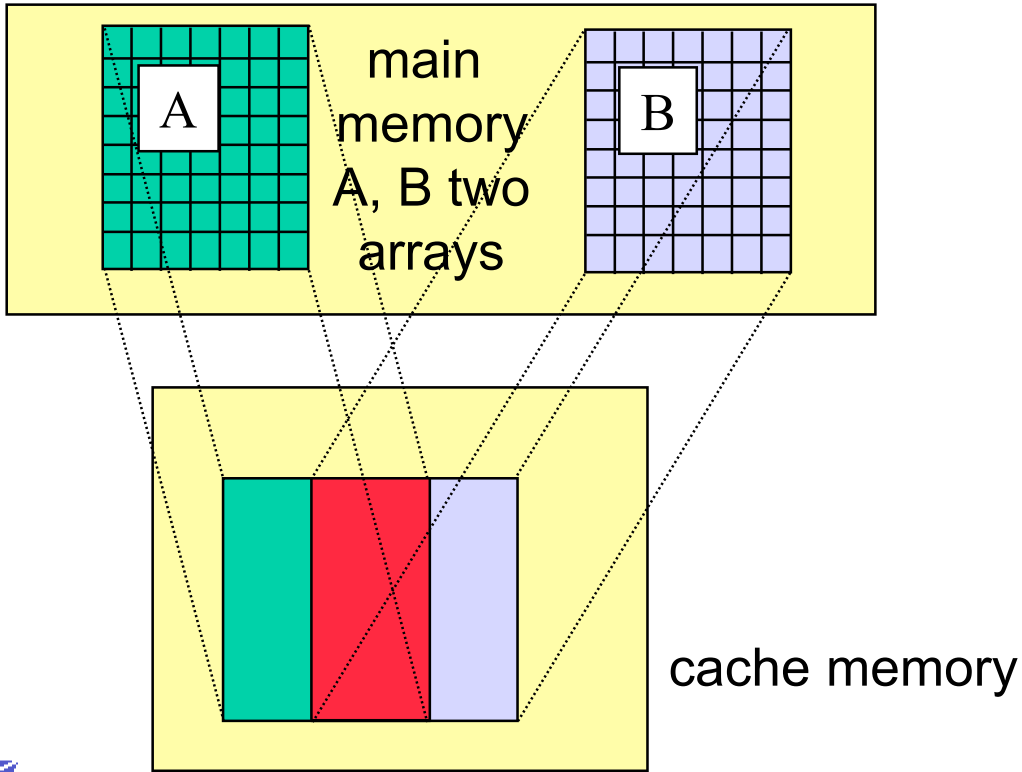
```
unsigned short x[N], y[N], z[N]
void sat(int n)
int i;
  for (i=0;i<n; i++){
     int t= x[i] + y[i];
     z[i]= (t < 65535) ?t: 65535;
  }
}
```

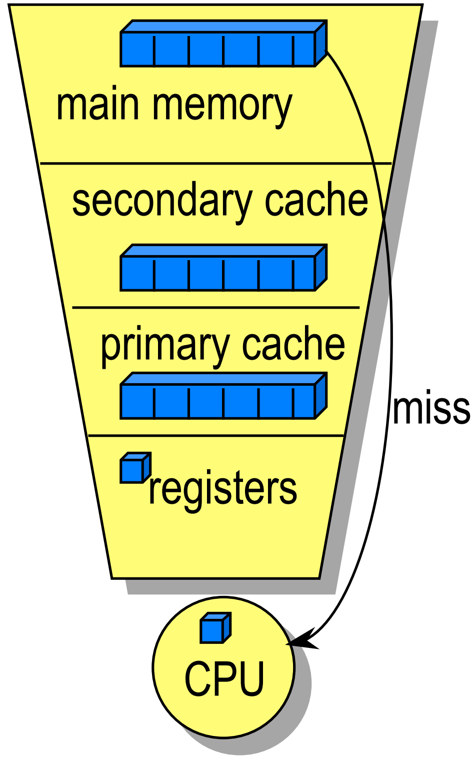
```
   xor eax,exa             ; i =0
L: movdqa xmm0, x[eax]     ; load 8 aligned words from x
   paddusw xmm0, y[eax]    ; add 8 words from y and
                           ; saturate
   movdqa  z[eax], xmm0    ; store 8 words into z
   add     eax, 16         ; increment 8x2
   cmp     eax, ecx        ; iterate n/8 times
   jb      L               ; followed by cleanup loop
```

Example from A. Bik 2004

Memory Hierarchy - Principle

- Efficient if data fits in the cache
- No interference

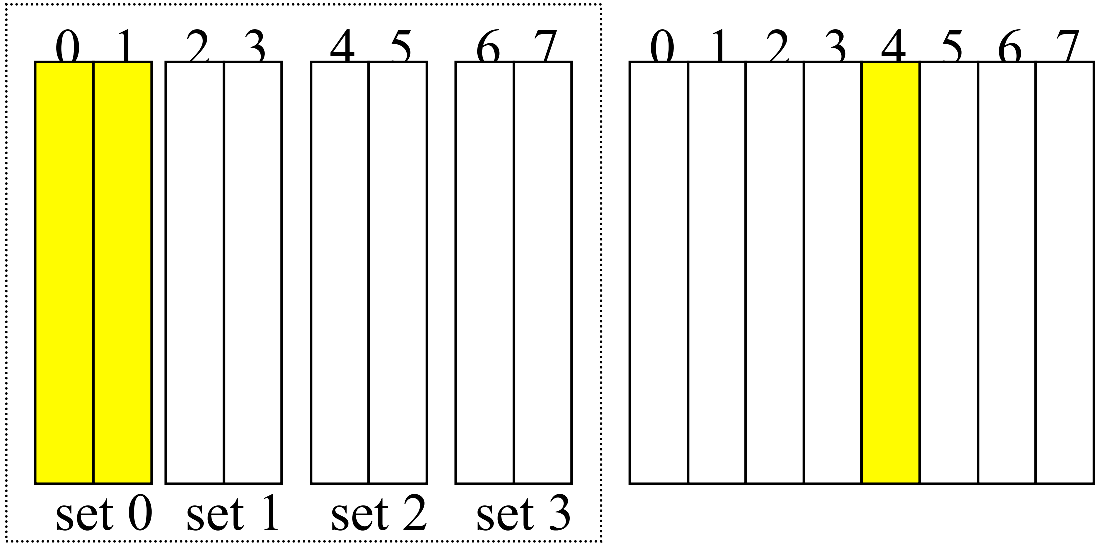




Rmk: loads are usually non blocking

Cache Memories

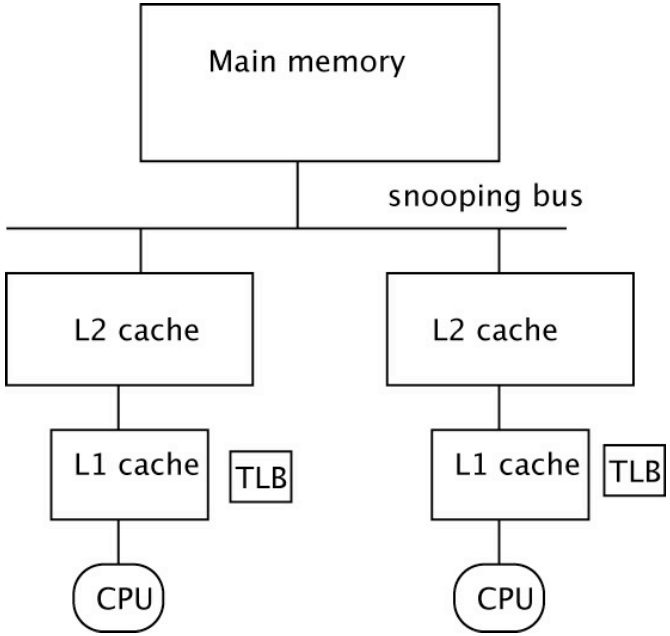
Various organization



k-way associative direct-mapped

block 12 's cache placements

Consistency issue



Data/Instruction Prefetch

- Hardware anticipate memory accesses for reducing memory latency

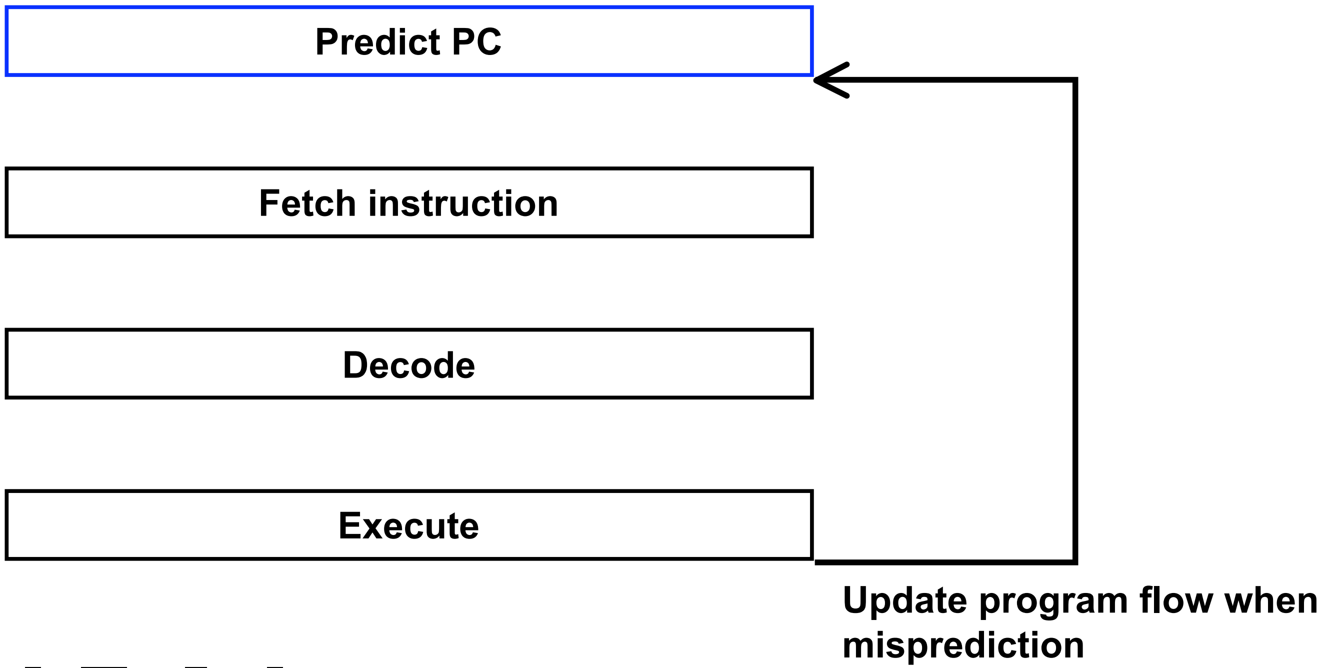
or

- Compiler issues prefetching instructions for loading data used later (but then what is the prefetch distance?)
- A major feature for high performance
- May cause cache pollution

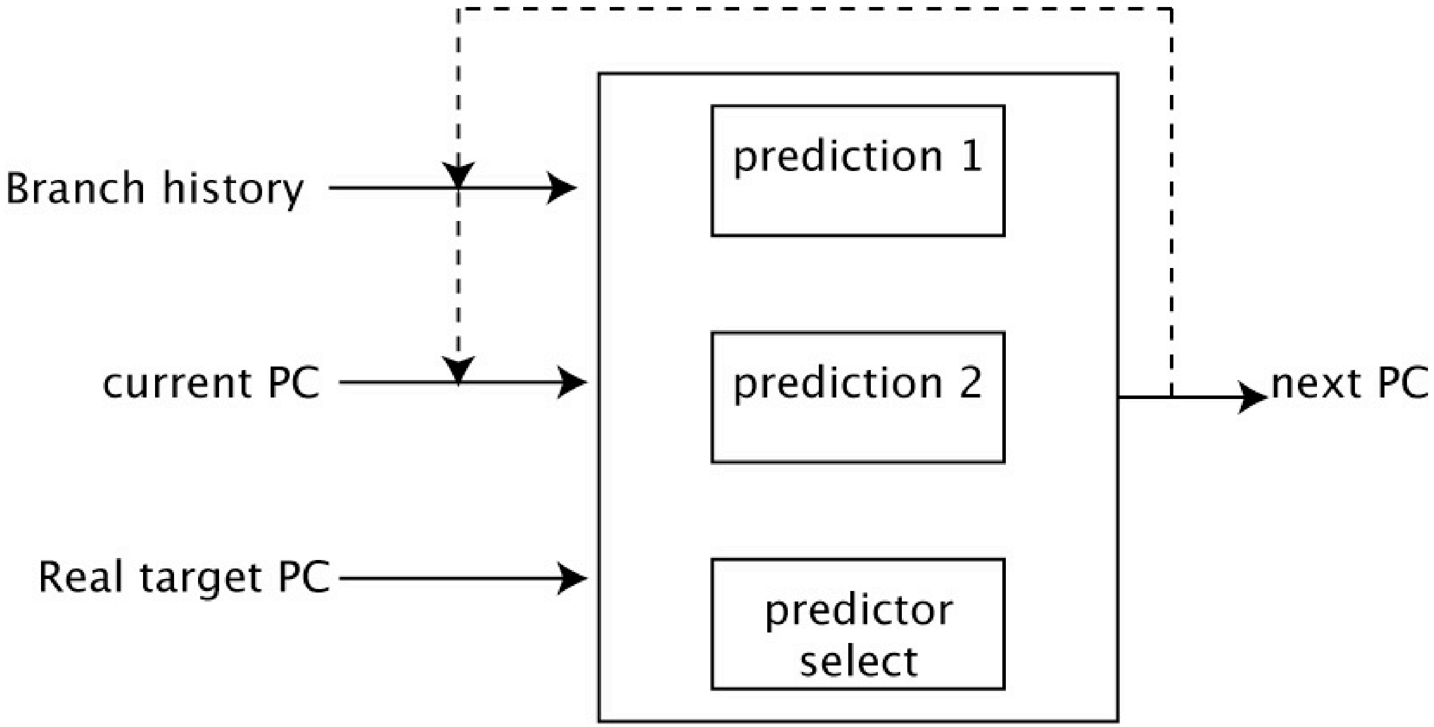
```
 do j=1, cols
//strip mining
    do ii = 1, row, blocksize
        prechargement(&(x(ii,j))+ blocksize)
        do i = ii, ii+blocksize-1
           sum = sum + x(i,j)
        enddo
    enddo
  enddo
```

Branch Prediction

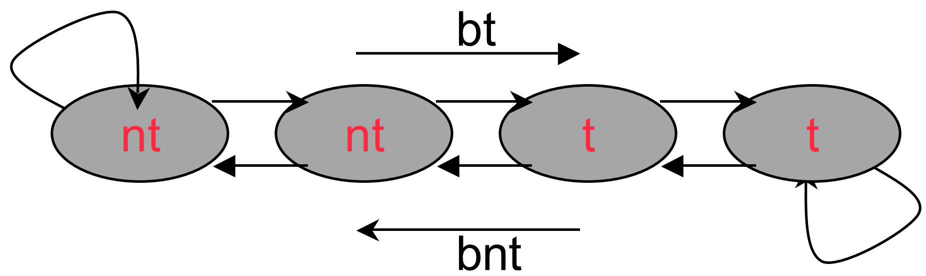
- Most branches are biased (a loop loops), some are correlated
- One of the key mechanisms for superscalar processors
- Anticipate branch computation: speculative execution



Branch Prediction Implementation

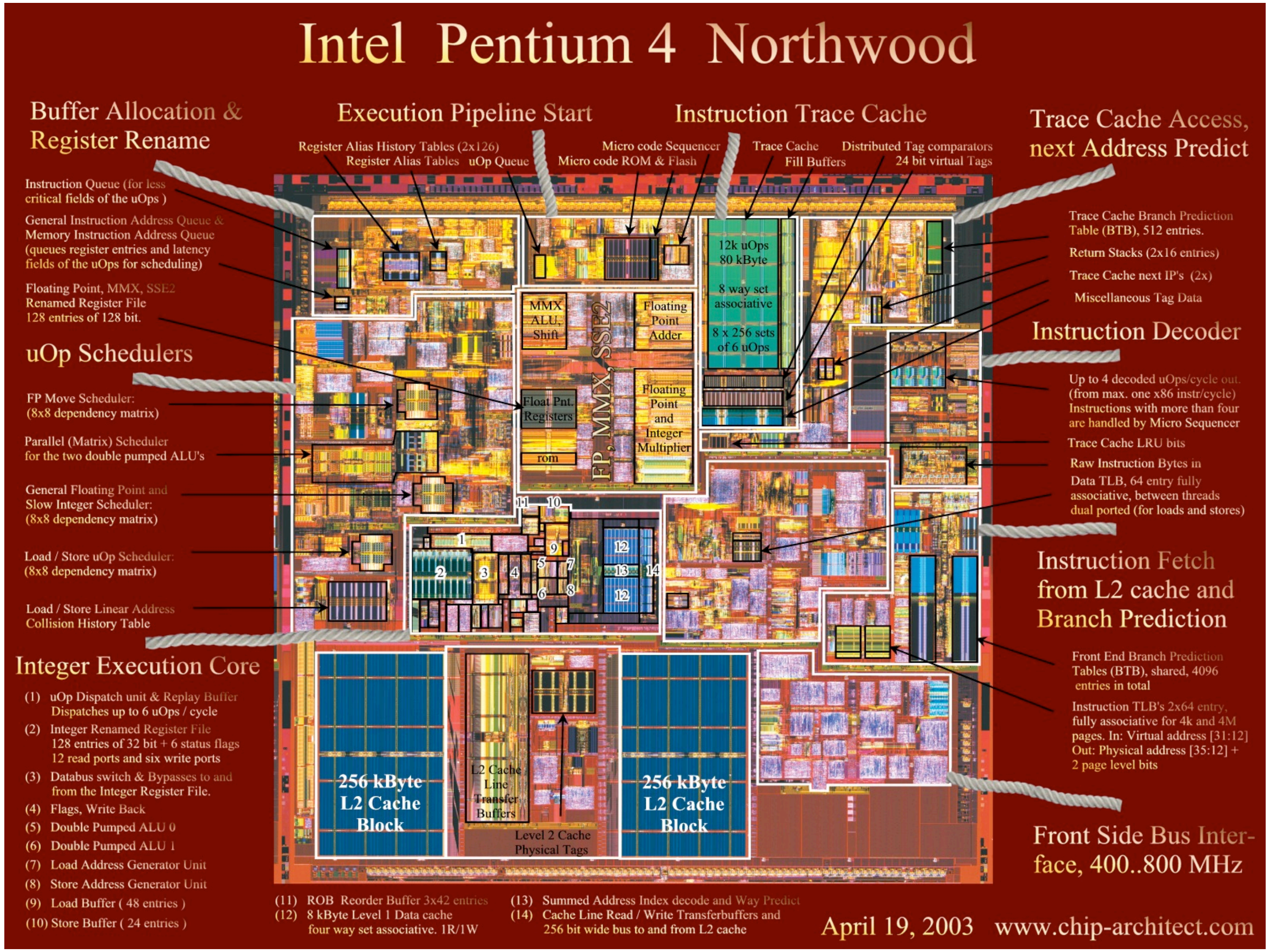



```
do i=1,n
   if (cond1) S1
   if (cond1 .and. cond2) S2
enddo
```



CAPS entreprise

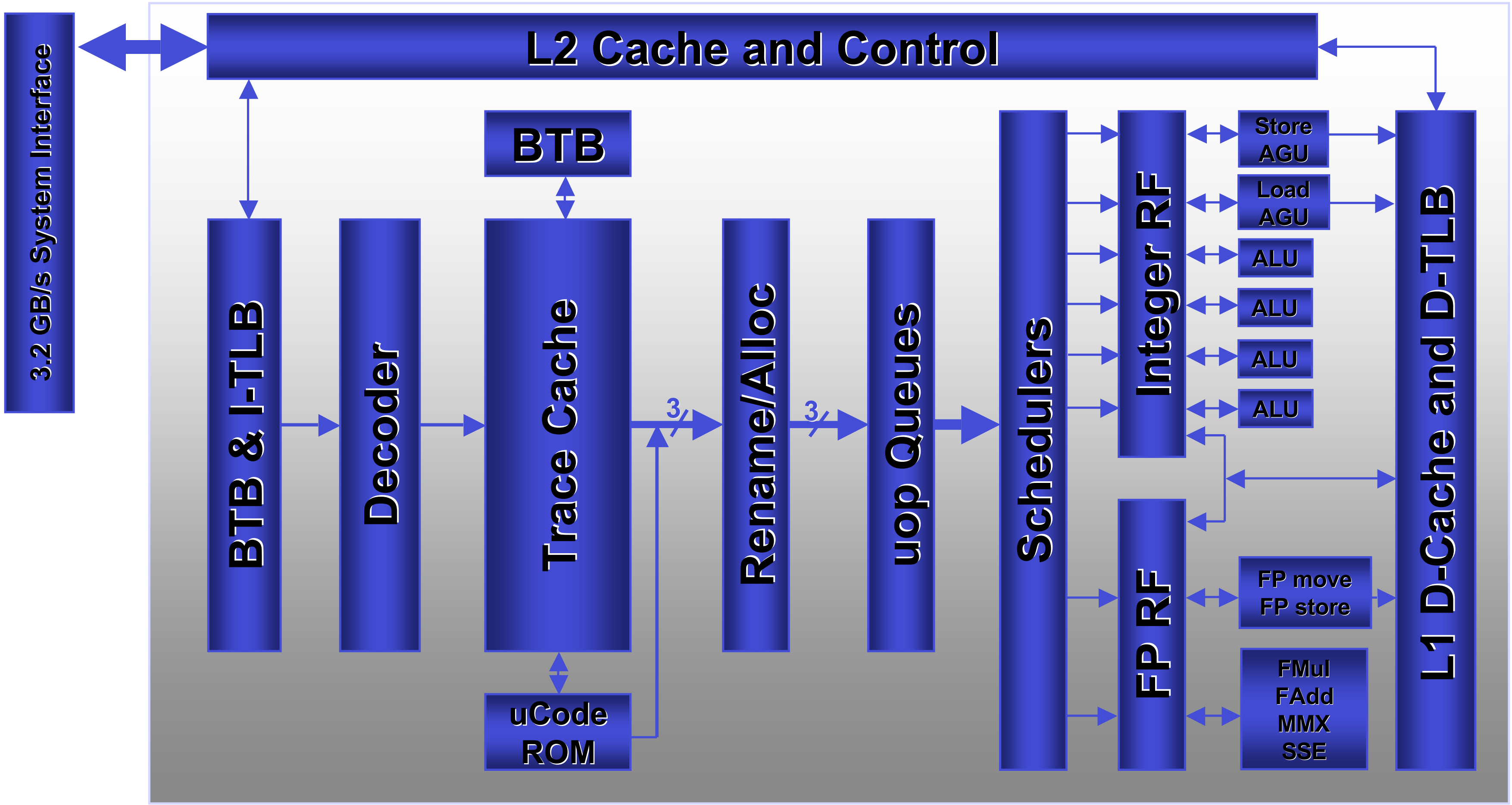
An Example Pentium4 Chip



http://www.chip-architect.com/

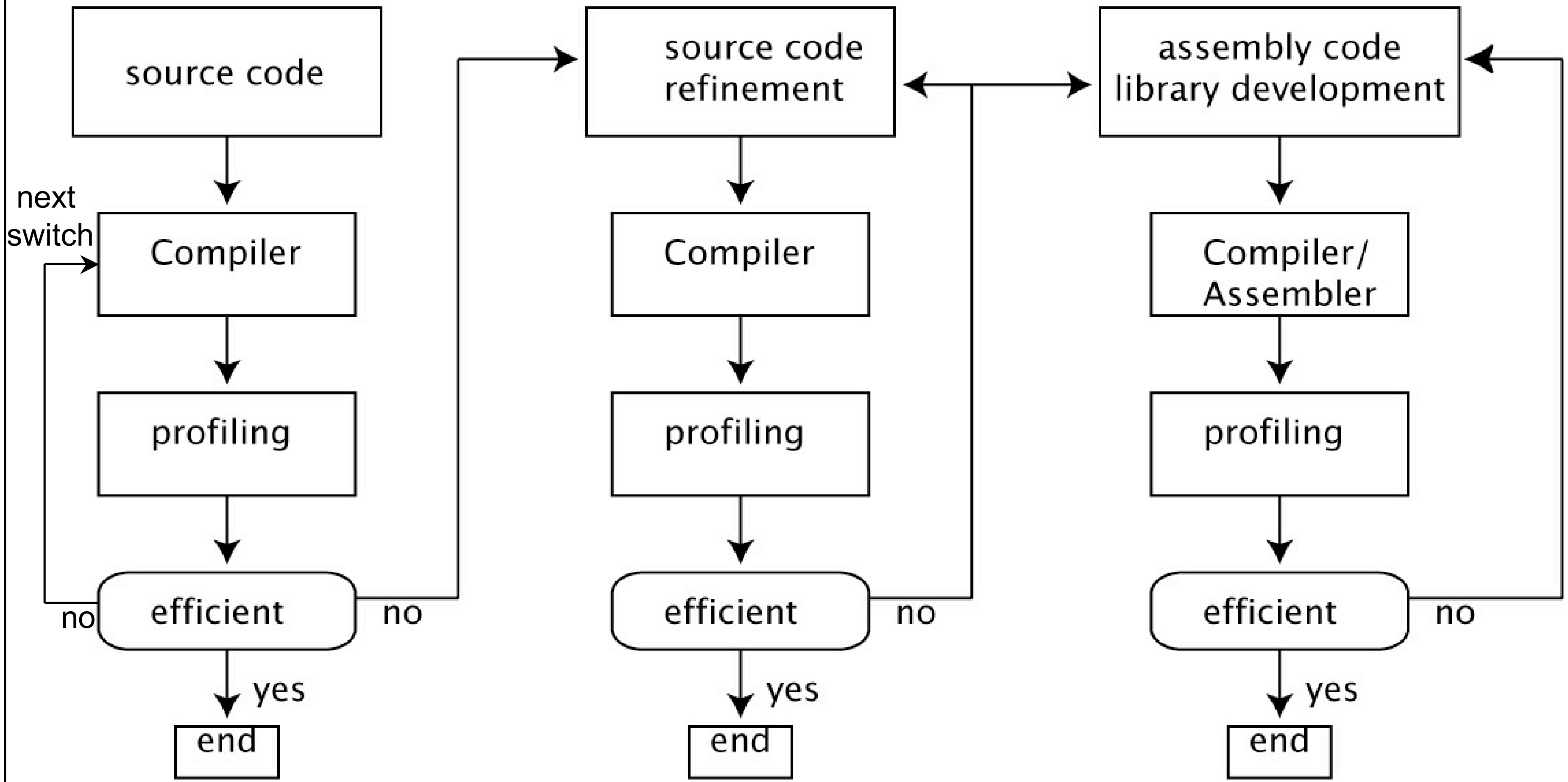
INRIA

Pentium 4 (from D. Carmean)



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Nxt IP		TC Fetch		Drive	Alloc	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive

Code Tuning


source code
Compiler
profiling
efficient
end
yes
no
next
switch
no
source code
refinement
Compiler
profiling
efficient
end
yes
no
assembly code
library development
Compiler/
Assembler
profiling
efficient
end
yes
no

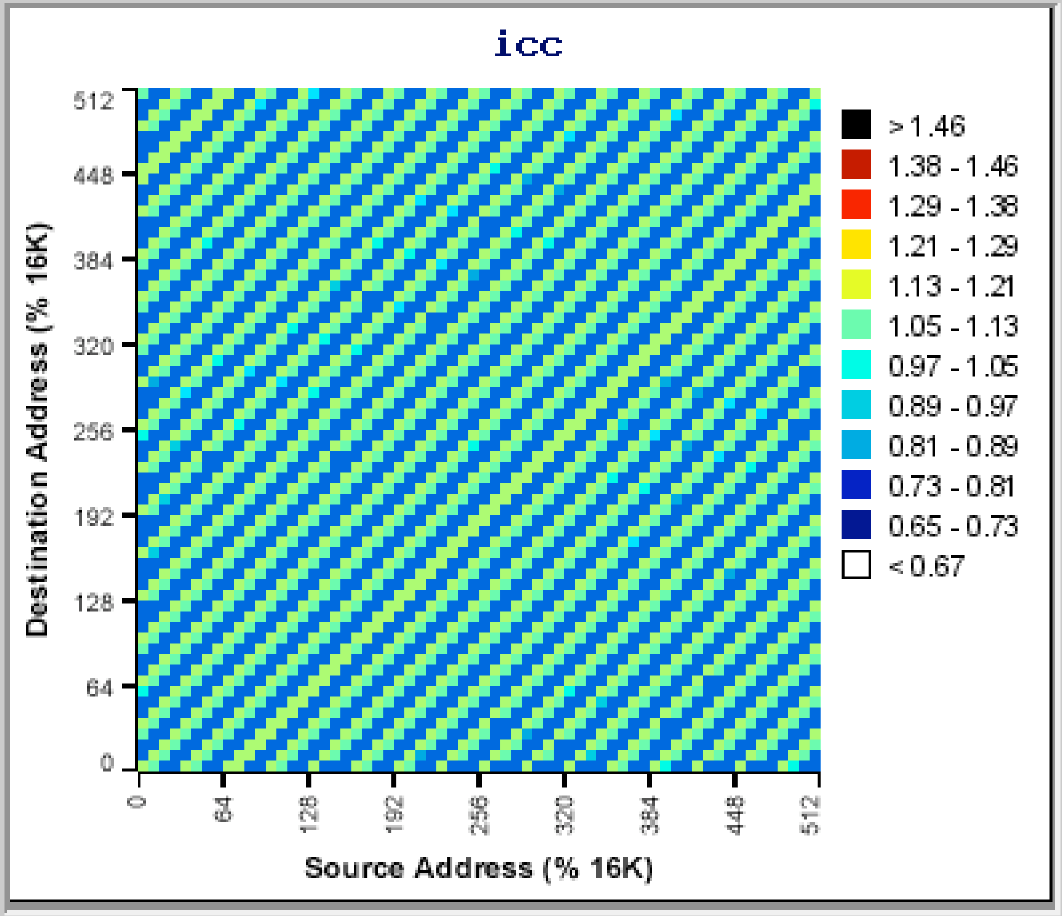
Issue in Code Tuning

- Program efficiency can vary a lot
  - an order of magnitude is common between an optimized and non optimized code
  - performance instabilities
- Identifying bottlenecks
- Code performance depends on
  - structure of the code
  - compiler
  - I/O

Performance instabilities

- Performance instabilities induced by data layout

daxpy (L2)
Itanium2
icc 8.1
CPI

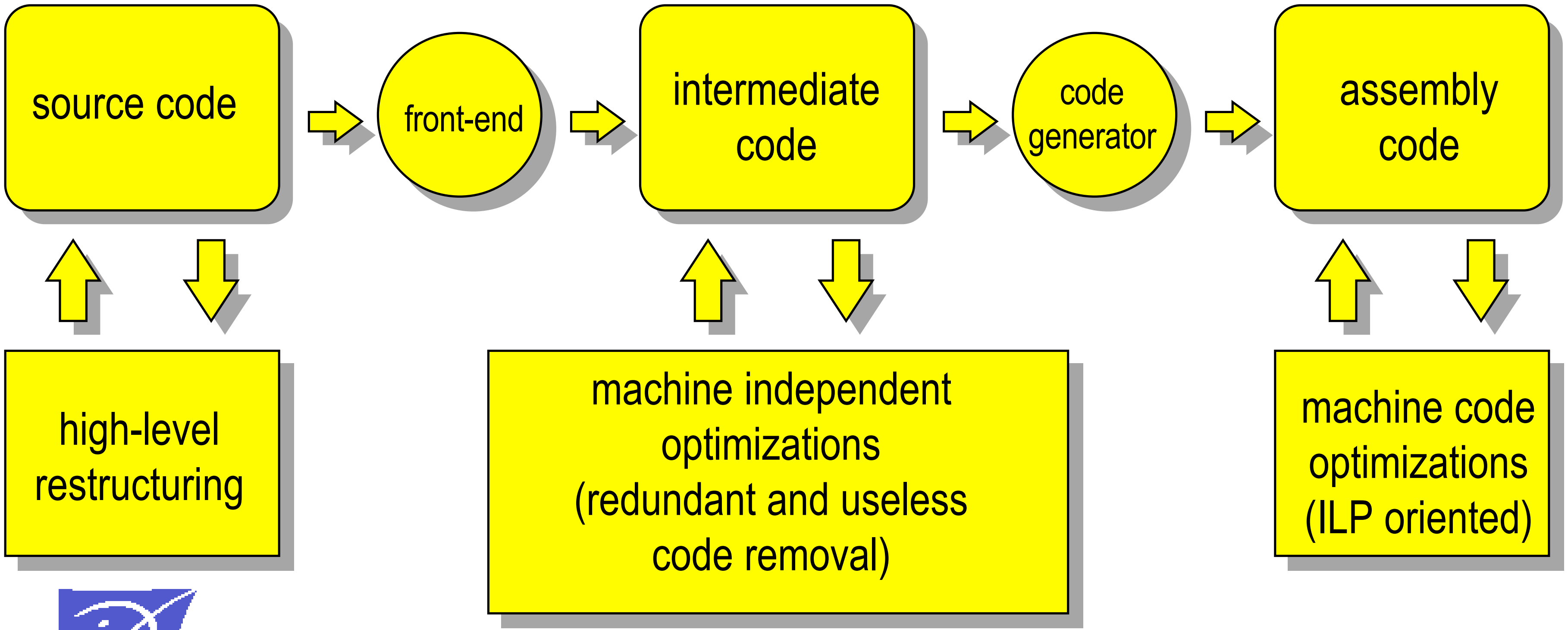


Identifying Bottlenecks

- Profiling tools
  - prof, gprof, …
    - sampling based
  - tcov, pixie, quantify, …
    - basic bloc instrumentation
  - Vtune, …
    - Sampling of hardware counters
    - efficient but may be difficult to interpret
    - event counting (miss/hit, etc.)

Compilers

- Target independent and dependant optimizations
- Relies on data flow and data dependence analysis
- Handles most optimizations but not all



What is a program transformation

- A change in the code that respect the program semantic
- Issue
  - What to change
  - When to change
  - Legal program transformation
- Base of target specific code optimizations
  - Change computation order to maximize pipeline throughput and memory access speed
  - Sequence of transformations are decided by the compiler according to its internal strategy and the compiler option switches

When is a program transformation correct?

- Respect code semantic for a program that respects language standard

```
PROGRAM  original
    DIMENSION A(100)
    CALL
    foo(A(2),A(1),100)
END
SUBROUTINE foo(v1,v2,n)
    DIMENSION v1(*), v2(*)
    INTEGER n
    DO i = 1,n-2
          v1(i) = v2(i)
    END DO
END
```

Legal

transformation

```
PROGRAM  transform,
    DIMENSION A(100)
    CALL foo(A(2),A(1),100)
END
SUBROUTINE  foo(v1,v2,n)
    DIMENSION v1(*), v2(*),
    tmp(n)
    INTEGER n
    DO i = 1,n-2
          tmp(i) = v2(i)
    END DO
    DO i = 1,n-2
          v1(i) = tmp(i)
    END DO
END
```

Data Flow and Data Dependence Analysis

- Compute production and usage of data/variable in the program (SSA)
  - partial order on statements
  - used to check that a transformation is conservative
- Common, equivalence, pointers, parameter aliasing inhibit optimizations
  - degrade analysis result
- Data dependencies based on integer linear algebra
  - handles well *affine* array index expressions (not A[B[n*i]])
- C more difficult than Fortran
  - pointer and subroutine parameter aliasing

Analysis Example

```
subroutine func(a,b,n,c)
integer n,c,a(n,n),b(n,n)
do i = 1,n
   do j =1,n
      a(i,j) = c*b(i,j)
   enddo
enddo
end
```

```
subroutine func(a,b,n,c,m)
integer n,m,i,j,c,a(*),b(*)
do i = 1,n
   do j =1,n
      a(i+m*j) = c*b(i+m*j)
   enddo
enddo
end
```

```
#define n 1000
...
int func(int a[n][n], int b[n][n], int c){
int i,j;
   for(i=0;i<n;i++){
      for(j=0;j<n;j++){
         a[j][i] = c*b[j][i];
      }
   }
}
```

C specific Issues - Restrict Pointers

- C99 allows to specify non aliased data structures
- Or using the compiler switch *-fno-alias*, …

```
void f_v2(int * restrict xint, int * restrict yint,
          int * restrict nx,   int * restrict ny,
          int * restrict xh,   int * restrict yh,
          int * restrict s)
{
  int    src, lx2, x, y, k;
  src = 17; lx2 = 3; y = 2; x = 4;
  for (k = 0; k < 100; k++) {
    xint[k] = nx[k]>>1;
    xh[k] = nx[k] & 1;
    yint[k] = ny[k]>>1;
    yh[k] = ny[k] & 1;
    s[k] = src + lx2*(y+yint[k]) + x + xint[k];
  }
```

Compiler optimization strategy

- Decide the sequence of program transformations to apply
  - Top to down, no backtracking
- Different according to the optimization level (compiler switches)
- Can be tuned for
  - Performance
  - Code size
  - Compiler time

Compiler Switches Issues

- Long list of switches
  - *Non linear* behaviour
  - Same options for all the files not always the best
  - The more aggressive optimization, the more risk to degrade performance
- Example from spec2000

SGI Altix 3700 Bx2 (1600MHz 9M L3, Itanium 2)

```
+FDO: PASS1=-prof_gen
      PASS2=-prof_use
Baseline optimization flags:
C programs:
  -fast -ansi_alias
  -IPF_fp_relaxed +FDO
Fortran programs:
  -fast -IPF_fp_relaxed +FDO
```

SGI Altix 3000 (1300MHz, Itanium 2)

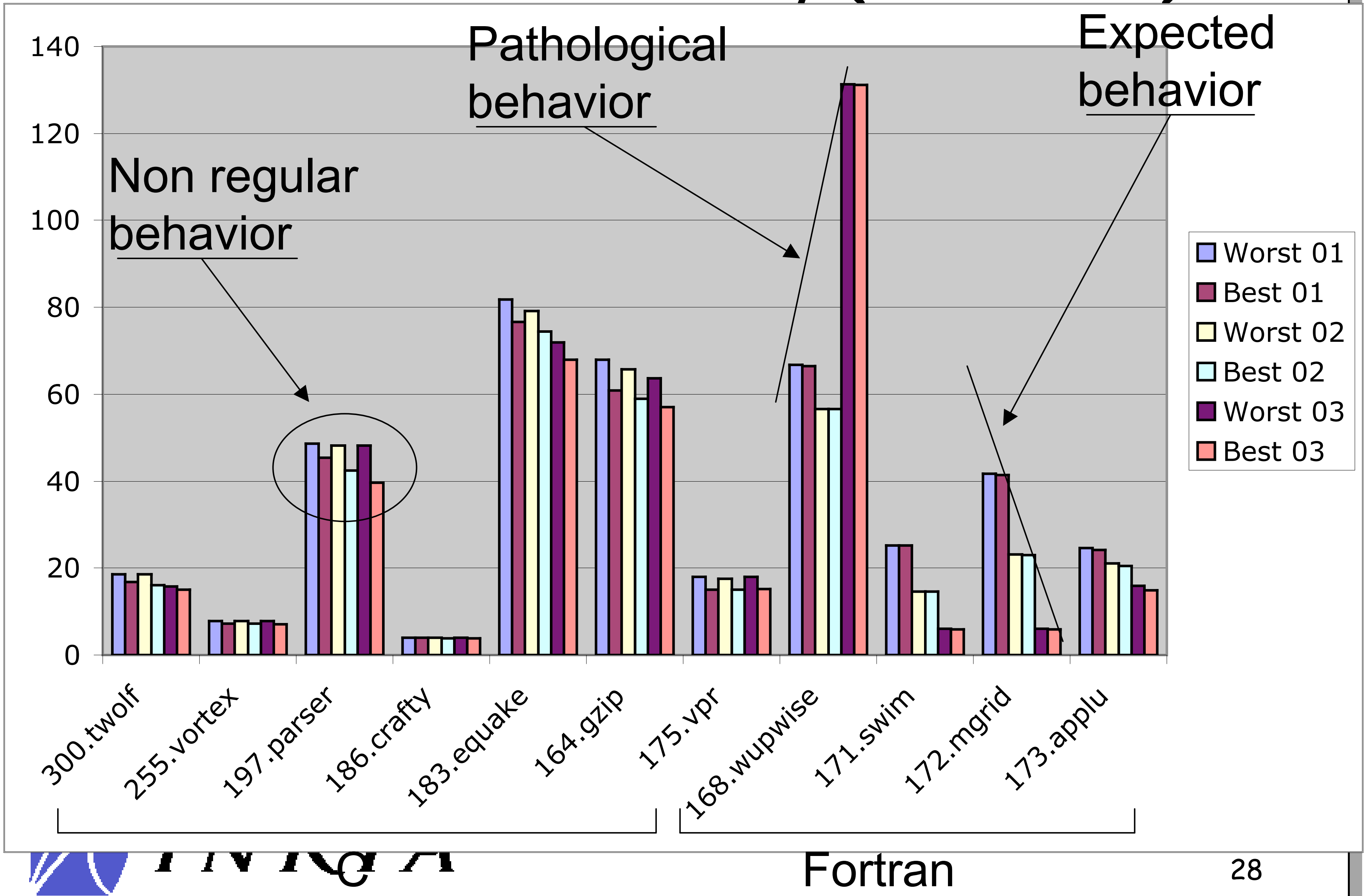
```
+FDO: PASS1=-prof_gen
      PASS2=-prof_use
Baseline optimization flags:
C programs:
   -ipo -O3 +FDO -ansi_alias
Fortran programs:
   -ipo -O3 +FDO
```

Examples

- SPEC2000
- Consider only most time consuming files
  - save compilation time
- Itanium 2 platform, Intel V8.0 compiler
  - tens of optimizations options
- Just a few options to keep it simple
  - -O0/-O1/-O2/-O3 -ip -prof_use -fno-alias
  - 25 settings
- Execution time in seconds

Performance Summary (exec time)

CAPS entreprise



Why does the compiler fail to optimize the code?

- Many unknown data
  - Execution parameters
  - Program analysis inaccuracy
  - No accurate predictive model of the architecture
  - Combining transformations is not always efficient, one transformation may cancel the benefit of another one
- Helping the compiler
  - Choosing the right switches
  - Improving the program analysis
  - Using profiling data
  - Adding "pragma"
  - Use optimize libraries

Architecture Dependant Optimizations

- Memory hierarchy improved hit ratio
  - For instance: loop blocking, unroll and jam
- Improved pipeline execution, instruction level parallelism
  - For instance: unrolling, software pipelining
- Use vector instructions
- *Huge optimization space*

Memory Hierarchy and Code Structure

- Exploit spatial locality
  - have stride 1 array accesses
- Exploit temporal locality
  - Make all usage of a data before going to the next one
- Limit cache interferences
  - Avoid data size that are 2^n
- Exploit program transformations
  - some/most performed by the compiler
  - hand tuning frequently needed

Example

- SGI ONYX, nmax = 1800, dimarray = 1800, t = 5.5 sec.
- SGI ONYX, nmax = 1800, dimarray = 2048, t = **29.8** sec.
- SUN ULTRA, nmax = 800, dimarray = 800, t = 3.58 sec.
- SUN ULTRA, nmax = 800, dimarray = 1024, t = **4.41** sec.

```
real*8 A(dimarray,nmax),B(dimarray,nmax)
do i=2,nmax-1
 do j=2,nmax-1
   A(j,i) = (A(j+1,i)+A(j-1,i)
     +A(j,i+1)+A(j,i-1)+ A(j+1,i+1)
     + A(j-1,i-1))
     *(1.D0/6.D0)+B(i,j)
 enddo
enddo
```

Array Padding

Poorly handled by compilers

```
REAL*8 A(512,512)
REAL*8 B(512,512)
REAL*8 C(512,512)
DO J = 1,512
  DO I = 1,512
    A(I,J) = A(I,J+1)&
        *B(I,J)+C(J,I)
  ENDDO
ENDDO
```

change declaration

```
REAL*8 A(515,512)
REAL*8 PAD1(n1)
REAL*8 B(515,512)
REAL*8 PAD2(n2)
REAL*8 C(515,512)
DO J = 1,512
  DO I = 1,512
    A(I,J) = A(I,J+1)&
        *B(I,J)+C(J,I)
  ENDDO
ENDDO
```

Array Dimension Exchanges

almost never performed by compilers

```
REAL*8 B(2,40,200)
DO I=1,2
  DO J= 1,40
    DO K=1,200
      B(I,J,K) = B(I,J,K)+...
      A( ...) = ...
    ENDDO
  ENDDO
ENDDO
```

```
REAL*8 B(200,40,2)
DO I=1,2
  DO J= 1,40
    DO K=1,200
      B(K,J,I) = B(K,J,I)+...
      A( ...) = ...
    ENDDO
  ENDDO
ENDDO
```

exchange array dimension

Loop Exchange

Sun Ultra 333.0 MHz: 12 sec.

```
real*8 a(500,500),b(500,500)
real*8 c (500,500)
do i=1,n
  do j= 1,n
    do k= 1,n
        a(j,i) = a(j,i)
               +b(j,k)*c(k,i)
    enddo
  enddo
enddo
```

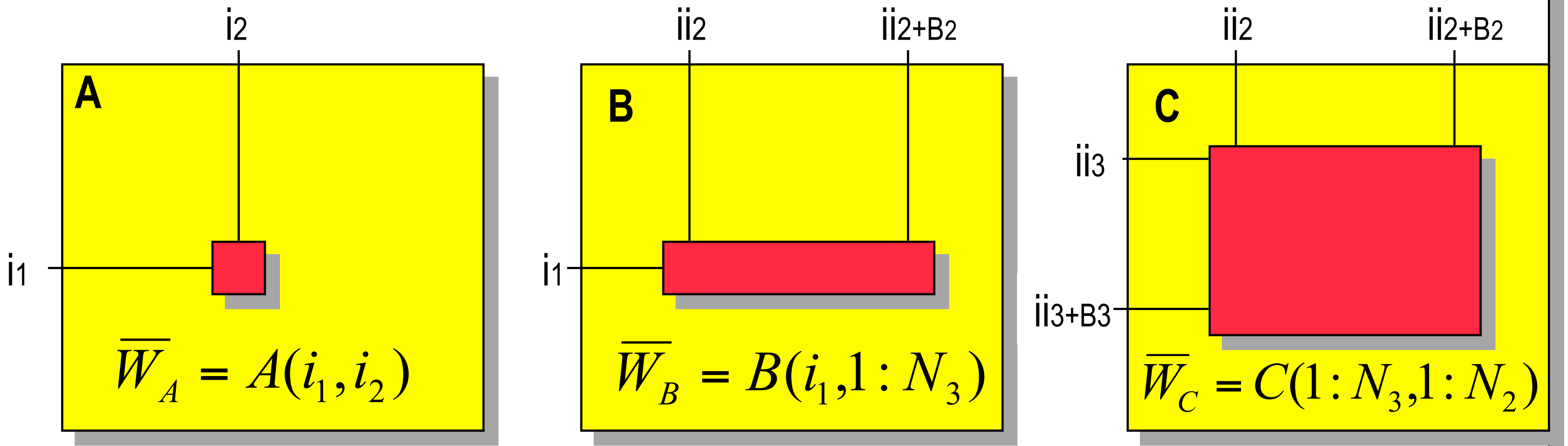
Sun Ultra 333.0 MHz: 3.8 sec.

```
real*8 a(500,500),b(500,500)
real*8 c (500,500)
do i=1,n
  do k= 1,n
    do j= 1,n
        a(j,i) = a(j,i)
               + b(j,k)*c(k,i)
    enddo
  enddo
enddo
```

Exchange loop order

Loop Blocking (temporal locality)

Sun Ultra 333.0 MHz: 1.8 sec.




```
DO 10 ii1 = 1, N1, B1
 DO 10 ii2 = 1, N2, B2
  DO 10 ii3 = 1, N3, B3
   DO 10 i1 = ii1, min(ii1 + B1 -1, N1)
    DO 10 i2 = ii2, min(ii2 + B2 -1,N2)
     DO 10 i3 = ii3 , min(ii3 + B3 -1,N3)
       A(i1,i2) = A(i1,i2) + B(i1,i3) * C(i3,i2)
10   CONTINUE
```

$$\begin{cases} 1 \leq B_3 + B_2 B_3 + 1 \leq T \\ 1 \leq B_1 \leq N_1 \\ 1 \leq B_2 \leq N_2 \\ 1 \leq B_3 \leq N_3 \end{cases}$$

Blocking for TLB

execution time : 1,93 s

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

```
DO JCHUNK=1,N,64
  DO ICHUNK=1,N,64
    DO I=ICHUNK,MIN0(ICHUNK+63,N)
      DO J=MAX(I,JCHUNK),MIN0(JCHUNK+63,N)
        A(I,J)=A(I,J)+B(J,I)
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

execution time :1,49s

```
DO JCHUNK=1,N,50
  DO ICHUNK=1,N,50
    DO I=ICHUNK,MIN0(ICHUNK+49,N)
      DO J=MAX(I,JCHUNK),MIN0(JCHUNK+49,N)
        A(I,J)=A(I,J)+B(J,I)
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

execution time: 0,499s

Unroll and Jam

Similar to block outer loops
and unroll it in that example
- exploits registers
- better pipelining
- exhibits redundant loads

```
DO 1 i1=1,N1
 DO 1 i2=1,N2
  DO 1 i3=1,N3
    A(i2,i1) = A(i2,i1)
          + B(i2,i3)
          * C(i3,i1)
1 CONTINUE
```

```
DO i1=ii1,ii1+NB-1,2
 DO i2=ii2,ii2+NB-1,2
   S00 = A(i2,i1)
   S01 = A(i2,i1 +1)
   S10 = A(i2+1,i1)
   S11 = A(i2+1,i1+1)
    DO  i3=ii3,ii3+NB-1
      S00 = S00 + B(i2,i3)
          * C(i3,i1)
      S01 =  S01 + B(i2,i3)
          * C(i3,i1+1)
      S10 =  S10 + B(i2+1,i3)
          * C(i3,i1)
      S11 = S11 + B(i2+1,i3)
          * C(i3,i1+1)
    ENDDO
   A(i2,i1) = S00
   A(i2,i1 +1) = S01
   A(i2+1,i1) = S10
   A(i1+1,i1+1) = S11
 ENDDO
ENDDO
```

Registers

- Mapping of variables on physical registers
  - How to assign a physical registers (few) to variables (many): can use same register if do not contain a live value at the same time
  - If not enough physical registers insert spill code (save/restore in memory)
- Large loops with multiple array references result in high register pressure
  - loop distribution may help improving performance
  - difficult to highlight

Example from NAS Mgrid

```
      do 600 i3=2,n-1
         do 600 i2=2,n-1
         do 600 i1=2,n-1
600      u(i1,i2,i3)=u(i1,i2,i3)
     >         +c(0)*( r(i1,   i2,   i3   ) )
     >         +c(1)*( r(i1-1,i2,   i3   ) + r(i1+1,i2,   i3   )
     >              +  r(i1,   i2-1,i3   ) + r(i1,   i2+1,i3   )
     >              +  r(i1,   i2,   i3-1) + r(i1,   i2,   i3+1) )
     >         +c(2)*( r(i1-1,i2-1,i3   ) + r(i1+1,i2-1,i3   )
     >              +  r(i1-1,i2+1,i3   ) + r(i1+1,i2+1,i3   )
     >              +  r(i1,   i2-1,i3-1) + r(i1,   i2+1,i3-1)
     >              +  r(i1,   i2-1,i3+1) + r(i1,   i2+1,i3+1)
     >              +  r(i1-1,i2,   i3-1) + r(i1-1,i2,   i3+1)
     >              +  r(i1+1,i2,   i3-1) + r(i1+1,i2,   i3+1) )
     >         +c(3)*( r(i1-1,i2-1,i3-1) + r(i1+1,i2-1,i3-1)
     >              +  r(i1-1,i2+1,i3-1) + r(i1+1,i2+1,i3-1)
     >              +  r(i1-1,i2-1,i3+1) + r(i1+1,i2-1,i3+1)
     >              +  r(i1-1,i2+1,i3+1) + r(i1+1,i2+1,i3+1) )
```

loop distribution

32x32x32	256x256x256
Original: **0,34** sec	Original: **206** sec
Loop distribution: **0.30** sec	Loop distribution: **182** sec

Avoid Short Loops

- Short loops do not behave well
  - better on recent processors (history based prediction)
  - unrolling may improve performance

```
do j= 1,10000
    do i=1,n
        y(i) = y(i) + a(i,j)*x(i)
    enddo
enddo
```

execution time (ultra sparc) : -O0:7.3s
-O2:1.4s
-O3:1.2s

```
   j= 1,10000
   i =1
   y(i) = y(i) + a(i,j)*x(i)
   i =2
   y(i) = y(i) + a(i,j)*x(i)
   i =3
   y(i) = y(i) + a(i,j)*x(i)
enddo
```

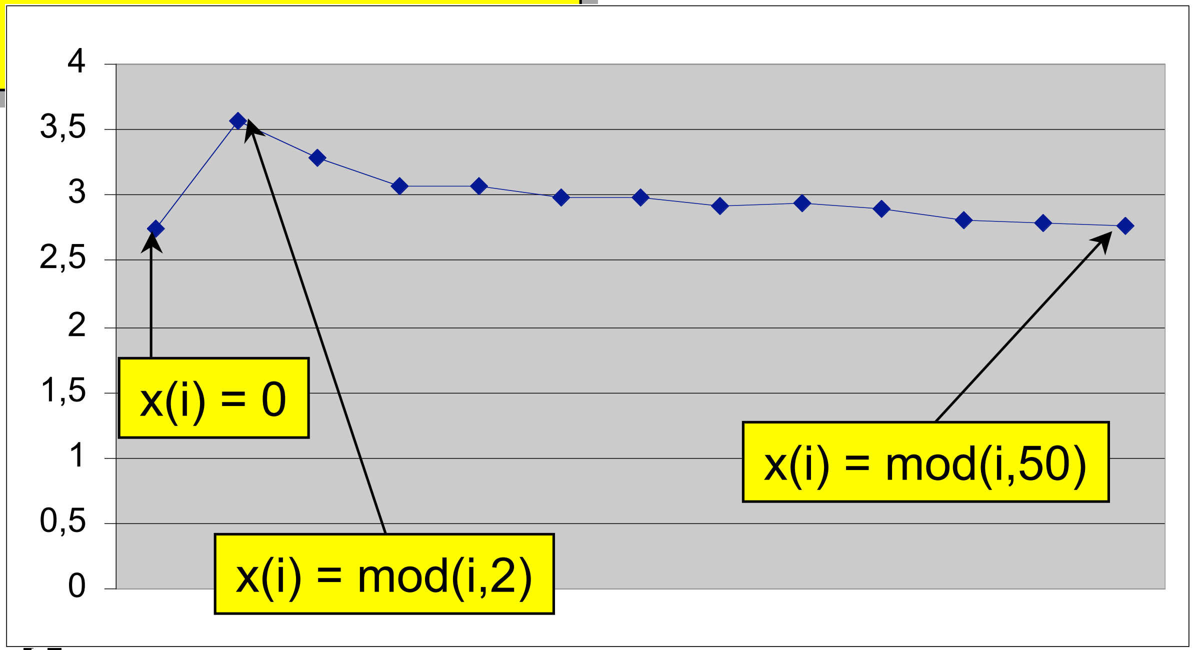
execution time (ultra sparc) : -O0: 4.9s, -O2:0.9s, -03:0.17s

Avoid Unpredictable Branches

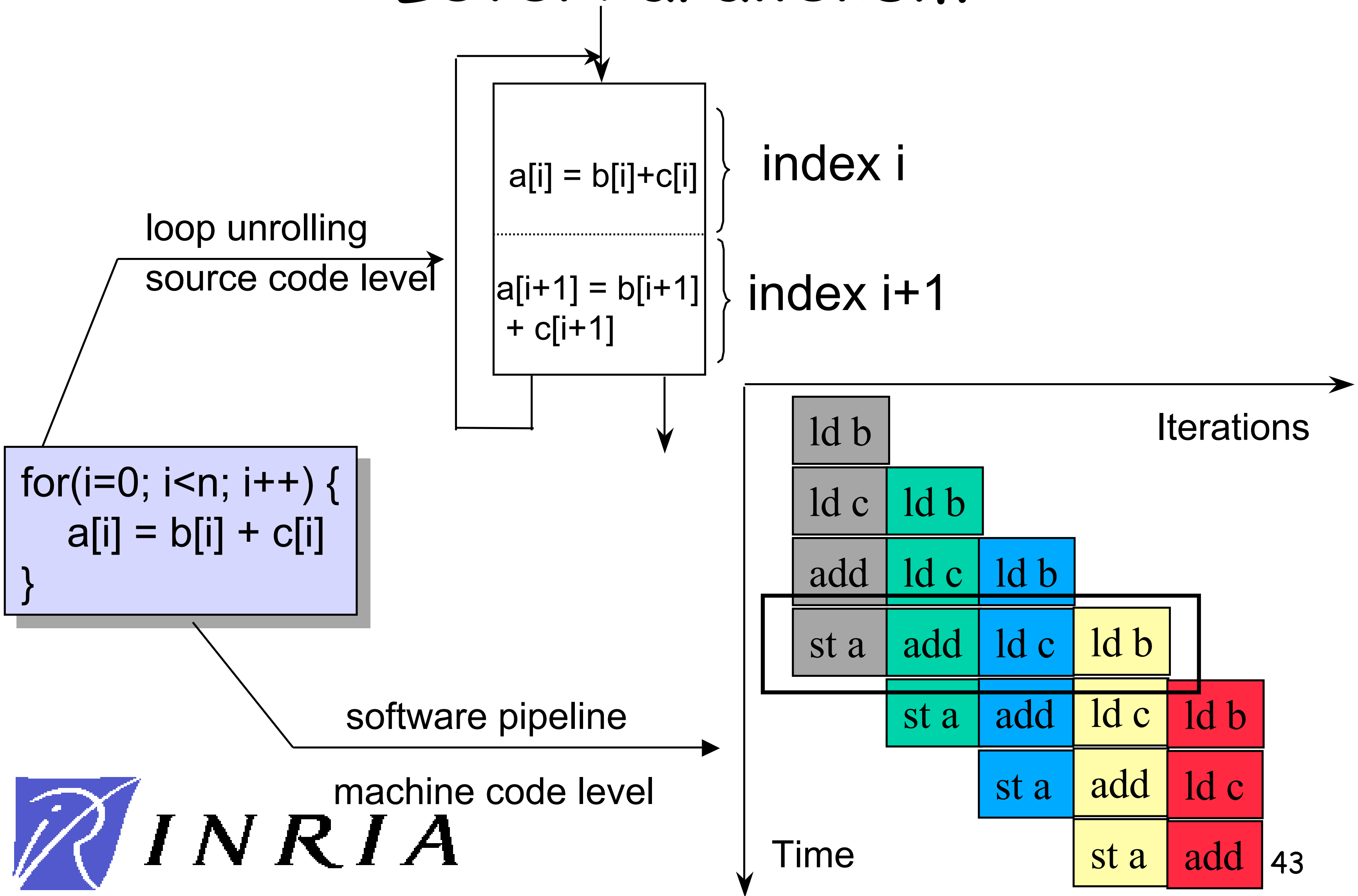
```
do j= 1,n
     do i=1,n
        if (x(i) .eq. 1) then
               y(i) = y(i) + a(i,j)
        else
               y(i) = y(i) - a(i,j)
        endif
     enddo
enddo
```

Can be solved with predicated instructions

execution time in sec. (ultra sparc)



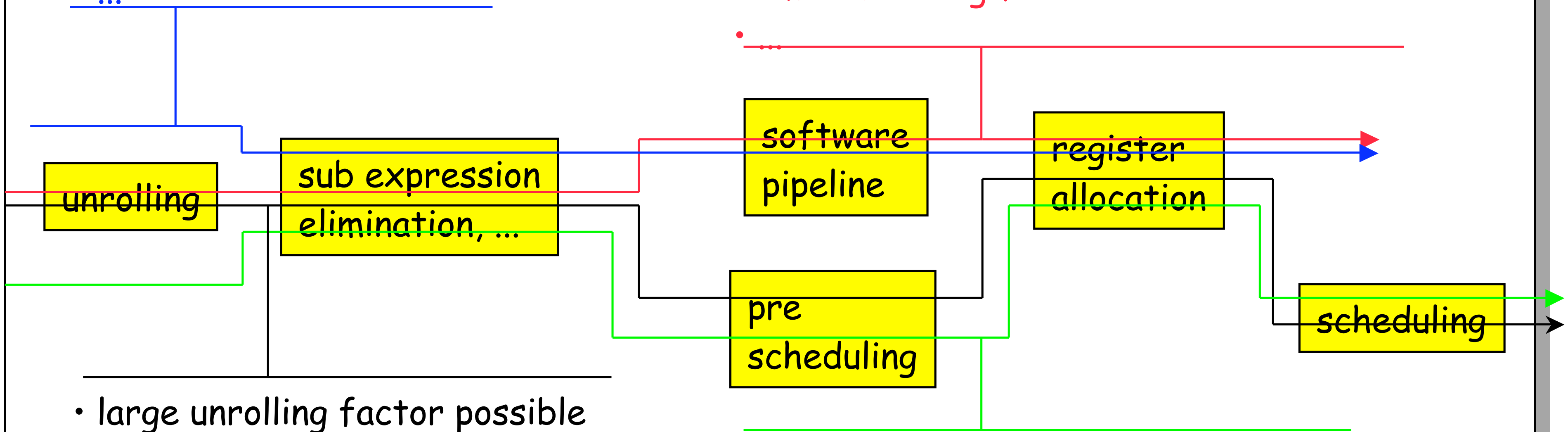
Improving Instruction Level Parallelism



Combining Loop Unrolling and SP

- large iteration number
- vector loop
- no control flow
- ...

- large code size
- sometime useful to reach optimal
- register allocation can fail
- not efficient for small iteration number
- small unrolling factor
- ...



- large unrolling factor possible
- register allocation may fail
- instruction cache overflow
- profiling dependent
- ...

- smallest code size
- sub-optimal
- useful if no ILP between iterations
- ...

Vectorizing techniques

- For using SIMD instructions
- Strongly connected components decomposition
- Strip-mining to adjust to vector length

```
do i=1,n
  a(i) = b(i) + c(i)
  sum = sum + a(i)
enddo
```

decomposition

```
do i=1,n
  a(i) = b(i) +
c(i)
enddo
do i=1,n
  sum = sum + a(i)
enddo
```

Strip-mining

```
do ii=1,n,64
 do i=ii,ii+64-1
   a(i) = b(i) + c(i)
 enddo
enddo
do i=1,n
  sum = sum + a(i)
enddo
```

Vector instruction Generation (SSE, Altivec, …)

```
do ii=1,n,64
   a(ii:ii+64-1) = b(ii:ii+64-1)
      + c(ii:ii+64-1)
enddo
do i=1,n
  sum = sum + a(i)
enddo
```

Using vector instructions

CAPS entreprise

```
float sFPDotProduct ( float sx[], float sy[], long int n ){
        long int i;
        float sDotProduct;                          /*   X•Y    */

        sDotProduct = 0;
        for ( i = 0; i < n; i++ )
                sDotProduct += sx[i] * sy[i];

        return sDotProduct;
}
```

Issue: data alignment

4 x 32 bit floats

```
float vFPDotProduct ( vector float x[], vector float y[], long int n ){

        long int i;
        vector float partialProduct, temp, sum;
        vector unsigned long minusZero;
        float vDotProduct;

        minusZero = vec_splat_u32 ( -1 );            /* create the -0.0 vector */
        minusZero = vec_sl ( minusZero, minusZero );  /* in vector integer      */
        partialProduct = ( vector float ) minusZero;  /* initialize to -0.0     */

        for ( i = 0; i < n ; i++ )                    /* the core of X•Y        */
                partialProduct = vec_madd ( x[i], y[i], partialProduct );

        temp = vec_sld ( partialProduct, partialProduct, 4 );
        sum  = vec_add ( partialProduct, temp );
        temp = vec_sld ( sum, sum, 8 );
        sum  = vec_add ( sum, temp);
        vec_ste ( sum, 0, &vDotProduct );
        return vDotProduct;
}
```

Conclusion

- Huge performance variation depending on code structure
- Hand tuning necessary in many cases
- Performance instabilities difficult to master
- Multiprocessor/Multithread/Multicore parallelism makes it worst