

Program Optimization Through Loop Vectorization



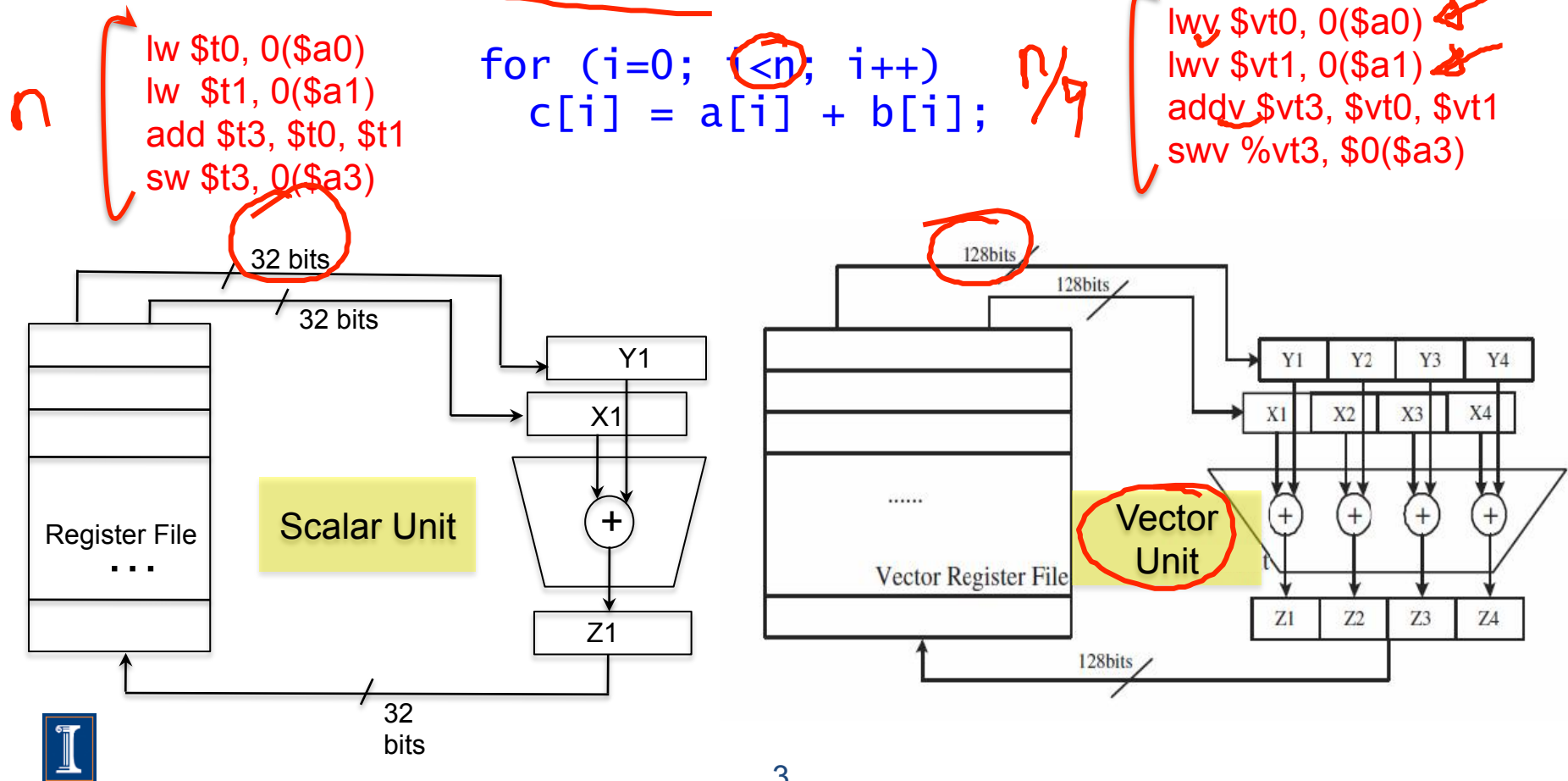
Topics covered

- What are the microprocessor vector extensions or SIMD (Single Instruction Multiple Data Units)
- Overcoming limitations to SIMD-Vectorization
 - Data Dependences
 - Data Alignment
 - Aliasing
 - Non-unit strides
 - Conditional Statements
- Vectorization with intrinsics



Simple Example

- Loop vectorization transforms a program so that the same operation is performed at the same time on several vector elements



SIMD Vectorization

- The use of SIMD units can speed up the program.
- Intel SSE and IBM AltiVec have 128-bit vector registers and functional units
 - 4 32-bit single precision floating point numbers
 - 2 64-bit double precision floating point numbers
 - 4 32-bit integer numbers
 - 2 64 bit integer
 - 8 16-bit integer or shorts
 - 16 8-bit bytes or chars
- Assuming a single ALU, these SIMD units can execute 4 single precision floating point number or 2 double precision operations in the time it takes to do only one of these operations by a scalar unit.
- Newer processors, such as Sandy or Ivy Bridge have AVX that support 256-bit vector registers.

float
double

INTEL



Experimental results

- Results are shown for different platforms with their compilers:
 - Report generated by the compiler ✓
 - Execution Time for each platform ✓

Platform 1: Intel Nehalem
Intel Core i7 CPU 920@2.67GHz
Intel ICC compiler, version 11.1
OS Ubuntu Linux 9.04

Platform 2: IBM Power 7
IBM Power 7, 3.55 GHz
IBM xlc compiler, version 11.0
OS Red Hat Linux Enterprise 5.4

The examples use single precision floating point numbers

32 bits

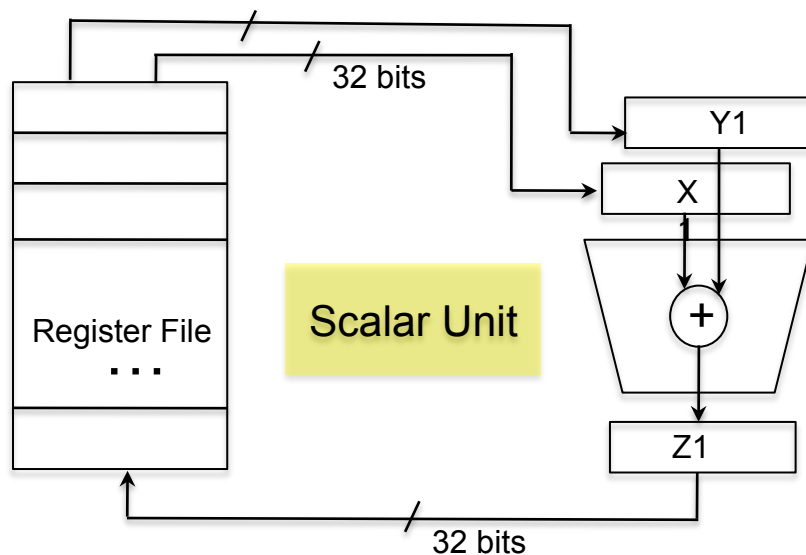


Executing Our Simple Example

```
for (i=0; i<n; i++)  
  c[i] = a[i] + b[i];
```

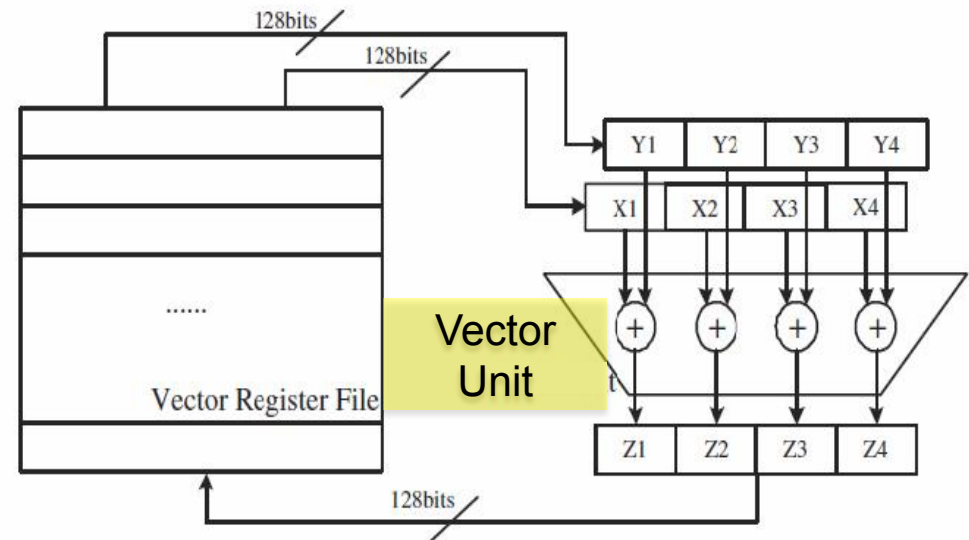
Intel Nehalem

Exec. Time scalar code: 6.1
Exec. Time vector code: 3.2
Speedup: 1.8



IBM Power 7

Exec. Time scalar code: 2.1
Exec. Time vector code: 1.0
Speedup: 2.1



How do we access the SIMD units?

- Three choices

1. C code and a vectorizing compiler

```
for (i=0; i<LEN; i++)  
    c[i] = a[i] + b[i];
```

2. Macros or Vector Intrinsics

vendor specific

```
void example(){  
    __m128 rA, rB, rC;  
    for (int i = 0; i < LEN; i+=4){  
        rA = _mm_load_ps(&a[i]);  
        rB = _mm_load_ps(&b[i]);  
        rC = _mm_add_ps(rA, rB);  
        _mm_store_ps(&c[i], rC);  
    }  
}
```

3. Assembly Language

```
..B8.5  
movaps    a(,%rdx,4), %xmm0  
addps     b(,%rdx,4), %xmm0  
movaps    %xmm0, c(,%rdx,4)  
addq      $4, %rdx  
cmpq      $rdi, %rdx  
jl        ..B8.5
```

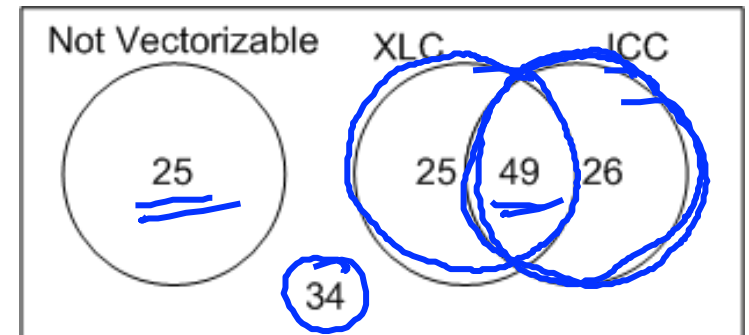


How well do compilers vectorize?

IBM

Compiler	XLC	ICC	GCC
Loops			
Total	159		
Vectorized	74	75	32
Not vectorized	85	84	127
Average Speed Up	<u>1.73</u>	<u>1.85</u>	<u>1.30</u>

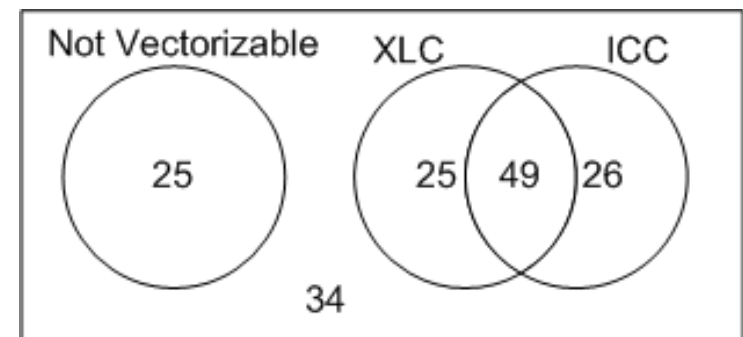
Compiler	XLC but not ICC	ICC but not XLC
Loops		
Vectorized	25	26



How well do compilers vectorize?

Compiler	XLC	ICC	GCC
Loops			
Total	159		
Vectorized	74	75	32
Not vectorized	85	84	127
Average Speed Up	1.73	1.85	1.30

Compiler	XLC but not ICC	ICC but not XLC
Loops		
Vectorized	25	26



By adding manual vectorization the average speedup was 3.78 (versus 1.73 obtained by the XLC compiler)



Why should the compiler vectorize?

1. Easier
2. Portable across vendors and machines
 - Although compiler directives differ across compilers
3. Better performance of the compiler generated code
 - Compiler applies other transformations

Compilers make your codes (almost) machine independent



Compiler Vectorization

- Compilers can vectorize for us, but they fail:

①. Code cannot be vectorized due to data dependences: vectorization will produce incorrect results.

②. Code can be vectorized, but the compiler fail to vectorize the code in its current form

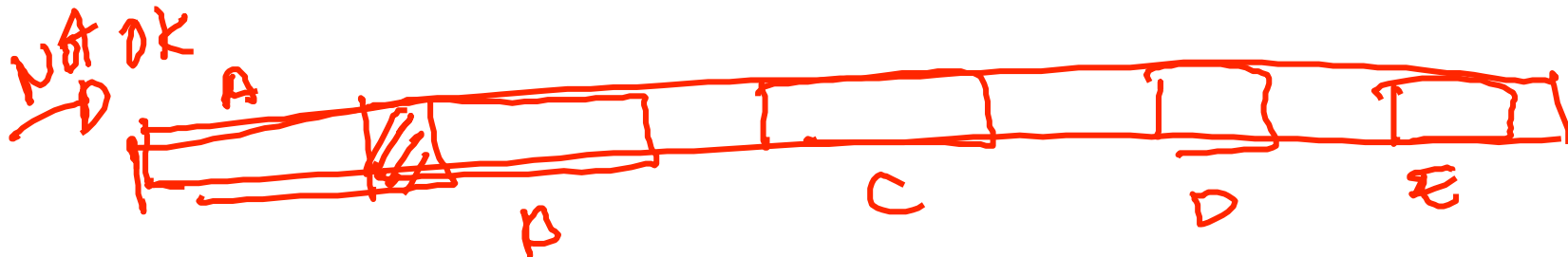
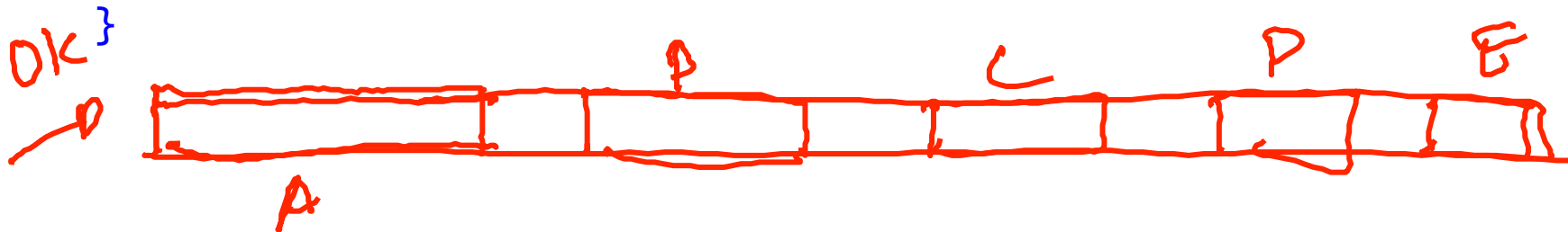
①. Programmer can use compiler directives to give the compiler the necessary information ✓

②. Programmer can transform the code



Example

```
void test(float* A, float* B, float* C, float* D, float* E)  
{  
    for (int i = 0; i < LEN; i++){  
        A[i]=B[i]+C[i]+D[i]+E[i];  
    }  
}
```



Compiler directives

```
void test(float* A, float* B, float*  
C, float* D, float* E)  
{  
    for (int i = 0; i < LEN; i++){  
        A[i]=B[i]+C[i]+D[i]+E[i];  
    }  
}
```

```
void test(float* __restrict__ A,  
float* __restrict__ B,  
float* __restrict__ C,  
float* __restrict__ D,  
float* __restrict__ E)  
{  
    for (int i = 0; i < LEN; i++){  
        A[i]=B[i]+C[i]+D[i]+E[i];  
    }  
}
```

Intel Nehalem

Compiler report: Loop was not
vectorized.

Exec. Time scalar code: 5.6

Exec. Time vector code: --

Speedup: --



Intel Nehalem

Compiler report: Loop was
vectorized.

Exec. Time scalar code: 5.6

Exec. Time vector code: 2.2

Speedup: 2.5

Compiler directives

```
void test(float* A, float* B, float*  
C, float* D, float* E)  
{  
    for (int i = 0; i < LEN; i++){  
        A[i]=B[i]+C[i]+D[i]+E[i];  
    }  
}
```

```
void test(float* __restrict__ A,  
float* __restrict__ B,  
float* __restrict__ C,  
float* __restrict__ D,  
float* __restrict__ E)  
{  
    for (int i = 0; i < LEN; i++){  
        A[i]=B[i]+C[i]+D[i]+E[i];  
    }  
}
```

Power 7

Compiler report: Loop was not
vectorized.

Exec. Time scalar code: 2.3

Exec. Time vector code: --

Speedup: --



Power 7

Compiler report: Loop was
vectorized.

Exec. Time scalar code: 1.6

Exec. Time vector code: 0.6

Speedup: 2.7

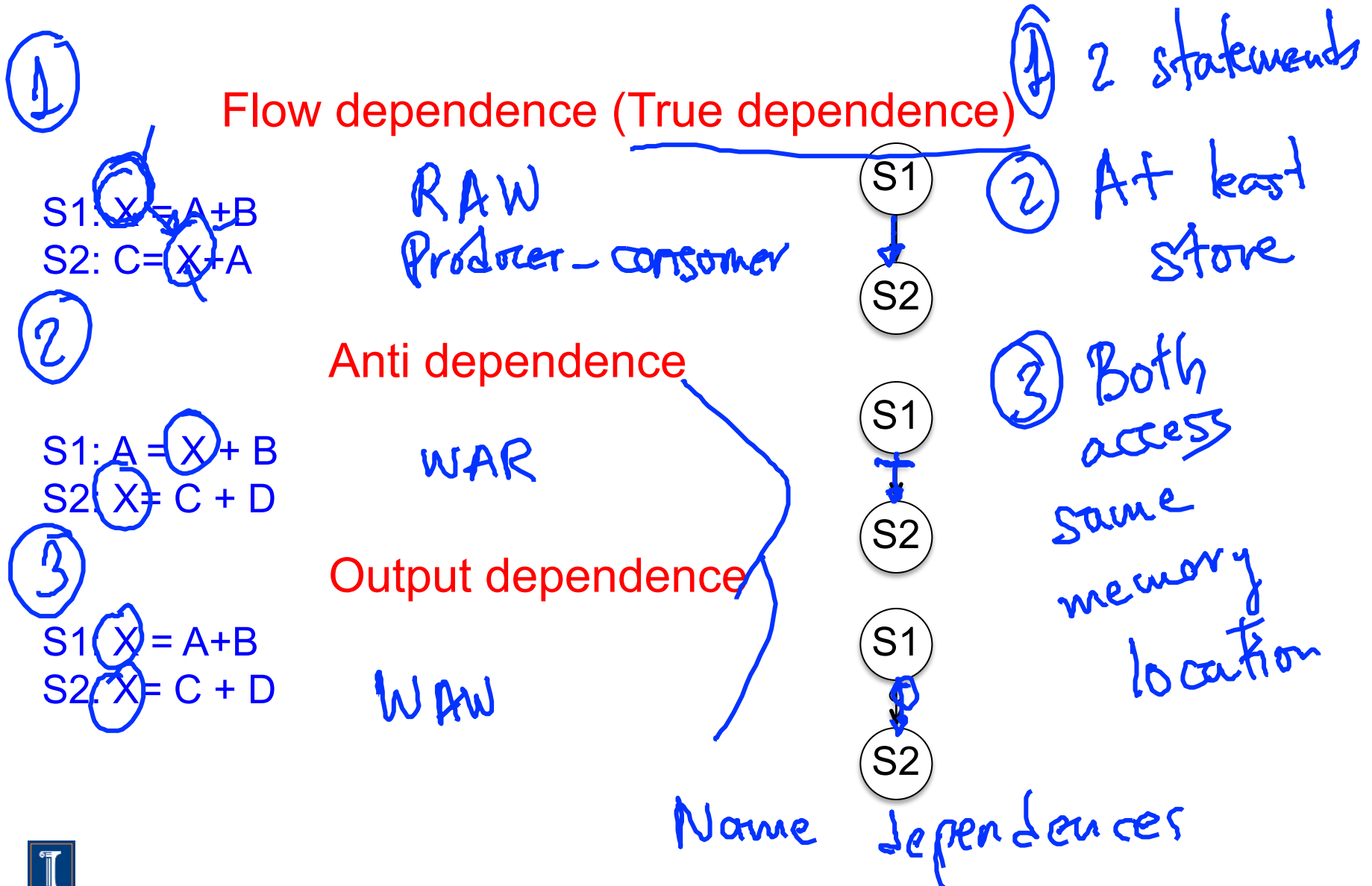
Vectorization is not always legal

- Vectorization of some codes could produce incorrect results
- Compilers (and programmers) can compute data dependences to determine if a program can be vectorized

① parallelized
② transformation



Tour on Data Dependencies



Data Dependencies

S1: $A = B + D$

S2: $C = A + T$

S3: $Z = P + T$

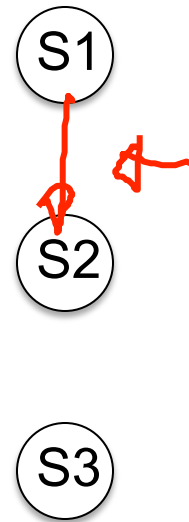
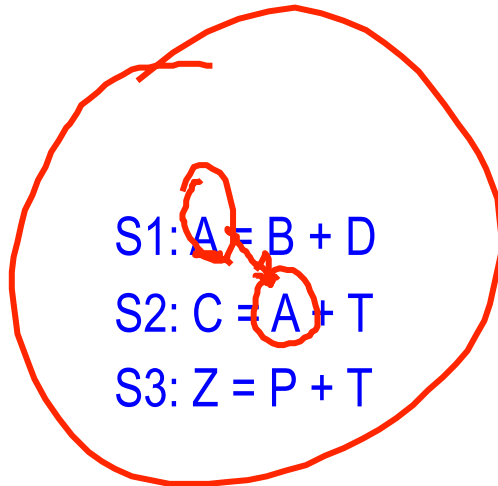
S1

S2

S3



Data Dependencies



① Legal reordering?

S1: $A = B + D$
 S3: $Z = P + T$
 S2: $C = A + T$

Yes ✓



No

②

S2: $C = A + T$
 S1: $A = B + D$
 S3: $Z = P + T$

Yes A

No B ✓



Definition of Dependence

- A statement S is said to be data dependent on statement T if
 - T executes before S in the original sequential/scalar program
 - S and T access the same data item
 - At least one of the accesses is a write.



Data Dependence

- ① • Dependences indicate an execution order that must be honored.
- ② • Executing statements in the order of the dependences guarantee correct results.
- ③ • Statements not dependent on each other can be reordered, executed in parallel, or coalesced into a vector operation.



Dependences in Loops (I)

- Dependences in loops are easy to understand if the loops are unrolled. Now the dependences are between statement “executions”.

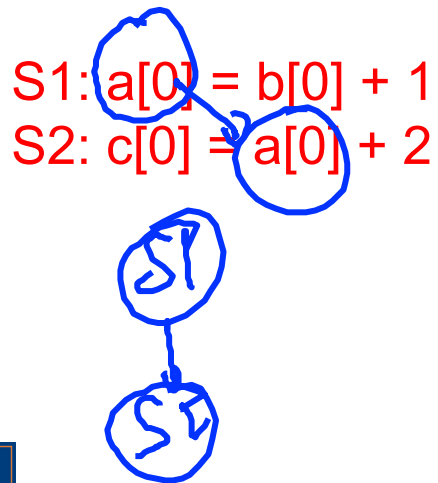
```
    for (i=0; i<n; i++){  
S1      a[i] = b[i] + 1;  
S2      c[i] = a[i] + 2;  
    }
```



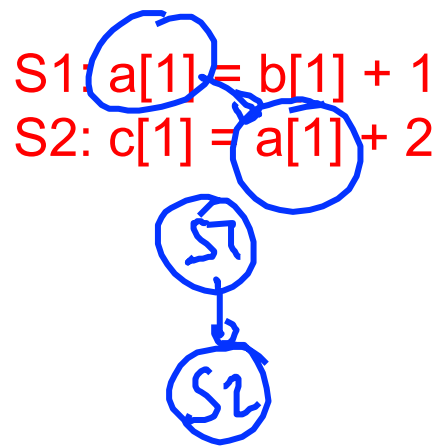
Dependences in Loops (I)

```
for (i=0; i<n; i++){  
  S1  a[i] = b[i] + 1;  
  S2  c[i] = a[i] + 2;  
}
```

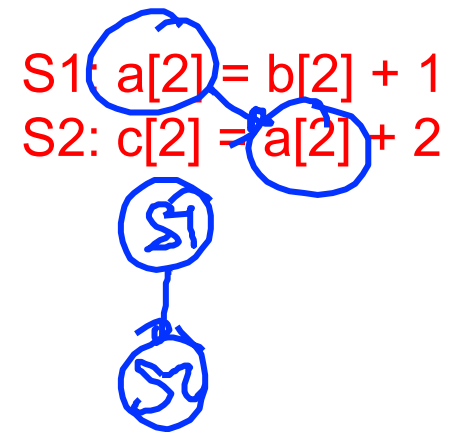
i=0



i=1

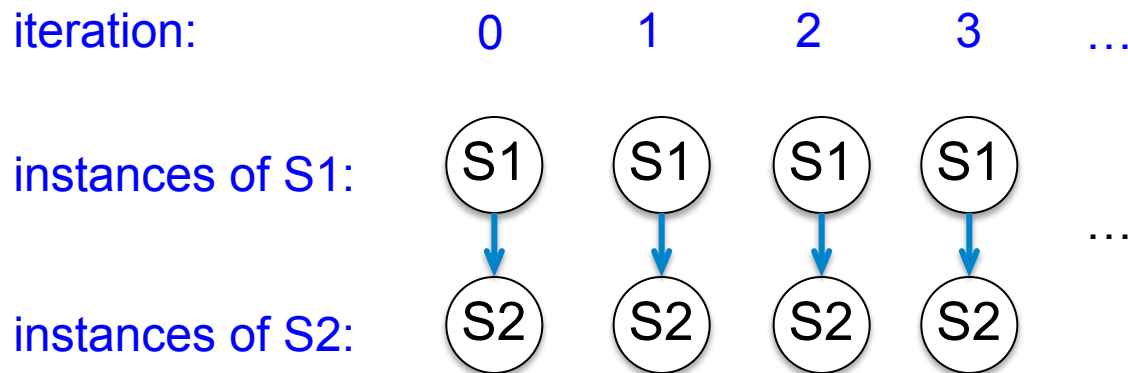


i=2



Dependences in Loops (I)

```
    for (i=0; i<n; i++){  
S1    a[i] = b[i] + 1;  
S2    c[i] = a[i] + 2;  
    }
```



Dependences in Loops (I)

```
for (i=0; i<n; i++){  
  S1  a[i] = b[i] + 1;  
  S2  c[i] = a[i] + 2;  
}
```

iteration:

0 1 2 3 ...

instances of S1:



instances of S2:

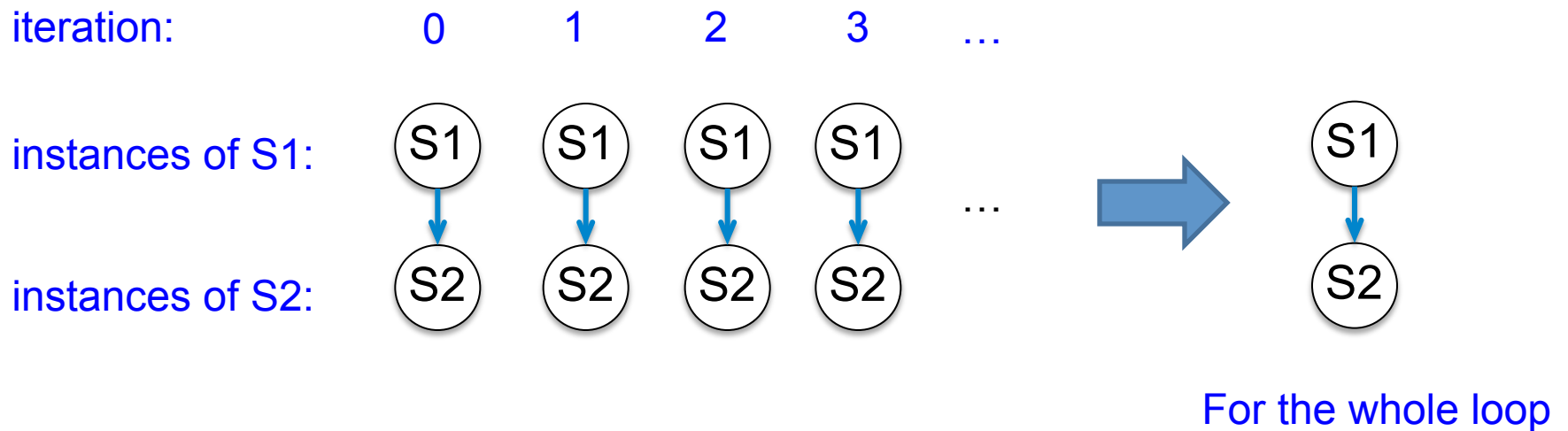


→ Loop independent dependence



Dependences in Loops (I)

```
for (i=0; i<n; i++){  
  S1  a[i] = b[i] + 1;  
  S2  c[i] = a[i] + 2;  
}
```



Dependences in Loops (I)

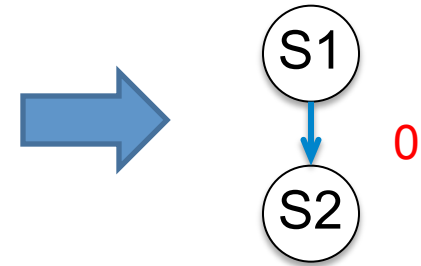
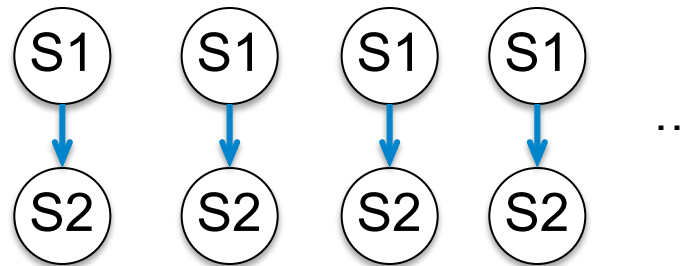
```

    for (i=0; i<n; i++){
S1      a[i] = b[i] + 1;
S2      c[i] = a[i] + 2;
    }

```

iteration: 0 1 2 3 ...

instances of S1:



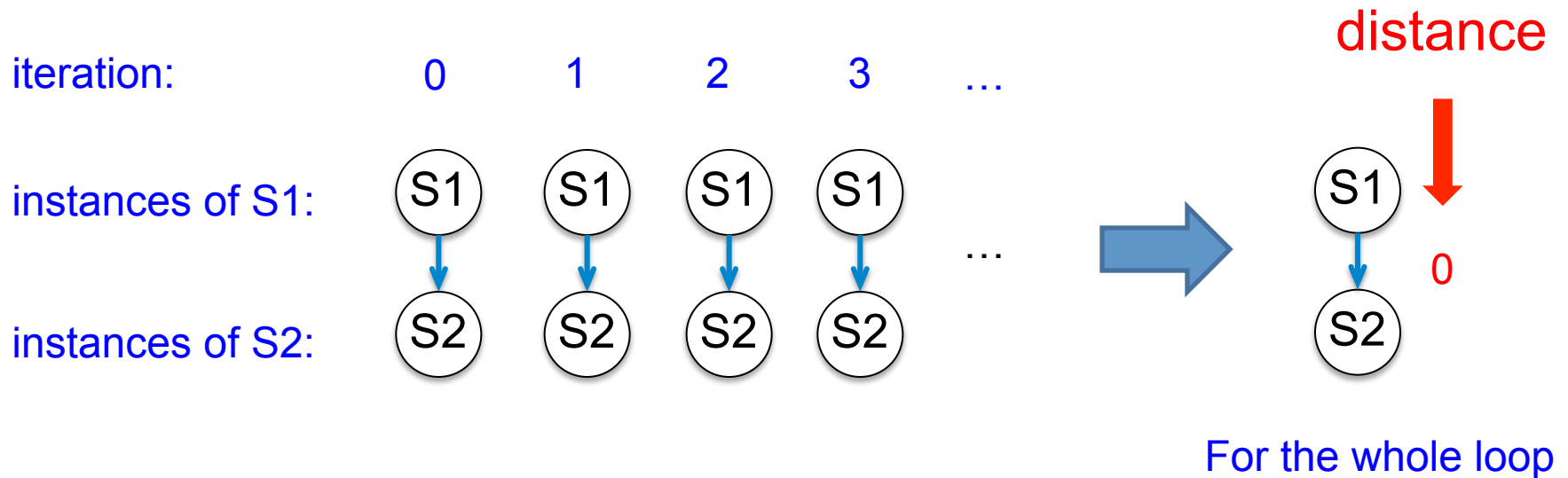
instances of S2:

For the whole loop

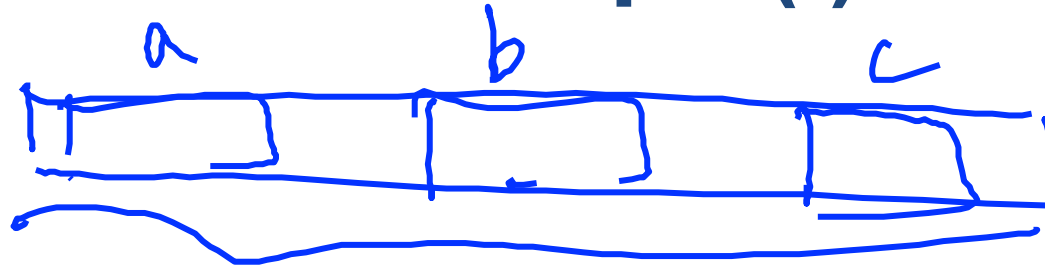


Dependences in Loops (I)

```
for (i=0; i<n; i++){  
  S1  a[i] = b[i] + 1;  
  S2  c[i] = a[i] + 2;  
}
```



Dependences in Loops (I)



```
for (i=0; i<n; i++){  
S1  a[i] = b[i] + 1;  
S2  c[i] = a[i] + 2;  
}
```


Memory.

For the dependences shown here, we assume that arrays do not overlap in memory (no aliasing). Compilers must know that there is no aliasing in order to vectorize.



Dependences in Loops (II)

```
    for (i=1; i<n; i++){  
S1    a[i] = b[i] + 1;  
S2    c[i] = a[i-1] + 2;  
    }
```



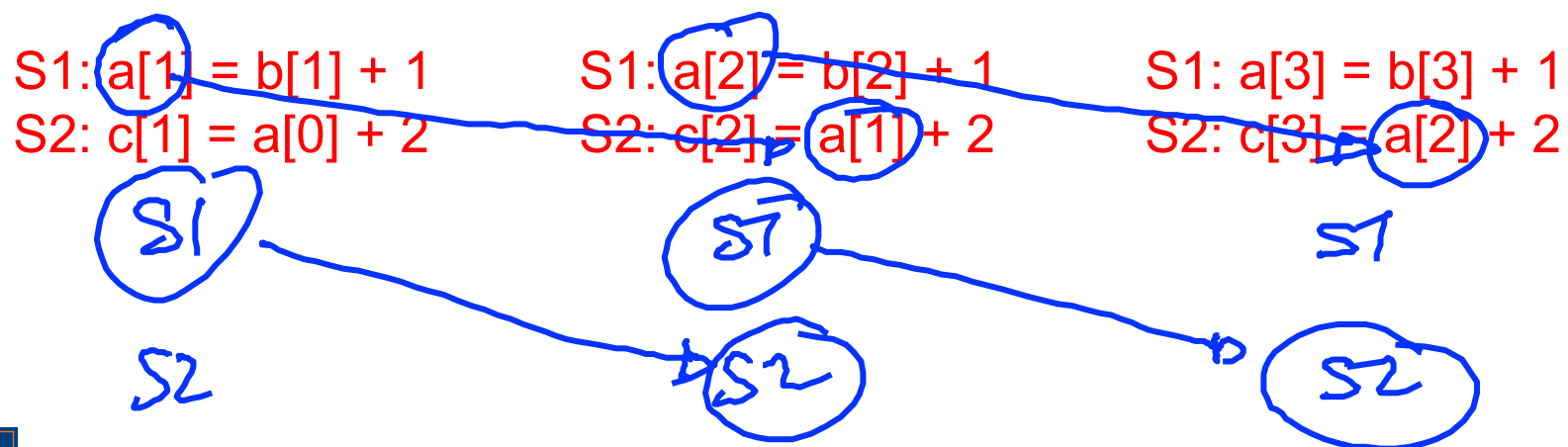
Dependences in Loops (II)

```
    for (i=1; i<n; i++){  
      S1  a[i] = b[i] + 1;  
      S2  c[i] = a[i-1] + 2;  
    }
```

i=1

i=2

i=3



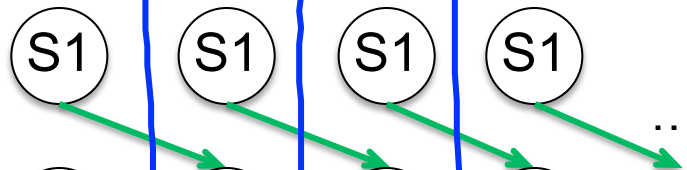
Dependences in Loops (II)

```
for (i=1; i<n; i++){  
  S1  a[i] = b[i] + 1;  
  S2  c[i] = a[i-1] + 2;  
}
```

iteration:

1 2 3 4 ...

instances of S1:



instances of S2:

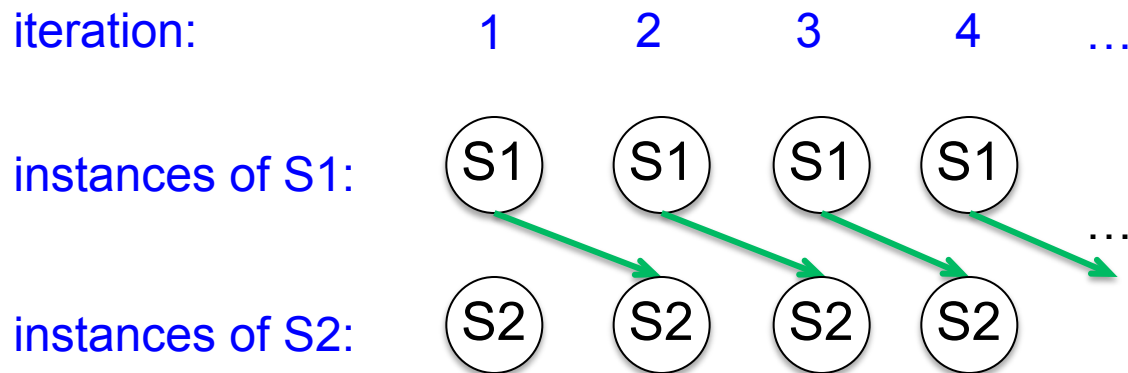


loop carried dependence



Dependences in Loops (II)

```
for (i=1; i<n; i++){  
  S1  a[i] = b[i] + 1;  
  S2  c[i] = a[i-1] + 2;  
}
```

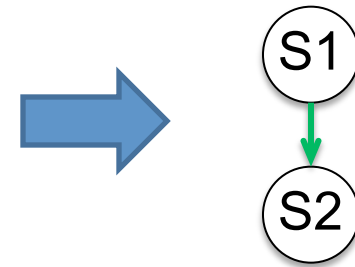
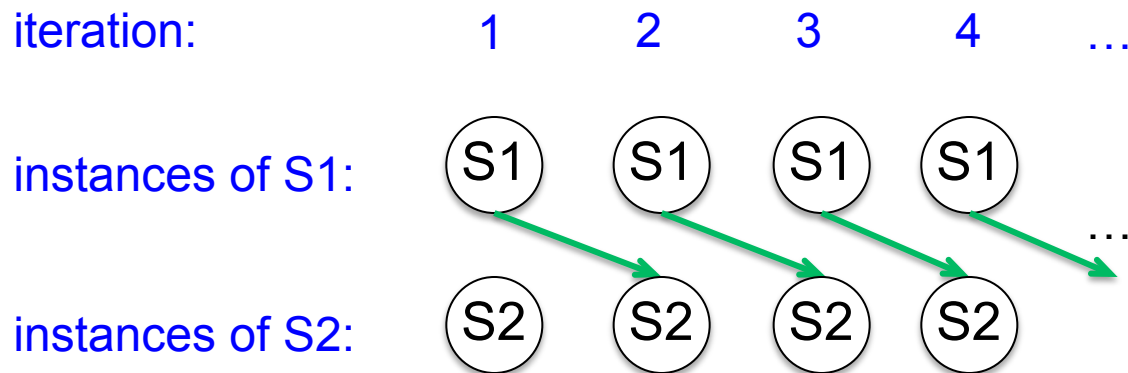


→ Loop carried dependence



Dependences in Loops (II)

```
for (i=1; i<n; i++){  
  S1  a[i] = b[i] + 1;  
  S2  c[i] = a[i-1] + 2;  
}
```

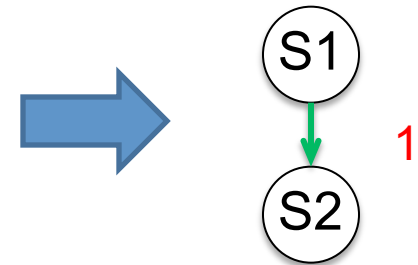
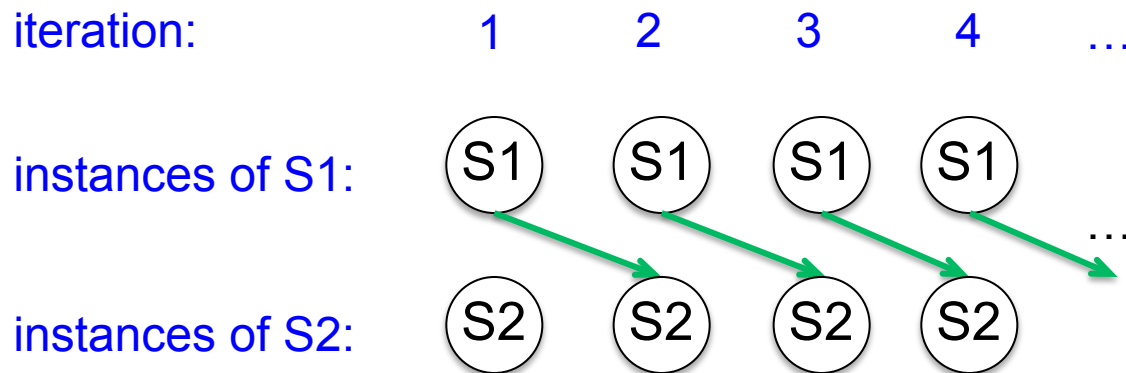


For the whole loop



Dependences in Loops (II)

```
for (i=1; i<n; i++){  
  S1  a[i] = b[i] + 1;  
  S2  c[i] = a[i-1] + 2;  
}
```



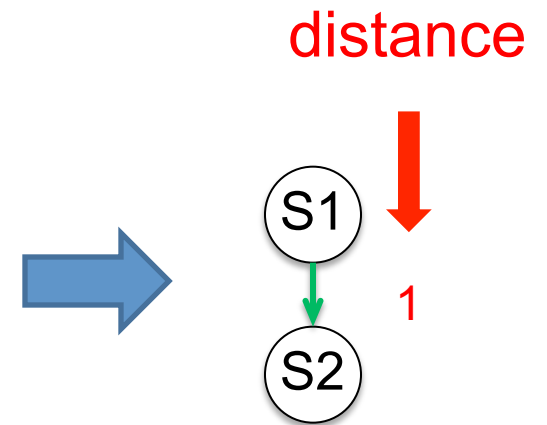
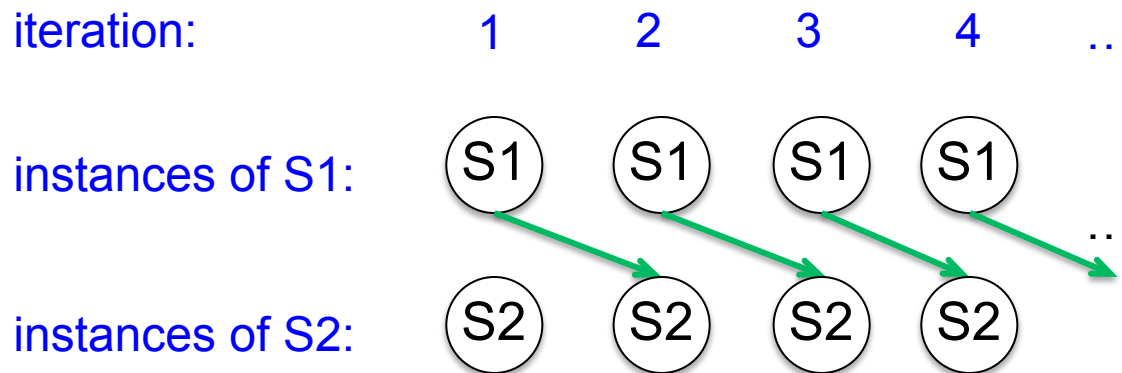
For the whole loop



Dependences in Loops (II)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement “executions”

```
for (i=1; i<n; i++){  
  S1  a[i] = b[i] + 1;  
      c[i] = a[i-1] + 2;  
}
```



For the whole loop



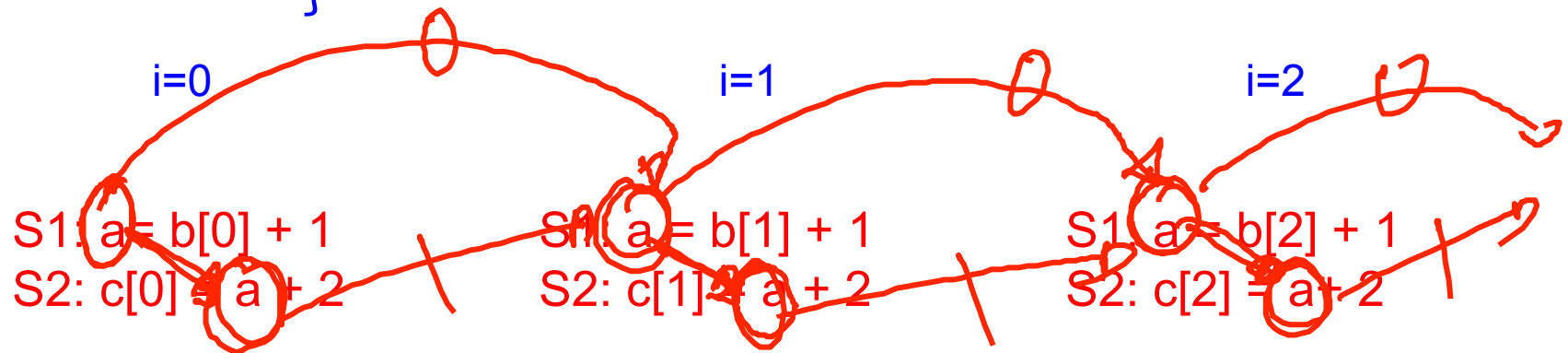
Dependences in Loops (III)

```
for (i=0; i<n; i++){  
S1   a = b[i] + 1;  
S2   c[i] = a + 2;  
}
```



Dependences in Loops (III)

```
for (i=0; i<n; i++){  
  S1  a = b[i] + 1;  
  S2  c[i] = a + 2;  
}
```



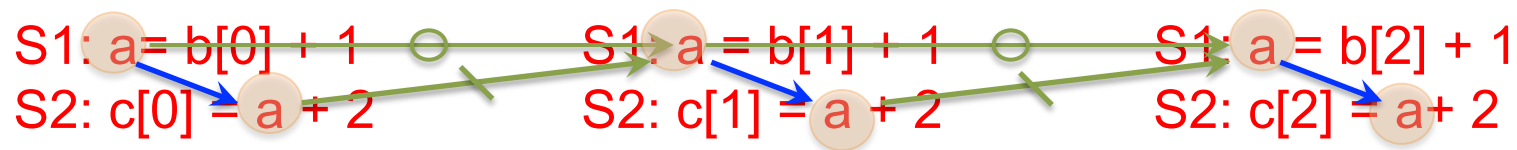
Dependences in Loops (III)

```
for (i=0; i<n; i++){  
  S1    a = b[i] + 1;  
  S2    c[i] = a + 2;  
}
```

i=0

i=1

i=2

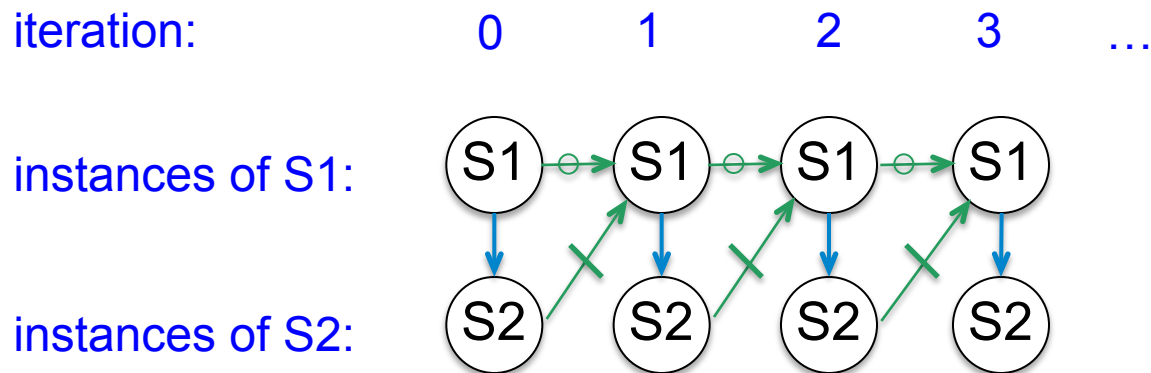


→ Loop independent dependence
→ Loop carried dependence



Dependences in Loops (III)

```
for (i=0; i<n; i++){  
  S1    a = b[i] + 1;  
  S2    c[i] = a + 2;  
}
```



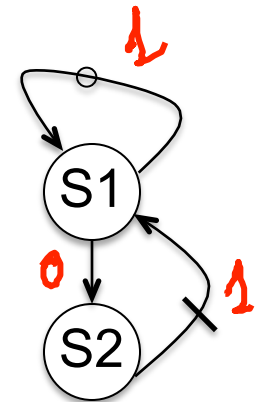
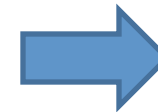
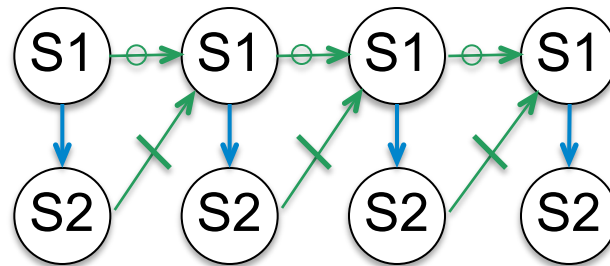
Dependences in Loops (III)

```
for (i=0; i<n; i++){  
  S1    a = b[i] + 1;  
  S2    c[i] = a + 2;  
}
```

iteration: 0 1 2 3 ...

instances of S1:

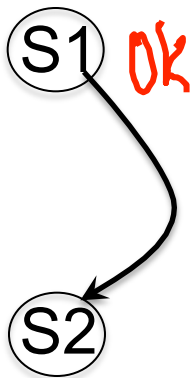
instances of S2:



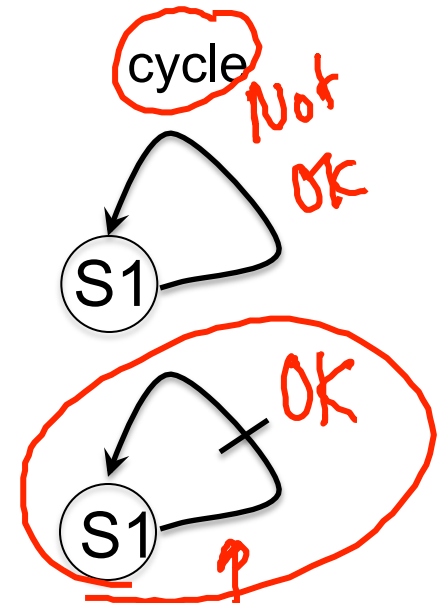
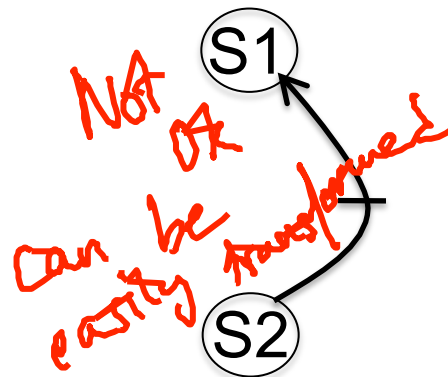
Loop Vectorization

- Loop Vectorization is not always a legal transformation.
 1. Compilers can vectorize when there are only forward dependences
 2. Compilers cannot vectorize when there is a cycle in the data dependences (with the exception of self-antidependence), unless a transformation is applied to remove the cycle
 3. Codes with only backward dependences can be vectorized, but need to be transformed

forward
dependence



backward
dependence



Simple Example

- Loop vectorization transforms a program so that the same operation is performed at the same time on several vector elements

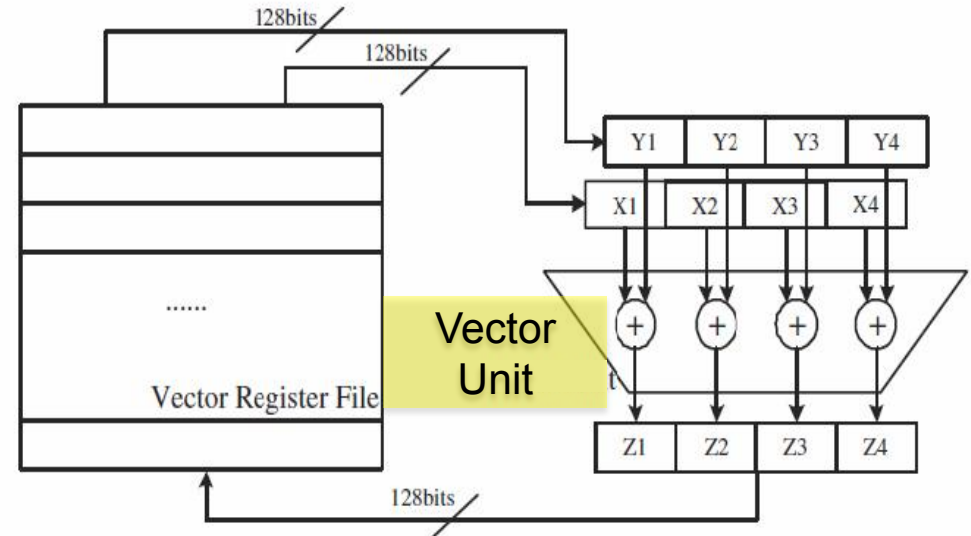
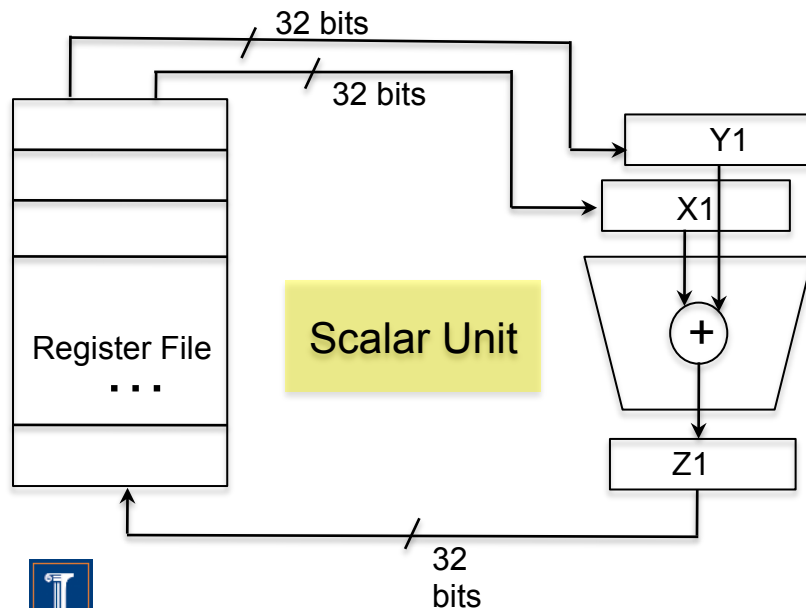
n times

```
lw $t0, 0($a0)
lw $t1, 0($a1)
add $t3, $t0, $t1
sw $t3, 0($a3)
```

```
for (i=0; i<n; i++)
    c[i] = a[i] + b[i];
```

$n/4$ times

```
lww $vt0, 0($a0)
lww $vt1, 0($a1)
addv $vt3, $vt0, $vt1
swv %vt3, $0($a3)
```



Loop Vectorization

- When vectorizing a loop with several statements the compiler need to strip-mine the loop and then apply loop distribution

①

```
for (i=0; i<LEN; i++){
  S1 a[i]=b[i]+(float)1.0;
  S2 c[i]=b[i]+(float)2.0;
}
```

②

Strip mine

```
for (i=0; i<LEN; i+=strip_size){
  for (j=i; j<i+strip_size; j++)
    a[j]=b[j]+(float)1.0;
    c[j]=b[j]+(float)2.0;
}
```



S1 S1 S1 S1

S2 S2 S2 S2

```
for (i=0; i<LEN; i+=strip_size){
  for (j=i; j<i+strip_size; j++)
    a[j]=b[j]+(float)1.0;
  for (j=i; j<i+strip_size; j++)
    c[j]=b[j]+(float)2.0;
}
```

Distributed



Loop Vectorization

- When vectorizing a loop with several statements the compiler needs to strip-mine the loop and then apply loop distribution

```
for (i=0; i<LEN; i++){  
S1 a[i]=b[i]+(float)1.0;  
S2 c[i]=b[i]+(float)2.0;  
}  
→  
for (i=0; i<LEN; i+=strip_size){  
  for (j=i; j<i+strip_size; j++)  
    a[j]=b[j]+(float)1.0;  
  for (j=i; j<i+strip_size; j++)  
    c[j]=b[j]+(float)2.0;  
}
```

i=0 i=1 i=2 i=3 i=4 i=5 i=6 i=7

(S1) (S1) (S1) (S1) (S1) (S1) (S1) (S1)

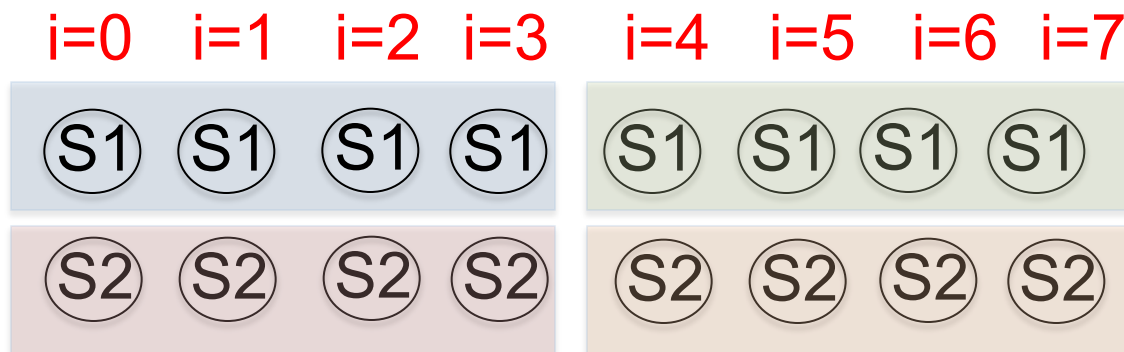
(S2) (S2) (S2) (S2) (S2) (S2) (S2) (S2)



Loop Vectorization

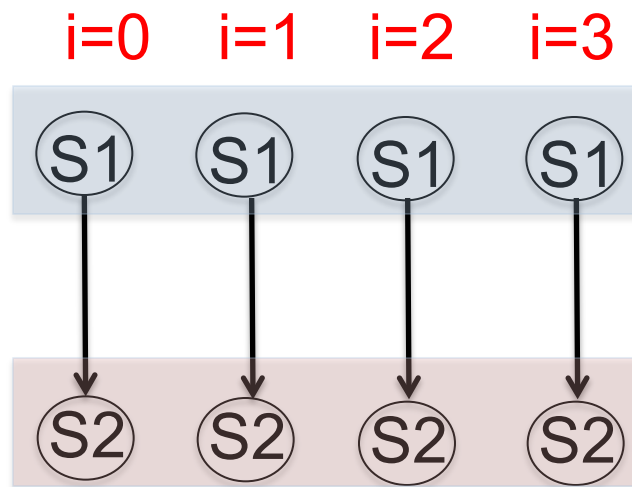
- When vectorizing a loop with several statements the compiler needs to strip-mine the loop and then apply loop distribution

```
for (i=0; i<LEN; i++){  
S1 a[i]=b[i]+(float)1.0;  
S2 c[i]=b[i]+(float)2.0;  
}  
→  
for (i=0; i<LEN; i+=strip_size){  
  for (j=i; j<i+strip_size; j++)  
    a[j]=b[j]+(float)1.0;  
  for (j=i; j<i+strip_size; j++)  
    c[j]=b[j]+(float)2.0;  
}
```

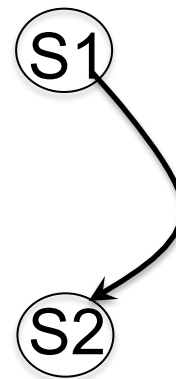


Acyclic Dependence Graphs: Forward Dependences

```
for (i=0; i<LEN; i++) {  
S1 a[i] = b[i] + c[i]  
S2 d[i] = a[i] + (float) 1.0;  
}
```



ok



forward
dependence



Acyclic Dependence Graphs: Forward Dependences

```
for (i=0; i<LEN; i++) {  
    a[i]= b[i] + c[i]  
    d[i] = a[i] + (float) 1.0;  
}
```

Intel Nehalem

Compiler report: Loop was
vectorized

Exec. Time scalar code: 10.2

Exec. Time vector code: 6.3

Speedup: 1.6

IBM Power 7

Compiler report: Loop was SIMD
vectorized

Exec. Time scalar code: 3.1

Exec. Time vector code: 1.5

Speedup: 2.0



Acyclic Dependence Graphs

Backward Dependences (I)

```
for (i=0; i<LEN; i++) {  
S1  a[i]= b[i] + c[i]  
S2  d[i] = a[i+1] + (float) 1.0;  
}
```

i=0 { S1: a[0] = b[0] + c[0]
S2: d[0] = a[1] + 1

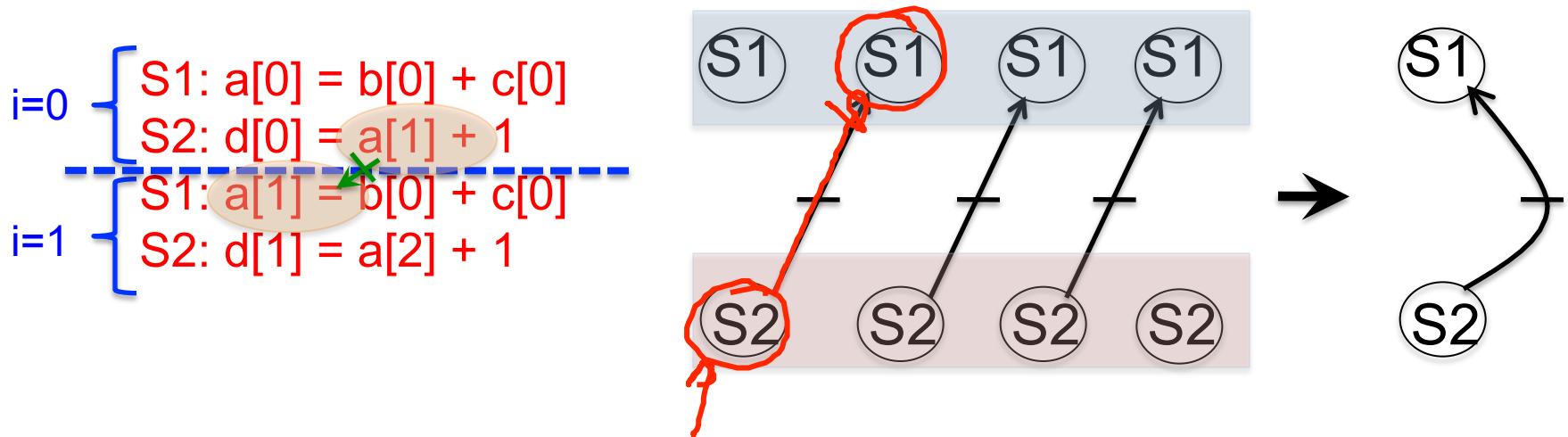
i=1 { S1: a[1] = b[0] + c[0]
S2: d[1] = a[2] + 1



Acyclic Dependenden Graphs

Backward Dependences (I)

```
for (i=0; i<LEN; i++) {  
S1  a[i]= b[i] + c[i]  
S2  d[i] = a[i+1] + (float) 1.0;  
}
```



This loop cannot be vectorized as it is

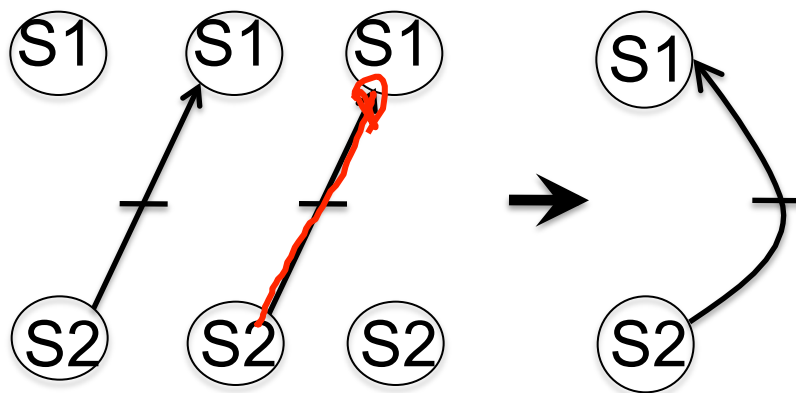


Acyclic Dependenden Graphs

Backward Dependences (I)

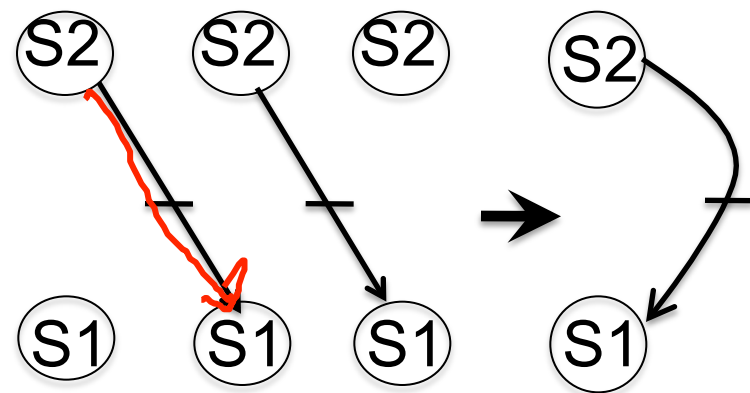
Reorder of statements

```
for (i=0; i<LEN; i++) {
S1  a[i]= b[i] + c[i]
S2  d[i] = a[i+1] + (float) 1.0;
}
```



backward
depedence

```
for (i=0; i<LEN; i++) {
S2  d[i] = a[i+1]+(float)1.0;
S1  a[i]= b[i] + c[i];
}
```



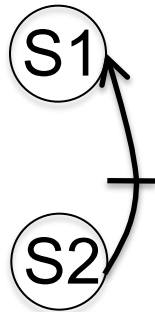
forward
depedence



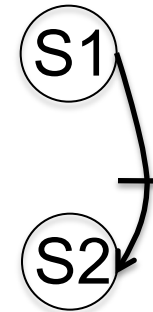
Acyclic Dependenden Graphs

Backward Dependences (I)

```
for (i=0; i<LEN; i++) {  
    a[i]= b[i] + c[i];  
    d[i] = a[i+1]+(float)1.0;  
}
```



```
for (i=0; i<LEN; i++) {  
    d[i] = a[i+1]+(float)1.0;  
    a[i]= b[i] + c[i];  
}
```



Intel Nehalem

Compiler report: Loop was not vectorized. Existence of vector dependence

Exec. Time scalar code: 12.6

Exec. Time vector code: --

Speedup: --

Intel Nehalem

Compiler report: Loop was vectorized

Exec. Time scalar code: 10.7

Exec. Time vector code: 6.2

Speedup: 1.72

Speedup vs non-reordered code: 2.03

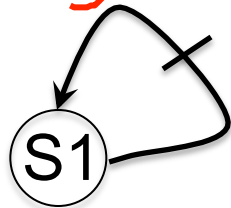


Cycles in the DG (III)

```
for (int i=0; i<LEN-1; i++){  
S1 a[i]=a[i+1]+b[i];  
}
```

S1
S1
S1
S1

$a[0] = a[1] + b[0]$
 $a[1] = a[2] + b[1]$
 $a[2] = a[3] + b[2]$
 $a[3] = a[4] + b[3]$



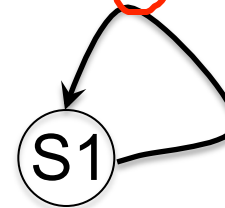
Ok

Self-antidependence
can be vectorized

```
for (int i=1; i<LEN; i++){  
S1 a[i]=a[i-1]+b[i];  
}
```

S1
S1

$a[1] = a[0] + b[1]$
 $a[2] = a[1] + b[2]$
 $a[3] = a[2] + b[3]$
 $a[4] = a[3] + b[4]$



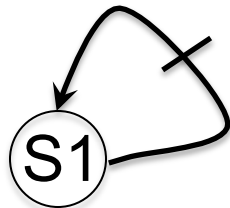
Not
Ok

Self true-dependence
can not vectorized
(as it is)

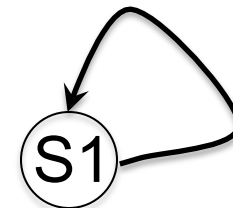


Cycles in the DG (III)

```
for (int i=0;i<LEN-1;i++){  
S1  a[i]=a[i+1]+b[i];  
}
```



```
for (int i=1;i<LEN;i++){  
S1  a[i]=a[i-1]+b[i];  
}
```



Intel Nehalem

Compiler report: Loop was vectorized

Exec. Time scalar code: 6.0

Exec. Time vector code: 2.7

Speedup: 2.2

Intel Nehalem

Compiler report: Loop was not vectorized. Existence of vector dependence

Exec. Time scalar code: 7.2

Exec. Time vector code: --

Speedup: --

