



Advanced Parallel Architecture

Lesson 10



Annalisa Massini - 2016/2017

Instruction-Level Parallelism

Hennessy, Patterson

Computer architecture A quantitative approach

Section 3.1

Instruction-Level Parallelism

- ▶ All processors since about 1985 use pipelining to overlap the execution of instructions and improve performance
- ▶ This potential overlap among instructions is called *instruction-level parallelism* (ILP), since the instructions can be evaluated in parallel
- ▶ There is a wide range of techniques for extending the basic pipelining concepts by increasing the amount of parallelism exploited among instructions

Instruction-Level Parallelism

- ▶ There are two largely separable approaches to exploiting ILP:
 - ▶ an approach that relies on hardware to help discover and exploit the parallelism dynamically
 - ▶ an approach that relies on software technology to find parallelism statically at compile time
- ▶ In the past few years, many of the techniques developed for one approach have been exploited within a design relying primarily on the other

Instruction-Level Parallelism

- ▶ It is interesting to discuss:
 - ▶ features of both **programs** and **processors** that limit the amount of parallelism that can be exploited among instructions,
 - ▶ the critical mapping between program structure and hardware structure, which is key to understanding whether a program property will actually limit performance and under what circumstances
- ▶ The CPI (cycles per instruction) for a pipelined processor is the sum of the base CPI and contributions from stalls:

Pipeline CPI = Ideal pipeline CPI +

+ Structural stalls + Data hazard stalls + Control stalls

Instruction-Level Parallelism

- ▶ The **amount of parallelism** available within a ***basic block*** - *code sequence with no branches in except to the entry and no branches out except at the exit* - **is quite small**
- ▶ To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks
- ▶ The simplest and most common way to increase the ILP is to **exploit parallelism among iterations of a loop**

Instruction-Level Parallelism

- ▶ This type of parallelism is often called *loop-level parallelism*
- ▶ Example: loop that adds two 1000-element arrays

```
for (i=0; i<=999; i=i+1)
    x[i] = x[i] + y[i];
```
- ▶ It is completely parallel
- ▶ Every iteration of the loop can overlap with any other iteration

Instruction-Level Parallelism

- ▶ Techniques for converting loop-level parallelism into instruction-level parallelism work by **unrolling the loop** either statically (compiler) or dynamically (hardware)
- ▶ An alternative method for exploiting loop-level parallelism is the use of SIMD in both vector processors and GPUs
 - ▶ A SIMD instruction exploits data-level parallelism by operating on a small to moderate number of data items in parallel (typically two to eight)
 - ▶ A vector instruction exploits data-level parallelism by operating on many data items in parallel using both parallel execution units and a deep pipeline

Instruction-Level Parallelism

- ▶ Example: loop that adds two 1000-element arrays

```
for (i=0; i<=999; i=i+1)
    x[i] = x[i] + y[i];
```
- ▶ This code sequence, requires seven instructions per iteration (two loads, an add, a store, two address updates, and a branch) for a total of 7000 instructions
 - ▶ It might execute in one-quarter as many instructions in a **SIMD architecture** where four data items are processed per instruction
 - ▶ On **vector processors**, this sequence might take only four instructions: two instructions to load the vectors x and y from memory, one instruction to add the two vectors, and an instruction to store back the result vector

Data Dependences and Hazards

- ▶ Determining how one instruction depends on another is **critical** to determining how much parallelism exists in a program and how that parallelism can be exploited
- ▶ To exploit instruction-level parallelism we must ***determine which instructions can be executed in parallel***
 - ▶ If two instructions are *parallel*, they can execute simultaneously in a pipeline of arbitrary depth without causing any stalls, assuming the pipeline has sufficient resources (and hence no structural hazards exist)
 - ▶ If two instructions are dependent, they are not parallel and must be executed in order
- ▶ The key is to determine *whether an instruction is dependent on another instruction*

Data Dependences and Hazards

- ▶ There are three different types of dependences:
 - ▶ *data dependences* (also called **true data dependences**)
 - ▶ *name dependences*
 - ▶ *control dependences*
- ▶ An instruction j is *data dependent* on instruction i if either of the following holds:
 - ▶ Instruction i produces a result that may be used by instruction j
 - ▶ Instruction j is data dependent on instruction k , and instruction k is data dependent on instruction i

Data Dependences

- ▶ The **second condition** simply states that one instruction is dependent on another if there exists a ***chain of dependences*** of the first type between the two instructions
- ▶ This dependence chain can be as long as the entire program
- ▶ Note that a dependence within a single instruction (such as `ADDD R1,R1,R1`) is not considered a dependence

Data Dependences

- ▶ Example: MIPS code sequence that increments a vector of values in memory (starting at 0(R1) and with the last element at 8(R2)) by a scalar in register F2

```
Loop: L.D F0,0(R1)      ;F0=array element
      ADD.D F4,F0,F2    ;add scalar in F2
      S.D F4,0(R1)     ;store result
      DADDUI R1,R1,#-8 ;decrement pointer 8 bytes
      BNE R1,R2,LOOP   ;branch R1!=R2
```

Data Dependences


▶ Example

```
Loop: L.D F0,0(R1)      ;F0=array element
      ADD.D F4,F0,F2    ;add scalar in F2
      S.D F4,0(R1)     ;store result
      DADDUI R1,R1,#-8 ;decrement pointer 8 bytes
      BNE R1,R2,LOOP   ;branch R1!=R2
```

The data dependences in this code sequence involve:


floating-point data

```
Loop: L.D F0,0(R1)
      ;F0=array element
      ADD.D F4,F0,F2
      ;add scalar in F2
      S.D F4,0(R1)
      ;store result
```



integer data

```
DADDUI R1,R1,#-8
      ;decrement pointer
      ;8 bytes (per DW)
      BNE R1,R2,Loop
      ;branch R1!=R2
```



Data Dependences

- ▶ In both of the above dependent sequences, as shown by the arrows, each instruction depends on the previous one
- ▶ The arrows show the order that must be preserved for correct execution
- ▶ The arrow points from an instruction that must precede the instruction that the arrowhead points to
- ▶ If two instructions are data dependent, they must execute in order and cannot execute simultaneously or be completely overlapped

Data Dependences

- ▶ The **dependence** implies that there would be a **chain of one or more data hazards** between the two instructions
- ▶ Dependences are a property of *programs*
- ▶ Whether a given dependence results in an **actual hazard** being detected and whether that hazard actually causes a **stall** are properties of the *pipeline organization*
- ▶ This difference is critical to understanding how instruction-level parallelism can be exploited

Data Dependences

- ▶ A dependence can be overcome in two different ways:
 - ▶ maintaining the dependence but avoiding a hazard
 - ▶ eliminating a dependence by transforming the code
- ▶ Scheduling the code is the primary method used to avoid a hazard without altering a dependence, and such scheduling can be done both by the compiler and by the hardware

Data Dependences

- ▶ A data value may flow between instructions either through **registers** or through **memory locations**
- ▶ When the data flow occurs in a register, detecting the dependence
 - ▶ **Is straightforward** since the register names are fixed in the instructions
 - ▶ Gets more complicated when branches intervene and correctness concerns force a compiler or hardware to be conservative

Data Dependences

- ▶ Dependences that flow through memory locations are more difficult to detect
- ▶ Two addresses may refer to the same location but look different
 - ▶ For example, $100(R4)$ and $20(R6)$ may be identical memory addresses
- ▶ In addition, the effective address of a load or store may change from one execution of the instruction to another (so that $20(R4)$ and $20(R4)$ may be different), further complicating the detection of a dependence

Name Dependences

- ▶ The second type of dependence is a ***name dependence***
- ▶ A name dependence occurs when two instructions use the same register or memory location, ***name***, but there is no flow of data between instructions associated with that name
- ▶ There are two types of name dependences between an instruction i that *precedes* instruction j in program order:
 - ▶ An ***antidependence*** between instruction i and instruction j occurs when instruction j writes a register or memory location that instruction i reads. The **original ordering** must be preserved **to ensure that i reads the correct value**
 - ▶ An ***output dependence*** occurs when instruction i and instruction j write the same register or memory location. The **ordering** between the instructions must be preserved **to ensure that the value finally written corresponds to instruction j**

Name Dependences

- ▶ Both **antidependences** and **output dependences** are name dependences, as opposed to *true data dependences*, since there is ***no value being transmitted between the instructions***
- ▶ **Solution** → Instructions involved in a name dependence can **execute simultaneously** or **be reordered**, if the ***name used in the instructions is changed so the instructions do not conflict***
- ▶ **Renaming** can be more easily done for **register operands**, where it is called ***register renaming***, and can be done either statically (compiler) or dynamically (hardware)

Data Hazards

- ▶ A hazard exists whenever:
 - ▶ there is a **name** or **data dependence** between instructions
 - ▶ instructions are close enough that the overlap during execution would change the order of access to the operand involved in the dependence
- ▶ Because of the dependence, we must preserve the ***program order*** → *order that the instructions would execute in if executed sequentially one at a time as determined by the original source program*
- ▶ The goal of both software and hardware techniques is to exploit parallelism by preserving program order ***only where it affects the outcome of the program***

Data Hazards

- ▶ **Detecting and avoiding hazards** ensures that necessary program order is preserved
- ▶ Data hazards may be classified depending on the order of read and write accesses in the instructions
- ▶ By convention, the hazards are named by the **ordering in the program that must be preserved by the pipeline**
- ▶ Consider **two instructions i and j** , with i preceding j in program order

Data Hazards

RAW (read after write)

Two instructions i and j , with i preceding j in program order

- ▶ j tries to read a source before i writes it, so j incorrectly gets the *old* value
- ▶ This hazard is the most common type and corresponds to a **true data dependence**
- ▶ Program order must be preserved to ensure that j receives the value from i

Data Hazards

WAW (write after write)

Two instructions i and j , with i preceding j in program order

- ▶ j tries to write an operand before it is written by i
- ▶ The writes end up being performed in the wrong order, leaving the value written by i rather than the value written by j in the destination
- ▶ This hazard corresponds to an **output dependence**
- ▶ WAW hazards are present only in pipelines that write in more than one pipe stage or allow an instruction to proceed even when a previous instruction is stalled

Data Hazards

WAR (write after read)

Two instructions i and j , with i preceding j in program order

- ▶ j tries to write a destination before it is read by i , so i incorrectly gets the *new* value
- ▶ A WAR hazard arises from an **antidependence**
- ▶ WAR hazards
 - ▶ cannot occur in most static issue pipelines —deeper pipelines or floating-point pipelines → reads are early and writes are late
 - ▶ occurs either when there are some instructions that write results early in the instruction pipeline **and** other instructions that read a source late in the pipeline, or when instructions are

Control Dependences

- ▶ A **control dependence** determines the ordering of an instruction i with respect to a **branch instruction** so that instruction i is executed in correct program order and only when it should be
- ▶ Every instruction, except for those in the first basic block of the program, is control dependent on some set of branches, and, in general, these control dependences must be preserved to preserve program order
- ▶ One of the simplest examples of a control dependence is the dependence of the statements in the **then** part of an if statement on the branch

Data Dependences and Hazards

Control dependences

- ▶ In general, two constraints are imposed by control dependences:
 - ▶ An instruction that is **control dependent** on a branch cannot be moved ***before*** the branch so that its execution ***is no longer controlled*** by the branch. For example, we cannot take an instruction from the then portion of an if statement and move it before the if statement
 - ▶ An instruction that is not control dependent on a branch cannot be moved ***after*** the branch so that its execution ***is controlled*** by the branch. For example, we cannot take a statement before the if statement and move it into the ***then*** portion

Loop-Level Parallelism

Hennessy, Patterson

Computer architecture A quantitative approach

Section 4.5

Detecting and Enhancing Loop-Level Parallelism

- ▶ **Loops** in programs are the fountainhead of **many of the types of parallelism**
- ▶ **Compiler technology** allows discovering the amount of parallelism that we can exploit in a program as well as hardware support for these compiler techniques
- ▶ We can define:
 - ▶ when a loop is parallel (or vectorizable)
 - ▶ how dependence can prevent a loop from being parallel
 - ▶ techniques for eliminating some types of dependences
- ▶ Finding and manipulating loop-level parallelism is critical to exploiting both DLP and TLP, as well as the more aggressive static ILP approaches

Detecting and Enhancing Loop-Level Parallelism

- ▶ Loop-level parallelism is normally analyzed at the **source level** or close to it, while **most analysis of ILP** is done once instructions have been **generated by the compiler**
- ▶ **Loop-level analysis** involves determining what *dependences exist among the operands in a loop across the iterations of that loop*
- ▶ We now consider only **data dependences**, which arise when an operand is written at some point and read at a later point
- ▶ **Name dependences** also exist and may be removed by the renaming techniques

Detecting and Enhancing Loop-Level Parallelism

- ▶ The **analysis of loop-level parallelism** focuses on determining whether *data accesses in later iterations are dependent on data values produced in earlier iterations*
- ▶ Such dependence is called a ***loop-carried dependence***
- ▶ Examples that have no loop-carried dependences are ***loop-level parallel***

Detecting and Enhancing Loop-Level Parallelism

- ▶ To see that a loop is parallel, let us first look at the source representation:

```
for (i=999; i>=0; i=i-1)
    x[i] = x[i] + s;
```

- ▶ In this loop, the two uses of **x[i]** are dependent, but this dependence is within a single iteration and is not loop carried
- ▶ There is a **loop-carried dependence between successive uses of i in different iterations**
- ▶ This dependence involves an induction variable that can be easily recognized and eliminated

Detecting and Enhancing Loop-Level Parallelism

- ▶ Dependences involving induction variables can be eliminated by loop unrolling
- ▶ Finding loop-level parallelism involves recognizing structures such as:
 - ▶ Loops
 - ▶ Array references
 - ▶ Induction variable computations
- ▶ The compiler can do this analysis more easily at or near the source level, as opposed to the machine-code level

Detecting and Enhancing Loop-Level Parallelism

▶ Example

```
for (i=0; i<100; i=i+1) {  
    A[i+1] = A[i] + C[i];    /* S1 */  
    B[i+1] = B[i] + A[i+1]; /* S2 */  
}
```

- ▶ Assume that A, B, and C are distinct, nonoverlapping arrays
- ▶ What are the data dependences among the statements S1 and S2 in the loop?

Detecting and Enhancing Loop-Level Parallelism

▶ Example

```
for (i=0; i<100; i=i+1) {  
    A[i+1] = A[i] + C[i];    /* S1 */  
    B[i+1] = B[i] + A[i+1]; /* S2 */  
}
```

▶ There are two different dependences:

- ▶ S1 uses a value computed by S1 in an earlier iteration, since iteration i computes $A[i+1]$, which is read in iteration $i+1$. The same is true of S2 for $B[i]$ and $B[i+1]$

Detecting and Enhancing Loop-Level Parallelism

▶ Example

```
for (i=0; i<100; i=i+1) {  
    A[i+1] = A[i] + C[i];    /* S1 */  
    B[i+1] = B[i] + A[i+1]; /* S2 */  
}
```

▶ There are two different dependences:

- ▶ S1 uses a value computed by S1 in an earlier iteration, since iteration i computes $A[i+1]$, which is read in iteration $i+1$. The same is true of S2 for $B[i]$ and $B[i+1]$
- ▶ S2 uses the value $A[i+1]$ computed by S1 in the same iteration

▶ These two dependences are different and have different effects

Detecting and Enhancing Loop-Level Parallelism

▶ Example

```
for (i=0; i<100; i=i+1) {  
    A[i+1] = A[i] + C[i];    /* S1 */  
    B[i+1] = B[i] + A[i+1]; /* S2 */  
}
```

- ▶ To see how they differ, let's assume that only one of these dependences exists at a time
- ▶ Because the dependence of statement S1 is on an earlier iteration of S1, this dependence is *loop carried*
- ▶ This dependence forces successive iterations of this loop to execute in series

Detecting and Enhancing Loop-Level Parallelism

▶ Example

```
for (i=0; i<100; i=i+1) {  
    A[i+1] = A[i] + C[i];    /* S1 */  
    B[i+1] = B[i] + A[i+1]; /* S2 */  
}
```

- ▶ The second dependence (S2 depending on S1) is within an iteration and is not loop carried
- ▶ Thus, if this were the only dependence, *multiple iterations of the loop could execute in parallel*, as long as each pair of statements in an iteration were kept in order

Detecting and Enhancing Loop-Level Parallelism

- ▶ Example 2 - It is also possible to have a loop-carried dependence that does not prevent parallelism

```
for (i=0; i<100; i=i+1) {  
    A[i] = A[i] + B[i];      /* S1 */  
    B[i+1] = C[i] + D[i];   /* S2 */  
}
```

- ▶ What are the dependences between S1 and S2?
- ▶ Is this loop parallel?
- ▶ If not, show how to make it parallel

Detecting and Enhancing Loop-Level Parallelism

▶ Example 2

```
for (i=0; i<100; i=i+1) {  
    A[i] = A[i] + B[i];      /* S1 */  
    B[i+1] = C[i] + D[i];   /* S2 */  
}
```

- ▶ Statement *S1* uses the value assigned in the previous iteration by statement *S2*, so there is a **loop-carried dependence** between *S2* and *S1*
- ▶ But *this loop can be made parallel*
- ▶ Unlike the earlier loop, this dependence is not circular; neither statement depends on itself, and although *S1* depends on *S2*, *S2* does not depend on *S1*

Detecting and Enhancing Loop-Level Parallelism

▶ Example 2

```
for (i=0; i<100; i=i+1) {  
    A[i] = A[i] + B[i];    /* S1 */  
    B[i+1] = C[i] + D[i]; /* S2 */  
}
```

- ▶ A loop is parallel if it can be written *without a cycle in the dependences*, since the absence of a cycle means that the dependences give a *partial ordering* on the statements
- ▶ Although there are no circular dependences in the above loop, *it must be transformed to conform to the partial ordering* and expose the parallelism

Detecting and Enhancing Loop-Level Parallelism

▶ Example 2

```
for (i=0; i<100; i=i+1) {  
    A[i] = A[i] + B[i];      /* S1 */  
    B[i+1] = C[i] + D[i];   /* S2 */  
}
```

- ▶ These two observations allow us to replace the loop above with the following code sequence:

```
A[0] = A[0] + B[0];  
for (i=0; i<99; i=i+1) {  
    B[i+1] = C[i] + D[i];  
    A[i+1] = A[i+1] + B[i+1];  
}  
B[100] = C[99] + D[99];
```

Detecting and Enhancing Loop-Level Parallelism

- ▶ We can observe that the analysis needs to begin by finding all loop-carried dependences
- ▶ This dependence information can be *inexact*, in the sense that it tells us that such dependence *may* exist
- ▶ Consider the following example:

```
for (i=0;i<100;i=i+1) {  
    A[i] = B[i] + C[i]  
    D[i] = A[i] * E[i]  
}
```

Detecting and Enhancing Loop-Level Parallelism

▶ Example:

```
for (i=0;i<100;i=i+1) {  
    A[i] = B[i] + C[i]  
    D[i] = A[i] * E[i]  
}
```

- ▶ The **second reference to A** in this example need not be translated to a **load instruction**, since we know that the value is computed and stored by the previous statement
- ▶ Hence, **the second reference to A** can simply be a reference to the register into which A was computed

Detecting and Enhancing Loop-Level Parallelism

▶ Example:

```
for (i=0;i<100;i=i+1) {  
    A[i] = B[i] + C[i]  
    D[i] = A[i] * E[i]  
}
```

- ▶ Performing this optimization requires knowing that the two references are *always* to the same memory address and that there is no intervening access to the same location
- ▶ Normally, data dependence analysis only tells that one reference *may* depend on another
 - ▶ A more complex analysis is required to determine that two references *must be* to the exact same address

Detecting and Enhancing Loop-Level Parallelism

- ▶ Example:

```
for (i=0;i<100;i=i+1) {  
    A[i] = B[i] + C[i]  
    D[i] = A[i] * E[i]  
}
```

- ▶ In this example, a simple version of this analysis suffices, since the two references are in the same basic block

Detecting and Enhancing Loop-Level Parallelism

Finding Dependences

- ▶ Finding the dependences in a program is important both:
 - ▶ To determine which loops might contain parallelism
 - ▶ To eliminate name dependences
- ▶ How does the compiler detect dependences in general?
- ▶ Nearly all dependence analysis algorithms work on the assumption that array indices are *affine*:
 - ▶ a one-dimensional array index is affine if it can be written in the form $a \times i + b$, where a and b are constants and i is the loop index variable

Detecting and Enhancing Loop-Level Parallelism

Finding Dependences

- ▶ Determining whether there is a dependence between two references to the same array in a loop *is equivalent* to determining whether *two affine functions can have the same value for different indices between the bounds of the loop*
- ▶ For example, suppose we have stored to an array element with index value $a \times i + b$ and loaded from the same array with index value $c \times i + d$, where i is the for-loop index variable that runs from m to n

Detecting and Enhancing Loop-Level Parallelism

Finding Dependences

- ▶ A dependence exists if two conditions hold:
 - ▶ There are two iteration indices, j and k , that are both within the limits of the for loop, that is $m \leq j \leq n$, $m \leq k \leq n$
 - ▶ The loop stores into an array element indexed by $a \times j + b$ and later fetches from that *same* array element when it is indexed by $c \times k + d$, that is $a \times j + b = c \times k + d$
- ▶ In general, we cannot determine whether dependence exists at compile time
- ▶ If a program contain primarily simple indices where a , b , c , and d are all constants, it is possible to devise reasonable compile time tests for dependence

Detecting and Enhancing Loop-Level Parallelism

Finding Dependences

- ▶ As an example, a simple and sufficient test for the absence of a dependence is the *greatest common divisor* (GCD) test
- ▶ It is based on the observation that if a loop-carried dependence exists, then $\text{GCD}(c, a)$ must divide $(d - b)$. (Recall that an integer, x , *divides* another integer, y , if we get an *integer quotient* when we do the division y/x and there is *no remainder*)

Detecting and Enhancing Loop-Level Parallelism

Finding Dependences

- ▶ **Example** Use the GCD test to determine whether dependences exist in the following loop:

```
for (i=0; i<100; i=i+1) {  
    x[2*i+3] = x[2*i] * 5.0;  
}
```

- ▶ **Answer** Given the values $a = 2$, $b = 3$, $c = 2$, and $d = 0$, then $\text{GCD}(a,c) = 2$, and $d - b = -3$
- ▶ Since 2 does not divide -3 , no dependence is possible
- ▶ The GCD test is sufficient to guarantee that no dependence exists; however, there are cases where the GCD test succeeds but no dependence exists

Detecting and Enhancing Loop-Level Parallelism

▶ Example

```
for (i=0; i<100; i=i+1) {  
    Y[i] = X[i] / c;    /* S1 */  
    X[i] = X[i] + c;    /* S2 */  
    Z[i] = Y[i] + c;    /* S3 */  
    Y[i] = c - Y[i];    /* S4 */  
}
```

- ▶ This loop has multiple types of dependences
- ▶ Find all the true dependences, output dependences, and antidependences, and eliminate the output dependences and antidependences by renaming

Detecting and Enhancing Loop-Level Parallelism

▶ Example

```
for (i=0; i<100; i=i+1) {  
    Y[i] = X[i] / c;    /* S1 */  
    X[i] = X[i] + c;    /* S2 */  
    Z[i] = Y[i] + c;    /* S3 */  
    Y[i] = c - Y[i];    /* S4 */  
}
```

- ▶ The following dependences exist among the four statements:
 - ▶ 1) There are **true dependences** from S1 to S3 and from S1 to S4 because of **Y[i]**. These are *not loop carried*, so they do not prevent the loop from being considered *parallel*. These dependences will **force S3 and S4 to wait for S1 to complete**

Detecting and Enhancing Loop-Level Parallelism

▶ Example

```
for (i=0; i<100; i=i+1) {  
    Y[i] = X[i] / c;    /* S1 */  
    X[i] = X[i] + c;    /* S2 */  
    Z[i] = Y[i] + c;    /* S3 */  
    Y[i] = c - Y[i];    /* S4 */  
}
```

- ▶ The following dependences exist among the four statements:
 - ▶ 2) There is an **antidependence** from S1 to S2, based on X[i]
 - ▶ 3) There is an **antidependence** from S3 to S4 for Y[i]
 - ▶ 4) There is an **output dependence** from S1 to S4, based on Y[i]

Detecting and Enhancing Loop-Level Parallelism

- ▶ The following version of the loop eliminates these false (or pseudo) dependences

```
for (i=0; i<100; i=i+1 {  
    T[i] = X[i] / c; /* Y renamed to T to remove  
                    output dependence */  
    X1[i] = X[i] + c; /* X renamed to X1 to remove  
                     antidependence */  
    Z[i] = T[i] + c; /* Y renamed to T to remove  
                     antidependence */  
    Y[i] = c - T[i];  
}
```

- ▶ After the loop, the variable X has been renamed X1
- ▶ In code that follows the loop, the compiler can simply replace the name X by X1

Detecting and Enhancing Loop-Level Parallelism

Eliminating Dependent Computations

- ▶ As mentioned above, one of the most important forms of dependent computations is a recurrence
- ▶ A dot product is a perfect example of a recurrence:

```
for (i=9999; i>=0; i=i-1)
    sum = sum + x[i] * y[i];
```

- ▶ This loop is not parallel because it has a **loop-carried dependence** on the **variable sum**
- ▶ We can transform it to a set of loops, one completely parallel and the other partly parallel

Detecting and Enhancing Loop-Level Parallelism

Eliminating Dependent Computations

- ▶ The first loop will execute the completely parallel portion of this loop:

```
for (i=9999; i>=0; i=i-1)
    sum[i] = x[i] * y[i];
```

- ▶ The **sum** has been expanded *from a scalar into a vector* quantity (a transformation called *scalar expansion*)
- ▶ Then we do the **reduce step**, which sums up the elements of the vector:

```
for (i=9999; i>=0; i=i-1)
    finalsum = finalsum + sum[i];
```