

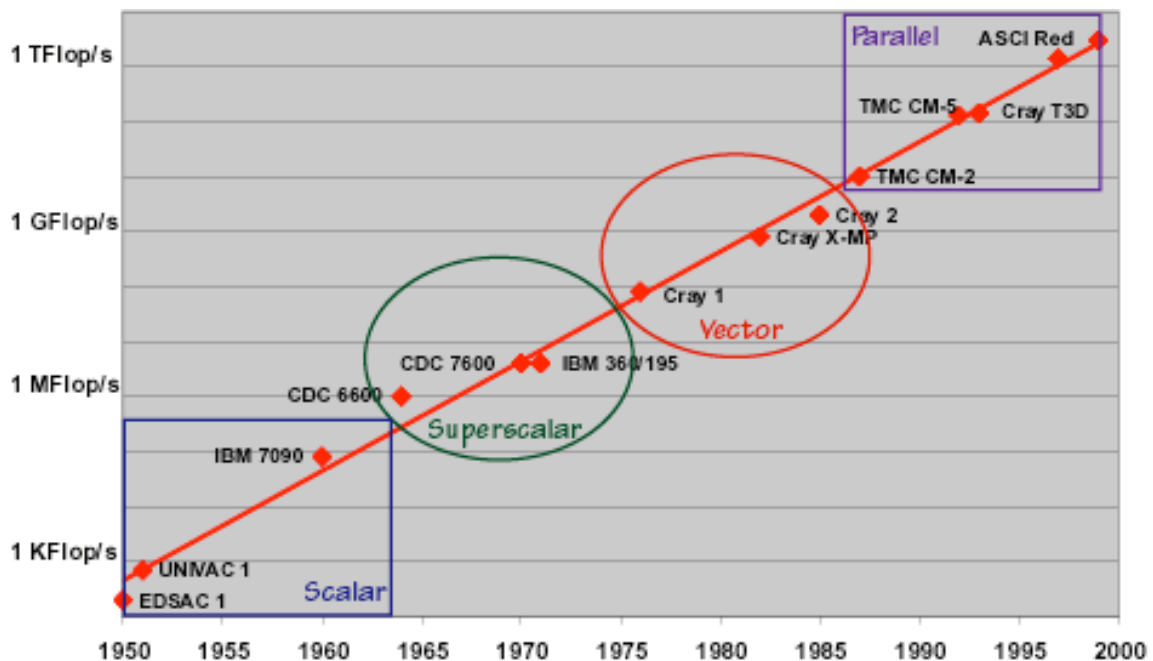
---

# Optimization Challenges for High Performance Architectures

Allen and Kennedy, Chapter 1

---

## Moore's Law



# Features of Machine Architectures

---

- **Pipelining**
- **Multiple execution units**
  - pipelined
- **Vector operations**
- **Parallel processing**
  - Shared memory, distributed memory, message-passing
- **VLIW and Superscalar instruction issue**
- **Registers**
- **Cache hierarchy**
- **Combinations of the above**
  - Parallel-vector machines

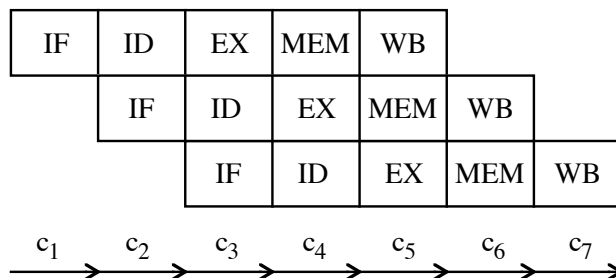
---

*Optimizing Compilers for Modern Architectures*

## Instruction Pipelining

---

- **Instruction pipelining**
  - DLX Instruction Pipeline



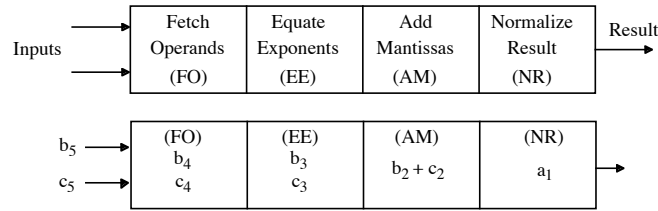
– What is the performance challenge?

---

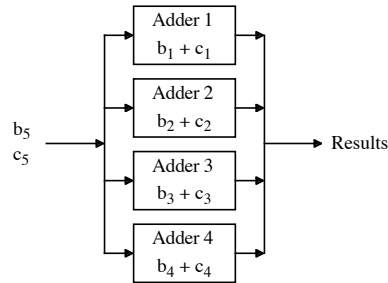
*Optimizing Compilers for Modern Architectures*

# Replicated Execution Logic

- **Pipelined Execution Units**



- **Multiple Execution Units**



What is the performance challenge?

# Vector Operations

- **Apply same operation to different positions of one or more arrays**

– Goal: keep pipelines of execution units full

– Example:

```
VLOAD V1,A
VLOAD V2,B
VADD V3,V1,V2
VSTORE V3,C
```

- **How do we specify vector operations in C, Java, Fortran 77 ?**

```
DO I = 1, 64
  C(I) = A(I) + B(I)
  D(I+1) = D(I) + C(I)
ENDDO
```

## VLIW

---

- **Multiple instruction issue on the same cycle**
  - Wide word instruction (or superscalar)
  - Usually one instruction slot per functional unit
- **What are the performance challenges?**

---

*Optimizing Compilers for Modern Architectures*

## VLIW

---

- **Multiple instruction issue on the same cycle**
  - Wide word instruction (or superscalar)
  - Usually one instruction slot per functional unit
- **What are the performance challenges?**
  - Finding enough parallel instructions
  - Avoiding interlocks
    - Scheduling instructions early enough

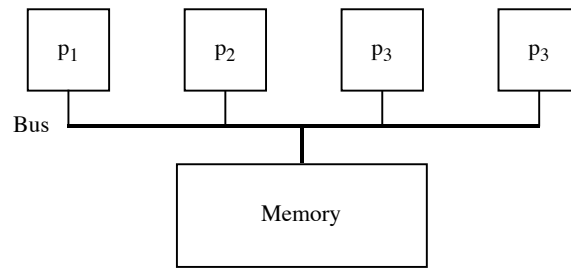
---

*Optimizing Compilers for Modern Architectures*

# SMP Parallelism

---

- Multiple processors with uniform shared memory
  - Task Parallelism
    - Independent tasks
  - Data Parallelism
    - the same task on different data



- 
- What is the performance challenge?

*Optimizing Compilers for Modern Architectures*

## Bernstein's Conditions

---

- When is it safe to run two tasks R1 and R2 in parallel?
  - If none of the following holds:
    1. R1 writes into a memory location that R2 reads
    2. R2 writes into a memory location that R1 reads
    3. Both R1 and R2 write to the same memory location
- How can we convert this to loop parallelism?
  - Think of loop iterations as tasks

---

*Optimizing Compilers for Modern Architectures*

## Memory Hierarchy

---

- Problem: memory is moving farther away in processor cycles
  - Latency and bandwidth difficulties
- Solution
  - Reuse data in cache and registers
- Challenge: How can we enhance reuse?

---

*Optimizing Compilers for Modern Architectures*

## Memory Hierarchy

---

- Problem: memory is moving farther away in processor cycles
    - Latency and bandwidth difficulties
  - Solution
    - Reuse data in cache and registers
  - Challenge: How can we enhance reuse?
    - Coloring register allocation works well
      - But only for scalars
- ```
DO I = 1, N
  DO J = 1, N
    C(I) = C(I) + A(J)
```
- Strip mining to reuse data from cache

---

*Optimizing Compilers for Modern Architectures*

# Distributed Memory

---

- **Memory packaged with processors**
  - Message passing
  - Distributed shared memory
- **SMP clusters**
  - Shared memory on node, message passing off node
- **What are the performance issues?**

---

*Optimizing Compilers for Modern Architectures*

# Distributed Memory

---

- **Memory packaged with processors**
  - Message passing
  - Distributed shared memory
- **SMP clusters**
  - Shared memory on node, message passing off node
- **What are the performance issues?**
  - Minimizing communication
    - Data placement
  - Optimizing communication
    - Aggregation
    - Overlap of communication and computation

---

*Optimizing Compilers for Modern Architectures*

# Optimization Techniques

---

- Program Transformations
  - Most of these architectural issues can be dealt with by restructuring transformations that can be reflected in source
    - Vectorization, parallelization, cache reuse enhancement
  - Challenges:
    - Determining when transformations are legal
    - Selecting transformations based on profitability
- Low level coding
- Some issues must be dealt with at a low level
  - Prefetch insertion
  - Instruction scheduling
- All require some understanding of the ways that instructions and statements depend on one another (share data)

---

*Optimizing Compilers for Modern Architectures*

## A Common Problem: Matrix Multiply

---

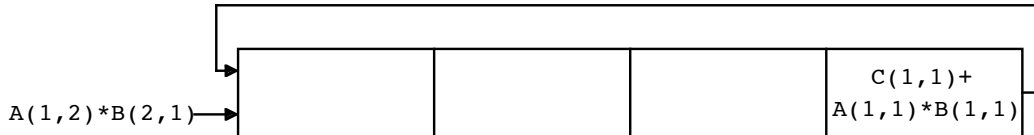
```
DO I = 1, N
  DO J = 1, N
    C(J,I) = 0.0
    DO K = 1, N
      C(J,I) = C(J,I) + A(J,K) * B(K,I)
    ENDDO
  ENDDO
ENDDO
```

---

*Optimizing Compilers for Modern Architectures*

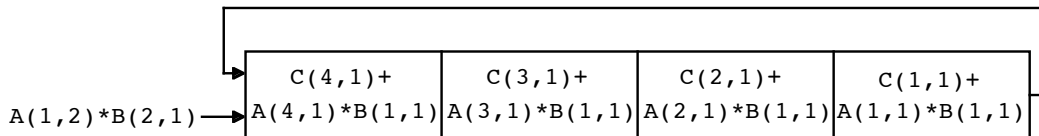
## Problem for Pipelines

- Inner loop of matrix multiply is a reduction



- **Solution:**

– work on several iterations of the J-loop simultaneously



*Optimizing Compilers for Modern Architectures*

## MatMult for a Pipelined Machine

```

DO I = 1, N,
  DO J = 1, N, 4
    C(J,I) = 0.0      !Register 1
    C(J+1,I) = 0.0   !Register 2
    C(J+2,I) = 0.0   !Register 3
    C(J+3,I) = 0.0   !Register 4
    DO K = 1, N
      C(J,I) = C(J,I) + A(J,K) * B(K,I)
      C(J+1,I) = C(J+1,I) + A(J+1,K) * B(K,I)
      C(J+2,I) = C(J+2,I) + A(J+2,K) * B(K,I)
      C(J+3,I) = C(J+3,I) + A(J+3,K) * B(K,I)
    ENDDO
  ENDDO
ENDDO

```

*Optimizing Compilers for Modern Architectures*

# Matrix Multiply on Vector Machines

---

```
DO I = 1, N
  DO J = 1, N
    C(J,I) = 0.0
    DO K = 1, N
      C(J,I) = C(J,I) + A(J,K) * B(K,I)
    ENDDO
  ENDDO
ENDDO
```

---

*Optimizing Compilers for Modern Architectures*

## Problems for Vectors

---

- Inner loop must be vector
  - And should be stride 1
- Vector registers have finite length (Cray: 64 elements)
  - Would like to reuse vector register in the compute loop
- Solution
  - Strip mine the loop over the stride-one dimension to 64
  - Move the iterate over strip loop to the innermost position
    - Vectorize it there

---

*Optimizing Compilers for Modern Architectures*

## Vectorizing Matrix Multiply

---

```
DO I = 1, N
  DO J = 1, N, 64
    DO JJ = 0, 63
      C(JJ, I) = 0.0

      DO K = 1, N

        C(J, I) = C(J, I) + A(J, K) * B(K, I)
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

---

*Optimizing Compilers for Modern Architectures*

## Vectorizing Matrix Multiply

---

```
DO I = 1, N
  DO J = 1, N, 64
    DO JJ = 0, 63
      C(JJ, I) = 0.0
    ENDDO
    DO K = 1, N
      DO JJ = 0, 63
        C(J, I) = C(J, I) + A(J, K) * B(K, I)
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

---

*Optimizing Compilers for Modern Architectures*

## MatMult for a Vector Machine

---

```
DO I = 1, N
  DO J = 1, N, 64
    C(J:J+63,I) = 0.0
    DO K = 1, N
      C(J:J+63,I) = C(J:J+63,I) + A(J:J+63,K)*B(K,I)
    ENDDO
  ENDDO
ENDDO
```

---

*Optimizing Compilers for Modern Architectures*

## Matrix Multiply on Parallel SMPs

---

```
DO I = 1, N
  DO J = 1, N
    C(J,I) = 0.0
    DO K = 1, N
      C(J,I) = C(J,I) + A(J,K) * B(K,I)
    ENDDO
  ENDDO
ENDDO
```

---

*Optimizing Compilers for Modern Architectures*

# Matrix Multiply on Parallel SMPs

---

```
DO I = 1, N  ! Independent for all I
  DO J = 1, N
    C(J,I) = 0.0
    DO K = 1, N
      C(J,I) = C(J,I) + A(J,K) * B(K,I)
    ENDDO
  ENDDO
ENDDO
```

---

*Optimizing Compilers for Modern Architectures*

## Problems on a Parallel Machine

---

- Parallelism must be found at the outer loop level
  - But how do we know?
- Solution
  - Bernstein's conditions
    - Can we apply them to loop iterations?
    - Yes, with dependence
  - Statement S2 depends on statement S1 if
    - S2 comes after S1
    - S2 must come after S1 in any correct reordering of statements
  - Usually keyed to memory
    - Path from S1 to S2
    - S1 writes and S2 reads the same location
    - S1 reads and S2 writes the same location
    - S1 and S2 both write the same location

---

*Optimizing Compilers for Modern Architectures*

## MatMult on a Shared-Memory MP

---

```
PARALLEL DO I = 1, N
  DO J = 1, N
    C(J,I) = 0.0
    DO K = 1, N
      C(J,I) = C(J,I) + A(J,K) * B(K,I)
    ENDDO
  ENDDO
END PARALLEL DO
```

---

*Optimizing Compilers for Modern Architectures*

## MatMult on a Vector SMP

---

```
PARALLEL DO I = 1, N
  DO J = 1, N, 64
    C(J:J+63,I) = 0.0
    DO K = 1, N
      C(J:J+63,I) = C(J:J+63,I) + A(J:J+63,K) * B(K,I)
    ENDDO
  ENDDO
ENDDO
```

---

*Optimizing Compilers for Modern Architectures*

# Matrix Multiply for Cache Reuse

---

```
DO I = 1, N
  DO J = 1, N
    C(J,I) = 0.0
    DO K = 1, N
      C(J,I) = C(J,I) + A(J,K) * B(K,I)
    ENDDO
  ENDDO
ENDDO
```

---

*Optimizing Compilers for Modern Architectures*

## Problems on Cache

---

- There is reuse of  $C$  but no reuse of  $A$  and  $B$
- Solution
  - Block the loops so you get reuse of both  $A$  and  $B$ 
    - Multiply a block of  $A$  by a block of  $B$  and add to block of  $C$
  - When is it legal to interchange the iterate over block loops to the inside?

---

*Optimizing Compilers for Modern Architectures*

## MatMult on a Uniprocessor with Cache

---

```
DO I = 1, N, S
  DO J = 1, N, S
    DO p = I, I+S-1
      DO q = J, J+S-1
        C(q,p) = 0.0
      ENDDO
    ENDDO
  ENDDO
  DO K = 1, N, T
    DO p = I, I+S-1
      DO q = J, J+S-1
        DO r = K, K+T-1
          C(q,p) = C(q,p) + A(q,r) * B(r,p)
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

*Diagram annotations:*

- A red arrow points from the text "ST elements" to the term  $A(q,r)$  in the assignment statement.
- A green arrow points from the text "ST elements" to the term  $B(r,p)$  in the assignment statement.
- A purple arrow points from the text " $S^2$  elements" to the entire assignment statement  $C(q,p) = C(q,p) + A(q,r) * B(r,p)$ .

---

*Optimizing Compilers for Modern Architectures*

## MatMult on a Distributed-Memory MP

---

```
PARALLEL DO I = 1, N
  PARALLEL DO J = 1, N
    C(J,I) = 0.0
  ENDDO
ENDDO
PARALLEL DO I = 1, N, S
  PARALLEL DO J = 1, N, S
    DO K = 1, N, T
      DO p = I, I+S-1
        DO q = J, J+S-1
          DO r = K, K+T-1
            C(q,p) = C(q,p) + A(q,r) * B(r,p)
          ENDDO
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

---

*Optimizing Compilers for Modern Architectures*

# Dependence

---

- **Goal:** aggressive transformations to improve performance
- **Problem:** when is a transformation legal?
  - Simple answer: when it does not change the meaning of the program
  - But what defines the meaning?
- Same sequence of memory states
  - Too strong!
- Same answers
  - Hard to compute (in fact intractable)
  - Need a sufficient condition
- We use in this book: **dependence**
  - Ensures instructions that access the same location (with at least one a store) must **not** be reordered

---

*Optimizing Compilers for Modern Architectures*

# Summary

---

- Modern computer architectures present many performance challenges
- Most of the problems can be overcome by transforming loop nests
  - Transformations are not obviously correct
- Dependence tells us when this is feasible
  - Most of the book is about how to use dependence to do this

---

*Optimizing Compilers for Modern Architectures*