COMP 515: Advanced Compilation for Vector and Parallel Processors

Prof. Krishna Palem
Prof. Vivek Sarkar
Department of Computer Science
Rice University
{palem,vsarkar}@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP515

Course Information

- Meeting time: TTh 10:50am – 12:05pm
- Meeting place: Martel College 103
- Instructors: Krishna Palem, Vivek Sarkar
- Web site: https://wiki.rice.edu/confluence/display/PARPROG/COMP515
- Prerequisite: COMP 412
- Textbook
  - Allen and Kennedy, Optimizing Compilers for Modern Architectures, Morgan-Kaufmann, Second Printing, 2005.
- Grading rubric
  - Homeworks (25%)
  - Exam 1 (20%)
  - Exam 2 (20%)
  - Class project (35%)
- Acknowledgment: Slides from previous offerings of COMP 515 by Prof. Ken Kennedy (http://www.cs.rice.edu/~ken/comp515/)

Dependence-Based Compilation

- Vectorization and Parallelization require a deeper analysis than optimization for scalar machines
  - Must be able to determine whether two accesses to the same array might be to the same location
- Dependence is the theory that makes this possible
  - There is a dependence between two statements if they might access the same location, there is a path from one to the other, and one access is a write
- Dependence has other applications
  - Memory hierarchy management—restructuring programs to make better use of cache and registers
    - Includes input dependences
  - Scheduling of instructions

Syllabus

- Introduction
  - Parallel and vector architectures. The problem of parallel programming. Bernstein's conditions and the role of dependence. Compilation for parallel machines and automatic detection of parallelism.
- Dependence Theory and Practice
  - Fundamentals, types of dependences. Testing for dependence: separable, gcd and Banerjee tests. Exact dependence testing. Construction of direction and distance vectors.
- Preliminary Transformations
  - Loop normalization, scalar data flow analysis, induction variable substitution, scalar renaming.

Syllabus (contd)

- Fine-Grain Parallel Code Generation
  - Loop distribution and its safety. The Kuck vectorization principle. The layered vector code-generation algorithm and its complexity. Loop interchange.
- Unimodular & Polyhedral loop transformation frameworks
  - New topics not covered in textbook
- Coarse-Grain Parallel Code Generation
  - Loop Interchange. Loop Skewing. Scalar and array expansion. Forward substitution. Alignment. Code replication. Array renaming. Node splitting. Pattern recognition. Threshold analysis. Symbolic dependence tests. Parallel code generation and its problems.
- Control Dependence
  - Types of branches. If conversion. Control dependence. Program dependence graph.

Syllabus (contd)

- Memory Hierarchy Management
  - The use of dependence in scalar register allocation and management of the cache memory hierarchy.
- Scheduling for Superscalar and Parallel Machines Machines
  - Role of dependence. List Scheduling. Software Pipelining. Work scheduling for parallel systems. Guided Self-Scheduling
- Interprocedural Analysis and Optimization
  - Side effect analysis, constant propagation and alias analysis. Flow-insensitive and flow-sensitive problems. Side effects to arrays. Inline substitution, linkage tailoring and procedure cloning. Management of interprocedural analysis and optimization.
- Compilation of Other Languages.
  - C, Verilog, Fortran 90, HPF.

Compiler Challenges for High Performance Architectures

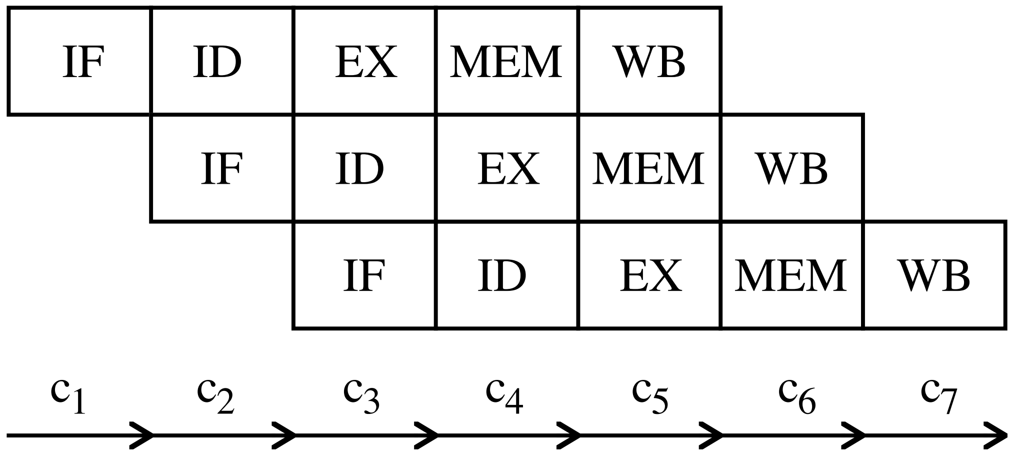
Allen and Kennedy, Chapter 1

Features of Machine Architectures

- Pipelining
- Multiple execution units
  - pipelined
- Vector operations
  - includes fine-grained SIMD vector parallelism
- Parallel processing
  - Multicore, shared memory, distributed memory, message-passing
- Superscalar instruction issue, software/hardware prefetch
- Registers
- Memory hierarchy
- Combinations of the above

Instruction Pipelining

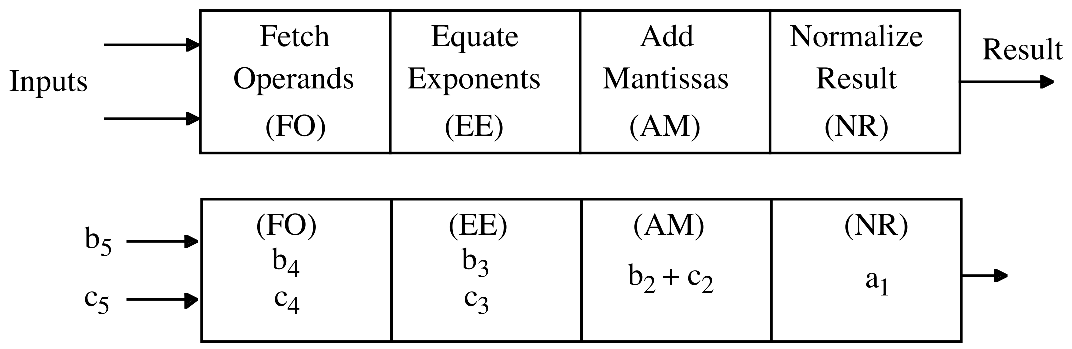
- **Instruction pipelining**
  - DLX Instruction Pipeline



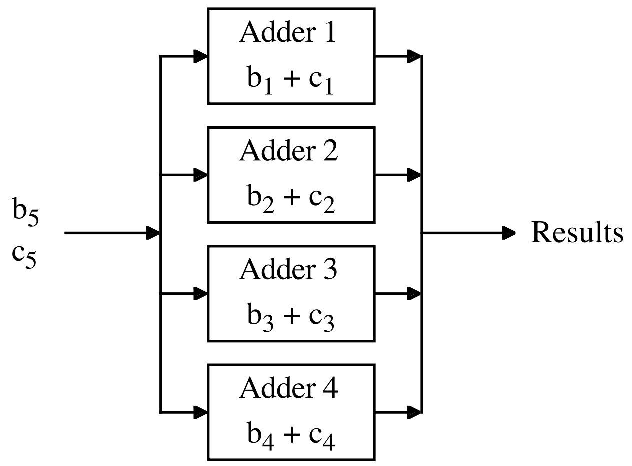
  - What is the performance challenge?

Replicated Execution Logic (Floating Point Adders)

- **Pipelined Execution Units**



- **Multiple Execution Units**



What is the performance challenge?

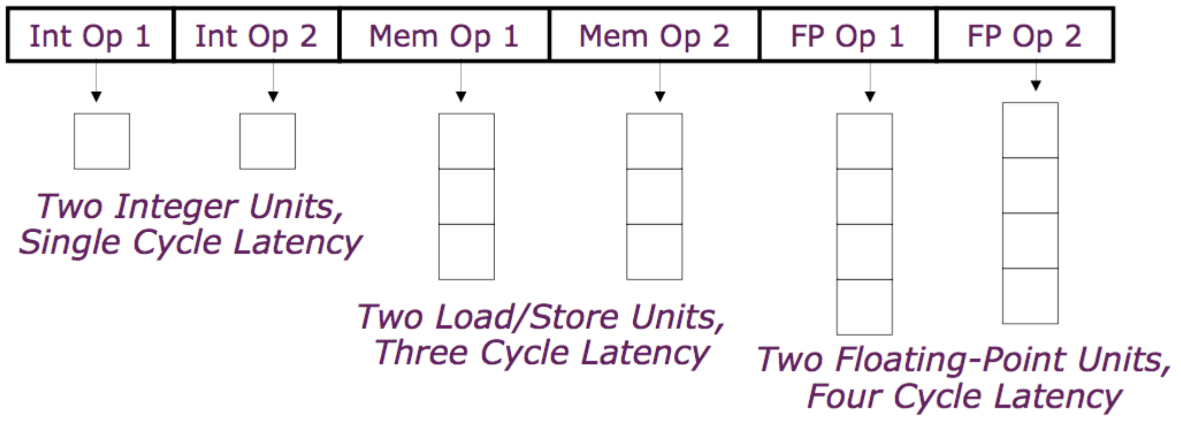
Vector Operations

- **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
```

Very Large Instruction Word (VLIW)

- Multiple instruction issue on the same cycle
  - Wide word instruction (or superscalar)
  - Designated functional units for instruction slots


Int Op 1
Int Op 2
Mem Op 1
Mem Op 2
FP Op 1
FP Op 2
Two Integer Units, Single Cycle Latency
Two Load/Store Units, Three Cycle Latency
Two Floating-Point Units, Four Cycle Latency

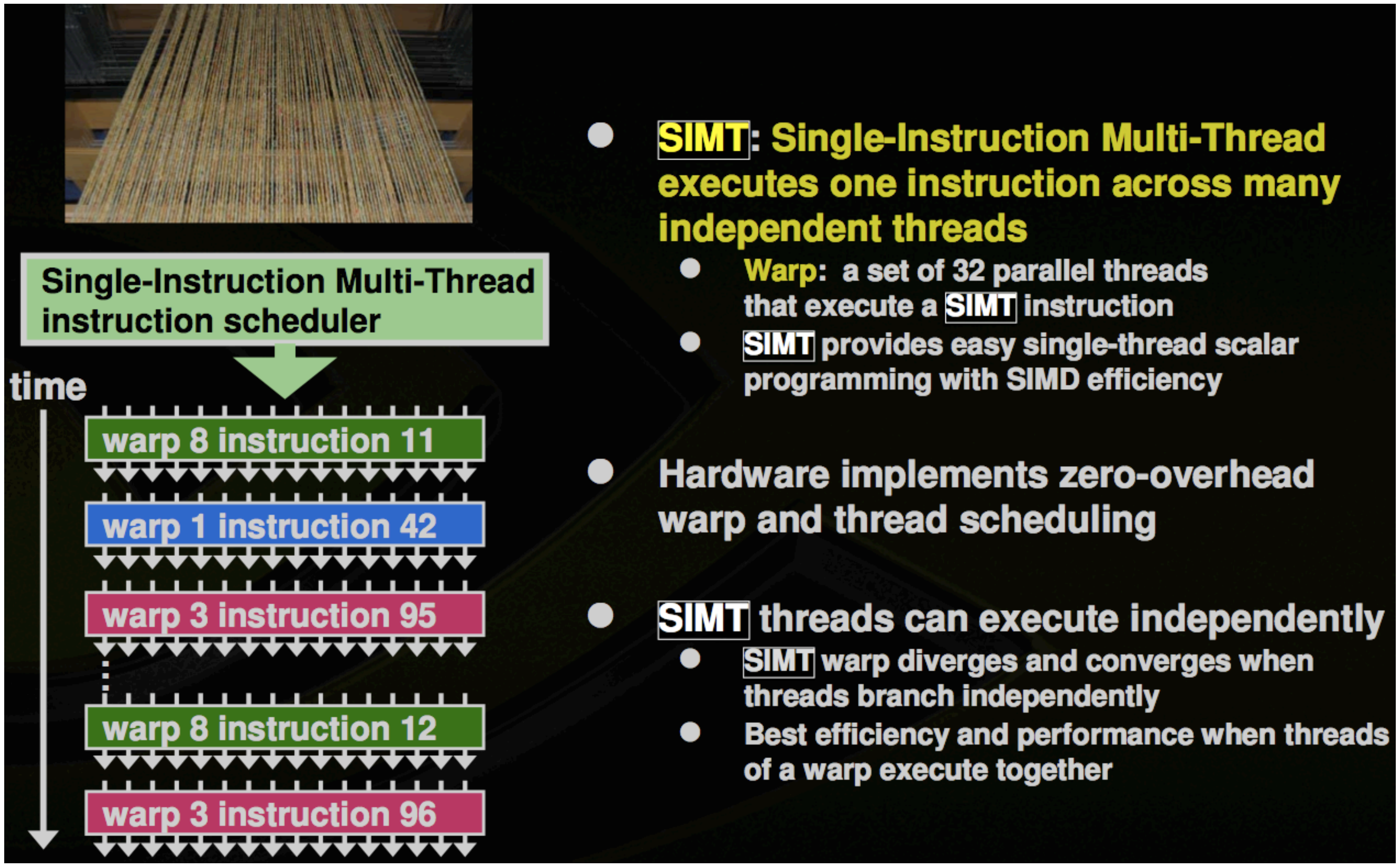
- Multiple operations packed into one instruction
- Each operation slot is for a fixed function
- Constant operation latencies are specified
- Architecture requires guarantee of:
  - Parallelism within an instruction => no x-operation RAW check
  - No data use before data ready => no data interlocks

Source: "VLIW/EPIC: Statically Scheduled ILP", Joel Emer,
http://ocw.mit.edu/NR/rdonlyres/Electrical-Engineering-and-Computer-Science/6-823Fall-2005/418A205E-6B93-4EB9-9080-0EDDB32E06D4/0/l21_vliw.pdf

SIMD (Single Instruction Multiple Data)

- Short SIMD architectures
  - E.g., MMX, SSE, AltiVec
  - Limited vector length (16 bytes for Altivec)
  - Contiguous memory access
  - Data alignment constraint (128-bit alignment for Altivec)

SIMT (Single Instruction Multiple Thread)

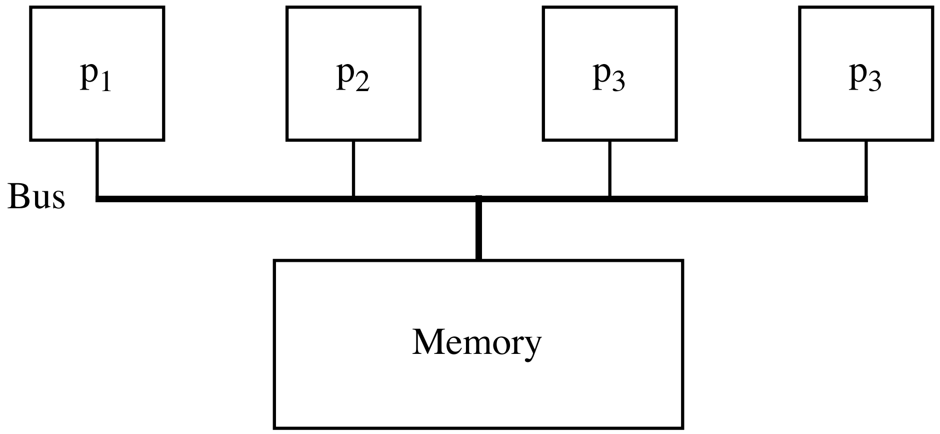


- SIMT: Single-Instruction Multi-Thread executes one instruction across many independent threads
  - Warp: a set of 32 parallel threads that execute a SIMT instruction
  - SIMT provides easy single-thread scalar programming with SIMD efficiency
- Hardware implements zero-overhead warp and thread scheduling
- SIMT threads can execute independently
  - SIMT warp diverges and converges when threads branch independently
  - Best efficiency and performance when threads of a warp execute together

Source: 10 Important Problems in Computer Architecture, David B. Kirk, http://isca2008.cs.princeton.edu/images/ISCA_DK.pdf

SMP Parallelism (Homogeneous Multicore)

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



- What is the performance challenge?

Distributed Memory

- Memory packaged with processors
  - Message passing
  - Distributed shared memory
- SMP clusters
  - Shared memory on node, message passing off node
- Distributed memory in multicore processors
  - Intel Single Chip Cloud (SCC) computer
  - Tilera
- What are the performance issues?
  - Minimizing communication
    - Data placement
  - Optimizing communication
    - Aggregation
    - Overlap of communication and computation

Compiler Technologies

- Program Transformations
  - Many of these architectural issues can be dealt with by restructuring transformations that can be reflected in source
    - Vectorization, parallelization, cache reuse enhancement
  - Two key challenges:
    - Determining when transformations are legal
    - Selecting transformations based on profitability
- Low level code generation
  - 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)

Fortran DO loop notation

```
DO I = 1, N                       DO 10 I = 1, N
  . . .            and               . . .
END DO                        10  CONTINUE
```

are equivalent to the following C-for loop:

```
for(I = 1; I <= N; I++) {
  . . .
}
```

Fortran column major vs. C row-major data layouts

Fortran: real*8 A(100,100)
A(1,1)
A(2,1)
…
A(100,1)
A(1,2)
A(2,2)
…
A(100,2)
…

C: double A[100][100]
A[0][0]
A[0][1]
…
A[0][99]
A[1][0]
A[1][1]
…
A[1][99]
…

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
```

MatMult for a Pipelined Machine

```
DO I = 1, N,
      DO J = 1, N, 4 // Unroll J loop 4 times
         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
```

Problems for Vectors

- Inner loop must be vector
  - And should be stride 1
  - Note that array layout is column-major for FORTRAN and row-major in C
- Vector registers have finite length (Cray: 64 elements, modern SIMD processor operands are in the 256-512 byte range)
  - 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

Vectorizing Matrix Multiply

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

            DO K = 1, N

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

Vectorizing Matrix Multiply

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

MatMult for a Vector Machine (using array language notation)

```
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
```

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
```

Bernstein's Conditions [1966]

- 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 apply this to loop parallelism?
  - Think of loop iterations as tasks

- How can we apply this to statement-level parallelism?
  - Think of statement instances as tasks

- Time for Worksheet #1 !

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

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
```

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
```

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?
  - Fortran example

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

  - Equivalent C/Java code

```
for  (int I = 1; I <= N; I++)
  for  (int J = 1; J <= N; J++)
     C[I] = C[I] + A[J];
```

  - Strip mining to reuse data from cache

Matrix Multiply for Cache Reuse

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

- J loop reuses C(I) and A(I), but not B(J)
- I loop reuses B(J), but not C(I) and A(I)
- Solution
  - Block/tile 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?
- Time for Worksheet #2 !

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
        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
```

ST elements

ST elements

S^2 elements

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

Summary

- Modern computer architectures present many performance challenges
- Most of the problems can be overcome by transforming loop nests
  - Transformations are not obviously correct --- need to formalize legality rules
- Dependence tells us when this is feasible
  - Most of the book is about how to use dependence to do this
- Next lecture
  - Chapter 2, Dependence: Theory and Practice

Course Project Logistics

- Today's "homework" to be completed by tomorrow (Aug 28)
  - Form project teams (pairs preferred)
  - Sign up on doodle poll to meet with instructors in DH 3131 during one of these slots on Aug 28th:
    - 9:00 – 9:30, 9:30 – 10:00, 10:00 – 10:30
- Goal of course project is to perform an in-depth study of a research problem related to the course
  - Should include a theoretical component
  - Practicality can be demonstrated by hand-coded source-to-source transformations
- We will try and assign you a senior PhD student as a mentor for your project
- September 17 & 19 are self-study days for you to develop your project proposal (due by Sep 20)
- Final project presentations scheduled in class on Nov 26, Dec 3, and Dec 5

Worksheet 1 (to be done in pairs)

Name 1: ____________________ Name 2: ____________________

```
DO I = 1, N
   T = A[I]             S1
   A[I] = B[I]          S2
   B[I] = T             S3
ENDDO
```

- Using Bernstein conditions, identify pairs of statement instances that can exhibit one of the following conditions (a different pair for each condition)
    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
    4. None of the above

Worksheet 2 (to be done in pairs)

Name 1: ____________________ Name 2: ____________________

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

1. Assuming a uniprocessor cache with one word per cache line, and unbounded ("infinite") capacity, how many cache misses are incurred by the above code?

2. How does your answer change if the cache can only hold 4 words?