Dependencies, Instruction Scheduling, Optimization, and Parallelism

Prof. James L. Frankel
Harvard University
Ordering of Execution of Instructions

• Although written by the programmer in a particular way, the language allows execution in another order so long as it meets the \textit{as-if} constraint

• A different order may allow faster execution because of
  • Delay slots
  • Pipelining advantages
  • Caching advantages
  • Prefetching
  • Multiple processing elements
  • Data locality
Delay Slot

• Pre-fetching of instructions is performed by the processor so it is not idle waiting for instructions to be read from memory
• If an unpredicted branch/jump occurs, it may cause a pipeline bubble
• Pre-fetching of instructions may not follow the execution path even if a processor is able to correctly predict whether a branch/jump will occur
• MIPS deals with this issue by executing one instruction that follows a branch/jump whether or not the branch/jump occurs
  • The location of that instruction following the branch/jump is referred to as the delay slot
Delay Slot Not Evident in Our MIPS Code

• We’ve been using SPIM in a default, simplified mode
  • SPIM is not emulating the delay slot feature of MIPS
• Switch `-delayed_branches` turns delay slot emulation on
Pipelining

• Present the CSCI E-93 MIPS Pipelining Slides
  • These are not available on-line
Caching

• See CSCI E-93 Caching slides
Types of Dependencies

• Control Dependence
  • Control flow of program determines what can execute when

• Data Dependence
  • Definition and use of variables determines a partial ordering
Control Dependencies

• Flow-of-control statements
  • If-then
  • If-then-else
  • For
  • While
  • Do-while
  • Switch-case
  • Function call
  • Return
  • Goto
  • Break
  • Continue
Control Dependencies

• Flow-of-control operators
  • ||
  • &&
  • ?:
Data Dependencies (1 of 3)

• True Dependence
  • A variable is written and then is read

  variable = ...

  ...

  ... = variable
Data Dependencies (2 of 3)

• Output Dependence
  • A variable is written and later is written again

```
variable = ...
...
variable = ...
```

• Can be removed by renaming (SSA form)
Data Dependencies (3 of 3)

• Anti-Dependence
  • A variable is read and then written

  … = variable
  ...
  variable = ...

• Can be removed by renaming (SSA form)
Complications in Determining Data Dependence

- *Array* accesses require analysis of the subscript expressions
- *Pointer* accesses require analysis of the pointers derivations
  - In addition to aliasing other pointers, pointers can also alias variables of other types
- *Unions* create aliases explicitly
Sequential Array Accesses (1 of 3)

\[
i = 5;
\]
\[
j = 6;
\]
\[
A[j-1] = \ldots;
\]
\[
\ldots = A[i];
\]

• Does \( j-1 \) equal \( i \)?
• Can be determined by copy propagation and constant folding
• What about \textit{across} basic blocks?
Sequential Array Accesses (2 of 3)

void f(int i, j) {
    A[j-1] = ...;
    ... = A[i];
}

• Does j-1 equal i?
• Requires symbolic evaluation and inter-procedural analysis (*i.e.*, analysis across the function call boundary)
void f(int i, j) {
    A[j-1] = ...;
    ... = A[i*3];
}

• Does j-1 equal i*3?
• Requires more complicated symbolic evaluation and inter-procedural analysis
Pointer Dereferencing (1 of 4)

```c
int A[10], p, q;
p = &A[0];
q = &A[1];
*p = ...;
... = *q;
```

• Do *p and *q alias each other?
int A[10], p, q;
p = &A[0];
q = &A[1];
*p = …;
… = *(q-1);

• Do *p and *(q-1) alias each other?
int A[10], p, q;
p = &A[0];
*p = ...;
q = f(...);
... = *q;

• Do *p and *q alias each other?
int i, p;
p = &i;
...
... = i; /* First reference to i */
*p = ...;
... = i; /* Second reference to i */

• Do *p and i alias each other?
• Do both references to i need to read the value of i or could i be kept in a register?
Unions (1 of 3)

union union_name {
    int i;
    float f;
} var;
var.i = ...;
... = var.f;

• Do var.i and var.f alias each other?
Unions (2 of 3)

union union_name {
    int i;
    short s;
    char c;
} var;
var.i = ...
... = var.s;

• Do var.i and var.s alias each other?
Unions (3 of 3)

union union_name {
    int i;
    short s[4];
    char c[6];
} var;
var.i = ...;
... = var.s[2];

• Do var.i and var.s[2] alias each other?
Sequential Data Dependency vs. Loop-Carried Data Dependency

• Sequential Data Dependency is directly reflected by the program without requiring analysis of loops

• Loop-Carried Data Dependency requires analysis of loops to be discovered
Simple Loop-Carried Data Dependence Example

n = 5;
product = 1;
while(n > 1) {
    product = product*n;
    n--;
}

• Both n and product have sequential and loop-carried dependencies
Difficulties in Data Dependence Analysis

• Usually analysis is more difficult because of more complex data types
• Determining if a reference is to the same data as another access is the problem of determining aliasing
• One access aliases another access, if the accesses overlap data in memory
• Array accesses require analysis of the subscript expressions
• Pointer accesses require analysis of the pointers derivations
• Unions create aliases explicitly
Loop-Level Parallelism (1 of 3)

• Compute the squares of the differences between elements in two arrays

```c
for(i = 0; i < n; i++) {
    Z[i] = X[i] - Y[i];
    Z[i] = Z[i] * Z[i];
}
```

• Contains independent iterations
Loop-Level Parallelism (2 of 3)

• Compute the squares of the differences between elements in two arrays

```c
for(i = 0; i < n; i++)
    Z[i] = X[i] - Y[i];
for(i = 0; i < n; i++)
    Z[i] = Z[i] * Z[i];
```

• Also contains independent iterations, but exhibits worse data locality than the program fragment on the previous slide
  • In the previous program fragment, operations can be performed while data is still in registers
Loop-Level Parallelism (3 of 3)

• Going back to the first fragment, with $M$ processors and with each processor numbered $p$ (zero origin), the previous loop can be rewritten, as follows:

\[
b = \text{ceil}(n/M);
\]

\[
\text{for}(i = b*p; i < \text{min}(n, b*(p+1)); i++) \{
    Z[i] = X[i] - Y[i];
    Z[i] = Z[i] * Z[i];
\}
\]

• Approximately equal size, independent iterations are created for each processor
FORTRAN PARALLEL DO

• FORTRAN has a PARALLEL DO statement that tells the compiler there are no dependencies across its iterations

PARALLEL DO I = 1, N
  A(I) = A(I) + B(I)
ENDDO
ISO C99 restrict

- ISO C99 has the **restrict** type qualifier *for pointers* to tell the compiler there are no aliases to access the object to which it points.

```c
void add(int n, int *restrict dest, int *restrict op1, int *restrict op2) {
    int i;
    for(i = 0; i < n; i++)
        dest[i] = op1[i] + op2[i];
}
```
Here is a slightly more complicated example of a loop-carried dependence:

double Z[100];
for(i = 0; i < 91; i++) {
    Z[i+10] = Z[i];
}

- Iteration 0 copies Z[0] into Z[10]
- ...
- Iteration 10 copies Z[10] into Z[20] -- This is a true dependent on iteration 0
- Iteration 11 copies Z[11] into Z[21] -- This is a true dependent on iteration 1
- ...
Loop-Carried Dependence (2 of 7)

- This program fragment copies the first ten locations of Z into each of the next ten locations of Z through to the end of Z
Loop-Carried Dependence (3 of 7)

• This example gives us a loop-carried dependence distance of 10
• And, a dependence direction of < (which means the direction is to a future iteration)

• These distances and directions can be computed for each nested loop iteration variable and for each statement in the loop

• For this example, the first 10 iterations can run with no dependencies
• Then, each iteration can run so long as the iteration 10 before it has completed
Loop-Carried Dependence (4 of 7)

- For which values of \(x\) and \(y\) does \(x+10\) equal \(y\) in the range \(0 \leq x, y < 91\)?
  - An exact test would tell us if there exists a solution in the specified range
  - An inexact test would tell us if there exists a solution, but not necessarily in the specified range

- This is an Integer Linear Program
- Diophantine analysis can give us an exact answer
- \textit{GCD (Greatest Common Divisor)} can give us an inexact answer
  - But, if GCD says \textbf{NO}, then that is very useful information because then there is no integer solution even outside the specified range!
Diophantine Equation

• Wikipedia: A **Diophantine equation** is a polynomial **equation**, usually in two or more unknowns, such that only the integer solutions are sought or studied (an integer solution is a solution such that all the unknowns take integer values)
Here is another example of a loop-carried dependence:

double Z[100];
for(i = 0; i < 91; i++) {
    Z[i] = Z[i+10];
}

- Iteration 0 copies Z[10] into Z[0]
- ...
- Iteration 10 copies Z[20] into Z[10]  -- This is anti-dependent on iteration 0
- Iteration 11 copies Z[21] into Z[11]  -- This is anti-dependent on iteration 1
- ...

---

Loop-Carried Dependence (5 of 7)
Loop-Carried Dependence (6 of 7)

• Unfortunately, these anti-dependences can’t be removed by renaming (converting into SSA form) because they are elements of an array

• This example gives us a loop-carried dependence distance of 10

• And, a dependence direction of < (which means the direction is to a future iteration)

• Once again, for this example, the first 10 iterations can run with no dependencies

• Then, each iteration can run so long as the iteration 10 before it has completed
Loop-Carried Dependence (7 of 7)

• Here is a more complicated example of a loop-carried dependence:

```c
double A[200];
for(i = 0; i < 100; i++) {
}
```

• Let’s apply the GCD test
• \(2i^{\text{dest}} + 2 = 2i^{\text{use}} + 1\)
• \(2i^{\text{dest}} - 2i^{\text{use}} = -1\)
• Does gcd(2, 2) divide 1?
• No; there is no dependency
Greatest Common Divisor

• The greatest common divisor of \(a_1, a_2, \ldots, a_n\) is denoted by \(\gcd(a_1, a_2, \ldots, a_n)\)
• It is the largest integer that evenly divides all \(a_1\) through \(a_n\)
• Use the Euclidean Algorithm to compute GCD; see Aho, Lam, Sethi, and Ullman, page 820 for details on the algorithm

• Theorem 11.32 in ALSU on page 819 states that
  • the linear Diophantine equation
    \[a_1x_1 + a_2x_2 + \ldots + a_nx_n = c\]
    • has an integer solution for \(x_1, x_2, \ldots, x_n\) if and only if \(\gcd(a_1, a_2, \ldots, a_n)\) divides \(c\)
  • Signs of the \(a\) terms and of \(c\) (i.e., if any of the \(a\) terms or \(c\) are negative) are irrelevant
Eager Evaluation

• Execute code to evaluate an expression when the result is assigned (bound) to a variable
• This is the usual evaluation methodology using in most programming languages
• Eager evaluation is a straight-forward implementation of the program
Futures/Lazy Evaluation/Call-by-Need

• Delayed evaluation until actually needed
  • Most common method of evaluation in executing Haskell programs
• Sometimes operations are performed, but only a portion of the result is needed
  • Example: array inversion, but only some elements needed
• Sometimes operations are performed, but control flow means the result may not be used
• Side-effects (e.g., input/output) must occur when expected
• May allow infinite-size data structures to be declared
• Causes the minimal amount of computation to be performed
Speculative Evaluation

• Execute code in advance of being needed if resources are available
• Take advantage of idle resources
• Have result immediately available, if needed
• Either side-effects must not occur (e.g., input/output) or must be able to be reverted or undone (e.g., changing values of variables)
• Overall more computation may be performed, but the overall time to completion of a program can be reduced
Locality of Data to Processor

• In a multi-processor system, having data local to a processor is very important
  • Data in registers is fastest
  • Data in memory is an order-of-magnitude slower
  • Data accessed over a network is slower
  • Data in mass storage is much slower

• Very important to appropriately locate data in a MIMD (Multiple Instruction Multiple Data) computer with local memory to each processing element
Task Parallelism

- Can run larger segments of code on separate processors
- These might be different function invocations
- These might be multiple independent loops

- Easy to exploit for small scale parallelization
- Not as attractive for large scale parallelization as loop iteration/data parallelism because
  - There isn’t the same degree of task parallelism
  - As the size of a data set increases, task parallelism doesn’t increase
  - Tasks are generally of unequal size
    - Not all processors are kept busy
    - Need to wait for the slowest processor
Data Parallelism

• For CPU intensive, long-running programs, there is a higher degree of data parallelism
• As the size of a data set increases, data parallelism increases
• Tasks are generally of equal size
  • Keeps all processors busy
  • No need to wait for the last processor to complete
Vector/SIMD/GPU Processors

• Same operation to multiple processing elements
  • SIMD == Single Instruction Multiple Data

• Compiler needs to uncover array-like operations and dole them out to each processor

• An equally big problem is locating the data in the appropriate processor
  • What if the data is used in different ways so that sometimes one assignment of data to processors was appropriate and other times a different assignment was appropriate?
Massively-Parallel Processor (MPP)

• Extremely large number of processors (e.g., 64K)
• Exploit parallelism in large data structures
  • Intended for very time-consuming computations
  • Almost all very time-consuming computations deal with massive amounts of data
• Distribute the data among the processors
• Perform (mostly) local operations on the data

• Explore C* as an example of how to program such machines
Data Flow Computation

• Present the Jack Dennis model of Data Flow Computation