Compiler techniques for leveraging ILP

Purshottam and Sajith

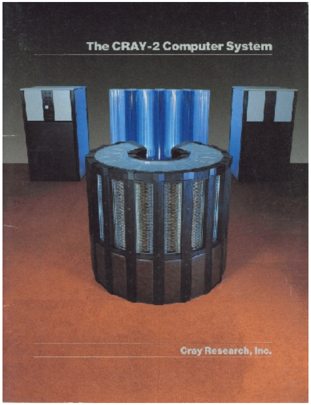
October 12, 2011

Parallelism in your pocket

- LINPACK is available in Android and iOS app markets. One reviewer says: “Have no Idea what it does and I am very very confused”

  http://www.netlib.org/linpack/

Parallelism NOT in your pocket



Cray-2: “Size of a washing machine, immersed in a tank of Fluorinert.”

Parallelism in your pocket

- NYT, May 2011: "Jack Dongarra's (University of Tennessee) research group has run the test on Apples new iPad 2, and it turns out that the legal-pad-size tablet would be a rival for a four-processor version of the Cray 2 supercomputer, which, with eight processors, was the worlds fastest computer in 1985."

  http://bits.blogs.nytimes.com/2011/05/09/the-ipad-in-your-hand-as-fast-as-a-supercomputer-of-yore/

Parallelism in your pocket

- NYT, May 2011: "Jack Dongarra's (University of Tennessee) research group has run the test on Apples new iPad 2, and it turns out that the legal-pad-size tablet would be a rival for a four-processor version of the Cray 2 supercomputer, which, with eight processors, was the worlds fastest computer in 1985."

  http://bits.blogs.nytimes.com/2011/05/09/the-ipad-in-your-hand-as-fast-as-a-supercomputer-of-yore/

- News in the street is that the new iPhone 4S can beat a 1993 vintage Cray in Linpack benchmarks.

How have compilers kept up?

Let's investigate!

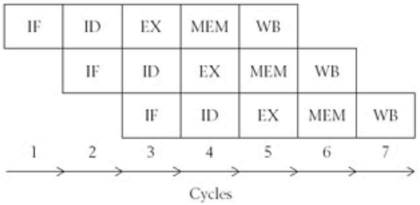
Many ways!

- Dependence testing
- Prelimimary transformations
- Enhancing fine-grained parallelism
- Creating coarse-grained parallelism
- Handling control flow
- Improving register usage
- Managing cache
- Scheduling
- Interprocedural analysis and optimization
- etc.

Overview

1. ILP overview
2. Compiling for scalar pipelines
3. Superscalar and VLIW processors
4. Vector architectures

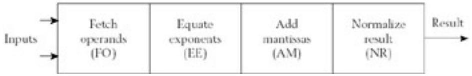
ILP overview
Pipelined instruction units



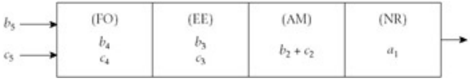
DLX instruction pipeline.

ILP overview

Pipelined execution units



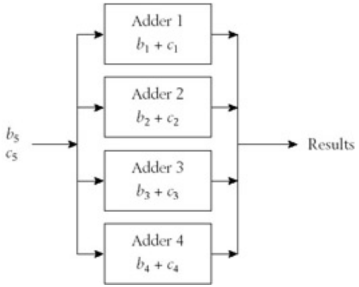
Typical floating point adder.



Snapshot of a pipelined execution unit computing ai = bi + ci.

ILP overview

Parallel functional units



Multiple functional units.

Compiling for scalar pipelines

- Key performance barrier is *pipeline stalls*, caused by one of these *hazards*:

Compiling for scalar pipelines

- Key performance barrier is *pipeline stalls*, caused by one of these *hazards*:
    - Structural hazards, where machine resources do not support all possible combinations of instruction overlap that might occur.

Compiling for scalar pipelines

- Key performance barrier is *pipeline stalls*, caused by one of these *hazards*:
    - Structural hazards, where machine resources do not support all possible combinations of instruction overlap that might occur.
    - Data hazards, where the result produced by one instruction is required by the subsequent instruction.

Compiling for scalar pipelines

- Key performance barrier is *pipeline stalls*, caused by one of these *hazards*:
    - Structural hazards, where machine resources do not support all possible combinations of instruction overlap that might occur.
    - Data hazards, where the result produced by one instruction is required by the subsequent instruction.
    - Control hazards, which occur because of the processing of branches.

Compiling for scalar pipelines

- Key performance barrier is *pipeline stalls*, caused by one of these *hazards*:
    - Structural hazards, where machine resources do not support all possible combinations of instruction overlap that might occur.
    - Data hazards, where the result produced by one instruction is required by the subsequent instruction.
    - Control hazards, which occur because of the processing of branches.
- The principal compiler strategy is to rearrange instructions so that the stalls never occur. This is called *instruction scheduling*.

Overview

1 ILP overview

2 Compiling for scalar pipelines

3 Superscalar and VLIW processors

4 Vector architectures

Superscalar and VLIW processors

- Vector operations complicate instruction set design.

Superscalar and VLIW processors

- Vector operations complicate instruction set design.
- ...if we could issue one or more pipelined instructions on each cycle, it might be possible to fill the execution unit pipelines...

Superscalar and VLIW processors

Multiple-issue instruction units

- Multiple-issue instruction units issues multiple “wide instructions” on each cycle. Each “wide instruction” holds several normal instructions, and each of them corresponds to an operation in a different functional unit.

Superscalar and VLIW processors

Compiling for multiple-issue processors

- Issues multiple instructions by executing a single "wide instruction on each cycle."

Superscalar and VLIW processors

Compiling for multiple-issue processors

- Issues multiple instructions by executing a single "wide instruction on each cycle."
- Statically Scheduled. Onus on the Compiler or the programmer to manage the execution schedule.

Superscalar and VLIW processors

Compiling for multiple-issue processors

- Issues multiple instructions by executing a single “wide instruction on each cycle.”
- Statically Scheduled. Onus on the Compiler or the programmer to manage the execution schedule.
- All hazards determined and indicated by the compiler (often implicitly).

Superscalar and VLIW processors

Compiling for multiple-issue processors... contd.

- No need of special look-ahead hardware as opposed to Superscalar processors. Hence, explicitly scheduled.

Superscalar and VLIW processors

Compiling for multiple-issue processors... contd.

- No need of special look-ahead hardware as opposed to Superscalar processors. Hence, explicitly scheduled.
- Compiler must recognize when operators are not related by dependence.

Superscalar and VLIW processors

Compiling for multiple-issue processors... contd.

- No need of special look-ahead hardware as opposed to Superscalar processors. Hence, explicitly scheduled.
- Compiler must recognize when operators are not related by dependence.
- Compiler must schedule instructions such that it requires fewest possible cycles.

Superscalar and VLIW processors

Compiler Techniques

- Loop unrolling
- Local scheduling
- Global scheduling - trace scheduling
- Software pipelining
- Superblock scheduling

Superscalar and VLIW processors

Examples

- 5 Operations
- 1 Integer operation (could be a branch)
- 2 PF operations
- 2 Memory references
- Instruction length 80 - 120

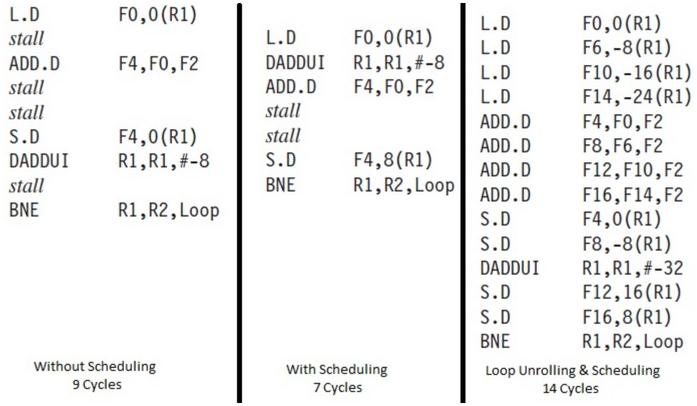
Superscalar and VLIW processors

Examples

```
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 (per DW)
       BNE     R1,R2,Loop     ;branch R1!=R2
```

Superscalar and VLIW processors

Comparison



```
L.D       F0,0(R1)
stall
ADD.D     F4,F0,F2
stall
stall
S.D       F4,0(R1)
DADDUI    R1,R1,#-8
stall
BNE       R1,R2,Loop
```

Without Scheduling
9 Cycles

```
L.D       F0,0(R1)
DADDUI    R1,R1,#-8
ADD.D     F4,F0,F2
stall
stall
S.D       F4,8(R1)
BNE       R1,R2,Loop
```

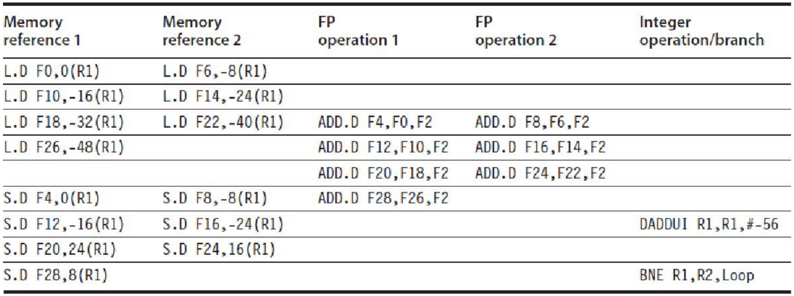
With Scheduling
7 Cycles

```
L.D       F0,0(R1)
L.D       F6,-8(R1)
L.D       F10,-16(R1)
L.D       F14,-24(R1)
ADD.D     F4,F0,F2
ADD.D     F8,F6,F2
ADD.D     F12,F10,F2
ADD.D     F16,F14,F2
S.D       F4,0(R1)
S.D       F8,-8(R1)
DADDUI    R1,R1,#-32
S.D       F12,16(R1)
S.D       F16,8(R1)
BNE       R1,R2,Loop
```

Loop Unrolling & Scheduling
14 Cycles

Superscalar and VLIW processors

Examples



Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
L.D F0,0(R1)	L.D F6,-8(R1)			
L.D F10,-16(R1)	L.D F14,-24(R1)			
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2	
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2	
		ADD.D F20,F18,F2	ADD.D F24,F22,F2	
S.D F4,0(R1)	S.D F8,-8(R1)	ADD.D F28,F26,F2		
S.D F12,-16(R1)	S.D F16,-24(R1)			DADDUI R1,R1,#-56
S.D F20,24(R1)	S.D F24,16(R1)			
S.D F28,8(R1)				BNE R1,R2,Loop

23 operations in 9 cycles (2.5 operations/cycle)

Superscalar and VLIW processors

Problems

- Code size
- Wasted bits in the instruction encoding
- Hazard detection
- Synchronization issue
- More bandwidth
- Binary code compatibility - overcome by EPIC approach

Superscalar and VLIW processors

Concepts to exploit

- Finding parallelism
- Reducing control and data dependences
- Speculation

Superscalar and VLIW processors

Compiling for Multiple Issue Processors

- Recognize dependencies
- Instruction scheduling

Superscalar and VLIW processors

Advantages of Compile-Time Techniques

- No burden on run-time execution
- Takes into account wider range of the program

Superscalar and VLIW processors

Disadvantages of Compile-Time Techniques

- Conservative without runtime information
- Assume Worst-Case

Superscalar and VLIW processors

Detecting and Enhancing Loop-Level Parallelism

- Determining data and name dependencies
- Loop-carried dependence

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

Dependencies:

- S1 uses a value computed by S1 in an earlier iteration
- S2 uses the value, A[i+1], computed by S1 in the same iteration

Superscalar and VLIW processors

Example 2

```
for (i=1; 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 dependencies.

```
A[1] = A[1] + B[1];
for(i=1; i<=99; i=i+1) {
   B[i+1] = C[i] + D[i];
   A[i+1] = A[i+1] + B[i+1];
}
B[101] = C[100] + D[100];
```

Superscalar and VLIW processors

Example 3

```
for (i=1;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.

Superscalar and VLIW processors

Example 4, recurrence

```
for(i=2;i<=100;i=i+1) {
   Y[i] = Y[i-1] + Y[i]; // Dependence distance of 1
}

for(i=6;i<=100;i=i+1) {
   Y[i] = Y[i-5] + Y[i]; // Dependence distance of 5
}
```

The larger the distance, the more potential parallelism can be obtained by unrolling the loop.

Superscalar and VLIW processors

Finding Dependences

- Affine functions
- GCD Test
- Points to analysis

Determining whether a dependence actually exists is an undecidable problem.

Superscalar and VLIW processors

Limitations in dependence analysis

- Restrictions in the analysis algorithms
- Need to analyze behavior across procedure boundaries to get accurate information

Superscalar and VLIW processors

Eliminating dependent computations

- Back substitution
- Copy propagation
- Tree height reduction

Superscalar and VLIW processors

Scheduling and structuring code for parallelism

- Software pipelining: symbolic loop unrolling
- Global code scheduling
    - Trace scheduling
    - Superblocks

Superscalar and VLIW processors

Hardware support for exposing parallelism

- Conditional or predicated instructions
- Compiler speculation
- Memory reference speculation

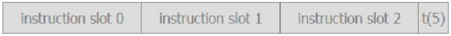
Superscalar and VLIW processors

The Intel IA-64 architecture and Itanium processor

- The IA-64 Register Model
    - 128 64-bit general-purpose registers
    - 128 82-bit floating-point registers
    - 64 1-bit predicate registers
    - 8 64-bit branch registers, which are used for indirect branches
    - a variety of registers used for system control, memory mapping, performance counters, and communication with the OS

Superscalar and VLIW processors

The Intel IA-64 architecture and Itanium processor ...contd.



- Instruction format and support for explicit parallelism
- Instruction groups
- Bundle - 128-bit long instruction words (called bundles) consisting of three 41-bit micro-operations and a 5-bit template field. Multiple bundles can be issued per clock cycle (number is defined by implementation).
- Template field:
    - Helps decode and route instructions
    - Indicates the location of stops that mark the end of groups of micro-operations that can execute in parallel

Superscalar and VLIW processors

Conclusion

- Same basic structure and similar sustained issue rates for the last 5 years.
- Clock rates are 1020 times higher, the caches are 48 times bigger, there are 24 times as many renaming registers, and twice as many load-store units!
- Result: Performance that is 816 times higher.

Overview

1 ILP overview

2 Compiling for scalar pipelines

3 Superscalar and VLIW processors

4 Vector architectures

Vector architectures

- Vector instructions

Vector architectures

- Vector instructions
- Hardware overview

```
VLOAD VR1, M
VADD VR3, VR2, VR1
```

Vector architectures

Compiling for vector pipelines

- Vector instructions simplify the job task of filling instruction pipelines, but they create challenges for compiler. Such as: ensuring vector instructions exactly implement the loops they're used to encode.

Vector architectures

Compiling for vector pipelines

- Vector instructions simplify the job task of filling instruction pipelines, but they create challenges for compiler. Such as: ensuring vector instructions exactly implement the loops they're used to encode.
- Languages with explicit array operations solve this problem to some extent.

Vectorization

Any single-statement loop that carries no dependence can be directly vectorized because that loop can be run in parallel.

Thus:

```
DO I = 1, N
   X(I) = X(I) + C
ENDDO
```

can be safely rewritten as:

```
X(1:N) = X(1:N) + C
```

Vectorization

... On the other hand, consider:

```
DO I = 1, N
   X(I+1) = X(I) + C
ENDDO
```

It carries a dependence. So the transformation to the statement...

```
X(2:N+1) = X(1:N) + C
```

... would be incorrect, since, on each iteration, the sequential version uses a value of X that is computed on the previous iteration.

Loop parallelization

There's a theorem about that...

Theorem

It is valid to convert a sequential loop to a parallel loop if the loop carries no dependence.

What about loop vectorization?

Is there a theorem about that?

Theorem

A statement contained in at least one loop can be vectorized by directly rewriting in Fortran 90 if the statement is not included in any cycle of dependencies.

Simple vectorization

```
procedure vectorize (L, D)

   // L is the maximal loop nest containing the statement.
   // D is the dependence graph for statements in L.

   find the set {S[1], S[2], ... , S[m]} of maximal strongly-connected
      regions in the dependence graph D restricted to L
      (use Tarjan's strongly-connected components algorithm);

   construct L[Pi] from L by reducing each S[i] to a single node and
      compute D[Pi], the dependence graph naturally induced on
      L[Pi] by D;

   let {Pi[1], Pi[2], ... , Pi[m]} be the m nodes of L[Pi] numbered in an order
      consistent with D[Pi] (use topological sort to do the ordering);

   for i = 1 to m do begin

       if P[i] is a dependence cycle then
          generate a DO-loop around the statements in Pi[i];
       else
          directly rewrite the single-statement Pi[i] in Fortran 90,
          vectorizing it with respect to every loop containing it;
       end

end
```

However...

Simple vectorization algorithm misses some opportunities for vectorization. Consider:

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

However...

Simple vectorization algorithm misses some opportunities for vectorization. Consider:

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

There is a dependence from S to itself with the distance vector (1,0) and direction vector $(<,=)$. Thus, statement S is contained in a dependence cycle, so the simple algorithm will not vectorize it.

However... (contd.)

Although we can vectorize the inner loop like so:

```
    DO I = 1, N
S       A(I+1,1:M) = A(I,1:M) + B
    ENDDO
```

However... (contd.)

Although we can vectorize the inner loop like so:

```
    DO I = 1, N
S       A(I+1,1:M) = A(I,1:M) + B
    ENDDO
```

Solution?

This suggests a recursive approach to the problem of multidimensional vectorization.

Solution?

This suggests a recursive approach to the problem of multidimensional vectorization.

- First, attempt to generate vector code at the outermost loop level.

Solution?

This suggests a recursive approach to the problem of multidimensional vectorization.

- First, attempt to generate vector code at the outermost loop level.
- If dependences prevent that, then run the outer loop sequentially, thereby satisfying the dependences carried by that loop, and try again one level deeper, ignoring dependences carried by the outer loop.

Multi-level vector code generation algorithm

```
procedure codegen(R, k, D)
   // R is the region for which we must generate code.
   // k is the minimum nesting level of possible parallel loops.
   // D is the dependence graph among statements in R..

   find the set {S[1], S[2], ... , S[m]} of maximal strongly-connected
      regions in the dependence graph D restricted to R
      (use Tarjan's algorithm);

   construct R[Pi] from R by reducing each S[i] to a single node and
      compute D[Pi], the dependence graph naturally induced on
      R[Pi] by D;

   let {Pi[1], Pi[2], ... , Pi[m]} be the m nodes of R numbered in an order
      consistent with D (use topological sort to do the numbering);
```

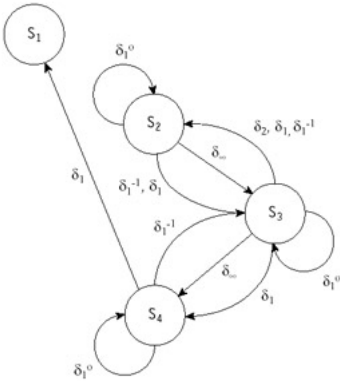
(next slide...)

Multi-level vector code generation algorithm (contd.)

```
   for i = 1 to m do begin

      if P[i] is cyclic then begin

         generate a level-k DO statement;

         let D[i] be the dependence graph consisting of all
            dependence edges in D that are at level k+1 or greater
            and are internal to Pi[i];

        codegen (Pi[i], k+1, D[i]);

        generate the level-k ENDDO statement;
    end

    else

       generate a vector statement for Pi[i] in Rho(P[i])-k+1 dimensions,
          where Rho(Pi[i]) is the number of loops containing Pi[i];

    end

end codegen
```

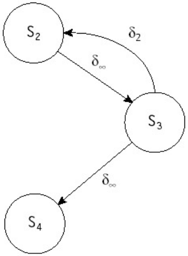
Illustration

```
    DO I = 1, 100
S1       X(I) = Y(I) + 10
       DO J = 1, 100
S2          B(J) = A(J,N)
          DO K = 1, 100
S3               A(J+1,K) = B(J) + C(J,K)
          ENDDO
S4          Y(I+J) = A(J+1, N)
       ENDDO
    ENDDO
```



Dependence graph.

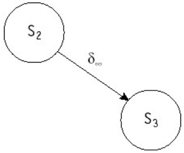
Illustration (contd.)



Dependence graph for S[2], S[3], S[4] after removing level-1 dependencies.

```
DO I = 1, 100
   DO J = 1, 100
      codegen({S2,S3},3})
   ENDDO
   Y(I+1:I+100) = A(2:101,N)
ENDDO
X(1:100) = Y(1:100) + 10
```

Illustration (contd.)



Dependence graph for S[2], S[3] after removing level-2 dependencies.

```
DO I = 1, 100
   DO J = 1, 100
      B(J) = A(J,N)
      A(J+1,1:100) = B(J) + C(J,1:100)
   ENDDO
   Y(I+1:I+100) = A(2:101,N)
ENDDO
X(1:100) = Y(1:100) + 10
```

Concluding remarks

- *Dependence* is the primary tool used by compilers in analysis.

Concluding remarks

- *Dependence* is the primary tool used by compilers in analysis.
- Any transformation that reorders the execution of statements in the program preserves correctness if the transformation preserves the order of source and sink of every dependence in the program.

Concluding remarks

- *Dependence* is the primary tool used by compilers in analysis.
- Any transformation that reorders the execution of statements in the program preserves correctness if the transformation preserves the order of source and sink of every dependence in the program.
- This can be used as an effective tool to determine when it is safe to parallelize or vectorize a loop.

References

- *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*, Randy Allen and Ken Kennedy.
- *Compiler Transformations for High-Performance Computing*, Bacon, David F., Susan L. Graham, and Oliver J. Sharp. http://portal.acm.org/citation.cfm?doid=197405.197406.