VU2 185.324

Compilation Techniques for VLIW Architectures

Dietmar Ebner ebner@complang.tuwien.ac.at
Florian Brandner brandner@complang.tuwien.ac.at

http://complang.tuwien.ac.at/cd/vliw

Last Lectures (1)

- Number representation
  - Floating-point vs. fixed-point
  - Floating-point emulation
- Embedded C extensions
  - Named address spaces
  - Saturated/fixed-point arithmetic

Last Lectures (2)

- Engineering a compiler
  - Long living (>10 years)
  - Design trade offs
  - Structure: Front-end, middle-end, back-end
- Optimization trade offs
  - Performance/runtime
  - Code-size
  - Power efficiency

Last Lectures (3)

- Profiling
  - Node vs. edge prof. / Call graph vs. CFG
  - Instrumentation, sampling, hw/simulator support
  - Problems: Representative input, keeping it up-to-date, may alter observed behavior
- Loop Unrolling
  - Duplicate the loop body
  - Pre/post-conditioning
  - Reductions
  - Trade offs: better performance, increased code-size

Assignments (1)

Assignment 8

In a clustered VLIW, we have the choice of providing implicit or explicit copy operations, for which in the implicit case operands must include encoding bits for the full register specifier, and in the explicit case we only need a cluster bit in the encoding, at the expense of requiring extra copy operations to move values between clusters. For a 4-way VLIW architecture with 32 registers per cluster draw a figure showing the number of bits in an instruction associated with the register specifier in the two cases. Compute how many (%) intercluster copy operations can be issued before the explicit copy mechanism becomes less efficient than the implicit copy.

Assume: 4 Clusters, 32 registers each vs. unified register file with 4x32 registers
Each operation has 3 register operands (2xR, 1xW)

Bits	Overhead	Operand 1	Operand 2	Operand 3
Unified	0	2+5	2+5	2+5
Clustered	1	5	5	5
Copy	0	0	2+5	5

```
n * 3 * 7 = n * (1 + 15) + n * c (1 + 12)
21 = 16 * c * 13
5/13 = c
```


Assignments (2)

Assignment 9

Write C code that implements the multiplication of two 24-bit fractional values to produce a 48-bit fractional value, assuming 32-bit registers.

```
Input: unsigned v1, unsigned v2
Output:  unsigned LO, unsigned HI

unsigned a, b, c, d, bd, ad, cb, ac;
unsigned mid, mid2, carry_mid = 0;

a = (v1 >> 16) & 0xffff;
b = v1 & 0xffff;
c = (v2 >> 16) & 0xffff;
d = v2 & 0xffff;

bd = b * d;
ad = a * d;
cb = c * b;
ac = a * c;
```

```
mid = ad + cb;
if (mid < ad || mid < cb)
  /* Arithmetic overflow or carry-out */
  carry_mid = 1;

mid2 = mid + ((bd >> 16) & 0xffff);
if (mid2 < mid || mid2 < ((bd >> 16) & 0xffff))
  /* Arithmetic overflow or carry-out */
  carry_mid += 1;

LO = (bd & 0xffff) | ((mid2 & 0xffff) << 16);
HI = ac + (carry_mid << 16) + ((mid2 >> 16) &
                                0xffff);
```

Source: SPIM Simulator

Assignments (3)

Assignment 10

Assume a 4-way VLIW architecture that has a unified register file with 32-bit registers (r0-r31), and the usual arithmetic and logic operations (+,-,*,/,&,|,etc.) and a multiply-accumulate (mac), 8 predicate registers (p0-p7), and according comparison operations (==, !=, >, <, etc.). Memory can be accessed using load/store operations (ld, st). The architecture supports partial predication, and offers a `select` instruction. There are no restrictions on the instruction bundling from the architecture or the encoding. Multiply, multiply-accumulate, and load operations take at least 3 cycles, all other operations finish within 1 cycle. There is no hazard detection implemented.

```
int foo(int a[], int n) {
  int i, min;
  min = UINT_MAX;
  for (i = 0; i < n; i++) {
   min = a[i] < min ? a[i] : min;
  }
  return min;
}
```

```
foo: r0 = UINT_MAX;
     p0 = r2 == 0;
     if (p0) goto L1;
     r2 = r2 * 4; // size of int
     nop
     nop          // wait for multiply
     r2 = r1 + r2;    // end of array
L0:  r3 = ld(r1);
     r1 = r1 + 4;     // next element
     p0 = r1 >= r2;   // end?
     p1 = r3 < r0;
     r0 = p1 ? r3 : r0; // min
     if (p0) goto L1; // exit loop
     goto L0;
L1:  return;
```


Assignments (4)

Assignment 10

– Unrolling with pre-conditioning, reductions and ILP

```
foo: p0 = r2 == 0;      r10 = r2 & 3;       r0 = UINT_MAX;      r2 = r2 & ~3;
     if (p0) goto Exit; p1 = r10 != 0;
     if (p1) goto L0;
     goto L1;

// pre-conditioning
L0: r3 = ld(r1);          r1 = r1 + 4;      r10 = r10 - 1;
    p1 = r10 != 0;
    nop   // wait for load
    p1 = r3 < r0;
    if (p1) goto L0;      r0 = p1 ? r3 : r0;

// prepare for unrolled loop
L1: p0 = r2 == 0;           r2 = r2 << 2;   r22 = UINT_MAX;   r23 = UINT_MAX;
    if (p0) goto Exit;      r2 = r1 + r2;   r24 = UINT_MAX;
```

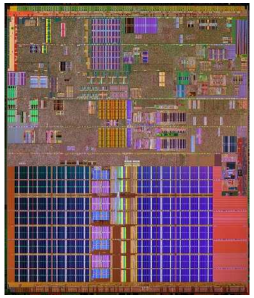
Assignments (5)

Assignment 10

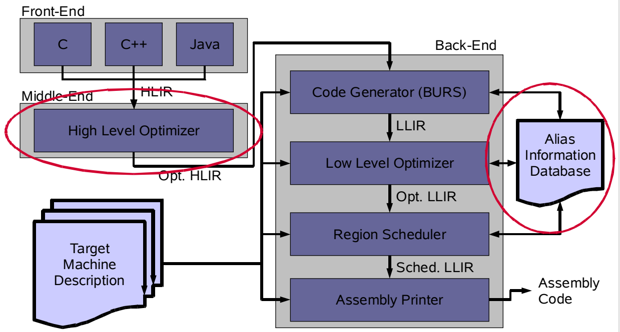
```
// unroll by 4, and use reductions to eliminate data dependencies
L2: r11 = ld(r1);        r12 = ld(r1 + 4);      r13 = ld(r1 + 8);       r14 = ld(r1 + 12);
    r1 = r1 + 16;        // next 4 elements
    p0 = r1 >= r2;       // end?
    p1 = r11 < r0;      p2 = r12 < r22;        p3 = r13 < r23;         p4 = r14 < r24;
    r0 = p1 ? r11 : r0; r22 = p2 ? r12 : r22; r23 = p3 ? r13 : r23;  r24 = p4 ? r14 : r24;
    if (p0) goto L2;     // exit loop?

// reduction
L3:
    p1 = r21 < r0;            p2 = r22 < r23;
    r0 = p1 ? r21 : r0;       r11 = p2 ? r22 : r23;
    p1 = r11 < r0;
    r0 = p1 ? r11 : r0;

// result in r0
Exit: return;
```


In Today's Lecture

- High-level Optimizations
  - Control Flow Graph
  - Function Inlining
  - Alias Analysis
- Loop Transformations
  - Definition of Loops
  - Loop Optimizations
- Data Cache Optimizations



Phases of an ILP Oriented Compiler

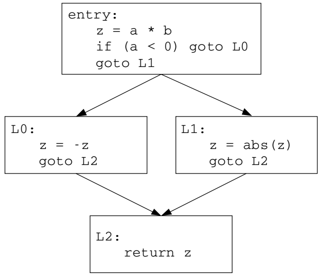


Control Flow Graph

- Basic block
  - Sequence of computations
  - One single entry, one single exit
- Control flow graph (CFG)
  - Nodes: Basic blocks
  - Edges: Jumps between blocks
  - Top most block usually called entry

Example: CFG

```
int foo(int a, int b) {
  int z = a * b;
  if (a < 0)
    z = -z;
  else if (b < 0)
    z = abs(z);
  return z;
}
```



Scalar Optimizations

- Mostly simple transformations
  - Usually improve performance & reduce code-size
  - Available in virtually every compiler
- Examples:
  - Subexpression elimination
  - Copy propagation
  - Copy elimination
  - Dead-code elimination
  - Strength reduction

Copy Propagation

```
if () {
  a = b;
  c = (a + e) * 1024;
  d = b + e;
  b = 7;
}
else {
  b = 7;
  x = a + c;
}
return b + c + d
```

```
if () {
  a = b;
  c = (b + e) * 1024;
  d = b + e;
  b = 7;
}
else {
  b = 7;
  x = a + c;
}
return b + c + d
```


Constant Propagation

```
if () {
  a = b;
  c = (b + e) * 1024;
  d = b + e;
  b = 7;
}
else {
  b = 7;
  x = a + c;
}
return b + c + d
```

```
if () {
  a = b;
  c = (b + e) * 1024;
  d = b + e;
  b = 7;
}
else {
  b = 7;
  x = a + c;
}
return 7 + c + d
```

Common Subexpr. Elimination

```
if () {
  a = b;
  c = (b + e) * 1024;
  d = b + e;
  b = 7;
}
else {
  x = a + c;
  b = 7;
}
return 7 + c + d
```

⇒

```
if () {
  a = b;
  tmp = b + e;
  c = tmp * 1024;
  d = tmp;
  b = 7;
}
else {
  x = a + c;
  b = 7;
}
return 7 + c + d
```


Dead Code Elimination

```
if () {
  a = b;
  tmp = b + e;
  c = tmp * 1024;
  d = tmp;
  b = 7;
}
else {
  x = a + c;
  b = 7;
}
return 7 + c + d
```

⇒

```
if () {
  a = b;
  tmp = b + e;
  c = tmp * 1024;
  d = tmp;
  b = 7;
}
else {
  x = a + c;
  b = 7;
}
return 7 + c + d
```


Strength Reduction

```
if () {
  tmp = b + e;
  c = tmp * 1024;
  d = tmp;
}
return 7 + c + d
```

⇒

```
if () {
  tmp = b + e;
  c = tmp << 10;
  d = tmp;
}
return 7 + c + d
```


Function Inlining

- Replace a function call by the functions body
  - ✓ Eliminates call overhead (argument passing, etc.)
  - ✓ Enlarges the scope for other optimizations
  - ✗ Increases code-size (effects on cache)
- Typically done using simple heuristics
  - Code-size (caller and callee)
  - Number of call sites
  - Profiling information
- Sometimes controlled by user (*inline* keyword)

Data Dependencies

- Arise from reading/writing data
  - Read-after-write (true dependence)
  - Write-after-Read (anti dependence)
  - Write-after-write (output dependence)
- Dependence information
  - Required for many optimizations
  - Determine if calculations are independent
  - Represented as graphs (data dependence graph)

Aliasing Problem

- Obtaining dependence information
  - Easy for scalar variables
  - Hard problem for memory locations
  - Pointer may refer to different locations at different program points

```
int x, y, z;
int *a = &z;

(*a) = 101;
```

```
if (z > 100)
  a = &x;
else
  a = &y;
(*a)++;
```


Alias Analysis

- Analysis tackling the aliasing problem
  - Determine to which memory locations a pointer may refer
- Possible memory locations
  - Local/global variables with address taken (& in C)
  - Heap references (malloc, new)
  - Arguments passed by reference

Alias Analysis (2)

- Flow-insensitive AA
  - Alias information independent of program locations
- Flow-sensitive AA
  - Alias information for each program point
  - More precise & more complex
- May vs. must aliasing
  - Determine if a pointer is guaranteed to refer to a particular memory location

Example: Flow Sensitivity

```
(1) int x, y, z;
(2) int *a = &z;
(3) (*a) = 101;
(4) if (z > 100)
(5)   a = &p;
(6) else
(7)   a = &q;
(8) (*a)++;
```

Flow-insensitive AA:
a **may** alias {*x*, *y z*}

Flow-sensitive AA:
(3) *a* **must** alias {*z*}
(6) a **must** alias {p}
(7) a **must** alias {q}
(8) a **may** alias {p, q}

Analysis Scope

- Intra procedural
  - Only consider the scope of a function
  - Conservative assumptions on incoming arguments
  - Similar for result values of calls
- Inter procedural
  - Consider the complete call graph
  - Context-sensitive vs. Context-insensitive analysis
  - More precise & more complex

Beyond Alias Analysis

- More precise information on heap objects
  - Statically detect dangling/NULL pointers
  - Detect memory leaks
  - Detect shared memory cells
  - Reachability of memory cells (garbage)
- Shape analysis
  - Determine a finite representation of dynamic data structures (lists, trees, DAGs, etc.)

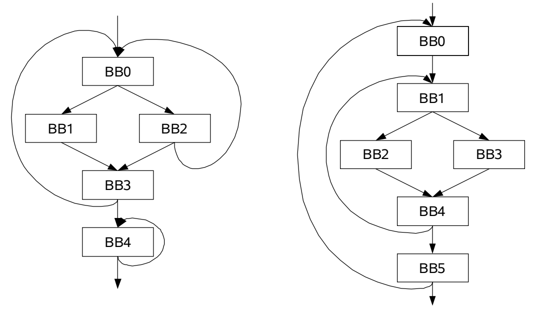
Loop Transformations

- Loops contribute a large amount of the execution time of programs
  - Optimizing loops is attractive
  - Limited scope, thus
    allows to use more sophisticated techniques
- What is a loop?

Loop Definition

- A set of basic blocks such that edges connecting these blocks form a cycle
  - Distinctive loop header
  - All blocks are reachable from the header
  - For any block there is a path to the header
  - All paths from a block outside the loop to a block inside the loop go through the header
- These loops are called natural loops

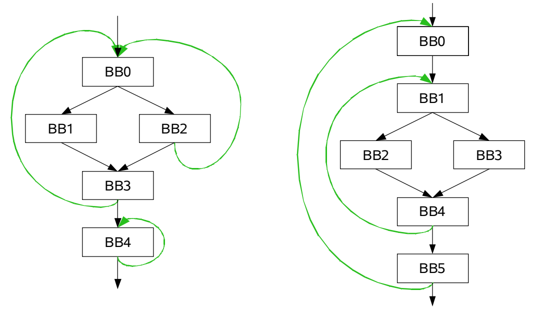
Example: Loops



Identifying Loops

- Using dominance relation
  - Block ***A*** dominates another block ***B*** iff all paths from the entry node to ***B*** go through ***A***
  - Efficiently calculated using depth first search (DFS)
- An edge of the CFG is a backegde iff the head of the edge dominates its tail
- The head of a backedge is a loop header

Example: Backedges

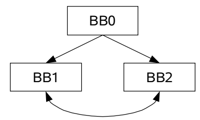


Reducible Flow Graphs

- A CFG is called reducible iff we can partition the edges into two sets:
  - backedges
  - forwardedges
- Considering only forwardedges, the CFG becomes a DAG
- A reducible CFG contains only natural loops
- Every cycle contains at least one backedge

Irreducible Flow Graphs

- Not all CFGs for real programs are reducible:



- These cases are rare, nevertheless one has to account for them
- Usually loop optimizations target natural loops

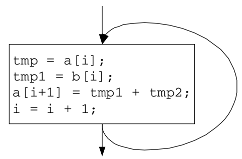
Dependencies in Loops

- Loops complicate dependence analysis
  - Loop carried dependencies
  - Aliasing of pointers/overlapping arrays
  - Distance vectors
- Dependence testing
  - Induction variables/subscript analysis
  - Array dependence analysis
  - Delta - test

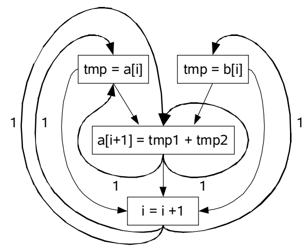
Example: Dependencies

```
for (i = 0; i < n; i++) {
  a[i+1] = a[i] + b[i];
}
```

Original C-code



Controll flow graph



Data dependence graph

Loop Optimizations

- Typical goals
  - Reduce the loop overhead
  - Increase data reuse - by modifying access patterns
  - Increase parallelism - by eliminating dependencies
- Examples:
  - Loop fusion, distribution, interchange
  - Loop skewing, peeling
  - Many, many, more

Renaming

```
for (i = 0; i < n; i++) {
  t = a[i] + b[i];
  c[i] = t + t;
  t = d[i] - b[i];
  a[i+1] = t * t;
}
```

⇒ Scalars

```
for (i = 0; i < n; i++) {
  t1 = a[i] + b[i];
  c[i] = t1 + t1;
  t2 = d[i] - b[i];
  a[i+1] = t2 * t2;
}
```

```
for (i = 0; i < n; i++) {
  a[i] = a[i-1] + x;
  y[i] = a[i] + z;
  a[i] = b[i] + c;
}
```

⇒ Arrays

```
for (i = 0; i < n; i++) {
  a1[i] = a[i-1] + x;
  y[i] = a1[i] + z;
  a[i] = b[i] + c;
}
```


Scalar Expansion

```
for (j = 0; j < m; j++) {
  for (i = 0; i < n; i++) {
    t = 0;
    for (k = 0; k < l; k++) {
      t = t + a[i][k] * b[k][j];
    }
    c[i][j] = t;
  }
}
```

⇒

```
for (j = 0; j < m; j++) {
  for (i = 0; i < n; i++) {
    T[i] = 0;
    for (k = 0; k < l; k++) {
      T[i] = T[i] + a[i][k] * b[k][j];
    }
    c[i][j] = T[i];
  }
}
```


Loop Distribution

```
for (j = 0; j < m; j++) {
  for (i = 0; i < n; i++) {
    T[i] = 0;
    for (k = 0; k < l; k++) {
      T[i] = T[i] + a[i][k] * b[k][j];
    }
    c[i][j] = T[i];
  }
}
```

⇒

```
for (j = 0; j < m; j++) {
  for (i = 0; i < n; i++) {
    T[i] = 0;
  }
  for (i = 0; i < n; i++) {
    for (k = 0; k < l; k++) {
      T[i] = T[i] + a[i][k] * b[k][j];
    }
  }
  for (i = 0; i < n; i++) {
    c[i][j] = T[i];
  }
}
```

Loop Interchange

```
for (j = 0; j < m; j++) {
  for (i = 0; i < n; i++) {
    T[i] = 0;
  }
  for (i = 0; i < n; i++) {
    for (k = 0; k < l; k++) {
      T[i] = T[i] + a[i][k] * b[k][j];
    }
  }
  for (i = 0; i < n; i++) {
    c[i][j] = T[i];
  }
}
```

```
for (j = 0; j < m; j++) {
  for (i = 0; i < n; i++) {
    T[i] = 0;
  }
  for (k = 0; k < l; k++) {
    for (i = 0; i < n; i++) {
      T[i] = T[i] + a[i][k] * b[k][j];
    }
  }
  for (i = 0; i < n; i++) {
    c[i][j] = T[i];
  }
}
```


Index-Set splitting

```
for (i = 0; i < 100; i++) {
  a[i+20] = a[i] + x;
}
```

Threshold analysis + strip mining

```
for (I = 0; I < 100; I+=20) {
  for (i = I; I < I + 19; i++)
    a[i+20] = a[i] + x;
}
```

```
for (i = 0; i < n; i++) {
  a[i] = a[i] + a[0];
}
```

Loop Peeling

```
a[0] = a[0] + a[0];
for (i = 1; i < n; i++) {
  a[i] = a[i] + a[0];
}
```

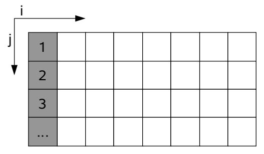

Cache Optimizations

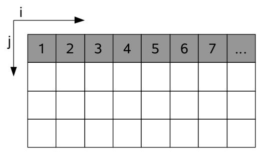
- Reduce the number of cache misses
  - Reorganize data
  - Reshape access patterns
  - Reduce the number of memory accesses
  - Prefetching
- In loops
  - Improve spatial and temporal locality

Loop Interchange (2)

```
for (i = 0; i < n; i++) {
  for (j = 0; j < m; j++) {
    a[i][j] = a[i][j] + b[i][j]
  }
}
```

```
for (j = 0; j < m; j++) {
  for (i = 0; i < n; i++) {
    a[i][j] = a[i][j] + b[i][j]
  }
}
```

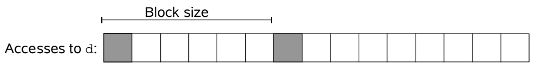




Loop Blocking

```
for (j = 0; j < m; j++) {
  for (i = 0; i < n; i++) {
    d[i] = d[i] + b[i][j]
  }
}
```

```
for (I = 0; I < n; I+=S) {
  for (j = 0; j < m; j++) {
    for (i = I; i < min(I+S-1,n); i++)
      d[i] = d[i] + b[i][j]
  }
}
```



Outlook

- Code layout
  - Block & function placement
- Code generation
  - Instruction selection on trees
  - DAG based approaches