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
Assignment 8
In a clustered VLIW, we have the choice of avoiding implicit or explicit copy operations, for which in the implicit case operands must include the register for the full register specifier, and in the explicit case we only need a cluster bit in the encoding, at the expense of register-to-register copy operations to move the register content. As an exercise in compiler optimization, write a VLIW assembly language figure showing the number of bits in an instruction associated with the register specifier in two cases. Compute how many instructions copy operations can be issued before the explicit copy mechanism becomes less efficient than the explicit copy.

Assumes 4 clusters, 32 registers each vs. unified register file with 4x32 registers.
Each operation has 3 register operands (24x, 16x).

\[
\begin{align*}
3 \times 7 + 3 & = (3 \times 7) + (3 \times 1) \\
& = 21 + 3 \\
& = 24
\end{align*}
\]

Assignment 9
Write a code that implements the multiplication of two 32-bit fractional values to produce a 48-bit fractional value, assuming 32-bit register.

Assignments (4)
Assignment 10
Unrolling with pre-conditioning, reductions and ILP

```c
#define MAX(a, b) (((a) > (b)) ? (a) : (b))
#define MIN(a, b) (((a) < (b)) ? (a) : (b))

int foo(int a[], int n) {
    int i, min;
    min = UINT_MAX;
    for (i = 0; i < n; i++) {
        int val = MIN(val, MAX(a[i], a[i] + 1));
    }
    return min;
}
```

Assignments (3)
Assignment 10
Assume a 4x4 VLIW architecture that has a unified register file with 32-bit registers (256-bit), and execute optimized arithmetic operations (<, >, +, -), load multiple-accumulate, store, load/store register (gr1), and according comparison operations (<, >, ==, etc.). Memory can be accessed using load/store operations. The architecture supports parallel instruction, and offers a select instruction. There are no restrictions on the instruction bundling within the architecture or the encoding. Multiple-accumulate, and load/store instruction take at least 3 cycles, all other instruction 1-2 cycles. There is no hazard detection implemented.

```c
int foo(int a[], int n) {
    int i, min;
    min = UINT_MAX;
    for (i = 0; i < n; i++) {
        int val = MIN(val, MAX(a[i], a[i] + 1));
    }
    return min;
}
```

Assignments (2)
Assignment 9
Write a code that implements the multiplication of two 32-bit fractional values to produce a 48-bit fractional value, assuming 32-bit register.

Assignments (1)
Assignment 8
In a clustered VLIW, we have the choice of avoiding implicit or explicit copy operations, for which in the implicit case operands must include the register for the full register specifier, and in the explicit case we only need a cluster bit in the encoding, at the expense of register-to-register copy operations to move the register content. As an exercise in compiler optimization, write a VLIW assembly language figure showing the number of bits in an instruction associated with the register specifier in two cases. Compute how many instructions copy operations can be issued before the explicit copy mechanism becomes less efficient than the explicit copy.

Assumes 4 clusters, 32 registers each vs. unified register file with 4x32 registers.
Each operation has 3 register operands (24x, 16x).

\[
\begin{align*}
3 \times 7 + 3 & = (3 \times 7) + (3 \times 1) \\
& = 21 + 3 \\
& = 24
\end{align*}
\]
Assignments (5)

Assignment 10

// unroll by 4, and use reductions to eliminate data dependencies
ll:  r1 = ld(r1);  r2 = ld(r1 + 4); r3 = ld(r1 + 8);    r4 = ld(r1 + 12);
    if (r4 < r2) goto ll; // unroll loop

l1:  p0 = r11 < r1;  p1 = r12 < r2;  p2 = r13 < r3;  p3 = r14 < r4;

// reduction
ll:  r1 = r1 + 16;    // next 4 elements
     p0 = r1 >= r2;    // end?
     p1 = r11 < r0;      p2 = r12 < r22;     p3 = r13 < r23;      p4 = r14 < r24;

// unroll by 4, and use reductions to eliminate data dependencies

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

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

Constant Propagation
Common Subexpr. Elimination

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

Dead Code Elimination

```c
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

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

```
if (x == 100)
    a = x;
else
    a = y;

(*a) = 101;
```

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 (x > 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 (x)
- (4) a must alias (x)
- (5) 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

Identifying Loops

- Using dominance relation
  - Block A dominates another block B if 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 backedge iff the head of the edge dominates its tail
- The head of a backedge is a loop header
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];
}

tmp = a[i];
tmp1 = b[i];
a[i+1] = tmp1 + tmp2;
i = i + 1;

Original code

Control 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

Scalar Expansion

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];
  }
}

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;
}

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;
}

Arrays

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

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

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

\[
\text{for } (i = 0; i < n; i++) \\
\quad \text{for } (j = 0; j < m; j++) \\
\quad \quad T[i][j] = 0; \\
\text{for } (i = 0; i < n; i++) \\
\quad \text{for } (k = 0; k < l; k++) \\
\quad \quad T[i][j] = T[i][j] + a[i][k] \times b[k][j]; \\
\text{for } (i = 0; i < n; i++) \\
\quad c[i][j] = T[i];
\]

Index-Set splitting

\[
\text{for } (i = 0; i < n; i++) \\
\quad \text{for } (j = 0; j < m; j++) \\
\quad A[i][j] = A[i][j] + B[i][j]; \\
\text{for } (i = 0; i < n; i++) \\
\quad c[i][j] = A[i][j];
\]

Loop Interchange (2)
Loop Blocking

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

```c
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