CS415 Compilers,

Code Generation,

Local Common Subexpression Elimination Optimization

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
Announcements

• Homework 7 has been posted

• Third project (local CSE) will be posted later this week, due May 2

• Second project has been posted; due date: Monday, April 23

• Midterm will be returned in recitation
  Midterm sample solutions (A and B versions) have been posted; Please review the sample solutions, and if you have any remaining question, please see the TA/me grader for the problem:
  1: Rajanya
  2: Ulrich Kremer
  3: Chen
  4: Jeff

If you have any concerns with respect to the midterm, please talk to TA and/or instructor by Monday, April 23. No more challenges after that date.
Announcements

Final exam: Tuesday, May 8, noon-3:00pm, Physics Lecture Hall, Busch Campus

Conflicts?

• More than two (2) final exams on one calendar day
• More than two (2) final exams scheduled in consecutive periods (ex: A student has exams scheduled for 4:00-7:00 pm and 8:00-11:00 pm on one day and 8:00-11:00 am on the following day.)
• Two final exams scheduled for the same exam period.

If you have a conflict, please send me the details of your conflict: class, email of instructor, time of scheduled exam
A compiler is a lot of fast stuff followed by some hard problems

→ The hard stuff is mostly in code generation and optimization
→ For superscalars, its allocation & scheduling that is particularly important
The key code quality issue is holding values in registers

- When can a value be safely allocated to a register?
  - When only 1 name can reference its value (no aliasing)
  - Pointers, parameters, aggregates & arrays all cause trouble

- When should a value be allocated to a register?
  - When it is both safe & profitable

Encoding this knowledge into the IR (register-register model)

- Use code shape to make it known to every later phase
- Assign a virtual register to anything that can go into one
- Load or store the others at each reference

Relies on a strong register allocator
Recursive Treewalk vs. Ad-hoc SDT

Goal : Expr { $$ = $1; } ;
Expr : Expr PLUS Term
      { t = NextRegister();
        emit(add,$1,$3,t); $$ = t; }
      | Expr MINUS Term {...}
      | Term { $$ = $1; } ;
Term : Term TIMES Factor
      { t = NextRegister();
        emit(mult,$1,$3,t); $$ = t; }
      | Term DIVIDES Factor {...}
      | Factor { $$ = $1; } ;
Factor : NUMBER
        { t = NextRegister();
          emit(loadI,val($1),none, t );
          $$ = t; }
        | ID
          { t1 = base($1);
            t2 = offset($1);
            t = NextRegister();
            emit(loadAO,t1,t2,t);
            $$ = t; }

int expr(node) {
    int result, t1, t2;
    switch (type(node)) {
        case ×,÷,+,— :
            t1 ← expr(left child(node));
            t2 ← expr(right child(node));
            result ← NextRegister();
            emit (op(node), t1, t2, result);
            break;
        case IDENTIFIER:
            t1 ← base(node);
            t2 ← offset(node);
            result ← NextRegister();
            emit (loadAO, t1, t2, result);
            break;
        case NUMBER:
            result ← NextRegister();
            emit (loadI, val(node), none, result);
            break;
        }
    return result;
}
\[ \text{lhs} \leftarrow \text{rhs} \]

Strategy

- Evaluate \( \text{rhs} \) to a value \((\text{an rvalue})\)
- Evaluate \( \text{lhs} \) to a location (memory address) \((\text{an lvalue})\)
  - If \( \text{lvalue} \) is a register \(\Rightarrow\) move \( \text{rhs} \)
  - If \( \text{lvalue} \) is an address \(\Rightarrow\) store \( \text{rhs} \)
- If \( \text{rvalue} \) & \( \text{lvalue} \) have different types
  - Evaluate \( \text{rvalue} \) to its "natural" type
  - Convert that value to the type of \( \text{lhs value, if possible} \)

Unambiguous scalars may go into registers (no aliasing)
Ambiguous scalars or aggregates go into memory (possible aliasing)
What if the compiler cannot determine the rhs’s type?

- This is a property of the language & the specific program
- If type-safety is desired, compiler must insert a run-time check
- Add a *tag field* to the data items to hold type information

Code for assignment becomes more complex

```plaintext
evaluate rhs
If lhs.type_tag ≠ rhs.type_tag then
    convert rhs to type(lhs) or signal a run-time error
lhs ← rhs
```

This is much more complex than if it knew the types
Compile-time type-checking

- Goal is to eliminate both the runtime check & the tag
- Determine, at compile time, the type of each subexpression
- Use compile-time types to determine if a run-time check is needed

Optimization strategy

- If compiler knows the type, move the check to compile-time
- Unless tags are needed for garbage collection, eliminate them
- If check is needed, try to overlap it with other computation (superscalar or multi-core architectures)
Handling Assignment (with reference counting)

Garbage Collection

The problem with reference counting

• Must adjust the count on each pointer assignment
• Overhead is significant, relative to assignment

Code for assignment becomes

```plaintext
evaluate rhs
lhs→count ← lhs→count - 1
lhs ← addr(rhs)
rhs→count ← rhs→count + 1
```

This adds 1 +, 1 -, 2 loads, & 2 stores

With extra functional units & large caches, this may become either cheap or free. What about power consumption?
How does the compiler handle $A[i, j]$?

First, must agree on a storage scheme

**Row-major order** (most languages)
- Lay out as a sequence of consecutive rows
- Rightmost subscript varies fastest
  - $A[1,1], A[1,2], A[1,3], A[2,1], A[2,2], A[2,3]$

**Column-major order** (Fortran)
- Lay out as a sequence of columns
- Leftmost subscript varies fastest
  - $A[1,1], A[2,1], A[1,2], A[2,2], A[1,3], A[2,3]$

**Indirection vectors** (Java)
- Vector of pointers to pointers to ... to values
- Takes much more space, trades indirection for arithmetic
- Not amenable to analysis
Laying Out Arrays

The Concept

<table>
<thead>
<tr>
<th></th>
<th>1,1</th>
<th>1,2</th>
<th>1,3</th>
<th>1,4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1,1</td>
<td>1,2</td>
<td>1,3</td>
<td>1,4</td>
</tr>
<tr>
<td></td>
<td>2,1</td>
<td>2,2</td>
<td>2,3</td>
<td>2,4</td>
</tr>
</tbody>
</table>

Row-major order

<table>
<thead>
<tr>
<th></th>
<th>1,1</th>
<th>1,2</th>
<th>1,3</th>
<th>1,4</th>
<th>2,1</th>
<th>2,2</th>
<th>2,3</th>
<th>2,4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1,1</td>
<td>1,2</td>
<td>1,3</td>
<td>1,4</td>
<td>2,1</td>
<td>2,2</td>
<td>2,3</td>
<td>2,4</td>
</tr>
</tbody>
</table>

Column-major order

<table>
<thead>
<tr>
<th></th>
<th>1,1</th>
<th>2,1</th>
<th>1,2</th>
<th>2,2</th>
<th>1,3</th>
<th>2,3</th>
<th>1,4</th>
<th>2,4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1,1</td>
<td>2,1</td>
<td>1,2</td>
<td>2,2</td>
<td>1,3</td>
<td>2,3</td>
<td>1,4</td>
<td>2,4</td>
</tr>
</tbody>
</table>

Indirection vectors

These have distinct & different cache behavior.
Computing an Array Address

Declaration: A[low .. high] of ...

\[ A[i] \]

- \[ \text{@A} + (i - \text{low}) \times \text{sizeof(A[1])} \]
- In general: base(A) + (i - low) \times \text{sizeof(A[1])}
Computing an Array Address

Declaration: A[low .. high] of ...

\[ A[i] \]
- \[ @A + (i - \text{low}) \times \text{sizeof}(A[1]) \]
- In general: \[ \text{base}(A) + (i - \text{low}) \times \text{sizeof}(A[1]) \]

\text{int } A[1:10] \Rightarrow \text{low is 1}
Make \text{low 0 for faster access (saves a - )}

Almost always a power of 2, known at compile-time
\Rightarrow \text{use a shift for speed}
Computing an Array Address

Declaration: \( A[\text{low}1 .. \text{high}1, \text{low}2 .. \text{high}2] \) of ...

\[ A[ i ] \]
- \( @A + ( i - \text{low} ) \times \text{sizeof}(A[1]) \)
- In general: \( \text{base}(A) + ( i - \text{low} ) \times \text{sizeof}(A[1]) \)

What about \( A[i_1, i_2] \) ?

Row-major order, two dimensions
\[ @A + (( i_1 - \text{low}_1 ) \times (\text{high}_2 - \text{low}_2 + 1) + i_2 - \text{low}_2) \times \text{sizeof}(A[1]) \]

Column-major order, two dimensions
\[ @A + (( i_2 - \text{low}_2 ) \times (\text{high}_1 - \text{low}_1 + 1) + i_1 - \text{low}_1) \times \text{sizeof}(A[1]) \]

Indirection vectors, two dimensions
\[ *(A[i_1])[i_2] \quad \text{— where } A[i_1] \text{ is, itself, a 1-d array reference} \]

This stuff looks expensive! Lots of implicit +, -, x ops
In row-major order
\[ @A + (i - \text{low}_1) \times (\text{high}_2 - \text{low}_2 + 1) \times w + (j - \text{low}_2) \times w \]

Which can be factored into
\[ @A + i \times (\text{high}_2 - \text{low}_2 + 1) \times w + j \times w - (\text{low}_1 \times (\text{high}_2 - \text{low}_2 + 1) \times w) + (\text{low}_2 \times w) \]

If \( \text{low}_i, \text{high}_i, \) and \( w \) are known, the last term is a constant

Define \( @A_0 \) as
\[ @A - (\text{low}_1 \times (\text{high}_2 - \text{low}_2 + 1) \times w + \text{low}_2 \times w \]

And \( \text{len}_2 \) as \( \text{(high}_2 - \text{low}_2 + 1) \)

Then, the address expression becomes
\[ @A_0 + (i \times \text{len}_2 + j) \times w \]

\( w = \text{sizeof}(A[1,1]) \)

Compile-time constants
One possible approach for code generation:

Loops
- Evaluate condition before loop (if needed)
- Evaluate condition after loop
- Branch back to the top (if needed)

Merges test with last block of loop body

While, for, do, & until all fit this basic model
for (i = 1; i < 100; i++) {
  \textit{body}
}

\textit{next statement}

\begin{itemize}
  \item \texttt{loadI} \texttt{1} \Rightarrow r_1
  \item \texttt{loadI} \texttt{1} \Rightarrow r_2
  \item \texttt{loadI} \texttt{100} \Rightarrow r_3
  \item \texttt{cmp\_GE} \texttt{r_1, r_3} \Rightarrow r_4
  \item \texttt{cbr} \quad r_4 \Rightarrow \texttt{L}_2, \texttt{L}_1
\end{itemize}

\texttt{L}_1: \textit{body}
\begin{itemize}
  \item \texttt{add} \quad r_1, r_2 \Rightarrow r_1
  \item \texttt{cmp\_LT} \quad r_1, r_3 \Rightarrow r_5
  \item \texttt{cbr} \quad r_5 \Rightarrow \texttt{L}_1, \texttt{L}_2
\end{itemize}

\texttt{L}_2: \textit{next statement}
Many modern programming languages include a `break`

- Exits from the innermost control-flow statement
  - Out of the innermost loop
  - Out of a case statement

Translates into a jump

- Targets statement outside control-flow construct
- Creates multiple-exit construct
- `Skip` in loop goes to next iteration
Control Flow

Case Statements
1. Evaluate the controlling expression
2. Branch to the selected case
3. Execute the code for that case
4. Branch to the statement after the case

Parts 1, 3, & 4 are well understood, part 2 is the key
Control Flow

Case Statements
1. Evaluate the controlling expression
2. Branch to the selected case
3. Execute the code for that case
4. Branch to the statement after the case (use break)

Parts 1, 3, & 4 are well understood, part 2 is the key

Strategies
• Linear search (nested if-then-else constructs)
• Build a table of case expressions & binary search it
• Directly compute an address (requires dense case set: jump table)

Surprisingly many compilers do this for all cases!
tries to improve quality of code (may fail in some cases)
optimizer typically consists of multiple passes
different optimization (code improvement) objectives:

- execution time reduction
- reduction in resource requirements (memory, registers)
- (peak) power and energy reduction

criteria for effectiveness of optimizations

- safety - program semantics must be preserved
- opportunity - how often can it be applied?
- profitability - how much improvement?
We will focus on two optimizations:

1. Common subexpression elimination (CSE - local, \textit{ILOC level})

2. Vectorization / parallelization (\textit{source level}) - will do this later

Local CSE reference: ALSU, chapter 8.5.2
Optimization: **Local Common Subexpression Elimination (CSE)**

Source code: \( a(i) \) (1-based indexing)

Basic Block DAG Construction

\[
\begin{align*}
4. & \quad t1 = \text{addr}(a) - 4 \\
5. & \quad t2 = i \times 4 \\
6. & \quad t3 = t1[t2] \\
\ldots
\end{align*}
\]
Optimization: **Local Common Subexpression Elimination (CSE)**

Source code: \[ a(i) \times a(i) \] (1-based indexing)

**DAG Construction**

4. \( t_1 = \text{addr}(a) - 4 \)
5. \( t_2 = i \times 4 \)
6. \( t_3 = t_1[t_2] \)
7. \( t_4 = \text{addr}(a) - 4 \)
8. \( t_5 = i \times 4 \)
9. \( t_6 = t_4[t_5] \)
10. \( t_7 = t_3 \times t_6 \)

\[ \ldots \]
Optimization: **Local Common Subexpression Elimination (CSE)**

Source code: \( a(i) \times a(i) \) (1-based indexing)

\[
\begin{align*}
4. & \quad t1 = \text{addr}(a)-4 \\
5. & \quad t2 = i \times 4 \\
6. & \quad t3 = t1[t2] \\
7. & \quad t4 = \text{addr}(a)-4 \\
8. & \quad t5 = i \times 4 \\
9. & \quad t6 = t4[t5] \\
10. & \quad t7 = t3 \times t6 \\
\end{align*}
\]

\[
\begin{align*}
\text{code generated:} & \quad t1 = \text{addr}[a]-4 \\
\quad & \quad t2 = i \times 4 \\
\quad & \quad t3 = t1[t2] \\
\quad & \quad t7 = t3 \times t3
\end{align*}
\]
How to add a subexpression into a partially constructed DAG:

\[
A = B + C
\]

Is there a node already for \(B + C\)?
- If so, add \(A\) to its list of labels.
- If not:
  - is there a node labeled \(B\) already?
    If not, create a leaf labeled \(B\).
  - Is there a node labeled \(C\) already?
    If not, create a leaf labeled \(C\).
  - Create a node labeled \(A\), for \(+\), with left child \(B\) and right child \(C\).

How to do this? HASHING \(<op, node(opd1), node(opd2)>\)
How to add a subexpression into a partially constructed DAG:

\[
A = B + C
\]

Is there a node already for \(B + C\)? \(<+, \text{node}(B), \text{node}(C)\>\) defined?
- If so, add \(A\) to its list of labels.
- If not:
  - is there a node labeled \(B\) already? \(\text{node}(B)\) defined?
    If not, create a leaf labeled \(B\).
  - Is there a node labeled \(C\) already? \(\text{node}(C)\) defined?
    If not, create a leaf labeled \(C\).
  - Create a node labeled \(A\), for +, with left child \(B\) and right child \(C\). Create node(+) with children \(\text{node}(B), \text{node}(C)\)

How to do this? **HASHING** \(<\text{op}, \text{node}(\text{opd1}), \text{node}(\text{opd2})\>\)
Basic Block DAG Construction

DAG Construction Algorithm

Summary:
- every expression is assigned a value number
  examples: node(a),
            node(4),
            node(<+, valNum1, ValNum2>)
- assignment changes value number associated with LHS variable

- implementation of value numbers
  • use pointers of nodes in DAG
  • use virtual register numbers (code shape encoding!)

You could do this in a single pass in our compiler!
Things to do and next class

Work on the project!

Intermediate representations
Read EaC: Chapter 5.1 - 5.3