A New Viewpoint on Code Generation for Directed Acyclic Graphs

Summary for FDA001

John Wilander
johwi@ida.liu.se

March 22, 2004

1 Contributions of the Paper

In this paper the authors ...

- Generate optimal code for one-register machines (accumulator-based)
- Optimize loads and spills by accounting for commutativity of operators
- Define better machine models regarding current hardware

2 Problems to be Solved

To be able to enhance the (optimal) code generation we need to change the machine model and thus allow for fewer loads and spills. The key idea is to account for commutativity of operators.

2.1 Commutative Operations

Some binary operators are commutative, for example $3 + 2 = 2 + 3$. One of the previous models for accumulator-based machines only had one such operation:

acc ← **acc** *op* **mem**

This meant that an instruction sequence such as `a := a + 1; c := c + a;` had to be scheduled with a spill of `a` and a load of `c` in-between, because the left operand had to be in the accumulator. Since addition is commutative this is not necessary, and the code could be optimized.

2.2 Non-Commutative Opertations

Other binary operators are not commutative, for example $3 - 2 \neq 2 - 3$. The other of the previous models for one register machines had two binary operators:

acc ← **acc** *op* **mem**
acc ← **mem** *op* **acc**

While covering for the commutative operators this model does not suffice for most real accumulator-based machines since they often require the left operad to be in the accumulator in noncommutative operations.

3 Worms, Partitions, and Binate Covering

Definition Let a subject DAG $D\langle V, E\rangle$ be given. A *worm* w in D is a subset $V \cup E$ forming a directed path, possibly of zero length, such that the vertices in the path will appear consecutively in the schedule.

Definition A *worm-partition* of D is a set of disjoint worms.

We cover the DAG of the whole program with a worm-partitioning and then optimize the code within each worm. The result will be globally optimal code.

Definition A *binate covering problem* is a triple $\langle X, Y, \mathbf{cost}\rangle$, where X is a set of Boolean variables, Y is a set of clauses and **cost** is a function that maps X to the nonnegative integers.

These kind of Boolean clauses can be added to formulate constraints for code generation. An example of a binate covering problem and its optimal solution is given below (+ is binary OR, $\overline{x_i}$ is unary NOT):

$$y_1 = x_0 + x_1 + x_2 + \overline{x_3}$$
$$y_2 = x_0 + x_2 + x_3 + x_4$$
$$y_3 = x_1 + \overline{x_2} + x_4 + x_5$$
$$y_4 = x_1 + x_2 + x_3 + \overline{x_5}$$

$\mathbf{cost}(x_0) = 4$, $\mathbf{cost}(x_1) = 2$, $\mathbf{cost}(x_2) = 1$,
$\mathbf{cost}(x_3) = 1$, $\mathbf{cost}(x_4) = 3$, $\mathbf{cost}(x_5) = 1$.

Optimal solution: $x_0 = 1$, $x_5 = 1$, and the other variables set to 0.

4 Binate Covering Formulation

We can now start to formulate Boolean clauses to build up a binate covering problem. The solution to this problem gives us optimal code.

4.1 Patterns for Different Operator Forms

To allow for a binary operation such as $a+b$ to be calculated as $b+a$ we construct two patterns for the +-operator, one with the accumulator as the left operand, and one with the accumulator as the right operand. If we denote these patterns m_0 and m_1 respectively we can formulate the option as a Boolean clause:

$$y_1 = m_0 + m_1$$

To formulate the consequence of choosing the wrong pattern we also add clauses such as:

$$y_2 = \overline{m_0} + e_1 + \mathbf{spill}(t)$$

Here the edge variable e_1 means src(e_1) should directly precede dest(e_1), and **spill**(t) implies a spill in-between src(e_1) and dest(e_1).

4.2 Clauses to Avoid Cyclic Dependencies

Directed cycles in the worm-partition graph makes code generation impossible since there is a deadlock in code dependencies. We have to avoid such illegal graphs.

For parts of the DAG not containing any undirected cycles (*u-cycles*) we can guarantee that we produce a worm-partitioning without directed cycles by ensuring that each vertex has at most one immediate predecessor and at most one immediate successor. For each pair of fan-out or fan-in edges e_i and e_j we add the following clause:

$$y_3 = \overline{e_i} + \overline{e_j}$$

When we do have u-cycles in the DAG we have to ensure that not all forward edges, nor all backward edges are chosen from a u-cycle into a worm. This requirement can also be formulated in Boolean clauses:

$$y_4 = \overline{f_1} + \overline{f_2} + \ldots + \overline{f_k}$$
$$y_5 = \overline{b_1} + \overline{b_2} + \ldots + \overline{b_l}$$

5 Empirical Results

The only costs optimizable by enhancing the modeling of commutative operations are loads and spill. Consideing only those Liao et al achieved a 59% code reduction, which corresponds to 27% reduction in overall size.