Using the heap to eliminate stack accesses

Zoltan Somogyi
Peter J. Stuckey

{zs,pjs}@cs.mu.oz.au
University of Melbourne

Background

Mercury is a purely declarative logic programming language.

It has strong types and strong modes. The compiler knows the state of instantiation of every variable at every program point.

Cells allocated on the heap are naturally immutable.

Instead of updating data structures in place, you create new versions.

The Mercury compiler is written in Mercury. Profiling shows that its most expensive predicates are the ones that search and update trees.

The general idea

We view procedure bodies as being composed of a set of program *segments*.

Segments are contiguous regions of code bounded by program points at which all variables must be flushed to the stack.

Some unifications assign the values of the fields of a memory cell to a bunch of variables. The program segments containing such unifications often save most of these variables on the stack.

We can avoid executing the code required for this if we are willing to get those field variables from the memory cell in other segments.

```
update(T0, K, V, T) :-
    % load K0, V0, L0, R0
    T0 => tree(K0, V0, L0, R0),
    % store K, V, K0, V0, L0, R0
    compare(Result, K, K0),
    ( % switch on Result
        Result => (<),
        % load L0, K, V
        update(L0, K, V, L),
        % load K0, V0, R0
        T <= tree(K0, V0, L, R0)
    ;
        Result => (=),
        T <= tree(K0, V, L0, R0)
    ;
        Result => (>),
        update(R0, K, V, R),
        T <= tree(K0, V0, L0, R)
    ).
```

```
update(T0, K, V, T) :-
    % load K0
    T0 => tree(K0, _, _, _),
    % store K, V, T0
    compare(Result, K, K0),
    ( % switch on Result
        Result => (<),
        % load T0, L0s2, K, V
        T0 => tree(_, _, L0s2, _),
        update(L0s2, K, V, L),
        % load T0, K0s3, V0s3, R0s3
        T0 => tree(K0s3, V0s3, _, R0s3),
        T <= tree(K0s3, V0s3, L, R0s3)
    ;
        Result => (=),
        T0 => tree(K0s4, _, L0s4, R0s4),
        T <= tree(K0s4, V, L0s4, R0s4)
    ;
        ...
```

Costs and benefits

The effects of accessing a field variable F (e.g. L0) via the cell variable C (e.g. T0) instead of via the stack:

+ We add a load of C in every segment that accesses F but not C.

+ We add a store of C to the stack in the initial segment if C is not live after the initial segment.

- We avoid a store of F to the stack in the initial segment.

- We avoid a load of F in the initial segment if F is not needed in the initial segment.

Interdependence

Cells often contain more than one field.

The benefits of accessing F1 via C are independent of whether we also access F2 via C.

The costs of accessing F1 via C are *not* independent of whether we also access F2 via C.

If a segment accesses F1 and F2 but not C, then a single added load of C is sufficient to provide a base address for accesses to both F1 and F2, provided the loaded value doesn't need to be spilled from its register.

The optimization graph

We want to select a set of field variables to access via the cell for which the number of loads/stores we can remove (the benefits) is equal to or greater than the number we have to add (the costs).

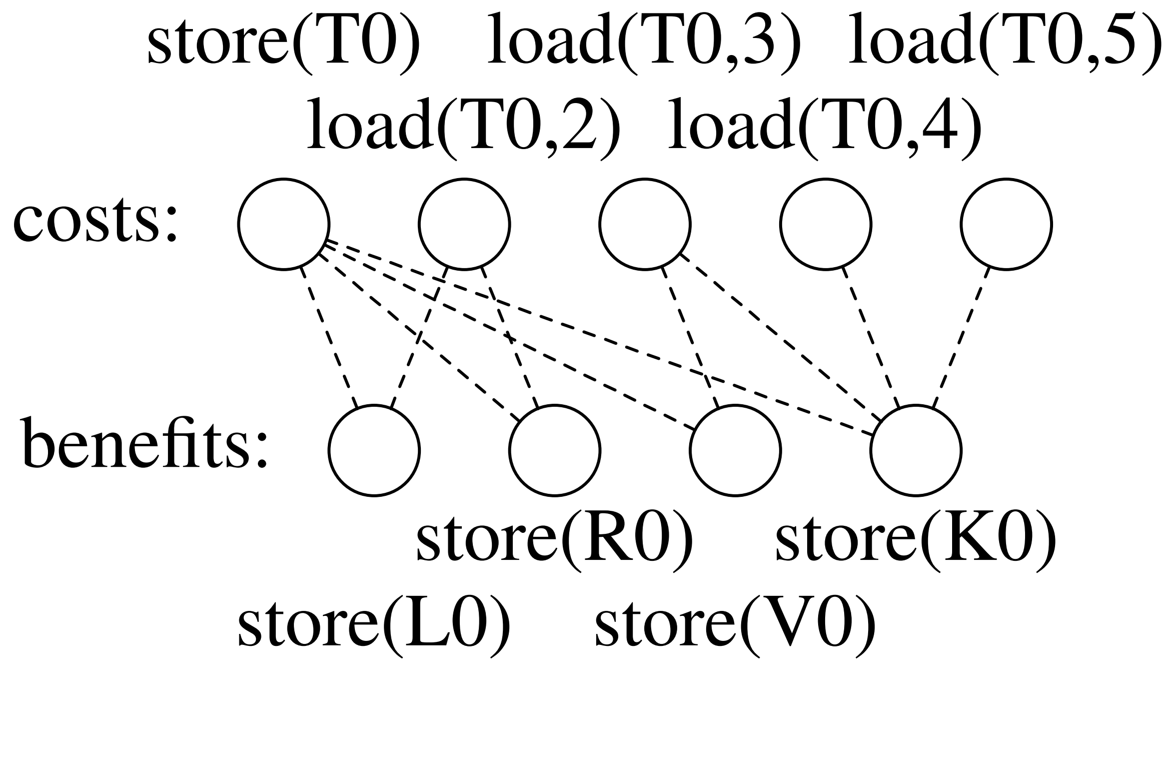
The optimization graph is a bipartite graph. One kind of node represents benefits, another represents costs.

It contains an edge linking a cost node and a benefit node if the two nodes represent parts of the costs and benefits of accessing the same field variable via the cell.

All the benefits and some of the costs are in the initial segment.

An example optimization graph

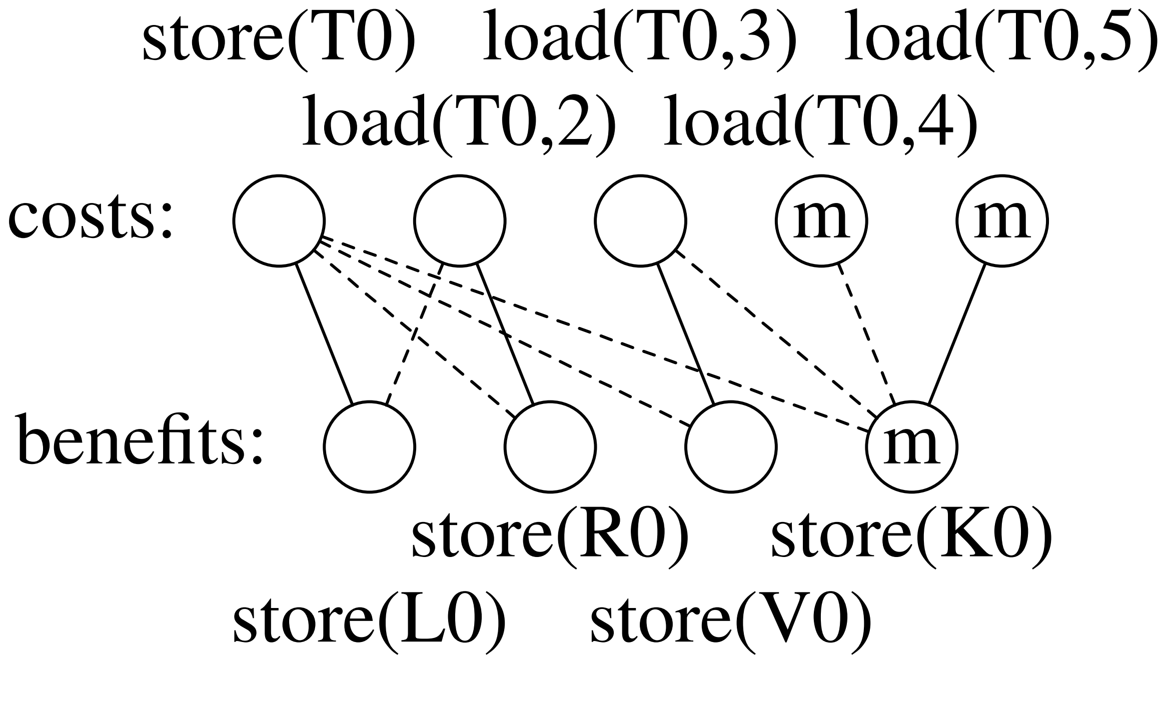
```
p(T0, C3) :-
    T0 => tree(K0, V0, L0, R0),
    dodgy(K0, V0, L0, R0),
    balanced(L0, R0),
    check(K0, V0, C1),
    check(K0, C1, C2),
    check(K0, C2, C3).
```



Maximal matching

The solid edges represent a maximal matching, i.e. a matching of maximal size.

Load(T0, 4) is an unmatched cost node, and the marked nodes are reachable from it via an alternating path.



Selecting field variables

We divide field variables into two classes: those whose benefits include a node marked by an alternating path from an unmatched cost node, and those whose benefits do not include such nodes.

We store variables in the first class in stack slots. We access variables in the second class via the cell variable, arranged through a source-to-source transformation.

We have a theorem that says this maximizes the net benefit.

Per-path analysis

Different paths through a procedure body execute different sets of segments.

The result of the costs/benefit analysis can therefore be different on each path through the procedure body.

We access a variable through its stack slot only if this doesn't yield a slowdown on *any* path.

This is a conservative approach. We could be more aggressive if we had information about how often execution follows each path.

Performance evaluation

We have implemented our algorithm in the Mercury compiler. Enabling it usually increases compile time by 0 to 2%.

The transformation consistently reduces executable sizes by 0.1 to 0.5%.

It reduces average stack frame sizes by up to 13%.

Its impact on execution time is variable. It can lead to slowdowns as well as speedups, although both are usually small (in the 2% range).

Occasionally, it yields significant (12%) speedups.

Conclusion

We should use program analysis and maybe profiling feedback to find out which heap references are likely to be cache misses, and take this into account in our cost/benefit analysis.

It would be nice if we could tell whether the transformation would extend the lifetime of the cell, and apply it only if it doesn't.

The transformation is applicable to any language whose implementation can find out (1) the relative frequency of execution of two program segments and (2) whether a heap cell is modified between two program points.