Using the Heap to Eliminate Stack Accesses

Zoltan Somogyi and Peter J. Stuckey
Department of Computer Science and Software Engineering
The University of Melbourne
3010, Australia
{zs,pjs}@cs.mu.oz.au

ABSTRACT

The value of a variable is often given by a field of a heap cell, and frequently the program will pick up the values of several variables from different fields of the same heap cell. By keeping some of these variables out of the stack frame, and accessing them in their original locations on the heap instead, we can reduce the number of loads from and stores to the stack at the cost of introducing a smaller number of loads from the heap. We present an algorithm that finds the optimal set of variables to access via a heap cell instead of a stack slot, and transforms the code of the program accordingly. We have implemented this optimization in the Mercury compiler, and our measurements show that it can reduce program runtimes by up to 12% while at the same time reducing program size. The optimization is straightforward to apply to Mercury and to other languages with immutable data structures; its adaptation to languages with destructive assignment would require the compiler to perform mutability analysis.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*compilers,optimization*; D.3.2 [**Programming Languages**]: Language Classifications—*constraint and logic languages*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*graph algorithms*

General Terms

Algorithms, Languages

Keywords

stack accesses, stack frames, heap cells, maximal matching

1. INTRODUCTION

Most compilers try to keep the values of variables in (perhaps virtual) registers whenever possible. However, procedure calls can (in the general case) modify the contents of all the registers. The standard solution to this problem is to allocate a slot in the stack frame to every variable that is live after the call, and to copy all these variables to the stack before the call.

In this paper we investigate the possibility of avoiding some of these allocations and the associated copies, exploiting the fact that some of the variable values we must save may already be available in memory locations that cannot be updated by the call. Suppose we obtained the value of a variable a from a field in an immutable cell on the heap pointed to by variable b. (From now on, we assume that all cells are immutable, unless we specifically say otherwise.) This allows us to avoid storing a on the stack, provided we can find its location on the heap everywhere we need to. In the case of a procedure in a single assignment language or of a procedure in an imperative language when expressed in a single-assignment form, this will be all places later in the procedure that refer to a.

Storing the value of b instead of the value of a on the stack may not look promising. In the worst case, it requires the same number of stack slots (one), the same number of stores to the stack before the call (one), and additional instructions to load b from the stack at all the places later in the procedure where we want to access a. However, it may happen that the program needs b itself after the call, in which case it needs a stack slot and the store instruction to fill that stack slot anyway, so accessing a via b needs no additional store. It may also happen that in every basic block in which the program accesses a it also accesses b, in which case accessing a via b requires no additional loads either. If both these things happen, then accessing a via b eliminates one stack slot and one store operation to fill in that stack slot. If a is not used between its definition and the first call after the definition, then accessing a via b also saves the load of a from the cell we would need before storing a in its stack slot. Overall, accessing a via b may be better than storing a in its own stack slot.

Of course, it may be that between two calls, the procedure accesses a but not b. In that case, accessing a via b incurs the cost of an additional load. However, it often happens that between two calls, a procedure does not access a variable pointing to a cell (e.g. b) but accesses more than one variable whose values came from the fields of that cell. A single load of b gives us access to a_1, a_2 and a_3, then the cost of the load of b needs to divided up among a_1, a_2 and a_3. This means that the cost of accessing a_1 via b instead of storing a_1 in its own stack slot depends on which of the other variables reachable from b we access via b. Indeed, it may happen

PPDP'02, October 6-8, 2002, Pittsburgh, Pennsylvania, USA.

that accessing just a_1 via b is not worthwhile, accessing just a_2 via b is not worthwhile, but accessing both a_1 and a_2 via b is worthwhile.

This interdependence of the decisions we need to make for the different variables reachable from the same cell significantly complicates the task of finding the optimal partition of the variables stored in the cell between those which should have their own stack slots and those which should be accessed via the cell. Compared to the partition in which all variables are stored in stack slots, the optimal partition may reduce the total number of stack accesses (loads and stores) performed by the program, it may reduce the number of stack slots required, or it may do both. Of course, it may also do neither, but then no optimization can *guarantee* a speedup.

Trying all partitions is the obvious algorithm of computing the best partition. Unfortunately this algorithm is not feasible, because some cells contain dozens of fields, which can require trying millions of partitions. In this paper we describe a much more efficient algorithm based on maximal matching in bipartite graphs. We have implemented the algorithm in the Mercury compiler, and our experiments show that it can give real benefits on real programs (e.g. the Mercury compiler itself).

In the next section we give a very brief introduction to Mercury, concentrating on the features that are relevant to our optimization. In section 3 we give a worked example explaining our technique. In section 4 we define the conditions on the applicability of the optimization, and show how we collect the information we need to implement the optimization. Section 5 gives our algorithm for computing optimal partitions, while section 6 explains the source-to-source transformation that we use to exploit optimal partitions. In Section 7 we give the results of a preliminary experimental evaluation of our optimization.

2. MERCURY

We assume familiarity with the basic concepts of logic programming, and Prolog syntax, which is also Mercury syntax.

Mercury is a purely declarative logic programming language designed for the construction of large, reliable and efficient software systems by teams of programmers [3, 6]. Mercury's syntax is similar to the syntax of Prolog, but Mercury also has strong module, type-, mode- and determinism systems, which catch a large fraction of programmer errors and enable the compiler to generate fast code. The main features of Mercury that are relevant for this paper are the following.

Every predicate has one or more *modes*, which say, for every argument, whether it is input or output. (The mode system is more sophisticated than that, but that is all that matters for this paper.) We call each mode of a predicate a *procedure*. The mode system insists on being able to find out at compile time what the state of instantiation of every variable is at every point in every procedure, and on being able to reorder conjunctions so that all goals that use the value of a variable come after the goal that generates the value of that variable. If it cannot perform either task, it rejects the program.

Mercury associates with every procedure a *determinism*, which expresses upper and lower bounds on the number of solutions the procedure can have. Procedures that are guaranteed to have exactly one solution have determinism *det*. Procedures that may have one solution for one set of inputs and zero solutions for other inputs have determinism *semidet*. Procedures that may have any number of solutions have determinism *nondet*.

The definition of a predicate in Mercury is a body made up of atoms, conjunctions, negations, disjunctions and if-then-elses. To simplify its algorithms, the compiler converts each body clause so that the only forms of atoms appearing are

$\texttt{p}(a_1,\ \ldots,\ a_n)$ $\qquad$ b = a $\qquad$ b = $\texttt{f}(a_1,\ \ldots,\ a_n)$

and all variables in an atom must be distinct.

During mode analysis, the compiler classifies all unifications into one of five types: *assignments*, where $b = a$ where one of a and b is input and the other is output; *tests*, where $b = a$ where both are input; *deconstructions*, where $b = f(a_1, \ldots, a_n)$ where b is input and each of $a_1, \ldots, a_n$ is output; *constructions*, where $b = f(a_1, \ldots, a_n)$ where each of $a_1, \ldots, a_n$ is input and b is output; and *complex unifications* otherwise. We use the respective notations b `:=` a, b `==` a, b `=>` $\texttt{f}(a_1, \ldots, a_n)$, b `<=` $\texttt{f}(a_1, \ldots, a_n)$ and `unify(`b`,` a`)` to indicate the mode of each unification.

In this paper, we shall be mainly concerned with deconstructions. The deconstruction b `=>` $\texttt{f}(a_1, \ldots, a_n)$ tests whether the principal functor of b is `f` if this hasn't been previously established, and, if $n > 0$, assigns the fields of the cell pointed to by b to the various a_i. Note that for field variables a_i that are not used later (such as anonymous variables) the compiler does not access the field.

The compiler also detects disjunctions in which each disjunct deconstructs the same input variable with different function symbols. In such disjunctions, the value of that variable on entry to the disjunction determines which disjunct, if any, can succeed; the others cannot succeed because they try to unify the variable with a function symbol it is not bound to. The compiler converts such disjunctions into *switches* (they resemble the switch construct of C).

The Mercury compiler has several backends, which translate Mercury into different target languages. The backend that we work with in this paper is the compiler's original backend, which translates Mercury into low level C code. This backend uses the execution algorithm described in [6]. This execution algorithm uses a virtual machine whose data structures consist of a heap, two stacks, a set of general purpose registers used for argument passing, and a set of special purpose registers such as heap and stack pointers. The differences between the two stacks are irrelevant for purposes of this paper.

The Mercury compiler is responsible for parameter passing, stack frame management, heap allocation, control flow (including the management of backtracking) and almost every other aspect of execution; the only significant tasks it leaves for the C compiler are instruction selection, instruction scheduling and register allocation within basic blocks. In effect, we use the C compiler as a high level, optimizing assembler. Besides achieving portability, this approach allows the Mercury compiler to perform optimizations that exploit semantic properties of Mercury programs (such as the immutability of ground terms) that cannot be conveyed to the C compiler.

The Mercury compiler assumes that every call can clobber every virtual machine register, so at every call site it flushes all live variables to their slots in the current procedure's

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

(a)

```
update(T0, K, V, T) :-
   ( % switch on T0                    % (A)
      T0 => empty,
      T <= tree(K, V, empty, empty)
   ;
      % load K0
      T0 => tree(K0, _, _, _),
      % store K, V, T0
      compare(Result, K, K0),         % (B)
      ( % switch on Result
         Result => (<),
         % load K, V, T0, L0_BC
         T0 => tree(_, _, L0_BC, _),
         update(L0_BC, K, V, L),      % (C)
         % load T0, K0_CE, V0_CE, R0_CE
         T0 => tree(K0_CE, V0_CE, _, R0_CE),
         T <= tree(K0_CE, V0_CE, L, R0_CE)
      ;
         Result => (=),
         % load V, T0, K0_BE, L0_BE, R0_BE
         T0 => tree(K0_BE, _, L0_BE, R0_BE),
         T <= tree(K0_BE, V, L0_BE, R0_BE)
      ;
         Result => (<),
         % load K, V, T0, R0_BD
         T0 => tree(_, _, _, R0_BD),
         update(R0_BD, K, V, R),      % (D)
         % load T0, K0_DE, V0_DE, L0_DE
         T0 => tree(K0_DE, V0_DE, L0_DE, _),
         T <= tree(K0_DE, V0_DE, L0_DE, R)
      )                               % (E)
   ).
```

(b)

Figure 1: `update` predicate in original form (a), and modified by our transformation (b).

stack frame. Similarly, at the starts of nondeterministic disjunctions, it flushes to the stack all the variables that are live at the start of the second or later disjuncts, since the starts of these disjuncts can be reached by backtracking long after further execution has clobbered all the virtual machine registers. We treat if-then-elses similarly, with the else branch corresponding to the second disjunct.

These are the only situations in which the Mercury execution algorithm requires all live variables to be flushed to the stack. The Mercury compiler therefore has a pass that figures out, for each of these flush points, which variables need to be flushed to the stack at that point. Variables which need to exist on the stack simultaneously must be stored in different stack slots, but one may be able to use the same stack slot to store different variables at different flush points. The compiler uses a standard graph colouring approach (see e.g. [2]) to assign variables to stack slots.

3. MOTIVATING EXAMPLE

Consider the Mercury predicate illustrated in Figure 1(a), which updates a binary search tree (`T0`) containing key-value pairs so that the new version of the tree (`T`) maps key `K` to value `V`. (Text after the symbol `%` is a comment.) Such predicates (predicates that search in and update various kinds of trees) are fairly important in many Mercury programs, because the declarative nature of Mercury encourages programmers to use them to represent dictionaries instead of arrays (which must be updated destructively).

In the original form of the predicate, all of the variables (`K0, V0, L0, R0`) whose values are produced in the deconstruction `T0 => tree(K0, V0, L0, R0)` are live after the call to `compare` that immediately follows the deconstruction. The compiler therefore allocates a stack slot to each of these variables and saves their values in those stack slots before the call. Saving the value of each of these variables requires that it be loaded into a register first from its original location in the memory cell pointed to by `T0`. `K` and `V` are also live after the call, and therefore must be stored on the stack, but they have already been put into registers by `update`'s caller.

Execution can follow one of three paths after `compare` returns. If `K` and `K0` are equal, then execution takes the second arm of the switch; the code there uses `K0`, `V`, `L0` and `R0` as inputs, so those four variables must be loaded from stack slots. If `K` is less than `K0`, then execution takes the first arm of the switch, which contains a call. Making the call requires `L0`, `K` and `V` to be loaded into registers from their stack slots. The call returns `L` in a register, so after the call we need to load into registers only `K0`, `V0` and `R0`. The third arm of the switch is analogous to the first.

We have added comments to indicate which variables the code stores to the stack, and which variables it loads from stack or from the heap. We can count the loads and stores of the variables involved in the deconstruction (the *cell* variable `T0` and the *field* variables `K0,V0,L0,R0`) that are required to make the values of those variables available along each path of execution that involves that deconstruction. If execution takes the first arm of the switch on `Result`, then we execute four loads and four stores involving those variables between program points A and B, one load between B and C, and three loads between C and E, for a total of eight loads and four stores. If execution takes the third arm of the switch on `Result`, then we similarly execute a total of eight loads and four stores. If execution takes the second arm of the switch on `Result`, then we execute four loads and four stores between A and B, and three loads between B and E, for a total of seven loads and four stores.

The key idea of this paper is the realization that the loads and stores between A and B are a significant cost, and that we can avoid this cost if we are willing to insert clones of the deconstruction later in the procedure body. These clones incur an extra cost, the load of `T0`, but as long as we choose to perform this transformation only if the initial saving is at least as big as the extra cost on all paths of execution, we have achieved a speedup.

Figure 1(b) shows the same predicate after our transformation. It has five clones of the original deconstruction, one for each region after the first that uses the field variables.

If execution takes the first arm of the switch on `Result`, then the transformed predicate executes one load and one store between A and B, two loads between B and C (loading `T0` from its stack slot to a register, and then loading `L0_BC` from the cell `T0` points to) and four loads between C and E (loading `T0` from its stack slot to a register, and then loading `K0_CE`, `V0_CE` and `R0_CE` from the cell `T0` points to) for a total of seven loads and one store. If execution takes the third arm of the switch on `Result`, the analysis is analogous and the total cost is again seven loads and one store. If execution takes the second arm of the switch on `Result`, then we execute one load and one store between A and B, and four loads between B and E, for a total of five loads and one store.

Overall, the transformation reduces the costs of the paths through the first and third arms from eight loads and four stores to seven loads and one store, and the cost of the path through the second arm from seven loads and four stores to five loads and one store. The transformation also reduces the number of stack slots required. The original code needed six stack slots for variables, one for each of `K,V,K0,V0,L0,R0`. The transformed code needs only three stack slots for variables, one for each of `K,V,T0`. (The number of stack slots needed to hold housekeeping information such as the return address is of course unchanged.)

The source of the speedup is that we only added one or two extra loads of `T0` into each path of execution, but we replaced four loads and four loads between A and B with one load and one store. The extra cost is always in the form of extra loads of the cell variable (`T0`) after the first stack flush following the deconstruction and possibly, as in this case, an extra store of the cell variable before that stack flush. The benefit is always in the form of eliminated stores of the field variables before that stack flush, and the eliminated loads of those field variables that are not needed before the stack flush. In this case, we must keep the load of `K0` but can eliminate its store, and we can eliminate the loads as well as the stores of `V0`, `L0` and `R0`.

The reason for the reduction in stack slot requirements is that saving `T0` on the stack preserves the values of all the field variables in `T0`'s heap cell across calls. and since the number of field variables in that cell is greater than one, we are using one stack slot to save the value of more than one variable across calls.

4. DETECTING OPPORTUNITIES FOR OPTIMIZATION

Before we can describe our algorithm for performing the transformation shown in the example above, we need to introduce some background information and definitions.

The body of a Mercury procedure is a goal. A goal may be an atomic goal or a compound goal. An atomic goal may be a unification, a builtin operation (e.g. arithmetic) or a call. (For the purposes of this paper, there is no distinction between first order calls, higher order calls and method calls.) A compound goal may be a conjunction, a disjunction, a switch, an if-then-else, a negation or an existential quantifier. In the rest of this paper, we will restrict our attention to the first four of those compound goal types. Our algorithms treat negation as a special case of if-then-else (`not(Goal)` is equivalent to `(Goal -> fail ; true)`), and they treat an existentially quantified goal as the goal itself. We call disjunctions, switches and if-then-elses *branched* control structures or branched goals.

Definition 1. A *flush point* is a point in the body of a procedure at which the code generator is required to store some variables in stack slots or in registers. In Mercury, there are four kinds of flush points.

- When execution reaches a call, the code generator must flush all variables that are live after the call to the stack. This because like most compilers, the Mercury compiler assumes that all calls can clobber all registers.
- When execution reaches the start of an if-then-else, the code generator must flush all variables that are live at the start of the else case. If the else case can be reached after a register may have been clobbered (e.g. by a call inside the condition), then the code generator must flush all these variables to the stack; otherwise, it can flush variables to registers as well as stack slots.
- When execution reaches the start of a disjunction, the code generator must flush all the variables that are live at the start of the second disjunct or a later disjunct. If a non-first disjunct can be reached via deep backtracking (i.e. after the failure of a call inside a previous disjunct or after the failure of a goal following the disjunction as a whole), then the code generator must flush all these variables to the stack; otherwise, it can flush variables to registers as well as stack slots.
- When execution reaches the end of a branched control structure, the code generator must store each variable that is live afterwards in a specific stack slot or in a specific register; the exact location is determined by a pre-pass of the code generator. This ensures that all branches leave those variables in the same place.

Definition 2. An *anchor* is one of the following:

- the start of the procedure body
- a call site
- the start of a branched control structure
- the end of the condition of an if-then-else
- the end of a branched control structure
- the end of the procedure body

All flush points are anchors, but not all anchors are flush points. In the example of Figure 1(a) the program points A,B,C,D, and E (which represent the start of the outer switch, the call to `compare`, the two calls to `update`, and the end of the inner switch respectively) are all anchors, and all but A are also flush points. The code fragment also contains two other anchors: the start of the inner switch and the end of the outer switch. Our example did not distinguish between the two anchors each at program points B and E.

Definition 3. An *interval* is a sequence of atomic goals delimited by a left-right pair of anchors, satisfying the property that if forward execution starts at the left anchor and continues without encountering failure (which would initiate backtracking, i.e. backward execution), the next anchor it reaches is the right anchor of the pair. We consider a call to be part of the atomic goals of the interval only if the call site is the right anchor of the interval, not the left anchor.

Definition 4. A *segment* is a maximal sequence of one or more intervals such that the right anchor of each interval in the sequence, except the last, is the left anchor of the interval following it in the sequence. The sequence must also satisfy the property that execution can get from the left anchor of the first interval to the right anchor of the last interval without the code generator throwing away its current record of the values of the live variables.

Most segments contain just one interval. However, if the right anchor of an interval is the start of an if-then-else, then the interval before the if-then-else and the interval at the start of the condition can belong to the same segment. If the right anchor of an interval is the start of a disjunction, then the interval before the disjunction, and the interval at the start of the first disjunct can belong to the same segment. If the right anchor of an interval is the start of a switch, then the interval before the disjunction, and the interval at the start of the *any* arm of the switch can belong to the same segment. Intervals whose right anchor is the start of a switch are the only intervals that can be part of more than one segment.

In the example of Figure 1(a) each of AB, BC, CE, BE, BD and DE are all segments. There is an empty interval (containing no atomic goals) between the end of the call to `compare` and the start of the following switch, which is part of all the segments starting at B.

Our transformation algorithm has three phases. In the first phase, we find all the intervals in the procedure body, and for each interval, record its left and right anchors and the set of variables needed as inputs in the interval. These include the inputs of the atomic goals of the interval, and, for intervals whose right anchor is the start of a switch, the variable being switched on. The set of variables needed in a segment s, which we denote $vars(s)$, is the union of the sets of variables needed by the component intervals of segment s. We also record, for each interval, the leftmost anchor of the segment(s) to which the interval belongs.

In the second phase, we traverse the procedure body backwards, looking at each deconstruction `b => f(`a_1`, ..., `a_n`)`. We call the a_i the *field* variables, as opposed to b, which is the *cell* variable. When we find a deconstruction, we try to find out which field variables we can avoid storing in stack slots, loading it from the heap cell pointed to by the cell variable instead wherever it is needed. We can access a field variable via the cell variable only if all the following conditions hold:

- The memory cell pointed to by the cell variable must be immutable; if it isn't, then the values of the fields of the cell may change between the original deconstruction and the copies of that deconstruction that the transformation inserts elsewhere in the procedure body. In Mercury, a cell is mutable only if the instantiation state of the cell variable at the point of the deconstruction states that it is the only pointer to the cell that is live at that point (so the compiler may choose to destructively update the cell).
- All the program points at which the value of the field variable is needed (as the input of an atomic goal, as a live variable to be flushed, or as an output argument of the procedure as a whole) must be within the effective scope of the deconstruction.

  Consider the variable `A` in the following program:

```
p(X,Y,Z) :-
   ( X > 0 ->
      Y => f(A, B),
      q(X,B)
   ;
      A <= a
   ),
   r(X,A,Z).
```

  `A` is accessible via `Y` only until the end of the then arm, but it is needed after the if-then-else; hence it must be stored in a stack slot.
- Every interval which needs the value of a field variable but not the value of the cell variable must be reachable at most once for each time execution reaches the deconstruction.

  Consider the variable `A` in the following program:

```
p(X, Y, Z) :-
   Y => f(A, B),
   q(X, A, B),
   r(X, A),
   s(X, Y, A, Z).
```

  If `A` were accessed indirectly through `Y`, then we need to add a load of `Y` to the segment between `q/3` and `r/2`. If `q/3` can succeed multiple times, then this load is executed once per success, and not is guaranteed to be compensated by the removal of the store of `A` before the call to `q/3`.
- The value of the field variable is required in a segment after the deconstruction, otherwise there is no point in investigating whether it would be worthwhile to access it via the cell variable in other segments.

For deconstructions in which all four conditions hold for at least some of the field variables (these form the set of *candidate* field variables), we then partition the set of candidates into two subsets: those we should access via the cell variable, and those we should nevertheless store in and access via stack slots. To do this, we first find the set of *maximal paths* that execution can take in the procedure body from the point of the deconstruction to the point at which the deconstruction goes out of scope.

Definition 5. A *path* is a sequence of segments starting at the segment containing the deconstruction, in which segment i can follow segment j if (a) the left anchor of the first interval in segment j is the right anchor of the last interval in segment i, or (b) execution can resume at the left anchor of the first interval in segment j after backtracking initiated within segment i. A path is *maximal* if isn't contained within another path.

A maximal path through a disjunction includes a maximal path through the first disjunct, then a maximal path through the second disjunct, then a maximal path through the third, etc. A maximal path through an if-then-else is either a maximal path through the condition followed by a maximal path through the then part, or a maximal path through the condition followed by a maximal path through the else part. A maximal path through a switch is a maximal path through one of the arms of the switch.

For the program of Figure 1(a), the maximal paths are [AB,BC,CE], [AB,BE] and [AB,BD,DE], since there is no backtracking in the switch.

For each maximal path starting at a given deconstruction which has a non-empty set of candidate field variables, we invoke the algorithm described in the next section to partition the candidate variables into the set that, from the point of view of an execution that takes that particular maximal path through the procedure body, it is better to access via the cell variable and the set that it is better to store in stack slots.

5. DECIDING WHICH VARIABLES TO LOAD FROM CELLS

5.1 Introduction to Maximal Matching

The algorithm we use for deciding which variable to load from cells make use of maximal matching algorithms for bipartite graphs. In this section we introduce terminology, and examples.

Definition 6. A *bipartite graph* G is made up of two disjoint sets of vertices B and C and edges E where for each $(b, c) \in E, b \in B$ and $c \in C$. In our application, the sets of vertices represent benefits and costs.

A *matching* of a bipartite graph G is a set of edges $M \subseteq E$ such that each vertex occurs at most once in M.

A *maximal matching* M of G is a matching such that for each other matching M' of G, $|M'| \leq |M|$.

There are efficient algorithms for maximal matching of bipartite graphs. They are all based on searching for augmenting paths.

Definition 7. Given a bipartite graph G and matching M, an *alternating path* is a path whose edges are alternatively in $E - M$ and M.

Define the set $reachable(u, M)$ as the set of nodes reachable from u by an alternating path.

Given a bipartite graph G and matching M, an *augmenting path* p is an alternating path where the first and last vertices are *free*, i.e. do not occur in M.

Given a bipartite graph G, matching M and augmenting path p,

$$M' \equiv (M - p) \cup (p - M)$$

is a matching where $|M'| = |M| + 1$.

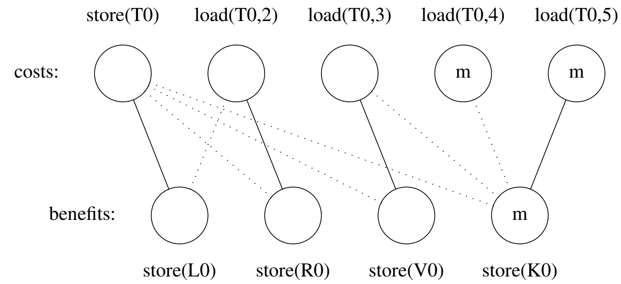

Figure 2: The stack optimization graph for the program of Figure 3 together with a maximal matching.

An important property of maximal matchings is their relationship with augmenting paths.

PROPERTY 1. *A matching M for G is maximal iff there exist no augmenting paths p for M and G.*

A straightforward algorithm for bipartite maximal matching based on searching for augmenting paths using breadth-first search is $O(|U| \times |E|)$ while more sophisticated algorithms are $O(\sqrt{|U|} \cdot |E|)$ [4].

Example 1. Figure 2 shows a bipartite graph, with the matching M illustrated by the solid arcs. The matching is $M = \{(store(L0), store(T0)),\ (store(R0), load(T0, 2)),\ (store(V0), load(T0, 3)),\ (store(K0), load(T0, 5))\}$. An example alternating path is $load(T0, 5) \rightarrow store(K0) \rightarrow store(T0) \rightarrow store(L0) \rightarrow load(T0, 2)$ defined by the edges: $\{(store(K0), load(T0, 5)),\ (store(K0), store(T0)),\ (store(L0), store(T0)),\ (store(L0), load(T0, 2))\}$. It is not an augmenting path as both endpoints are matched. Indeed the matching M is a maximal matching.

5.2 Minimizing Stack Operations

Our aim is to find, for each deconstruction unification `b => f(`$a_1, \ldots, a_n$`)`, the set of variables involved which should be stored on the stack in order to minimize the number of stack operations required. Let $F \subseteq \{a_1, \ldots, a_n\}$ be the *candidate* field variables.

For each maximal path from the deconstruction we assume we are given a list of segments $1, \ldots, m$ and a function $vars(i)$ which determines the variables whose values are required in program segment i. We determine the set of variables that require their own stack slot independently for each maximal path.

Definition 8. The *costs* that must be incurred if we access candidate variable f via the cell variable b instead of via the stack are:

- $load(b, i)$: we need to add a load of b in every segment i where $f \in vars(i)$ and $b \notin vars(i)$; and
- $store(b)$: we need to add a store of b in the first segment, if b is not live after the initial segment. [1]

We call this set $cost(f)$.

The *benefits* that are gained if we access candidate variable f via the cell variable b instead of via the stack are:

[1]If b *is* live after the initial segment, then even the original program would need to store b on the stack, so this store is not an extra cost incurred by accessing a field variable via b.

- $store(f)$: we avoid storing f in the initial segment; and
- $load(f, 1)$: we avoid loading f in the initial segment if $f \notin vars(1)$.

We call this set $benefit(f)$.

We can use these to model the total costs and benefits of choosing to access a given subset V of F via the cell variable instead of storing them on the stack.

The total set of costs we incur if we choose to access a given subset V of F via the cell variable instead of storing them on the stack is $cost(V) = \cup_{f \in V} cost(f)$, while the total set of benefits of that choice is $benefit(V) = \cup_{f \in V} benefit(f)$. Note that, while the benefits for each f are independent, the costs are not, since the cost of the load of b in a given segment is incurred only once, even if it is used to access more than one candidate variable. We therefore cannot decide for each candidate variable individually whether it should be stored on the stack or accessed via the cell variable, but must consider a set of candidates at a time.

We need an efficient algorithm for finding a set $V \subseteq F$, such that $benefit(V)$ are greater than or equal to $cost(V)$. For the time being we will assume that all loads and stores cost the same. We discuss relaxing this assumption in Section 7.

Hence we are searching for a set $V \subseteq F$ such that $net(V) = |benefit(V)| - |cost(V)|$ is maximal. Further if we have two choices V_1 and V_2 where $net(V_1) = net(V_2)$ and $|V_1| > |V_2|$, we prefer V_1 since it requires fewer stack slots.

Our algorithm reduces most of the stack optimization problem to the maximal matching problem for bipartite graphs, for which efficient algorithms are known.

Definition 9. The *stack optimization graph* for a deconstruction b `=>` $\mathtt{f}(a_1, \ldots, a_n)$ over segments $\{1, \ldots, m\}$ is given by the bipartite graph G whose vertex sets are $B = \cup_{f \in F} benefit(f)$ and $C = \cup_{f \in F} cost(f)$, and edges are defined as $E = \{(b, c) \mid f \in F, b \in benefit(f), c \in cost(f)\}$.

Each node in the graph represents a load or store instruction, and the edges represent the benefits one can gain if one is willing to incur a given set of costs. In our diagrams, the cost nodes are at the top and the benefit nodes are at the bottom.

Example 2. Consider the program shown in Figure 3(a). The default compilation requires 14 loads and stores. The deconstruction `T0 => tree(K0,V0,L0,R0)` has a single maximal path (the entire procedure). All the field variables are candidates. The segments are anchored by the end of each call. The *vars* information is given by:

vars(1) = K0, V0, L0, R0
vars(2) = L0, R0
vars(3) = K0, V0
vars(4) = K0
vars(5) = K0

The costs and benefits for each of the field variables are given by

	cost	*benefit*
`K0`	$\{store(T0), load(T0,3),$ $load(T0,4), load(T0,5)\}$	$\{store(K0)\}$
`V0`	$\{store(T0), load(T0,3)\}$	$\{store(V0)\}$
`L0`	$\{store(T0), load(T0,2)\}$	$\{store(L0)\}$
`R0`	$\{store(T0), load(T0,2)\}$	$\{store(R0)\}$

Note that since `T0` is not required after the deconstruction, it is a cost for each candidate, and since each candidate is required in the initial segment there are no load benefits.

The stack optimization graph for the deconstruction is shown in Figure 2.

The algorithm starts by finding a maximal match M in the stack optimization graph. (Figure 2 shows the edges in the maximal matching in solid lines.) It then marks each unmatched cost node and each node reachable from these nodes using an alternating path with respect to M. The cost nodes this marks represent costs which are not "paid for" by corresponding benefits. The benefit nodes which are not marked are those where the benefit equals or outweighs the corresponding costs. The algorithm partitions the candidates into those whose benefits include marked nodes, and those whose benefits do not include marked nodes. The result V of variables we want to access via the cell variable is the latter set.

In fact the benefit nodes for each candidate variable will either be all marked or all unmarked. This is a consequence of the following lemma.

LEMMA 1. *Let G be the stack optimization graph, where $b_1 \in B$ and $b_2 \in B$ are adjacent to the same subset $A \subseteq C$ and M is a maximal matching of G. Let $MC = \{c \mid (b, c) \in M\}$ be the matched nodes in C. Let $\overline{MC} = C - MC$. Let $R = \cup_{c \in \overline{MC}} reachable(c, M)$ be the nodes reachable by an alternating path from an unmatched node in C. Then $b_1 \in R$ if and only if $b_2 \in R$.*

PROOF. Suppose to the contrary that w.l.o.g. $b_1 \in R$ and $b_2 \notin R$. Since $b_1 \in R$, there is an alternating path from some $c \in \overline{MC}$ to b_1. Hence there is an alternating path from c to some $a \in A$. Since $c \in \overline{MC}$, the first edge in this path cannot be in M; since the path must have an even number of edges, the last edge must be in M. Now b_2 is also adjacent to a. If $(b_2, a) \notin M$, then we can extend the alternating path from c to a to reach b_2, which is a contradiction. Alternatively $(b_2, a) \in M$, but since there is an alternating path from c to a, this means the path from c to a must use this edge, and hence there is an alternating path from c to b_2, which is again a contradiction. $\square$

Example 3. Figure 2 shows the stack optimization graph for the program of Figure 3(a), together with a maximal matching. We mark (with an 'm') all the nodes reachable by an alternating path starting from an unmatched node in C (in this case, the only such node is $\{load(T0, 4)\}$). In Figure 2, the marked nodes are $load(T0, 4)$, $store(K0)$ and $load(T0, 5)$. The set V defined by the matching is the set of candidate variables all of whose benefit nodes are unmarked. In this case $V = \{L0, R0, V0\}$. The resulting optimized program is shown in Figure 3(c) and requires 14 loads and stores. Note that accessing all field variables through the cell results in the program in Figure 3(b), which requires 15 loads and stores.

We can show not only that the choice V is no worse than the default of accessing every candidate through a stack slot, but that the choice is optimal.

THEOREM 1. *Let G be the stack optimization graph, and M a maximal matching of G. Let $MC = \{c \mid (b, c) \in M\}$ be the matched nodes in C, and $\overline{MC} = C - MC$ be the*

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

(a)

```
p(T0, C3) :-
    % load K0, V0, L0, R0
    T0 => tree(K0, V0, L0, R0),
    % store T0
    dodgy(K0, V0, L0, R0),
    % load T0, L0_2, R0_2
    T0 => tree(_, _, L0_2, R0_2),
    balanced(L0_2, R0_2).
    % load T0, K0_3, V0_3
    T0 => tree(K0_3, V0_3, _, _),
    check(K0_3, V0_3, C1),
    % load T0, K0_4
    T0 => tree(K0_4, _, _, _),
    check(K0_4, C1, C2),
    % load T0, K0_5
    T0 => tree(K0_5, _, _, _),
    check(K0_5, C2, C3).
```

(b)

```
p(T0, C3) :-
    % load K0, V0, L0, R0
    T0 => tree(K0, V0, L0, R0),
    % store T0, K0
    dodgy(K0, V0, L0, R0),
    % load T0, L0_2, R0_2
    T0 => tree(_, _, L0_2, R0_2),
    balanced(L0_2, R0_2),
    % load K0, T0, V0_3
    T0 => tree(_, V0_3, _, _),
    check(K0, V0_3, C1),
    % load K0
    check(K0, C1, C2),
    % load K0
    check(K0, C2, C3).
```

(c)

Figure 3: (a) original program, (b) maximal stack savings version, and (c) optimal version.

unmatched nodes in C. Let $R = \cup_{c \in \overline{MC}} reachable(c, M)$. Let $V = \{f \in F \mid benefit(f) \cap R = \emptyset\}$. Then $net(V)$ is maximal.

Proof. Let $R_B = B \cap R$, $R_C = C \cap R$, $\bar{R}_B = B - R$ and $\bar{R}_C = C - R$.

Each node $c \in \bar{R}_C$ is matched, otherwise it would be in $\overline{MC}$, and it must be matched to a node in $\bar{R}_B$. Suppose to the contrary, that c is matched with $b \in R_B$. Then there is an alternating path from some $c' \in \overline{MC}$ to b which ends in an unmatched edge (since it starts with an unmatched edge from C to B). Hence we can extend this path using the matching edge (b, c), hence $c \in R$. Contradiction.

Let $MB = \{b \mid (b, c) \in M\}$, and $\overline{MB} = B - MB$.

Clearly $b \in \overline{MB}$ implies that $b \in \bar{R}_B$ otherwise there would be an augmenting path from some $c \in \overline{MC}$ to b. Now $|\bar{R}_C| + |\overline{MB}| = |\bar{R}_B|$ since each node in $\bar{R}_B$ either matches a node in $\bar{R}_C$ or is unmatched.

By definition $benefit(V) = \bar{R}_B$ since V contains exactly the f such that $benefit(f) \cap R = \emptyset$.

We now show that $cost(V) = \bar{R}_C$. Now $cost(V)$ is all the nodes in C adjacent to a node in $benefit(V) = \bar{R}_B$. Since each node in $\bar{R}_C$ is matched with a node in $\bar{R}_B$, it must be adjacent to a node in $\bar{R}_B$, thus $\bar{R}_C \subseteq cost(V)$.

Suppose to the contrary there is a node $c \notin \bar{R}_C$ adjacent to $b \in \bar{R}_B$. Now $c \in R_C$, hence there is an alternating path from some $c' \in \overline{MC}$ to c which ends in an edge in M. But then the alternating path can be extended to b since (b, c) is not in M, and hence $b \in R$, which is a contradiction. Thus $cost(V) \subseteq \bar{R}_C$.

Now $net(V) = |benefit(V)| - |cost(V)| = |\bar{R}_B| - |\bar{R}_C| = |\overline{MB}|$.

Consider any set of variables $V' \subseteq F$. Let $MB' = MB \cap benefit(V')$, $MC' = MC \cap cost(V')$, $\overline{MB}' = benefit(V') - MB'$, and $\overline{MC}' = cost(V') - MC'$. By definition each node $b \in benefit(V')$ is unmatched or matches a node in $cost(V')$. Hence $|MC'| \geq |MB'|$. Also clearly $\overline{MB}' \subseteq \overline{MB}$ since nodes in $\overline{MB}'$ are unmatched.

Now $net(V') = |benefit(V')| - |cost(V')| = |MB'| + |\overline{MB}'| - |MC'| - |\overline{MC}'| \leq |\overline{MB}'| - |\overline{MC}'| \leq |\overline{MB}| = net(V)$. □

5.3 Merging Results from Different Maximal Paths

Example 4. In the program in Figure 1(a), there are 3 maximal paths following `T0 => tree(K0,V0,L0,R0)`: [AB,BC,CE], [AB,BD,DE], and [AB,BE]. The stack optimization graphs for each maximal path are shown in Figure 4. None of the maximal matchings leave unmatched cost nodes in V, and we get the same result $V = \{K0, V0, L0, R0\}$ along each maximal path. We will therefore access all the variables in $\{K0, V0, L0, R0\}$ via $T0$ along every maximal path. The resulting optimized program in shown in Figure 1(b).

However, in general we may compute different sets V along different maximal paths. If the value of V computed along a given maximal path does not include a given field variable, then accessing that field variable via the heap cell along that maximal path may lead to a slowdown when execution takes that maximal path. Accessing a field variable via the cell on some maximal path and via a stack slot on other maximal paths doesn't make sense: if we need to store that field variable in a stack slot in the first interval for use on the maximal paths in which it is accessed via the stack slot, we gain nothing and may lose something if we access it via the cell along other maximal paths. Since we try to make sure that our optimization never slows down the program ("first, do no harm"), we therefore access a field variable via the cell only if all maximal paths prefer to access that field variable via the cell, i.e. if the field variable is in the sets V computed for all maximal paths.

The value of V we compute along a given maximal paths guarantee that accessing the variables in V via the cell instead of via stack slots will not slow the program down. However, there is no similar guarantee about subsets of V: accessing a *subset* of the variables in V via the cell instead of via stack slots *can* slow the program down. It would therefore not be a good idea simply to take the intersection of the sets V computed along the different maximal paths and access the variables in the intersection via the cell.

What we do instead is restrict the candidate set by removing from it all the variables that are not in the intersection,

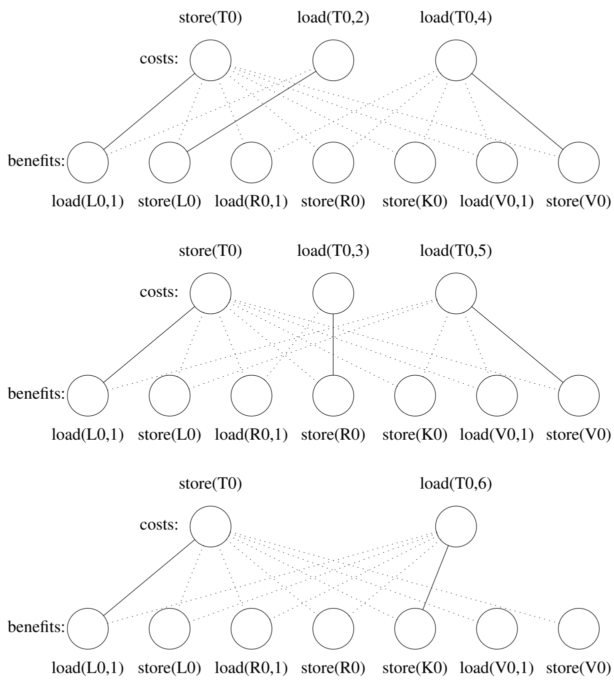

Figure 4: The stack optimization graphs for maximal paths [AB,BC,CE], [AB,BD,DE], and [AB,BE] of the program in Figure 1(a).

and restart the analysis from the beginning with this new candidate set, and keep doing this until we get the same set V for all maximal paths. Each time we restart the analysis we remove at least one variable from the candidate set. The size of the initial candidate set thus puts an upper bound on the number of times we need to perform the analysis.

5.4 Cost of Operations

Until now, we have assumed that all loads and stores cost the same. While this is reasonably close to the truth, it is not the whole truth. Our optimization deals with two kinds of stores and four kinds of loads. The kinds of stores we deal with are (1) the stores of field variables to the stack, and (2) the stores of cell variables to the stack. The kinds of loads we deal with are (1) loading field variables into registers so we can store them on the stack (in the initial segment); (2) loading a cell variable from the stack into a register (in later segments) so that we can use that register as a base for loading a field variable from that cell; (3) loading a field variable from a cell; and (4) loading a variable from a stack slot.

Our transformation adds type 2 loads and possibly a type 2 store while removing type 1 stores and possibly some type 1 loads; as a side effect, it also turns some type 4 loads into type 3 loads. The stores involved on either side of the ledger go the current stack frame, which means that they are likely to be cache hits. Type 1 loads are clustered, which means that they are also likely to be cache hits. For example, if a unification deconstructs a cell with five arguments on a machine on which each cache block contains four words, then the five type 1 loads required to load all the arguments in the cell into registers will have at most two cache misses. Type 2 loads, occurring one per cell per segment, are not clustered at all, and are therefore much more likely to be cache misses. Type 3 loads are also more likely to be cache misses than type 4 loads.

Loads of type 1 will typically be followed within a few instructions by a store of the loaded value. Loads of type 2 will typically be followed within a few instructions by a load of type 3 using the loaded value as the cell's address. Our optimization can turn a type 4 load into a type 3 load, but when it does so, it does nothing to change the distance between the load instruction and the next instruction that needs the loaded value.

Both types of stores have the property that the value being stored is not likely to be accessed in the next few instructions, making a pipeline stall from a data hazard most unlikely. Type 1 and 2 loads, on the other hand, have a significant chance of causing a data hazard that results in a stall. What this chance is and what the cost of the resulting stall will be depends on what other, independent instructions can be scheduled (by the compiler or by the hardware) to execute between the load and the first instruction that uses the loaded value. This means that the probability and cost (and thus the average cost) of such stalls is dependent on the program and its input data.

Since the relative costs of the different types of loads and stores depend on the average number and length of the cache misses and stalls they generate, their relative costs are program-dependent, and to a lesser extent data-dependent as well. We have therefore extended our optimization with four parameters that give the relative costs of type 1 and 2 loads and type 1 and 2 stores. (The cost parameter for type 2 loads is also supposed to account for the associated cost of turning some type 4 loads into type 3 loads.) The parameters are in the form of small integers. Our extension consists of replicating each node in the stack optimization graph c times where c is the cost parameter of the type of operation represented by the node. All replicas of a given original node have the same connectivity, so (according to Lemma 1) we retain the property that all copies of the node will either be marked or not, and hence the set V remains well defined, and the theorem continues to hold. However, the matching algorithm will now generate a solution whose net effect is e.g. the addition of n type 2 (cell variable) loads and the removal of m type 1 (field variable) stores only if $n \times CellVarLoadCost \geq m \times FieldVarStoreCost$. In our experiments, we set $CellVarLoadCost$ to 3, and the other three parameters ($CellVarStoreCost$, $FieldVarLoadCost$ and $FieldVarStoreCost$) to 1.

6. TRANSFORMING THE CODE

Once we have determined the set V of field variables that should be accessed through the cell variable from a deconstruction b `=>` $\mathtt{f}(a_1, \ldots, a_n)$, we transform the program by adding clones of the deconstruction.

We perform a forward traversal of the procedure body starting from the deconstruction, applying a current substitution as we go. Initially the current substitution θ is the identity substitution. When we reach the beginning of a segment i where $V \cap vars(i) \neq \emptyset$, we add a clone of the deconstruction, with each field variable f replaced by a new variable f'. We construct a substitution $\theta = \{f \mapsto f' \mid f \in V\}$ which has the effect of replacing each variable we will access through the cell by the copy in the clone deconstruction.

The remaining new variables in the clone deconstruction will never be used. We then proceed applying the substitution until we reach the end of the segment.

Example 5. For the program given in Figure 3(a), where $V = \{V0, L0, R0\}$. Traversing forward from the deconstruction, we reach segment 2, the segment following the call to `dodgy/4`. Since $vars(2) \cap V$ is not empty, we add a clone of the deconstruction `T0 => f(K0_2, V0_2, L0_2, R0_2)` and construct the substitution $\theta = \{V0 \mapsto V0\_2, L0 \mapsto L0\_2, R0 \mapsto R0\_2\}$. Continuing the traversal, we apply the substitution to `balanced(L0, R0)`, replacing it with `balanced(L0_2, R0_2)`. Note that `K0_2` and `V0_2` are never used. On reaching the end of segment 2 and the start of segment 3, we insert a new clone deconstruction, `T0 => f(K0_3, V0_3, L0_3, R0_3)` and construct a new current substitution $\theta = \{V0 \mapsto V0\_3, L0 \mapsto L0\_3, R0 \mapsto R0\_3\}$. The processing of the later segments is similar.

For segments that share intervals (which must be an interval ending at the start of a switch), this transformation inserts the clone unification after the (unique) anchor that starts all those segments. In Figure 1(a), this would mean inserting a single clone deconstruction immediately after the call to `compare` instead of the three clone deconstructions at the starts of the three switch arms we show in Figure 1(b). However, the Mercury code generator does not load variables from cells until it needs to. The code generated by the transformation therefore has exactly the same effect as the code in Figure 1(b).

The optimization extends in a straightforward manner to cases where a cell variable b is itself the field variable of another deconstruction, e.g. c `=> g(`b`,`b_2`, ... ,`b_k`)`. Simply applying the optimization first to the deconstruction b `=> f(`a_1`, ... ,`a_n`)`, and then applying the optimization to c `=> g(`b`,`b_2`, ... ,`b_k`)` will achieve the desired effect.

Our implementation of the optimization has two passes over the procedure body. The first pass is a backwards traversal. It builds up data structures describing intervals and segments as it goes along. When it reaches a deconstruction unification, it uses those data structures to find the candidate variables, applies the matching algorithm to find which candidates should be accessed via the cell variable, and then updates the data structures to reflect what the results of the associated transformation *would* be, but does not apply the transformation yet. Instead the transformations required by all the optimizable deconstructions are performed all at once by the second, forward traversal of the procedure.

7. PERFORMANCE EVALUATION

We have implemented the optimization we have described in this paper in the Melbourne Mercury compiler. In our initial testing, we have found it necessary to add two tuning parameters.

- If the *one-path node ratio threshold* has a value OPR, then we accept the results of the matching algorithm on a given path only if the ratio between the number of benefit nodes and the number of cost nodes in the computed matching is at least $OPR\%$.
- If the *all-paths node ratio threshold* has a value APR, then we accept the results of the matching algorithm only if the ratio between the total number of benefit nodes and total number of cost nodes on all paths is at least $APR\%$.

To be accepted, a result of the matching algorithm must pass both thresholds. If it fails one or both thresholds, the algorithm will not use that cell as an access path to its field variables, and will store all its field variables on the stack.

Example 6. Consider the program in Figure 3. This contains only one path, whose matching is shown in Figure 2. The ratio of the (unmarked) benefit nodes to the (unmarked) cost nodes (which correspond to the benefits and costs of accessing $\{L0, R0, V0\}$ via cell variable $T0$) is 3/3 = 100%. If $OPR > 100$, this optimization will be rejected and each of $\{L0, R0, V0\}$ will be stored in its own stack slot. Since there is only one path, the all-paths ratio is the same.

Example 7. The program in Figure 1(a) has three paths: [AB,BC,CE], [AB,DB,DE] and [AB,BE]. The matchings are shown in Figure 4. The ratio of the (unmarked) benefit nodes to the (unmarked) cost nodes for [AB,BC,CE] and for [AB,DB,DE] is 7/3 = 233.3%, while the ratio for [AB,BE] is 7/2 = 350%, so the one-path node threshold will not reject the transformation leading to the code in Figure 1(b) unless $OPR > 233$. While the three paths share all their benefit nodes, they share only one cost node, so when one looks at all paths, the benefit node to costs node ratio is only 7/6 = 116.7%. Hence the all-path node threshold will reject the transformation if $APR > 116$.

Increasing the one-path node ratio threshold beyond 100% has the same kind of effect as increasing the numbers of nodes allocated to cell variable loads and stores relative to the numbers of nodes allocated to field variable loads and stores. Its advantage is that setting this threshold to (say) 125% is significantly cheaper in compilation time than running the matching algorithm on graphs which have five copies of each cost node and four copies of each benefit node.

The all-path node ratio threshold poses a different test to the one-path node ratio threshold, because different paths share their benefits (the elimination of field variable stores and maybe loads) but not the principal component of their costs (the insertion of cell variable loads into segments) in controlling the impact of the optimization on executable size. The all-path node ratio threshold is useful in controlling the impact of the optimization on executable size. If all operations have the same number of nodes, then setting this parameter to 100% virtually guarantees that the optimization will not increase the size of the executable; if the cost operations have more nodes than the benefit operations, then setting this parameter to 100% virtually guarantees that any application of the transformation will strictly decrease the size of the executable. (One cannot make a concrete guarantee because it is the C compiler, not the Mercury compiler, that has the final say on executable size.)

There are two reasons why we found these thresholds necessary. First, the impacts of the pipeline effects and cache effects we discussed in Section 5.4 vary depending on the circumstances. Sometimes these variations make the transformed program faster than the original; sometimes they make it slower. The thresholds allow us to filter out the applications of the transformation that have the highest chance of slowing down the program, leaving only the applications that are very likely to yield speedups. Second, even

program	lines	no opt	opt: 100/125		opt: 133/133		opt: 150/100		opt: 150/125	
		time	time	ratio	time	ratio	time	ratio	time	ratio
mmc	262844	50.31	44.84	89.1%	44.05	87.6%	44.94	89.3%	45.55	90.5%
compress	689	15.66	15.67	100.0%	15.66	100.0%	15.65	100.0%	15.66	100.0%
icfp2000	6139	34.95	35.16	100.6%	35.35	101.1%	35.19	100.7%	35.01	100.2%
icfp2001	932	18.49	18.30	99.0%	18.31	99.0%	18.31	99.0%	18.31	99.0%
nuc	3475	35.63	36.45	102.3%	36.45	102.3%	34.68	97.3%	34.69	97.4%
ray	2102	13.42	13.31	99.2%	13.29	99.0%	13.30	99.1%	13.29	99.0%

Table 1: Performance evaluation

if the original program and the transformed program have the same performance with respect to cache and pipeline effects, we have reason to prefer the original program. This reason concerns what happens when a cell variable becomes dead while some but not all of its field variables are still alive. With the original program, the garbage collector may be able to reclaim the storage occupied by the cell's dead field variables, since there may be no live roots pointing to them. With the transformed program, such reclamation will not be possible, because there *will* be a live root from which they are reachable: the cell variable, whose lifetime the transformation artificially extends. It would be interesting to try applying the transformation only if it doesn't extend the lifetime of the cell, but the Mercury compiler doesn't (yet) have the analysis this would require.

We have therefore tested each of our test programs with a several sets of parameter values. (Unfortunately, the whole parameter space is so big that searching it even close to exhaustively is not really feasible.) Due to space limitations, we cannot present results for all of these parameter sets. However, the four we have chosen are representative of the results we have; the comments we make are still true when one looks at all our results to date.

All four sets of parameter values had the cost of a cell variable load set to three while all other operations had cost one, since our preliminary investigations suggested these as roughly right. The sets differed in the values of the one-path and all-path node ratio thresholds. The four combinations of these parameters' values we report on are 100/125, 133/133, 150/100 and 150/125 (one-path/all-path).

Our test programs are the following. The `mmc` test case is the Melbourne Mercury compiler compiling six of the largest modules in its own code. `compress` is a Mercury version of the 129.compress benchmark from the SPECint95 suite. The next two entries involve our group's entries in recent ICFP programming contests. The 2000 entry is a ray tracer that generates .ppm files from a structural description of a scene, while the 2001 entry is a source-to-source compression program for a hypothetical markup language. `nuc` is a Mercury version of the pseudoknot benchmark, executed 1000 times. `ray` is a ray tracing program generating a picture of a helix and a dodecahedron. The benchmark machine was a Dell PC: 1.6 MHz Pentium IV, 512 Mb, Linux 2.4.16.

Table 1 shows the results. The first column identifies the benchmark program, and the second gives its size in source lines of code (measured by the word count program `wc`). The third column gives the time taken by the program when it is compiled without stack slot optimization, while the following four groups of two columns give both the time it takes when compiled with stack slot optimization and the indicated set of parameter values, and the ratio of this time to the unoptimized time. All these times were derived by executing each benchmark program eight times, discarding the highest and lowest times, and averaging the remaining times.

The table shows a wide range of behaviors. On compress, the optimization has no effect; `compress` simply doesn't contain the kind of code that our optimization applies to. On two programs, `icfp2001` and `ray`, stack slot optimization consistently gives a speedup of about 1%. On `icfp2000`, stack slot optimization consistently gives a slowdown of around 1%. On `nuc`, stack slot optimization gives a speedup of a bit more than 2% for some sets of parameter values and a slowdown of a bit more than 2% for other sets of parameter values, which clearly indicates that some of the transformations performed by stack slot optimization are beneficial while others are harmful, and that different parameter values admit different proportions of the two kinds. In general, raising the threshold reduces the probability of a slowdown but also reduces the amount of speedup available; one cannot guarantee that a given threshold value excludes undesirable applications of the transformation without also guaranteeing that it also excludes the desirable ones. On `icfp2000`, the parameter values we have explored (all in the range 100 to 150) admitted too many of the wrong kind; using parameter values above 150 or using higher costs for cell loads and stores may yield speedups for `icfp2000` also.

On `mmc`, stack slot optimization achieves speedups in the 9% to 12% range. When one considers that one doesn't expect any program to spend much more than 30% of its time doing stack accesses (it also has to allocate memory cells, fill them in, make decisions, perform calls and returns, do arithmetic and collect garbage), and these results show the elimination of maybe one quarter of stack accesses, these results look very impressive. Interestingly, previous benchmark runs (which may be more typical) with slightly earlier versions of the compiler yielded somewhat smaller speedups, around 5-9%, but these are still pretty good results.

We think there is a reason why it is the largest and most complex program in our benchmark suite that gives by far the best results: it is because it is also by far the program that makes the most use of complex data structures. Small programs tend to use relatively simple data structures, because the code to traverse a complex data structure is also complex and therefore big. There is therefore reason to believe that the performance of the stack slot optimization on other large programs is more likely to resemble its behavior on `mmc` than on the other benchmark programs. However, the fact that stack slot optimization can sometimes lead to slowdowns means that it may not be a good idea for compilers to turn it on automatically; it is probably better to let the programmer do so after testing its effect.

Our benchmarking also shows that stack slot optimization usually reduces the sizes of executables, usually by 0.1% to 0.5%; in the few cases it increases them by a tiny amount (less than 0.1%). This is nice, given that many optimizations improve speed only at the cost of additional memory.

Enabling stack slot optimization slows down compilation only slightly. In most cases we have seen, the impact on compilation time is in the 0%-2% range. In a few cases, we have seen it go up to 6.5%; we haven't seen any higher than that.

We have also looked at the effect of our optimization on the sizes of stack frames. The tested version of the Mercury compiler has 8815 predicates whose implementations need stack frames. With the parameter values we explored, stack slot optimization was able reduce the sizes of the stack frames of up to 1331 of those predicates, or 15.1%. It also reduced the average size of a stack frame, averaged over all 8815 predicates, by up to 13.3%, from 5.50 words to 4.77 words. Oddly enough, the optimization leads to only a trivial reduction (less than 0.1%) in the stack space required to execute the compiler; at the point where the compiler requires maximum stack space, virtually all of the frames in the stack are from predicates whose stack frame sizes are not affected by our optimization.

8. CONCLUSION AND RELATED WORK

The optimization we have described replaces a number of stack accesses with an equivalent or smaller number of heap accesses. In many cases, the optimization will reduce the number of memory accesses required to execute the program. The optimization can also reduce the sizes of procedures' stack frames, which improves locality and makes caches more effective. The optimization can lead to significant performance improvements in programs manipulating complex data structures. A default usage of this optimization needs to be conservative in order to avoid slowdowns; the parameter values 150/125 would appear to be suitable for this. To gain maximum benefit from this optimization, the programmer may need to explore the parameter space, since conservative thresholds can restrict the benefit of the optimization.

The optimization we define is somewhat similar to rematerialization for register allocation (e.g. [1, 2]), where sometimes it is easier to recompute the value of a variable, than to keep it in a register. Both rematerialization and our optimization effectively split the lifetime of a variable in order to reduce pressure on registers or stack slots. The key difference is that the rematerialization of a variable is independent of other variables, whereas the complexity of our problem arises from the interdependence of choices for stack slots. In fact, the Mercury compiler has long had an optimization that took variable definitions of the form b `<=` `f` where `f` is a constant, and cloned them in the segments where b is needed in order to avoid storing b on the stack, substituting different variables for b in each segment. Unlike the optimization presented in this paper, that optimization requires no analysis at all.

Also somewhat related is partial dead code elimination [5]. The savings on loads in the initial segment effectively result from "sinking" the calculation of a field variable to a point later in the program code. We restrict the candidate field variables to ensure this sinking does not add overhead.

It is worth discussing how the same optimization can be applied to other languages. The optimization is applicable to Prolog even without mode information, since even though the first occurrence of a unification $b = f(a_1, \dots, a_n)$ may be neither a construct or a deconstruct, after its execution all copies added by our algorithm will be deconstructs. A good Prolog compiler can take advantage of this information to execute them efficiently. Of course without information on determinism, the optimization is more dangerous, but simply assuming code is deterministic is reasonable for quite a large proportion of Prolog code. Unfortunately any advantage is not likely to be visible in WAM-based compilers, since the "registers" of the WAM are themselves in memory. For strict functional languages such as ML the optimization is straightforwardly applicable, since mode information is syntactically available and the code is always deterministic.

It should also be possible to apply this optimization to imperative languages. It would require the compiler to perform mutability analysis; in particular, it cannot access a field variable via its containing structure unless it can prove that this field of that structure is not modified between the assignment from that field to the variable (the imperative equivalent of a deconstruction) and the last access to the variable. Note that we need not require the structure to be fully immutable; the optimization doesn't care if it is updated outside this interval. The natural adaptation of our algorithm to imperative languages would actually simplify it, since such languages don't have backtracking. For example, the natural equivalent of a Mercury interval is a basic block, and the task of finding sets of maximal paths is standard control flow analysis.

The optimization will be available in the next major release of the Mercury system; it has been available in releases-of-the-day of the Mercury system at `<http://www.cs.mu.oz.au/mercury/>` since April.

We would like to thank the Australian Research Council for its support.

9. REFERENCES

[1] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. In *Proceedings of PLDI '92*, pages 311–321, 1992.

[2] Gregory J. Chaitin. Register allocation spilling via graph coloring. In *Proceedings of the 1982 Symposium on Compiler Construction*, pages 98–105, 1982.

[3] Fergus Henderson, Thomas Conway, Zoltan Somogyi, David Jeffery, Peter Schachte, Simon Taylor, and Chris Speirs. The Mercury language reference manual. Available from `<http://www.cs.mu.oz.au/mercury/>`, 2000.

[4] J.E. Hopcroft and R.M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal of Computing*, 4:225–231, 1973.

[5] Jens Knoop, Oliver Ruthing, and Bernhard Steffen. Partial dead code elimination. In *Proceedings of PLDI '94*, pages 147–158, 1994.

[6] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 26(1-3):17–64, 1996.