RETURN VALUE PLACEMENT AND TAIL CALL OPTIMIZATION IN HIGH LEVEL LANGUAGES *

PETER A. BIGOT AND SAUMYA DEBRAY

▷ This paper discusses the interaction between tail call optimization and the placement of output values in functional and logic programming languages. Implementations of such languages typically rely on fixed placement policies: most functional language implementations return output values in registers, while most logic programming systems return outputs via memory. Such fixed placement policies incur unnecessary overheads in many commonly encountered situations: the former are unable to implement many intuitively iterative computations in a truly iterative manner, while the latter incur a performance penalty due to additional memory references. We describe an approach that determines, based on a low-level cost model for an implementation together with an estimated execution profile for a program, whether or not the output of a procedure should be returned in registers or in memory. This can be seen as realizing a restricted form of inter-procedural register allocation, and avoids the disadvantages associated with the fixed register and fixed memory output placement policies. Experimental results indicate that it provides good performance improvements compared to existing approaches. ◁

* This work was supported in part by the National Science Foundation under grant number CCR-9123520. The first author was also supported by graduate fellowships from the U.S. Office of Naval Research and AT&T Bell Laboratories. A preliminary version of this paper appeared in *Proc. Eleventh International Conference on Logic Programming*, Santa Margherita Ligure, Italy, June 1994.

Address correspondence to Saumya K. Debray, Department of Computer Science, University of Arizona, Tucson, AZ 85721, USA. E-mail: `debray@cs.arizona.edu`

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 0743-1066/93/$3.50

1. INTRODUCTION

Programs in functional and logic programming languages tend to be procedure call intensive. Because of this, implementations of such languages must handle the data and control transfers at procedure calls and returns efficiently in order to get good performance. The data transfer overhead is usually reduced by placing the arguments to procedures—and, in many systems, the values returned by procedures—in registers. A very important component of techniques that reduce the control transfer overhead is tail call optimization. This paper examines the interaction between the data passing optimization of placing arguments and return values in registers and the control passing optimization of tail call optimization.

Implementations of functional languages typically adopt fixed register usage conventions for passing arguments to functions and returning values from them. A common approach is to use a fixed mapping from the position of a value in an argument sequence to the register in which that value is passed: the first argument to a function is passed in register 1, the second argument in register 2, etc.; the first return value is returned in register 1, the second return value in register 2, and so on (see, for example, [3, 8, 11]; the S-1 Common Lisp compiler uses this approach for numerical return values [6]). A similar situation arises in systems such as Standard ML of New Jersey [1] that use continuation passing style, and which pass arguments to "known" functions in registers: since functions in continuation passing style do not actually return any values to their caller, but pass them instead as arguments to a continuation, the placement of these "return values" is determined by the scheme used for passing arguments into a function. The advantage of such fixed schemes is uniformity and simplicity. They have two disadvantages: first, as we will show in Section 2, an *a priori* commitment to pass return values in registers may force a program to incur unnecessary space and time overheads; and second, a fixed positional mapping of values to registers can require additional register shuffling to move a value into the appropriate register. The second problem, namely, register shuffling, can be addressed to some extent by techniques such as register targeting [1, 2, 11], but these do not address the additional space overheads that can be incurred by such schemes.

It is interesting to contrast such register-return models, commonly used in functional programming systems, with implementations of logic programming languages such as Prolog. Prolog procedures do not, in general, have any notion of input and output arguments, and a particular argument to a procedure can be an input argument in one invocation and an output argument in another. Because of this, it is simplest to pass all arguments to a procedure in registers, with each unbound variable—usually corresponding to an output argument—passed by reference, as a pointer to the cell occupied by that variable. An output value is returned by binding it to a variable, i.e., by writing to the corresponding memory location. This works well in some cases, but incurs unnecessary overheads in others because of the additional memory references incurred in initializing the output locations, writing values to them, and then reading these values back at the point of use.

At first glance, the placement of return values would seem to be a rather small and, presumably, unimportant aspect of an implementation of a programming language. It turns out that because of interactions between return value placement and tail call optimization, placement decisions can have a surprisingly large impact on execution speed. Moreover, no single fixed placement scheme is good for all programs: many commonly encountered programs do better with register placements, and many others run faster with memory placements. What is desirable is a method whereby a compiler can determine, for each procedure in a program, which placement scheme is best for it. This paper discusses an algorithm that accomplishes this, by taking into account execution frequency estimates and the relative costs of various low level operations to evaluate the costs and benefits of various alternatives, and choosing placements for the different output arguments for different procedures in a program in a way that attempts to minimize the overall execution time of the program. The assumptions made by our algorithm are fairly weak, and are applicable to a reasonably wide variety of languages and systems. The most fundamental assumption we make is that tail call optimization is implemented. In other words, when the last action performed by a procedure p is a call to another procedure q—a situation that is referred to as a *tail call*—any environment allocated for p is no longer needed and can therefore be reclaimed, once the arguments to the call to q have been computed into the appropriate locations. This allows the call to q to be implemented as a simple jump, thereby avoiding unnecessary state saving and a procedure call and return. We assume also that input arguments to a procedure call are passed in registers; the mapping that determines which parameter gets passed in which register need not be the same for all functions. This assumption is satisfied by most modern implementations of high level languages. Experimental results indicate that our algorithm generally makes the right decisions, choosing register placements for procedures that benefit from having their outputs returned in registers, and memory placements for procedures for which this is better.

2. OUTPUT VALUE PLACEMENT AND TAIL CALL OPTIMIZATION

Consider the following Scheme function to count the length of a list:

```
(define (length x)
   (if (null? x) 0 (+ 1 (length (cdr x))))
)
```

Suppose the recursive call to `length` returns its value in a register. The next action that has to be taken, upon return from this call, is to increment the value returned, and, since it is already available in a register, this can be done by a simple register increment operation. If, on the other hand, the returned value had been placed in memory, it would be necessary to incur several memory operations—which are considerably more expensive—to achieve the same effect. In this case, therefore, the natural place to put the return value is in a register.

As this simple example illustrates, there are, in many cases, significant performance advantages to returning output values in registers rather than in memory. However, the situation is complicated by the interaction of this optimization with tail call optimization. The problem is that if a tail call returns a value to its caller in a register r, but the caller wants that value in a different location x, then it is necessary to insert *move* or *load* instructions after the tail call to reconcile the return locations of caller and callee, and this inserted code precludes tail call optimization. This can be seen in the context of procedures that recursively construct data structures, which are common in functional and logic programming languages.

In many implementations, such structures are allocated on the heap. In these cases, if the recursive calls that construct the "rest" of the structure return their values in registers, additional code is necessary to store the values into memory, rendering tail call optimization inapplicable and increasing the memory requirements of programs. To see this, consider the following Scheme function to double each element of a list:

```
(define (ldbl x)
   (if (null? x) () (cons (* 2 (car x)) (ldbl (cdr x))))
)
```

This function creates and returns a list, which naturally resides on the heap; thus, the longer the input list, the more space it will need to create its output. However, the computation performed by this function is, intuitively, iterative in nature—it simply traverses a list, performing some computation on each element—and one might expect that such a computation would use only the amount of storage necessary for the data structures it creates. In other words, given a list of length n, one would hope that this computation would be carried out using $O(1)$ space for environments. Unfortunately, in most implementations this computation will require $O(n)$ storage for environments. For example, assuming that primitive arithmetic operations, as well as the list operations `car` and `cdr`, are performed in-line, a possible execution sequence might be as follows:

1. allocate an environment;
2. evaluate the expression `(* 2 (car x))`;
3. save this value—call it z—in the environment;
4. recursively evaluate `(ldbl (cdr x))`;
5. load z from the environment into a register;
6. allocate a cons cell on the heap and set its head to z and its tail to the value returned by the recursive call;
7. load the address of this cons cell into the appropriate register and return

This requires the allocation of an environment at each level of recursion, which is expensive in both time and space.

Superficially, the reason this function is not executed in an iterative manner is that it is not syntactically tail recursive. However, this explanation is overly simplistic. The definition of the `ldbl` function, read declaratively, states that the value of `(ldbl x)`, where `x` is a nonempty list with head y and tail z, is a list whose head is $2y$ and whose tail is `(ldbl` z`)`. If an implementation insists on returning values—in this example, in particular, the value of the recursive call—in registers, then it has no option but to insert an assignment after the recursive call to write the value returned by this call from a register into the memory location at the tail of the cons cell, and this, of course, precludes tail call optimization. However, the declarative reading of the function does not demand any particular temporal ordering between the creation of the cons cell and the recursive call in the body of the function. Thus, suppose we were to implement the function to take, as an additional (compiler-introduced) argument, a memory address `addr` into which its output should be stored. The computation could then proceed as follows:

1. compute the value of `(* 2 (car x))`;
2. allocate a cons cell on the heap and store the value so computed into the head of this cell;
3. store a NULL value into the tail of this cons cell (to help the garbage collector);
4. assign the address of this cons cell into the location given by `addr`;
5. set `addr` to the address of the tail of this cons cell;
6. make the recursive call, which can now be done with a simple jump instruction[1]

In fact, the Prolog version of this function given below would, under most implementations, realize this behavior almost exactly—the computation would be iterative and use $O(1)$ space other than space used for data structures created on the heap. The mode declaration '`:- mode ldbl(in, out)`' accompanying the definition of the procedure specifies that the first argument of the procedure `ldbl` is an input argument and the second argument is an output argument.

```
:- mode ldbl(in, out).
ldbl([], []).
ldbl([H1|L1], [H2|L2]) :- H2 is 2*H1, ldbl(L1, L2).
```

Apart from the additional memory requirements discussed above, the register-return model can also incur a secondary cost in the form of additional runtime checks. In the `ldbl` function above, for example, if garbage collection is initiated via explicit tests on the heap and/or stack pointers, the register-return version will require at least two overflow checks at each level of recursion: one, before the recursive call, to determine whether there is enough space to allocate an environment; and another, after the recursive call, to verify that there is enough space to allocate a cons cell. These checks cannot be coalesced: for example, we cannot use a single check before the recursive call to determine whether there is enough space for an environment and a cons cell, because even if enough space is available before the recursive call is made, in general it cannot be guaranteed that the space for the cons cell will still be available when control returns from the recursive call. In general, this situation occurs whenever different memory allocations are separated by a function call. In the memory-return model, however, if the points at which memory allocations occur can be moved so that different allocations are not separated by function calls, the overflow checks for the different allocations can be coalesced into a single test—this is true of the memory-return version of the `ldbl` function above, independent of whether or not an environment is allocated.

A similar problem arises if we have procedures with multiple return values. Again, if output arguments are returned in registers, then it is impossible to avoid deoptimizing some tail calls in some cases, regardless of what approach is taken for output register assignment and code generation. This is illustrated by the following example. Consider a Prolog procedure defined by the clauses

```
:- mode p(out, out), q(out, out).
```

[1] If other functions rely on the convention that return values are passed in registers, it would be necessary, after the execution of the memory-return function has finished, to load the value computed into a register, but this is easily done using a wrapper function and not too expensive.

```
p(X,Y) :- q(X,Y).
p(X,Y) :- q(Y,X).
```

As indicated by the "`:- mode` ..." declaration, both the arguments of each of the procedures `p` and `q` are output values. The first clause defines `X`, the first output value of `p`, to be the first output value of `q`; and `Y`, the second output value of `p`, to be the second output value of `q`. In the second clause, the order of outputs is reversed: the first output value of `p` is the second output value of `q`, and the second output of `p` is the first output of `q`.

It is not difficult to see that if either `p` or `q` returns either of its outputs in registers, at least one of the clauses defining `p` will have to give up tail call optimization. However, if both `p` and `q` return their outputs in memory, tail call optimization can be retained by permuting the addresses of the output locations in the second clause before making the tail call.

As illustrated by the `ldbl()` function discussed earlier, a procedure call that is not syntactically a tail call may nevertheless, in some circumstances, be implemented as a tail call: this can be done if the only action taken by the caller after returning from that call is to store the returned value into memory, after possibly allocating memory for this purpose. In that case, we can instead allocate the memory beforehand if necessary, then pass a memory address into the call and have the callee store the return value into the corresponding location. In general, this can be done even if a call returns more than one output value, as long as the only action of the caller after returning from the call is to possibly allocate memory, then store each return value into memory. Based on this, we classify a call in the body of a procedure as a *potential tail-call* if (*i*) it occurs syntactically as a tail call; or (*ii*) the only actions taken by the caller, after control returns from that call, are to store its return values into memory.

3. OUTPUT VALUE PLACEMENT POLICIES

Most implementations of functional and logic programming languages use a "homogeneous" output placement policy: output values are always placed in one class of locations—i.e., always in registers, or always in memory.[2] This obviates the need to make complicated decisions about the "best" location for a return value, thereby simplifying compilation. Several homogeneous policies are plausible.

3.1. Fixed Register Returns

The simplest way to assign registers to output values is to adopt a fixed mapping from outputs to registers. For example, we may use a convention similar to that used for the input arguments, with the first return value being placed in register 1, the second in register 2, and so on. The simplicity of this approach makes it the method of choice in many functional language implementations [3, 6, 8, 11].

[2] In reality, implementations have only a bounded number of registers available to them. Because of this, a system that would otherwise pass a value in a register may be forced, due to an inadequate number of available registers, to pass it in memory. We consider such placement decisions—which would change if we could somehow increase the number of available registers—to be homogeneous.

It also has the merit that, in the absence of recursive data structure construction and multiple return values, a call that appears syntactically in a tail call position can be guaranteed to be implementable as a tail call (as illustrated in Example 3.2, this is not true of schemes such as register targeting [1, 2, 11], which relax the fixed positional association between return values and registers in order to reduce the shuffling of data between registers). Unfortunately, it suffers from two disadvantages. First, and most serious, is the fact, illustrated in Section 2, that it may preclude the use of tail call optimization under some circumstances—specifically, in computations involving the creation of data structures, even if the computations are intuitively iterative in nature, and in computations involving multiple return values. The second disadvantage is that because of the fixed mapping from the position of an output value to the register it is returned in, additional register shuffling may be necessary to move it to the register that it needs to be in. Both of these problems are illustrated by the following example.

Example 3.1. Consider the following implementation of the quicksort algorithm in Prolog:

```
:- mode qsort(in, out).
qsort([], Sorted) :- Sorted = [].
qsort([H|L], Sorted) :-
   partition(H, L, Big, Small),
   qsort(Small, SmallS),
   qsort(Big, BigS),
   append(SmallS, [H|BigS], Sorted).

:- mode partition(in, in, out, out).
partition(X, [Y|L], Big, Small) :-
   Y >= X,
   Big = [Y|Bs],
   partition(X, L, Bs, Small).
partition(X, [Y|L], Big, Small) :-
   Y <  X,
   Small = [Y|Sms],
   partition(X, L, Big, Sms).
partition(_, [], Big, Small) :-
   Big = [], Small = [].

:- mode append(in, in, out).
append([], L, Lout) :- Lout = L.
append([H|L1], L2, Lout) :- Lout = [H|L3], append(L1, L2, L3).
```

Suppose that we use the fixed register placement policy described above for output values. Consider the first clause for the procedure `partition`: the recursive call in the body will return the value of `Bs` in register 1 and `Sms` in register 2. After control returns from the recursive call, however, it will be necessary to take the value of `Bs` and create the cons pair `[Y|Bs]`: this will result in a loss of tail call optimization for this clause. A similar consideration, applied to the value of `Sms`, will preclude tail call optimization in the other recursive clause for this procedure. Similarly, the procedure `append` will not be tail recursive because of the need to create the cons pair `[H|L3]` after the recursive call returns the value of `L3` in register 1.

In the recursive clause for the procedure `qsort`, the call to `partition` will return the value of `Big` in register 1 and that of `Small` in register 2. However, our parameter passing convention demands that the value of `Small`, which is the first argument of the the next call, `qsort(Small, SmallS)`, be in register 1. This requires additional data movement between registers that might have been avoided with a more flexible output placement policy. ■

3.2. Flexible Register Returns

The discussion of fixed register return policies identified two problems: first, fixed register returns are unable to realize some intuitively iterative computations in a truly iterative way; and second, they sometimes incur additional register shuffling to move a value from the register it was returned in to that where it is needed. The second of these problems can be avoided using *flexible register return policies*, where the positional association between values and registers is relaxed. This can be accomplished, for example, using register targeting techniques [1, 2, 11] or inter-procedural register allocation [15]. However, flexible register returns exacerbate the problem with tail call deoptimization due to mismatches in return register choices. In particular, unlike the fixed register return case, tail call optimization can be blocked even in the absence of multiple return values and data structures constructed on the heap. This is illustrated by the following example.

Example 3.2. Consider a function `f` that returns a value returned to it by another function `g`:

```
(define (f x) (g (h x)) )
```

Suppose that, in our desire to avoid register moves, we decide to place `f`'s output in register 2 based on an examination of its call sites. Similarly, suppose that, based on `g`'s call sites (this one, and others), we decide to place `g`'s output in register 3. This decision forces `f` to give up tail call optimization, since additional code must now be inserted in `f` to move `g`'s return value from register 3 to register 2. ■

What this means, in practice, is that when deciding register assignments in flexible register return policies, it is not enough simply to inspect the various call sites for a function to see which position the return value is used in: it is necessary also to take into account the possibility of tail call deoptimizations due to mismatched decisions, and the costs of such deoptimizations (possibly weighted by expected execution frequency). Moreover, flexible register return schemes do not address the first problem discussed above, namely, the inability to implement intuitively iterative computations that involve computations of components of data structures in a truly iterative manner.

3.3. Memory Returns

Unlike functional language systems, implementations of logic programming languages have typically returned output values in memory. A commonly used policy, originating in the Warren Abstract Machine [21], is to pass the ith argument in register i: if the ith argument happens to be a variable (which typically corresponds to an output argument), the value passed is a pointer to the location of the variable (which may be either on the stack or on the heap). In effect, this policy passes output arguments by reference. The policy is motivated by the fact that, in general, Prolog procedures do not have any notion of input and output arguments, and a particular argument to a procedure can be an input argument in one invocation and an output argument in another. Returning outputs in memory allows a simple and uniform treatment of communication between procedure activations under these circumstances.

The main advantage of a memory return policy, apart from simplicity, is that it never prevents tail call optimizations, since one memory location is as good as any other. Because of this, there is no need to insert code to move a value to a preferred location, as in Example 3.2. Thus, both the `partition` and `append` procedures in Example 3.1 can be implemented with tail call optimization under this policy.

The biggest disadvantage of a homogeneous memory return policy is its cost. For each assignment of a return value into memory, we must do two memory writes, one to initialize the location (to allow garbage collection and, in logic programming languages, to allow general-purpose unification routines to work correctly), and one for the eventual assignment; a memory read at the use point; and possibly other operations such as tagging and untagging of pointers. Furthermore, in logic programming languages there will typically be an additional memory read for dereferencing pointer chains that could arise as a result of unification. This disadvantage is exemplified by the following example.

Example 3.3. Consider the following Prolog procedure to compute the factorial of a given number:

```
:- mode fact(in, out).
fact(0, 1).
fact(N, F) :- N > 0, N1 is N-1, fact(N1, F1), F is N*F1.
```

At each level of recursion, the variable `F1`, which corresponds to the output argument of the recursive call, is allocated a slot in the stack frame: this has to be initialized as an unbound variable, which costs a memory write. When the recursive call returns after assigning its return value into `F1`—this costs another memory write—the value of `F1` is retrieved from memory—costing at least a memory read—and used to compute the expression `N*F1`, and the result stored back into memory. This sequence of events is repeated all the way up the chain of recursion. This leads to two sources of overhead: a space overhead because environments on the stack must allocate space for the output variables of procedures, and a time overhead because of the increased memory traffic. It is not difficult to see that the repeated loads and stores of the output argument in the example above are not necessary: it can be computed into a register at each level of recursion and returned in that register. ∎

In dynamically typed languages such as Prolog and Scheme, values in memory typically require associated type descriptors, or "tags." Many implementations of such languages implement tagged floating point values as boxed objects: the values themselves are allocated on the heap, and a pointer to the value is passed around.

Input:

1. A set of candidate procedures, with input and output arguments determined via mode analysis (if necessary), and execution frequency estimates for each potential tail call;
2. for each procedure p in the program, a set of registers that is preserved by p;
3. values for the cost parameters of Table 1.

Output: A placement decision for each return value of each candidate procedure.

Method:

0. Initialize all placement costs to 0.
1. [*Pass 1: Local Cost Computation*]
   For each candidate procedure p do:
   (a) for each return value v of p, add in the costs for each placement for v based on how v will be used;
   (b) for each return value v of p, add in the costs for each placement for v based on the points at which v is defined.
2. [*Pass 2: Assigning Output Placements*]
   For each candidate procedure in decreasing order of execution frequency do:
   (a) Update the cost vector of p to account for tail call deoptimizations;
   (b) Use the updated cost estimates to choose a placement for each return value of p.

Figure 1. Overview of Placement Algorithm

Another disadvantage of memory returns is that a value that could have been returned in unboxed form using register returns (e.g., a floating point value that is returned in a floating point register) may require boxing if it is returned in memory. This incurs both space and time overheads: apart from the fact that memory operations are generally more expensive than operations on registers, creating a boxed value may also require additional tests to determine whether or not there is enough space available on the heap.

4. A HETEROGENEOUS OUTPUT PLACEMENT ALGORITHM

As the discussion of the previous section suggests, an output placement policy aimed at generating efficient code should have the following characteristics: it should be *heterogeneous*, so that it can avoid the expensive memory reference behavior of homogeneous memory return policies, and yet be able to realize intuitively iterative computations involving data structure components in a truly iterative fashion; it should be *flexible* in register assignment, so that values are placed in registers where they will be needed next; and it should take into account the expected frequency of execution of various procedures, so that rarely executed code is not optimized at the expense of frequently executed code. This section describes an algorithm we have developed that has these characteristics and that has been incorporated into a compiler that we have implemented for Janus, a committed-choice logic programming language [10]. The compiler uses inter-procedural dataflow analyses [9] to determine the input and output arguments of each procedure, and identify procedures and variables that must use the default output placement policy (e.g., procedures whose execution can suspend and subsequently be resumed, and variables that may be used as logical variables, i.e., "used" before they are defined). The details of these analyses are orthogonal to this paper and are not discussed here. A procedure that meets the criteria for heterogeneous output placement will be referred to as a *candidate* procedure. Our output placement algorithm also assumes that relative execution frequencies for each call site in the program have been obtained separately.

The algorithm has two passes. The first pass assigns costs to various output locations based on the amount of work that would have to be done if those locations were chosen without assuming anything about placements in other procedures. The second pass processes procedures in decreasing order of execution frequency (obtained either using heuristics based on program structure [5, 20] or using execution profiles generated from "training runs" of the program) and does a greedy bottom-up assignment of output locations. A high-level overview of the method appears in Figure 1. The costs incurred by different placements are determined by considering the features of the various contexts in which values are defined and used.

4.1. Pass 1: Determining Output Location Costs

The first pass estimates the costs associated with each potential return location for each output value of a procedure without assuming anything about the output placements of other procedures. It associates a vector of cost information, indexed by potential placement (memory and registers), with each output of a particular procedure. The costs are *incremental*, in the sense that they characterize the additional expense of choosing a particular placement over the best case; and *distributed*, in the sense that they associate the components of a cost induced by choosing a particular location with the program point at which the cost is paid. Costs associated with a particular program point are weighted by the estimated frequency with which control reaches that point. The estimation of the costs of different placements involves looking at two sets of program points separately: the points in a procedure definition where an output value is defined, and the points where the returned value is used. For this, we need to know what registers might be affected by a procedure call:

Definition 4.1. Given a register r, a procedure p is said to be *r-preserving* if the contents of r will be preserved across any call to p. ∎

A register r may be preserved by a procedure p either if r is not modified by p, or if r is saved by p before it is defined, and restored subsequently. Our current implementation uses caller-saved registers uniformly, and as an approximation to register-preservation we assume that each non-primitive procedure defines all registers. However, it is not difficult to see how this heuristic might be improved.

An output value in a procedure is defined either by a primitive operation, or by a call to another user-defined procedure. Within this, we distinguish two types of definition point: one is a definition of a variable local to a procedure, and the other is a definition of a variable which is an output parameter of the procedure in which the definition appears. The later case is more complex, because if the definition point itself is a procedure call, we must take into account the multiple steps that separate the declaration and initialization of the variable from its base definition point.

If an output value is computed into a register r at a point in a procedure p, and there is a subsequent call to a procedure q that is not r-preserving, then the contents of r have to be saved across the call to q and subsequently restored. If q happens to be a tail call, the restoration may prevent tail call optimization. It is important to point out that the costs associated with deoptimizing a tail call are incurred only once for each tail call that is so deoptimized, regardless of how many different reasons might have contributed to the decision to deoptimize it. As an example, consider the following Scheme program fragment:

```
(define (p x) (cons x (q x)))
```

The value returned by `(p x)` is a cons pair whose head is the value of `x` and whose tail is the value returned by `(q x)`. It is not difficult to see that if either `p` or `q` places its return value in a register, it will not be possible to implement the call from `p` to `q` as a tail call. Now consider the situation where both `p` and `q` place their return values in registers. In this case, there are two independent reasons for the loss of tail call optimization in `p`: first, because `p` places its return value in a register; and second, because `q` places its return value in a register. A naive cost computation might examine `q`, consider the fact that a register placement for its return value precludes tail call optimization, and count the cost incurred thereby; then examine `p`, consider the fact that a register placement for its return value would preclude tail call optimization, and count the cost so incurred. This would count twice the cost incurred for losing a single opportunity for tail call optimization in the function `p`, even though in reality this cost is paid only once. To avoid distorting our cost estimates with such miscalculations, we need to keep track of two kinds of information. First, in order to know when to add in the costs associated with giving up tail call optimization, we need to know which placements of a return value will preclude tail call optimization. This is managed by maintaining, for each potential output placement for each output value in a tail call, a flag that indicates whether choosing that location for that value will prevent a tail call optimization for that call. Second, we need to know when the costs associated with a tail call deoptimization have been taken into account already and therefore need not be accounted for again: this is managed using a flag associated with each potential tail call that indicates whether it has already been deoptimized.

In summary, our algorithm maintains the following data structures:

1. for each procedure p, a cost vector $Cost_p(v, l)$ that gives, for each return value v for p and each return location l, the cost of placing v in l;

2. for each potential tail call C, a flag $C.tc\_deopt(v, l)$ that indicates, for each

Parameter	*Description*
rmove	Cost of moving a value from one register to another
rstore	Cost of Storing a value from a register into memory
rload	Cost of loading a value from memory into a register
initmem	Cost of initializing a memory location
call_ta	Cost of a call and return plus environment allocation
call_tn	Cost of a call and return, with no environment allocation

Table 1. Parameters to cost model

return value v for p and each return location l, whether returning v in location l would cause a loss of tail call optimization for C; and

3. for each potential tail call C, a flag $C.tc\_optimizable$ that indicates whether the costs associated with a loss of tail call optimization for C have been accounted for.

4.1.1. Cost Considerations at Use Point For simplicity of exposition, we assume that a strict primitive operation can compute its result into any of an appropriate set of registers (e.g., a floating point operation may compute its result into any floating point register), and that the cost of the operation does not depend on the particular register it computes its result into. This assumption is satisfied by most modern architectures, and it is not difficult to extend our approach to cover situations where it is not.

The costs of preparing for and using a returned value depend on the contexts of the definition and use in a procedure body. Consider a value v that is returned from a call to q in the body of a procedure p: i.e., the definition point for v is a procedure call. There are two distinct and orthogonal kinds of "uses" we have to consider. First, v may be used in an expression or another procedure call at a later point within the body of p. In this case, the costs of different placements for v when returned from q will depend on the context in which v is used. Second, v may be returned by p to its caller. While returning the value is a "use" of v, in that p may be required to move the returned value to another location for the return, it is better interpreted as a multi-step definition of the output from p. The cost, within p, of each placement location for v when returned from q depends on where p is expected to return the value; i.e., on parent call-sites to p. This information is not available without a global analysis similar to the one we are describing in this paper. Therefore, if v is not used, in the former sense, in the portion of p following its defining call to q, the material in this section does not apply; some of the associated costs for the latter sense will be captured in the definition-point considerations in the next section.

If there are multiple uses of v in the body of p, the cost computation considers the first use. This means that if there are multiple uses of v, the costs associated with uses after the first are not taken into account (this may happen, for example, if the later uses require loads from memory into a register).

The actions of the first phase of Pass 1, which considers the costs of different placements for the return values of a procedure based on how those values will be used, are shown in Figure 1, with various low-level cost parameters as described in Table 1. First, if a return value is to be placed in memory and the parameter

```
begin /* Compute costs for procedure q based on uses of q's return values */
  for each call site C for q do
    for each return value v of q do
      Cost_q(v, memory) := Cost_q(v, memory) + freq(C) × rload;
      if v is not an output of the procedure p in which C occurs then
        Cost_q(v, memory) := Cost_q(v, memory) + freq(C) × initmem;
      fi
      for each register r do
        if there is a call to a procedure s between C and a subsequent use of v
            such that s is not r-preserving then
          Cost_q(v, r) := Cost_q(v, r) + freq(C) × (rstore + rload);
        fi
        if v must be passed in register r' as an argument to a call and r ≠ r' then
          Cost_q(v, r) := Cost_q(v, r) + freq(C) × rmove;
        fi
      od
    od
  od
end
```

Figure 1. Pass 1 of Placement Algorithm: Use Point Considerations

is a local variable (will not be returned from p), then the caller needs to initialize, with cost initmem, the corresponding memory slot—this is necessary, for example, so that the garbage collector does not become confused.[3] If the parameter were returned from p, this is a *chained* definition, and the cost of initializing memory will be accounted for at the topmost call site. In either case, the value must be loaded into a register for the following use point, adding an additional rload to the cost of placing v in memory.

For each register that v may be returned in, the cost of using the register depends on the context of the succeeding use of v in p, and what happens between the definition and use points. There are two major cases:

1. If the register is not preserved by all intervening calls, the cost of using it is that of preserving the value across the destructive calls, and loading it again at the use point; i.e. rstore + rload.
2. If the register is preserved, there are again two cases:
   (a) If the use is in a procedure call and the register is not the one in which the call expects the corresponding parameter, we must move the value to the proper register at the use call site, incurring cost rmove.
   (b) Otherwise (the use is in the right register, or is an expression which is accepting of any register), no additional cost is incurred.

All these costs, for both registers and memory, are scaled by the frequency with which the clause is executed.

[3] Such initialization may not be necessary if the uninitialized memory cells can be recognized as such, e.g., by using a special tag on pointers to such cells [4]. It is straightforward to modify our algorithm to account for this.

```
begin /* Compute costs for procedure p based on definitions of p's return values */
  for each return value v of p do
    (i) Compute the cost of communicating return locations to definition points :
      if there is a procedure call between the entry to p and any definition point
        for v which is not r preserving for the register r in which the memory
        pointer for v would be passed then
        Cost_p(v, memory) := Cost_p(v, memory) + freq(p) × (rstore + rload);
      fi
    (ii) Compute the cost of placing the return value in the return location :
      if the definition for v is through a primitive action then
        Cost_p(v, memory) := Cost_p(v, memory) + freq(p) × rstore;
      fi
      for each register r do
        if there is a call to a procedure q between a definition point of v and
            the return from p such that q is not r-preserving then
          Cost_p(v, r) := Cost_p(v, r) + freq(p) × (rstore + rload);
        fi
      od
  od
end
```

Figure 2. Pass 1 of Placement Algorithm: Definition Point Considerations

4.1.2. Cost Considerations at Definition Point The actions of the second phase of Pass 1, which considers the costs of different placements for the return values of a procedure based on where those values are defined, are shown in Figure 2. There are two distinct costs we need to consider at the point where a return value is defined: first, that of communicating the location where the return value is to be placed; and second, that of actually placing the return value into this location.

First, consider the cost of communicating the return location to the definition point. If a return value v of a procedure p is to be returned in memory, one of the inputs to p must be a pointer to the memory location where it should be returned. Given our assumption that arguments are passed in registers, this pointer will be passed in some register r. If there is any procedure call that is not r-preserving between the entry to p and the point(s) where v is defined, this pointer must be saved across such calls, then loaded into a register at v's definition point to permit an indirect store. In this case, therefore, a memory return costs an additional $\mathsf{rstore} + \mathsf{rload}$. Otherwise, if v is to be returned in a register, or it is to be returned in memory but there is no need to save and restore the address of the corresponding memory location, this cost is 0.

Next, consider the cost of placing the return value into the location where it is to be returned. If the value v is defined by a primitive operation, we have the following cases:

1. v is returned in memory. In this case the cost is that of storing a value into memory, i.e. rstore.[4]

[4]Depending on the language, additional costs may be incurred for this case: for example, in logic programming languages it is necessary to deal with the possibility of pointer chains created via unification, which requires an additional dereference operation.

2. v is returned in a register r. We have the following sub-cases:
   (a) If there is a procedure call that is not r-preserving between the point at which v is defined and the point(s) at which control returns from p, the value of v must be saved across the procedure call and reloaded, potentially deoptimizing a tail call. Since the costs associated specifically with tail call deoptimization are accounted for elsewhere, this case incurs cost $\mathsf{rstore} + \mathsf{rload}$.
   (b) If there is no such call, the local cost of using r is 0.

If the value of v is computed and returned by a call to some other procedure q, memory returns incur no cost within the body of p. However, the cost of register returns depend on where q returns the value. Since this information isn't available yet, we do nothing in this case, adding in what costs we can in the final pass where some of the callee return locations will have already been assigned.

4.2. Pass 2: Choosing Output Locations

At the end of the first pass, we have determined output placement costs that are independent of particular output value placements of different procedures. We next visit each procedure in turn, and assign to each of its outputs the location that yields the smallest incremental cost to the program as a whole.

As noted previously, fixing the locations for the return values of one procedure can affect the optimal choice for another (e.g., in tail calls). One way to avoid the difficulties that arise from this would be to use an iterative approach, going back to reconsider previous decisions when an assignment that might affect them is made. It is not obvious that such iteration will reach a fixpoint. We have opted instead for a greedy approach that processes procedures, and potential tail calls within a procedure body, in order of decreasing execution frequency. For each procedure, we first determine, for each of its return values, which placements of that value would cause a loss of tail call optimization in a potential tail call in its body. After this, we factor in the additional costs associated with any such possible loss of tail call optimization. Finally, we examine the cost vectors and choose the placements for its return values.

As mentioned in Section 4.1, each potential tail call in a procedure is associated with a collection of flags—one flag for each possible placement of each output value of that procedure—whose purpose is to indicate whether or not that particular placement will prevent tail call optimization at that call site. Initially, these flags are optimistically set to indicate that tail call optimization is possible. In pass 2, we first set these flags for each procedure by examining each potential tail call in its body. Assume that we are processing a procedure p, and consider a potential tail call C in its body to a procedure q. For each output value v of p, we have the following cases:

1. v is defined at a program point preceding C. If v is returned in a register r and q is r-preserving, then this placement of v does not cause a loss of tail call optimization for this call. However, if q is not r-preserving, it is necessary to load the value of v into a register after control returns from the call, and this precludes tail call optimization.

```
begin
  for each procedure p, in decreasing order of execution frequency, do
    1. Update the cost vector for p to account for tail call deoptimizations:
      for each potential tail call C in p, in decreasing order of frequency, do
        (i) Compute C.tc_deopt(v,l) for each return value v and return location l;
        (ii) Estimate costs associated with tail call deoptimization:
          if C.tc_optimizable then
            for each return value v and return location l do
              if C.tc_deopt(v,l) then
                Cost_p(v,l) := Cost_p(v,l) + freq(C) × tcdeopt_cost(C);
              fi
              for each potential tail call D to p from a procedure r whose
                  return value placements have already been determined do
                if placing v in l causes D to lose tail call optimization then
                  Cost_p(v,l) := Cost_p(v,l) + freq(D) × tcdeopt_cost(D);
                fi
              od
            od
          fi
      od
    2. Choose return value placements for p:
      while there are return values of p with unassigned placements do
        let v be an unassigned return value of p, and l a location, such that
          Cost_p(v,l) ≤ Cost_p(v,l') for every return location l', and
          Cost_p(v,l) ≥ min_l' Cost_p(w,l') for every unassigned return value w of p;
        assign return location l to return value v;
        for each potential tail call D to p do
          if placing v in l causes D to lose tail call optimization then
            D.tc_optimizable := FALSE;
          fi
        od
      od
  od
end
```

Figure 3. Pass 2 of the Placement Algorithm

For an output value defined before the potential tail call, therefore, for each register r such that q is not r-preserving, the flag $C.tc\_deopt(v, r)$, corresponding to a placement of v in register r, is tagged as preventing tail call optimization.

2. v is defined by q. We have two sub-cases:

   (a) The output placements for q have been determined already. Suppose q returns the value v in location l_q. For each placement l_p in which p could return v, if $l_p \neq l_q$ it will be necessary to add code to move the value of v from l_q to l_p after control returns from q, and this will preclude tail call optimization. Therefore, for each return location l_p that does not match the location l_q in which v is returned by q, the flag $C.tc\_deopt(v, l_p)$ is set to indicate that tail call optimization is prevented.

   (b) q has not yet had its output placements determined. In this case we have no way of telling which locations will eventually cause loss of tail call optimization, so the flags are left unmodified, i.e., indicate that tail call optimization may still be possible. When we subsequently process q, any placement of q's outputs that would cause a loss of tail call optimization here (in p) will be noted, and the corresponding cost accounted for in the cost vector of q.

Once the flags indicating placements that cause loss of tail call optimization have been set, we compute, for each output value, the cost associated with different possible placements for it. Initially, this cost is set to the cost computed for that placement in the first pass. After this, the costs associated with tail call deoptimization are factored in. In general, the cost associated with a loss of tail call optimization can depend on whether this causes an environment to be allocated. If the caller would have allocated an environment anyhow, the additional cost associated with a loss of tail call optimization is that of a procedure call and return, and is given by call_tn; otherwise, there is also a cost associated with the allocation of an environment, with total cost call_ta. As a first approximation, our implementation currently takes these values to be system-dependent constants; however, it is not difficult to see how this could be extended to make finer evaluations, e.g., by taking into account the number of environment locations that have to be initialized when an environment is allocated, or the number of values that have to be saved in the environment as a result of a loss of tail call optimization. To this end, we express the cost associated with a loss of tail call optimization at a potential tail call C as $tcdeopt\_cost(C)$:

$$tcdeopt\_cost(C) = \begin{cases} \mathsf{call\_ta} & \text{if tail call deoptimization of } C \text{ causes} \\ & \text{an environment to be allocated} \\ \mathsf{call\_tn} & \text{otherwise} \end{cases}$$

Tail call deoptimization costs for various return value placements are counted as follows: for each potential tail call C for which the flag $C.tc\_optimizable$ is true,

1. each placement that prevents tail call optimization gets an additional cost of $tcdeopt\_cost(C)$, weighted by the execution frequency of C; and

2. for each potential tail call D from a procedure whose placements have been decided already, each placement that would force a loss of tail call optimiza-

tion for D incurs, in the cost vector of C, the cost $tcdeopt\_cost(D)$, weighted by the execution frequency of D.

Finally, once the cost vector of different output placements for each potential tail call in a procedure has been computed, we are in a position to choose placements for the output arguments of that procedure. In general, a procedure may have more than one return value, each with a choice of return locations. The assignments for these values can interfere with each other. For example, suppose a procedure has two return values, x and y: it may happen that an assignment of a particular register to x incurs a small savings, but prevents the use of that register for y, thereby incurring a much higher cost for the next best choice for y. To lessen the effects of such interference, we look for the output value whose minimum cost location is the most expensive amongst all minimum cost output placements: assigning any other output's location will certainly not decrease this output's minimum cost, and may well increase it if the assignment prevents the corresponding location from being chosen when this output is finally assigned.[5] This assignment is then set. Any potential tail call D that is forced to give up tail call optimization because of this placement decision has the flag $D.tc\_optimizable$ set to false, so that the costs associated with losing tail call optimization are not charged multiple times. The search and assignment is repeated until all outputs have an assigned location. The actions for this phase are summarized in Figure 3.

4.3. Interactions with Garbage Collection

The system in whose context our ideas have been implemented, and for which we present performance numbers (see Section 6), is a logic programming system; our results suggests that these ideas can improve the performance of implementations of logic programming languages. At the low implementation level addressed by this paper, similar aspects of different languages—e.g., tail calls—are often implemented in essentially similar ways, and for this reason it is possible that these ideas may also be useful for functional language implementations.[6] Because of the way lazy data constructors work, lazy recursive data construction functions already share many of the advantages that memory returns, as described in this paper, give. For strict functional languages, the optimization described in this paper can, in principle, adversely affect garbage collection. Strict, mostly pure functional languages such as Standard ML have the property that pointers are almost always from younger objects to older objects, and implementations may take advantage of this in several ways, e.g., by doing generational garbage collection. However, our transformation would cause this property to be violated frequently, and this could offset some or all of the savings resulting from our optimization, especially when the garbage collector employs a write barrier. Fortunately, this turns out not to be a problem in practice, since the issue can be addressed using a technique developed by Cheng and Okasaki [7], who maintain a list of pointers into data structures that have survived

[5] In the case of ties between different placements, our implementation chooses memory over registers of the same cost, because memory will less often destroy a tail call opportunity. However, in an implementation where the default output value placement is in registers, one could just as well consider choosing the default register placement in the case of ties.

[6] We are indebted to an anonymous referee for these observations about functional language implementations.

one of more garbage collections, and process this list during every collection. They observe that only writes through such pointers can cause inter-generational references, and the number of such pointers (typically only one or two per collection) is usually very small compared to the total number of writes, making this approach faster than using write barriers even when the cost per surviving pointer is higher than the cost of a write barrier per write. Using an implementation based on TIL [16], an optimizing compiler for Standard ML, Cheng and Okasaki show that using an approach similar to that proposed here can lead to significant improvements in running time, both when garbage collection time is included in the measurements, and when it is not; they also show that in many cases, such an approach leads to improvements in the total time required for garbage collection. A similar interaction with generational garbage collection also arises in lazy functional languages because of the way closures are updated with values once they get evaluated; techniques proposed to address this problem for lazy languages may be applicable to our optimization as well (see, for example, [14]).

4.4. Complexity

Assuming that p is an upper bound on the number of output arguments of any procedure in the program, the complexity of the first pass is $O(p(S+C))$ and that of the second pass is $O((1+p^2)(S+C))$, where S is the number of call sites in the program and C is the number of potential tail calls. Hence, the algorithm is essentially linear in the size of the program.

5. TRANSFORMING THE PROGRAMS

If we resort to a heterogeneous return placement policy, procedure call interfaces can no longer rely on a simple uniform placement policy. Additional work may be needed, e.g., to pass the address of a memory location where a return value is to be placed instead of relying on a default policy where values are returned in registers, or to retrieve a return value from a register instead of relying on a default policy of having return values placed in memory. This may affect tail calls as well, as discussed earlier. Thus, once placements have been determined for all of the return values of each procedure in a program, we have to transform the code to deal consistently with different return placements. This section examines the transformations necessary for the two most commonly encountered default return placement policies: register returns, employed in most functional language implementations; and memory returns, employed in most logic programming systems.

5.1. Default Register Returns

If values are returned in registers by default, functions that are to return some values in memory need to be changed to take additional arguments that point to the locations where these values are to be placed, and calls to such functions have to be modified to supply pointers to appropriate memory locations. It may be possible, in addition, to take advantage of tail call optimization by reordering computations in some situations. In situations where this happens, a function that would have needed to allocate an environment before the transformation may, after transformation, no longer need to do so (see Example 5.1 below).

First, for each function that returns at least one value in memory, we fix an ordering among the memory-placed return values of the function. For each function f, the transformation then proceeds as follows:

1. If f returns k values in memory, add k new formal parameters $z_1, \ldots, z_k$ that are pointers to values of the appropriate type. The order of these new formals is determined by the ordering previously determined for the memory return values of f.

2. At each return point in the body of f, insert code such that the i^{th} memory return value is assigned into the location pointed to by z_i.

3. For each call C in the body of f, where the called function g returns m values in memory, where $m > 0$, do:

   (a) If C is followed by an operation S that is a non-strict data construction operation, such that any value that is used by S and defined by C is returned by the callee g in memory, reorder the computations so that S precedes C. For each field of S that is defined by C, insert code immediately after S to initialize that field appropriately. Repeat this step if there are multiple such operations.

   (b) If, as a result of this reordering step, there are no more computations following C, mark C as tail recursive.

   (c) Define m addresses $addr_1, \ldots, addr_m$ as follows:

      (i) If the i^{th} value that C returns in memory happens to be the j^{th} value f returns in memory, then $addr_i \equiv z_j$. In other words, we pass the pointer to the return location, which was passed to f as the parameter z_j, into g.

      (ii) If the i^{th} value returned by C in memory is used as a field in a data structure whose construction was moved before C in the previous step, then $addr_i$ is the address of that field.

      (iii) Otherwise, $addr_i$ is the address of a new variable w_i that is introduced to hold the i^{th} memory return value of g.

   Pass these addresses $addr_1, \ldots, addr_m$ as additional arguments to C, in the order determined by the ordering that has been determined for g's memory return values.

   (d) For each value that is returned in memory by C, modify any subsequent references to that value as necessary to refer to the appropriate memory location.

4. If one or more calls in the body of f became tail calls as a result of this transformation, it is possible that f no longer needs to allocate an environment. Examine the transformed definition of f to determine whether this is so, and update the appropriate information if necessary.

Example 5.1. Consider the function `ldbl` discussed in Section 2. Assume that it has been transformed into an appropriate internal representation, such as an abstract syntax tree, that might be rendered as in Figure 1(a). Suppose we have

```
proc ldbl(x) :-
  local u, v;

  if (null? x)
    return nil
  else
    u := 2 * (car x);
    v := (ldbl (cdr x) );
    return cons(u, v);
  fi
end
```

(*a*) Before transformation

```
proc ldbl(x, z)
  local u;

  if (null? x)
    @z := nil
  else
    u := 2 * (car x);
    @z := cons(u, v);
    return ldbl((cdr x), &v);
  fi
end
```

(*b*) After transformation

Figure 1. Example transformation for default register returns

decided that `ldbl` should return its result in memory. After the transformation described above, we get the program of Figure 1(*b*). The transformed version is tail recursive, and can be executed without allocating an environment. ■

5.2. Default Memory Returns

In the case where the default policy is to place return values in memory, procedures that return some values in registers have to be modified to load these values into registers before returning. This may compromise tail call optimization in some cases, and possibly require the allocation of an environment. Finally, if a procedure that returns some of its output values in registers, there is no need to pass it pointers to the memory locations where those values would have to be placed.

Each procedure p in the program is transformed as follows:

1. For each return value v of p that is returned in a register r, insert additional code to ensure that v is returned in r:
   (a) For each v-defining tail call C in the body of p, suppose that the called procedure places v in location s (which may be a register or memory). If $r \neq s$, add code after C to move v from s to r.
   If tail call optimization becomes blocked as a result of this code insertion, mark C as not amenable to tail call optimization.
   (b) For each return point D in the body of p that is not a tail call, insert code to move v into register r (if code can be generated to place v in r directly, this move can be optimized away later).
2. For each call C in the body of p:
   (a) For each return value v of C that is returned in a register, any subsequent use of v should refer to the appropriate register rather than memory.
   (b) If a value returned in a register needs to be preserved across a procedure call, insert code to save its value before any such call and restore it before any use.

```
proc fact(N, F) :-
   local N1, F1;

   if (N = 0)
      @F := 1;
      return;
   else
      if (N > 0)
         N1 := N-1;
         call fact(N1, &F1);
         @F := N*F1;
         return;
      fi
   fi
end
```

(b) Before transformation

```
proc fact(N) :-
   local N1, F1;

   if (N = 0)
      r1 := 1;
      return;
   else
      if (N > 0)
         N1 := N-1;
         call fact(N1);
         r1 := N*r1;
         return;
      fi
   fi
end
```

(b) After transformation

Figure 2. Example transformation for default memory returns

(c) If the called procedure passes some of its return values in registers, there is no need to pass, as arguments to the call, pointers to memory locations corresponding to such return values.

3. For each return value of p that is returned in a register, delete the corresponding formal parameter that is a pointer to the memory location where that value should be assigned.

Example 5.2. Consider the factorial program of Example 3.3, whose abstract syntax tree might be as shown in Figure 2(a). Suppose that we decide to place the return value `F` of this procedure in register `r1`. After transformation, we get the definition of Figure 2(b).

It is not difficult, during subsequent processing, to notice that this procedure does not need to allocate any memory for either of the local variables `N1` and `F1`. ■

6. EXPERIMENTAL RESULTS

To test the efficacy of our algorithm, we tested it on a number of benchmark programs, broadly classified into three groups: *simple loops*, which perform simple iterative computations and where the choice of output placement does not make a significant difference to performance; *scalar computations*, where the preferred output placements are in registers; and *list computations*, where the preferred placements are in memory. The benchmarks were small to medium sized programs, to make it possible to isolate the effects of different output placements and allow them to be compared in a reasonable way. The system used for these experiments was `jc`, a sequential implementation of a committed-choice logic programming language where procedure bodies are executed from left to right as in Prolog. The `jc` compiler translates Janus programs to C and then uses a C compiler (the performance numbers in

Program	Execution time (μsec)			Speedups		
	Mem	Reg	Het	Reg/Mem	Het/Mem	Het/Reg
bessel	11031	11119	11236	1.008	1.019	1.010
muldiv	12489	12466	12487	0.998	1.000	1.002
nand	4613	4612	4612	1.000	1.000	1.000
pi	11960	12144	12151	1.016	1.016	1.000
sum	1692	1693	1692	1.015	1.000	1.000
Geometric Mean:				1.007	1.007	1.002

Key :

Mem:	memory returns only;
Reg:	register returns only;
Het:	heterogeneous memory+register returns

Table 1. Performance Results: Simple Loops

this paper correspond to `gcc` 2.6.3 invoked with `-O2 -fomit-frame-pointer`) to compile the resulting program to executable code (the current system uses heuristics based on the structure of the program to estimate execution frequencies [20]: a detailed discussion of the heuristics used appears in [5]; in principle, this information could also be obtained using profile information obtained from "training runs" of the programs, but we have not implemented this yet). An early version of the system is described in [10], and a prototype of the system as well as the code for the benchmarks is available by anonymous FTP from `ftp.cs.arizona.edu`.

The programs were run on a 40 MHz Sun SPARC IPC, with 32 Mbytes of physical memory and 64 Kbytes of combined instruction and data cache, running Solaris 2.3. Execution times were obtained using the *gettimeofday*(2) system call to obtain microsecond-resolution measurements of execution time, with the testing being the only active process. For each benchmark program, a single "run" consisted of executing a test query one hundred times for each output placement policy and, in each case, taking the shortest measured query execution time. Queries were designed to be large enough to exercise the programs, yet small enough to able to execute in a single timeslice with no system interruptions; taking the minimum measurement avoids bias when one or more query runs nonetheless happened to be interrupted. A single experiment consisted of a single run of each benchmark program, with the different benchmarks executed in random order within each experiment so as to avoid systemic bias from disk and memory cache effects. Nine such experiments were performed, and for each benchmark the median execution time for each execution policy was taken.

The performance results for the various classes of benchmarks are as follows:

Simple Loops: These are simple computations where a value is computed iteratively and returned at the end of the loop. The benchmarks used were the following:

- *bessel* computes the Bessel function $J_{75}(3)$, and evaluates both integer (for factorial) and floating point (for exponentiation) expressions;
- *muldiv* exercises integer multiplication and division, doing 5000 of each;

Program	Execution time (μsec)			Speedups		
	Mem	Reg	Het	Reg/Mem	Het/Mem	Het/Reg
aquad	28190	20368	20358	0.722	0.722	0.999
binomial	5747	5543	5538	0.964	0.964	0.999
chebyshev	8894	11422	8894	1.284	1.000	0.779
fib	11073	4453	4483	0.402	0.405	1.007
log	15745	16582	16595	1.053	1.053	1.001
mandelbrot	24249	23752	23758	0.979	0.978	1.000
mcint	16642	15977	15977	0.960	0.960	1.000
tak	13457	5344	5343	0.397	0.397	1.000
zeta	18116	18808	18864	1.038	1.041	1.003
Geometric Mean:				0.808	0.786	0.973

Key :

Mem:	memory returns only;
Reg:	register returns only;
Het:	heterogeneous memory+register returns

Table 2. Performance Results: Scalar Computations

- *nand* is an electrical circuit design program, taken from [17];
- *pi* computes the value of π to a precision of 10^{-3} using the expansion $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$;
- *sum* adds the integers from 1 to 10,000—it is essentially similar to a tail-recursive factorial computation, except that it can perform a much greater number of iterations before incurring an arithmetic overflow.

Since most of the computation in these programs is performed in loop bodies, with output values returned only at the end, one would expect that in this case there should not be a significant difference between memory placements and register placements. The benchmark results, given in Table 1, verify that this is indeed the case: the differences between the various placement policies is less than 1% on the average.

Scalar Computations: These are computations of scalar values. The programs are more complex than for the simple loops considered above, and some involve extensive floating point computations. Because of this, the preferred locations for the placement of return values are registers. The benchmarks used were the following:

- *aquad* performs a trapezoidal numerical integration $\int_0^1 e^x dx$ using adaptive quadrature;
- *binomial* computes the binomial expansion $\sum_{i=0}^{30} x^i y^{30-i}$ at $x = 2.0,\ y = 1.0$;
- *chebyshev* computes the Chebyshev polynomial of degree 10000 at 1.0;
- *fib* computes the Fibonacci value $F(16)$;
- *log* computes $\log_e(1.999)$ using the expansion $\log_e(1+x) = \sum_{i \geq 0} (-1)^{i+1} x^i / i$;

Program	Execution time (μsec)			Speedups		
	Mem	Reg	Het	Reg/Mem	Het/Mem	Het/Reg
bsort	16422	32708	16512	1.992	1.005	0.505
disj	29643	30369	30375	1.025	1.025	1.000
fft	30119	42053	31401	1.396	1.043	0.747
hanoi	15302	15441	15429	1.009	1.008	0.999
lr1gen	22446	24844	22238	1.107	0.991	0.895
matmult	29873	32876	30198	1.100	1.011	0.918
nrev	6985	21582	7059	3.128	1.011	0.327
pascal	9122	15255	9065	1.672	0.994	0.594
prime	10714	17171	10716	1.603	1.000	0.624
qsort	11347	26631	11255	2.347	0.992	0.428
queen	6563	8195	6563	1.249	1.000	0.801
Geometric Mean:				1.499	1.007	0.649

Key :

Mem: memory returns only;
Reg: register returns only;
Het: heterogeneous memory+register returns

Table 3. Performance Results: Structure Computations

- *mandelbrot* computes the Mandelbrot set on a 17×17 grid on an area of the complex plane from $(-1.5, -1.5)$ to $(1.5, 1.5)$;
- *mcint* uses Monte Carlo integration to estimate the mass of a body of irregular shape, adapted from [13];
- *tak*, from the Gabriel benchmarks, is a heavily recursive program involving integer addition and subtraction;
- *zeta* computes the Euler-Riemann zeta function, defined by the series $zeta(x) = 1 + 2^{-x} + 3^{-x} + \cdots$ (where x is real-valued), at $x = 2.0$;

In this case, a homogeneous memory placement policy is, as expected, considerably slower than a register placement policy. It turns out, as shown in Table 2, that returning outputs in memory is, on the average, about 20% slower than returning outputs in registers.

Structure Computations: These programs involve a significant amount of recursive data structure computation. The benchmarks used were the following:

- *bsort* is a bubble sort program on a list of 100 integers;
- *disj* converts a propositional formula to disjunctive normal form;
- *fft* is an iterative one-dimensional fast Fourier transform, adapted from [13]. The program computes the fast Fourier transformation and its inverse on a vector of size 64;
- *hanoi* is the Towers of Hanoi program: the numbers given are for *hanoi*(10);

Program	Execution Time (μsecs)				Relative performance		
	J	gcc:2	cc:2	cc:4	J/gcc:2	J/cc:2	J/cc:4
aquad	20569	16604	28883	26433	1.238	0.712	1.119
bessel	12364	12644	20635	20123	0.978	0.599	0.614
binomial	5720	5075	8894	6098	1.127	0.643	0.938
chebyshev	8500	7207	18067	18065	1.179	0.470	0.470
fib	4711	4727	4598	4584	0.997	1.025	1.028
log	17198	17487	35029	35029	0.984	0.491	0.491
mandelbrot	23942	19403	78423	46195	1.234	0.305	0.518
muldiv	12705	10605	11688	11669	1.193	1.087	1.089
pi	12144	11998	22528	22520	1.012	0.529	0.529
sum	1694	1606	1606	406	1.055	1.055	4.172
tak	5340	4384	4085	4070	1.218	1.298	1.303
zeta	18864	18029	38962	38792	1.046	0.484	0.486
Geometric Mean :					1.100	0.665	0.838

Key : J : `jc -O`; `gcc:2` : `gcc -O2`; `cc:2` : `cc -O2`; `cc:4` : `cc -O4`

Table 4. The speed of `jc` compared to optimized C

- *lr1gen* is the core of an LR(1) parser generator;
- *matmult* is an integer matrix multiplication program;
- *nrev* is an $O(n^2)$ naive reverse program on an input list of length 100;
- *pascal* is a benchmark, by E. Tick, to compute Pascal's triangle;
- *prime* computes prime numbers up to 200 using the Sieve of Eratosthenes;
- *qsort* is a quicksort program (see Example 3.1), executed on a list of length 100;
- *queen* is the n-queens program: the numbers given are for 6 queens.

The data structures manipulated, in most cases, were lists: the only exceptions were the *disj* program, which used nested arrays, and *fft*, which implemented updatable arrays using binary trees. For these programs, the loss in tail call optimization resulting from a homogeneous register return policy leads to a significant loss of performance. Because of this, as shown in Table 3, returning values in registers turns out to be about 50% slower than a homogeneous memory return policy.

Table 4 compares the execution speed of our system with optimized C code, written in a "natural" C style wherever possible (i.e., using iteration instead of recursion, and with destructive update.) It can be seen that the baseline performance of our system—with the default homogeneous memory placement policy—is reasonably good: it is, on the average, only about 20% slower than C code compiled with `gcc -O2`. It is easy to take a poorly engineered system with a lot of inefficiencies and get huge performance improvements by eliminating some of these inefficiencies. The point of these numbers, when evaluating the efficacy of our optimizations, is that we were careful to begin with a system with good performance so as to avoid drawing overly optimistic conclusions.

It is clear from these results that homogeneous output placement policies—i.e., where return values are returned in either always in registers, or always in memory—perform well on some programs but poorly on others. For example, for scalar computations the homogeneous memory placement policy commonly used in logic programming systems is considerably slower than a policy where outputs are always returned in registers. Much the opposite is true for list computations: the homogeneous return policy commonly used in implementations of functional languages is very often much slower than a homogeneous memory return policy. This supports our claim that for best performance, it is necessary to use a heterogeneous output placement policy that is able to choose between registers and memory in a flexible manner depending on their relative costs and benefits.

Further, for either group of benchmarks, it can be seen that our algorithm generally chooses the output placement method one intuitively expects. In particular, even though our algorithm may occasionally choose to give up tail call optimization in favor of a cheap placement for the output values of a procedure, no program has significantly worse performance using our algorithm than with the best placement. Overall, for scalar computations we find that on the average, the code generated using our algorithm for output placement is about 22% faster than that resulting from a homogeneous memory placement, and very slightly faster than that obtained using a homogeneous register placement policy (Table 2 shows it to be about 2.6% faster for this class of programs, but this is due almost entirely to a single benchmark: if the *chebyshev* program is ignored, the two policies produce essentially identical performance). This performance improvement is due primarily to two reasons: first, a reduction in the number of memory references due to placing values in registers; and second, the ability to maintain values in unboxed form in registers in situations where writing them to memory would have required boxing them. For list computations, the code produced using our algorithm is, on the average, about 35% faster than that resulting from a homogeneous register placement, and almost identical in performance to code obtained using a uniform memory placement. The performance gain in this case is due almost entirely to the fact that memory placements allow the use of tail call optimizations in some situations where register placements would not.

7. RELATED WORK

The work most closely related to this is the output placement algorithm described by Van Roy [18] and used in the Aquarius Prolog compiler, and the "destination passing style" described by Larus [12].

Van Roy's scheme is heterogeneous, i.e., can choose between register and memory placements. When register returns are chosen, it uses a fixed positional mapping to determine which register an output value should be returned in. It also does not take into account relative execution frequencies, and does not consider relative costs of losing a tail call optimization versus storing values into memory. For these reasons, the output placements obtained using Van Roy's algorithm are generally not as good as those obtained using our algorithm.

Larus's destination passing style is very similar to our approach to turning potential tail calls that are followed by a set of memory assignments into proper tail calls by passing addresses of memory locations into the call. However, it is motivated by very different considerations, namely, increasing the amount of parallelism in Lisp programs by removing certain kinds of dependencies. Because of this, the cost/benefit criteria relevant to Larus's work are very different from ours: whereas we are concerned with the savings in time (and, indirectly, space) accruing from tail call optimization in a sequential context, and the costs associated with returning values in memory, Larus is concerned primarily with the amount of parallelism that can be extracted from programs. Because of this, Larus's transformation is defined solely with respect to tail recursive functions, rather than tail calls in general: the transformation discussed in Section 5.1 can be seen as a straightforward generalization of that defined by Larus. Another direct consequence of this difference in motivation is that Larus's work does not rely on a cost model to evaluate tradeoffs and determine whether or not destination passing style is desirable in a particular context, nor does it empirically investigate the effects, on sequential performance, of returning values in registers or in memory. An idea similar to destination passing style, though motivated by different concerns—namely, the elimination of intermediate lists in applicative programs—and somewhat more restricted in scope, is described by Wadler, who refers to it as *tail recursion modulo cons* [19].

8. CONCLUSIONS

Most implementations of functional and logic programming languages take a fixed approach to how values computed by procedures are returned: return values are usually placed either always in registers, or always in memory. Neither of these choices is uniformly desirable: they are good in some situations, and not so good in others. The reason is that register placements can be accessed without any memory operations, but can sometimes compromise tail call optimization; on the other hand, memory placements do not interfere with tail call optimization, but are more expensive in terms of memory accesses.

This paper gives an algorithm for return value placement that attempts to attain the best of both worlds. It uses cost estimates for various placement alternatives, weighted by execution frequency estimates, to determine a "good" output location assignment for each procedure in a program. Our experiments indicate that it usually makes the right decisions: in situations where outputs are best returned in registers, it chooses register returns, while in situations where memory returns are better, it typically chooses memory placements. Overall, this results in significant speed improvements compared to traditional fixed output placement schemes.

REFERENCES

1. A. W. Appel, *Compiling with Continuations*, Cambridge University Press, 1992.

2. A. W. Appel and Z. Shao, "Callee-save Registers in Continuation-passing Style", *Lisp and Symbolic Computation* (5) 191–221, 1992.

3. J. M. Ashley and R. K. Dybvig, "An Efficient Implementation of Multiple Return Values in Scheme", *Proc. ACM Conference on Lisp and Functional Programming*, 1994, pp. 140–149.

4. J. Beer, "The Occur-Check Problem Revisited", *J. Logic Programming* vol. 5 no. 3, Sept. 1988, pp. 243–261.

5. P. A. Bigot, D. Gudeman, and S. K. Debray, "Output Value Placement in Moded Logic Programs", Technical Report 94-03, Department of Computer Science, The University of Arizona, Tucson, Jan. 1994.

6. R. A. Brooks, R. P. Gabriel, and G. L. Steele, Jr., "S-1 Common Lisp Implementation", *Proc. ACM Symp. on Lisp and Functional Programming*, Pittsburgh, PA, Aug. 1982, pp. 108–113.

7. P. Cheng and C. Okasaki, "Destination-Passing Style and Generational Garbage Collection", unpublished manuscript, School of Computer Science, Carnegie Mellon University, Pittsburgh, Nov. 1996.

8. W. D. Clinger and L. T. Hansen, "Lambda, the Ultimate Label, or A Simple Optimizing Compiler for Scheme", *Proc. ACM Conference on Lisp and Functional Programming*, 1994, pp. 128–139.

9. S. K. Debray, D. Gudeman and P. A. Bigot, "Detection and Optimization of Suspension-free Logic Programs", *Journal of Logic Programming* (Special Issue on High Performance Implementations), vol. 29 nos. 1–3, Nov. 1996, pp. 171–194.

10. D. Gudeman, K. De Bosschere, and S.K. Debray, "`jc`: An Efficient and Portable Sequential Implementation of Janus", *Proc. Joint International Conference and Symposium on Logic Programming*, Washington DC, Nov. 1992, pp. 399–413. MIT Press.

11. D. Krantz, *ORBIT: An Optimizing Compiler for Scheme*, Ph.D. Dissertation, Yale University, 1988. (Also available as Technical Report YALEU/DCS/RR-632, Dept. of Computer Science, Yale University, Feb. 1988.)

12. J. R. Larus, *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors*, Ph.D. Dissertation, University of California, Berkeley, 1989. Also available as Technical Report UCB/CSD 89/502, Computer Science Division (EECS), University of California, Berkeley, May 1989.

13. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C*, Cambridge University Press, 1992.

14. N. Röjemo, "Generational Garbage Collection for Lazy Functional Languages with Temporary Space Leaks", *Proc. International Workshop on Memory Management*, 1995. Springer Verlag.

15. P. A. Steenkiste and J. L. Hennessy, "A Simple Interprocedural Register Allocation Algorithm and its Effectiveness for Lisp", *ACM Transactions on Programming Languages and Systems*, vol. 11 no. 1, Jan. 1989, pp. 1–32.

16. D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee, "TIL: A type-directed optimizing compiler for ML", *Proc. SIGPLAN '96 Conference on Programming Language Design and Implementation.* ACM, New York, pp. 181–192.

17. E. Tick, *Parallel Logic Programming*, MIT Press, 1991.

18. P. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, University of California at Berkeley, 1990.

19. P. Wadler, "Listlessness is Better than Laziness: Lazy evaluation and garbage collection at compile-time", *Proc. ACM Symposium on Lisp and Functional Programming*, 1984, pp. 45–52.

20. D. W. Wall, “Predicting Program Behavior Using Real or Estimated Profiles”, *Proc. SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991, pp. 59–70.

21. D. H. D. Warren, “An Abstract Prolog Instruction Set”, Technical Note 309, SRI International, Menlo Park, CA, Oct. 1983.