Output Value Placement in Moded Logic Programs *

Peter A. Bigot David Gudeman Saumya Debray
Department of Computer Science
University of Arizona
Tucson, AZ 85721, USA
{pab, gudeman, debray}@CS.Arizona.EDU

Technical Report # TR94–03

January 31, 1994

Abstract

Most implementations of logic programming languages treat input and output arguments to procedures in a fundamentally asymmetric way: input values are passed in registers, but output values are returned in memory. In some cases, placing the outputs in memory is useful to preserve the opportunity for tail call optimization. In other cases, this asymmetry can lead to a large number of unnecessary memory references and adversely affect performance. When input/output modes for arguments are known it is often possible to avoid much of this unnecessary memory traffic via a form of interprocedural register allocation. In this paper we discuss how this problem may be addressed by returning output values in registers where it seems profitable to do so. The techniques described have been implemented in the `jc` system, but are also applicable to other moded logic programming languages, such as Parlog, as well as languages like Prolog when input/output modes have been determined via dataflow analysis.

*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.

1 Introduction

In the Warren Abstract Machine (WAM), which is used as the basis for a large number of implementations of Prolog and other logic programming languages, there is a fundamental asymmetry in the treatment of the input and output values of a procedure: input values are passed in argument registers, while output values are returned in memory. The reason for this is clear in the context of Prolog, for which the WAM was originally designed: 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 into a procedure in registers, with each uninstantiated 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 an uninstantiated variable to a value, i.e., by writing to the corresponding memory location.

While this scheme is simple and works in general, it may incur unnecessary overheads. To see this, consider the following 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 (which must be initialized as an unbound variable), and a pointer to the slot is passed into the recursive call. When the call returns, the value of `F1` is retrieved from memory, 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. Indeed, since implementations of functional languages typically put the return value of a function in a (hardware) register (see, for example, [5, 13]), the behavior described above can be a source of performance disadvantage for logic programming languages compared to functional languages. Finally, since the number of procedure returns at runtime must be equal to the number of procedure calls, the benefits of reducing the cost of procedure returns, in particular via careful management of registers, can be significant.

It is obvious that knowledge about which arguments of a procedure are output arguments is necessary for this optimization to be carried out. For languages such as Parlog [6] this information is available from user annotations. For other logic programming languages, this information may be obtained via dataflow analysis (e.g., see [8] for mode analysis of Prolog programs, [21, 23] for mode analysis of FGHC programs). However, a number of other issues must also be addressed in order to carry out the kind of interprocedural register allocation discussed in the `fact`/2 example above, including how the merits of alternative output placements should be compared, and how output placement decisions interact with the ability to carry out tail call optimization. The remainder of the paper discusses these and related issues and describes how they have been addressed in the context of `jc`, a sequential implementation of a variant of Janus [12].[1] However, the techniques we have developed are not peculiar to Janus in any way, and are applicable also to other logic programming languages where mode information about procedure arguments is available. To minimize

[1] An alpha release of the system is available by anonymous ftp from `ftp.cs.arizona.edu`, file `janus/jc/jc-2.0.tar`.

syntactic hurdles for the reader, we will use a Prolog-like syntax for our examples, with input and output arguments of procedures indicated by mode declarations as in the `fact`/2 example above.

2 Output Value Placement: Implementation Considerations

There are a variety of implementation concerns which interact with output value placement. If an output placement algorithm is chosen naïvely, the overhead of extra work induced by unfortunate placements can overwhelm any benefits of returning values in registers. Some of those concerns are considered here.

2.1 Interactions with Tail Call Optimization

While program performance can be improved significantly by returning output values in registers rather than in memory, the situation is complicated by the interaction of this optimization with *tail call optimization* (TCO). This optimization collapses procedure returns when the last body goal is a call, immediately followed by a procedure return. By replacing this *tail call* with a jump to the called procedure and re-using the caller's frame, one avoids a redundant return and frame allocation. The interaction problem is illustrated by the following example.

Example 2.1 Consider a predicate `p`/3 defined by the following clause:

```
:- mode p(in, out, out).
p(X,Y,Z) :- q(0,X,Z,Y).
```

Suppose that `q`/4 has the mode `q(in,in,out,out)`, and that in a call `q(U,V,Z,Y)` the output `Z` is returned in register 1 while `Y` is returned in register 2. Suppose also that in a call `p(X,Y,Z)` the output `Y` is returned in register 2 and `Z` in register 3. Because the output placement for `q`/4 does not match that of `p`/3, additional code must be inserted in the clause above to move the output `Z` from register 1 (where it is returned by `q`/4) to register 3 (where it has to be returned by `p`/3), and the implementation of this clause has the form:

`p(X,Y,Z) :- q(0,X,Z,Y)`, ⟨*code to move* `Z` *from register* 1 *to register* 3⟩.

It is evident that the implementation of this clause cannot take advantage of tail call optimization on the goal `q(0,X,Y,Z)`. □

Notice that the problem illustrated in this example arises whenever the output placement of the head of a clause differs from that of a tail call in the clause: in the example above, the same problem would have arisen if `q`/4 had returned its outputs in registers while `p`/3 returned its outputs in memory in the traditional manner, or vice versa. The cost of this "de-optimization" must be weighed against the expected savings accruing from returning output values in registers when deciding whether or not the outputs of a procedure should be returned in registers.[2] In general, if output arguments are returned in registers, then it is impossible to avoid deoptimizing some tail calls in some clauses, regardless of what approach is taken for output register assignment and code generation. The problem is illustrated by the following example:

[2]However, the cost of this de-optimization may not be as bad as one might expect: our experimentally-determined cost model, summarized in Table 1, suggests that the cost of sacrificing a tail call optimization where an environment has already been allocated (C_{tn}) is between 15% less and 25% more than total time cost of returning a value in memory, including the cost of initialization, assignment, and loading the value ($C_p + C_a + C_l$). Losing a tail call where an environment must be allocated (C_{ta}) costs only 45% to 60% more than the cost of returning the value in memory.

Example 2.2 Consider a procedure defined by the clauses

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

It is not difficult to see that if either p/2 or q/2 returns either of its outputs in registers, at least one of the clauses defining p/2 will have to give up tail call optimization. □

2.2 Interactions with Procedure Suspension

Committed choice languages such as Parlog [6], GHC [22], and Janus [19] typically require that if a procedure activation cannot make progress because its inputs are not adequately instantiated, it should suspend until its inputs are available. Many modern Prolog systems also provide facilities whereby an activation can be made to suspend until certain of its arguments become instantiated. Since a procedure that suspends cannot return an output value, it is necessary to take such suspension behavior into account when considering whether to return output values in registers. As an example, consider the following clause:

```
p(X,Y) :- q(X,Z), Y is Z+X.
```

Suppose that the procedure q/2 has the mode `q(in, out)`, and we choose to return the output `Z` of the call `q(X,Z)` in a register r. If the execution of this call suspends, then it is necessary to ensure that r is set to a value that causes the computation `Y is Z+X` to suspend as well. Moreover, the suspension structure for the goal `Y is Z+X` has to be set up in such a way that this computation is resumed when the computation of `q(X,Z)` eventually computes a value for `Z`. This can complicate handling procedure suspension and resumption considerably. For example, it is possible to augment the code for q/2 so that on suspension, it creates an unbound variable on the heap and returns a reference to it in register r, but this complicates the implementation of assignment: it becomes necessary to distinguish between an activation that has not suspended (in which case an output value computed in a register can simply be passed along), and one where a suspension has occurred (in which case the output value must be stored into memory where other code is looking for it; this may block tail call optimization). One can imagine various baroque schemes for dealing with these problems; it was not clear to us that the complications and runtime overhead incurred were justified by the benefits of returning in registers the output values of procedures that may suspend, especially given the already high cost of suspension and resumption. For this reason, we have chosen to return output values of a call to a procedure p in registers only if all of the computation underneath that call can be guaranteed to not suspend: such calls are said to be *transitively non-suspending*. Since the suspension behavior of a particular call depends on its arguments, we create two copies of candidate procedures: one that can be guaranteed to not suspend and may return its outputs in registers, and another that may or may not suspend and returns its outputs in memory in the traditional way. At compile time, calls are routed to one or the other of these copies, depending on the results of suspension analysis (see Section 5.1). If the analysis determines that only one of these versions is used, only that version is kept.

3 Output Value Placement Methods

A variety of placement algorithms exist, each with its own advantages and disadvantages. We will discuss several of these methods in turn, using them both as examples to highlight features that are to be sought or

avoided, and to introduce various components of the cost model we will ultimately use to choose a placement for return values.

3.1 Homogeneous Placements

The simplest way to assign output value locations is to choose one fixed form of return, and use it throughout the program. This obviates the need to make complicated decisions about the best location to hold a return value, thereby speeding and simplifying compilation, possibly at the expense of run-time performance. Several homogeneous schema are plausible.

3.1.1 Memory

The classical place to return output values is in memory. As noted above, returning a value in memory is necessary in the absence of mode information, or when the suspension behavior of a procedure cannot be predicted. A homogeneous memory return scheme has the added advantage of never preventing a tail call optimization, since one memory location is as good as any other, and no moves to preferred locations need be inserted.

The disadvantage of returning values in memory is exemplified by the factorial program discussed in Section 1: the processor will store a value into memory, only to reload it immediately on return to the caller. The cost of returning a value in memory can be broken down into several components:

- allocating space for the return value on the stack or heap;
- preparing the return value slot (storing an "uninitialized variable" marker);
- at the callee, dereferencing the variable before assignment;
- assigning the value into memory; and
- loading it from memory again at the use-point.

For a homogeneous return scheme, the time taken to allocate space for a variable is not a major concern, since space can be reserved in the activation record of the clause which uses the value. In the case of the heterogeneous schemes examined below, the cost of allocation becomes significant when a disagreement in return location prevents tail call optimization, resulting in allocation of a previously unnecessary stack frame. For the purpose of comparing relative costs between alternatives when determining how an output value should be returned, the cost of allocating a new environment in this way is best absorbed by the cost of losing tail call optimization.

There are three reasons for initializing the return location by storing within it some special marker that indicates that it contains an uninitialized variable: first, for the correct working of general-purpose unification routines; second, so that garbage collection routines can recognize every item on the heap; and third, for concurrent implementations which might attempt to access the value before it has been computed. This initialization is usually slightly more expensive than a simple register-to-memory store, since some sort of tagging operation must be performed on the value to be stored.

Dereferencing a pointer chain to reach the final unassigned memory slot involves, in most cases, simply looking at the storage slot to ensure it contains the uninitialized marker (rather than a binding to another variable), but in the general case it can require unbounded dereferencing. It is relatively expensive, nevertheless, since it requires at least a load from memory, a test, and a conditional branch. In our cost model,

we combine the cost of determining the address and performing the store into one parameter, because the two correspond to one virtual machine instruction. The load of the value at the point of its use corresponds to a simple memory load.

There are other cases where simple memory stores and loads are required, such as when saving a value across a procedure call. We separate these two costs into two parameters, because reading and writing memory may require different overheads, especially on shared or distributed memory parallel architectures (due to overhead of memory-consistency mechanisms), but even on sequential uniprocessors like the Sparc (see the discussion in Section 6.1).

For each assignment of a return value into memory, we must do two memory writes (the initialization and the eventual assignment) and at least one memory read (for dereferencing), and possibly other operations such as tagging and untagging; in addition, there will be a read at the use point which might otherwise be avoided. Since memory operations can be expensive, and having the value in memory is often unnecessary, it is natural to consider returning all values in registers instead.

3.1.2 Fixed Register Placement

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 output argument being returned in register 1, the second in register 2, and so on. While simple, this scheme has the problem that it may require additional data movement to get the output values into the registers where they are needed. This is illustrated by the following example.

Example 3.1 Consider the following procedure, taken from a program to compute and evaluate Chebyshev polynomials:

```
:- mode cheb(in,in,out,out).
cheb(0,_,Y1,Y2) :- Y1=1, Y2=0.
cheb(1,X,Y1,Y2) :- Y1=X, Y2=1.
cheb(N,X,Y1,Y2) :- N > 1, N1 is N-1, cheb(N1,X,Z1,Z2), c1(X,Z1,Z2,Y1,Y2).
```

The third and fourth arguments of `cheb`/4 are output arguments. Suppose that we use the convention that the first output argument of a procedure is returned in register 1, the second in register 2, and so on. In this case, this means that in the literal `cheb(N,X,Z1,Z2)` in the body of the recursive clause, the value of `Z1` is returned in register 1 while that of `Z2` is returned in register 2. However, given the parameter passing convention of the WAM, the call `c1(X,Z1,Z2,Y1,Y2)` requires that `Z1` should be in register 2 and `Z2` in register 3. Thus, two additional register-to-register moves are necessary at each level of recursion to set up the arguments to the next call correctly. □

In most functional languages, where a function has only a single value, this scheme does not suffer from the problem of sometimes being unable to use tail call optimization. Even in functional languages where a function may return multiple values, such as Common Lisp, the simplicity of this scheme makes it the approach of choice: for example, the S-1 Common Lisp implementation returns up to 8 numeric output values in registers reserved for that purpose [3]. Unfortunately, this scheme interferes with tail call optimization in the presence of any of a number of features that are common in logic programming languages:

1. If a procedure can have more than one output argument, then returning output values in registers may sometimes require that TCO be sacrificed. This is illustrated by Example 2.2.

2. If a procedure uses partially instantiated structures, then tail call optimization may preclude returning output values in registers, and vice versa. This is illustrated by the following example:

   Example 3.2 Consider the following procedure:

   ```
   :- mode p(out).
   p([a|X]) :- q(X).
   ```

   The execution of this procedure creates a partially instantiated structure `[a|X]` and then calls `q/1`, which fills in the value of `X`. If `q/1` returns the value of `X` in a register, it is necessary to add an explicit store instruction at the end of the clause for `p/1` to ensure that the cons cell created by `p` has the correct values. If `p/1` wants to return (a pointer to) its output in a register, it is necessary to add an explicit load instruction at the end of its clause. In either case, tail call optimization is lost. □

3. If it is possible for procedure activations to suspend because of under-instantiated inputs, and suspendable procedures return their outputs in memory (as discussed in Section 2.2), then returning output values in registers may require inserting code to wait for the tail call to resume and compute a value, which is then placed into a register. This code insertion will block TCO.

3.1.3 Varied Register Placements

Many of the problems noted in the fixed scheme are a result of the rigidity of register assignment (and the ensuing need to move values to the desired location, as in Example 3.1). A natural extension is to consider choosing the appropriate return register to avoid as many of those moves as possible, by seeing where the value will be needed by calling procedures. We did not examine homogeneous varied schemes of this sort directly because (*i*) they require estimation of execution frequencies to determine which of several use points is the most important to satisfy, hence become much more complicated than a fixed scheme; (*ii*) choosing a return location for one procedure may require recomputation of preferred locations for a previously assigned procedure (especially in the presence of non-trivial strongly connected components in the call graph), resulting in an iterative computation which may not reach a fixpoint; and (*iii*) there are still cases, such as that in Example 2.2, where no homogeneous register return scheme will preserve opportunities for tail call optimization. This last problem is avoided by adopting a heterogeneous output placement method, which allows different procedures to return values in either memory or registers, depending on their callers. The machinery involved in our method of deciding whether to use memory or register for the return location naturally extends to indicate which register will require the fewest additional moves, addressing the first concern as well. By applying the heterogeneous scheme in a bottom-up order (assigning return locations to the most frequently executed and highest saving candidates first), we avoid the second problem's concern with recomputing locations while preserving good savings overall.

3.2 Heterogeneous Assignments

Perhaps the best-known heterogeneous return location assignment is that used in the Aquarius Prolog compiler, and described by Van Roy in [24]. The Aquarius scheme can be considered a fixed register heterogeneous method, in which all return values of non-suspending procedures are candidates for being returned in a register, and are removed from candidacy (placed into memory) when some condition is no longer met. Candidates that survive the winnowing process are then returned in the register in which

the unassigned variable pointer would have been passed. This fixed assignment avoids the need to re-visit previously assigned procedures as the candidacy of return values from called procedures changes, as would be required with the policy described in Section 3.1.2.

An output variable is removed from candidacy if its definition point in a procedure body appears before any call in the body, since calls may destroy register contents. This condition is sufficient for a correct heterogeneous scheme, but is too permissive because it allows register returns which will require additional moves to ensure proper placements, potentially blocking tail call optimization. It is also somewhat stronger than necessary, since if the last call in the body is followed by an in-line computation (which precludes tail call optimization), the value of the variable can be saved in the activation record and restored to a register after the last procedure call in the clause body, potentially removing the initialization and dereference costs from the assignment. The loss of opportunities for tail call optimization is reduced by adding two additional, independent conditions:

- If the definition point for a procedure output is in a tail call, and the argument position of the value in the callee differs from that of the caller, return in memory rather than deoptimize the last call to move the value between registers. This affects the candidacy of the linked arguments in both the caller and the callee.
- If the last goal of a body is a call which has candidate outputs that are not candidate outputs of the caller, the tail call must be deoptimized to store the extra return values into memory. In the case where the called procedure is "fast" (essentially, a leaf procedure) the extra return values are removed from candidacy in the called procedure to avoid this deoptimization.

These conditions reduce the number of times that a tail call optimization is blocked, but do not eliminate them. The first of these two conditions is designed to address the permissivity of the basic rule, by retaining opportunities for TCO. Unfortunately, the fixed register return policy conspires with this condition to force memory placements for many outputs, because of position disagreements in tail calls.

The Aquarius method does not take into account frequency of execution, and may unwittingly choose to return values in memory in a frequently called procedure in order to retain a tail-call optimization opportunity in a rarely called one. Nor does it consider relative costs of losing a tail call optimization versus storing values into memory: the overhead of preparing memory slots and dereferencing chains can make the two costs fairly close. Van Roy's analysis of this method indicated that approximately one third of all outputs can be converted to register placement with this scheme. Our implementation of the method in `jc` confirms good speedups for small programs, but shows only a negligible improvement over homogeneous memory for more complicated programs where few outputs satisfy the restrictions for register placement (see Tables 2 and 3).

4 An Algorithm for Output Value Placement

The discussion of the previous section suggests that a good return location assignment should have the following characteristics: (*i*) it should be *heterogeneous*, so as to avoid both losing tail call optimization opportunities and excess memory accesses; (*ii*) it should be *varied* in register assignment, so that values are placed in registers where they will be needed next; (*iii*) 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; and (*iv*) it should be parameterized for the costs of certain primitive operations, so that it can be implemented in compilers for a variety of architectures which can have drastically different memory

subsystem behaviors. This section describes an algorithm we have developed that has these characteristics and that has been implemented in the `jc` system [12]. It assumes that information about input/output modes, possible suspension behavior for procedures, and relative execution frequencies for procedures and clauses has been obtained separately; see Section 5. 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, and the second pass does a bottom-up assignment, choosing at each point the (apparently) best return location while avoiding tail call deoptimizations that would overwhelm the benefits of previous choices.

4.1 Pass 1: Determining Output Location Costs

For the first pass, we want to find the costs associated with each potential return location for each output value of a procedure without assuming anything about other procedures. This involves looking at two program points separately: the point following a defining call where the returned value is used, and the point in a clause where a return value is defined.

Output values are defined either by calls or via explicit assignment operations;[3] definitions by calls ultimately are grounded in assignment operations. Assignments compute the value to be assigned into a register, and we assume that the local code generator can arrange to place the value in any specified register without incurring additional cost. We follow the terminology of Van Roy [24] and classify body goals as either *survive* goals, which preserve the contents of registers, or *non-survive* goals, which can destroy the contents of registers. If a clause body contains a non-survive goal following a definition into a register, the value of that register has to be saved over the non-survive goals and restored at the clause end, possibly preventing tail call optimization.

The first pass of the algorithm 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 location 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.

Example 4.1 Consider the following code:

```
:- mode p(out, out).          :- mode q(out).          :- mode r(out).
p(A, B) :- q(A), r(B).        q(A) :- A = 3.           r(A) :- A = 4.
```

Assume that `p/2` is to return its first value (`A`) in a register. Because calls are non-survive goals, `A` must be saved into memory across the call to `r/1`. If `q/1` returns its value in a register, then this causes an incremental cost of one load and save within `p/2`. If the value is returned from `q/1` in memory, `p/2` adds the cost of allocating and initializing a memory slot for the value, then loading the value again at the end. The cost of doing the actual assignment into memory is associated with the assignment statement in `q/1`. No matter where `q/1` returns its value, the tail call to `r/1` cannot be optimized, because the return result of `q` must be reloaded afterwards.

On the other hand, if `p/2` is to return its value in memory, then it has been passed a pointer to a memory location by some ancestor: the program point where that location was reserved accrued the cost

[3] In general, variables are given bindings via unification, but since we assume a moded language, this degenerates to assignment.

Parameter	Description	Typical Value	
		cc	cc -O4
C_x	Move a value from one register to another	1	1
C_s	Store a value from a register into memory	441	238
C_l	Load a value from memory into a register	158	89
C_a	Dereference a pointer and assign a register into memory	594	374
C_p	Initialize an unassigned variable slot	417	283
C_{ta}	Call and return instead of jump, plus environment allocation	1683	1204
C_{tn}	Call and return instead of jump, no environment allocation	1015	922

Table 1: Parameters to cost model (values in microseconds, on a SPARC)

of allocation and initialization. If `q/1` returns its value in a register, then `p/2` incurs the cost of doing the pointer dereference and assignment after control returns from `q/1`. If `q/1` returns its value in memory, then `p/2` incurs no additional cost at all, since the cost of doing the assignment is attached to `q/1`. □

Costs associated with a particular program point are multiplied by the estimated execution frequency of the clause in which the point occurs, so that we get an estimate of the overall time spent by the program as a whole due to the choice of a particular return location.

It is important to point out that the costs associated with deoptimizing a tail call are incurred only once for each clause, no matter how many potential decisions might separately force this deoptimization. For example, the code in Example 3.2 may block TCO for two reasons: `q/1` returning its value in a register, or `p/1` returning its value in a register. Either is sufficient to block TCO and incur the corresponding cost, but the cost should not be charged twice if both conditions hold. Therefore, rather than add tail call deoptimization costs into the cost arrays now, we simply set, for each potential output placement for a clause, a flag that indicates whether choosing that location for that value will prevent a tail call optimization in that clause. In the second pass, tail call opportunities are further constrained as return locations are assigned in other procedures, and the flags are updated accordingly. At the final assignment for a particular output in the second pass, the cost of losing a tail call opportunity is added exactly once for each placement that would force such a loss where the opportunity existed before.

4.1.1 Cost Considerations at Use Point

The costs of preparing for and using a returned value depend on the contexts of the definition and use in a clause body. At a particular call site that defines a variable, there are several possible places for the use of that variable: (*i*) in a body goal following the definition point, (*ii*) in a body goal prior to the definition point, or (*iii*) returned from the function. There is also the potential that it is not used at all, or that it is used multiple times in different ways. We consider each of these cases in turn.

Consider the following sequence of body goals:

$$g(\vec{t}) \text{ :- } p_1, d(\ldots, x, \ldots), p_3, u(\ldots, x, \ldots), p_5. \tag{1}$$

where the p_i encapsulate zero or more body goals (survive or non-survive), d is a procedure call which defines some variable x, and u is either a procedure call or an expression which uses the value of x computed in the call d.

- If u is an expression and p_3 contains no non-survive goals, then d can return x in any register, and it will be available for use in u at no additional cost. For memory returns, the cost is that of preparing a memory space for d to assign into, and loading the memory value back into a register for use at u. We summarize this using the following notation, where r_i refers to (any arbitrary) register i, M refers to memory, and the cost parameters are as described in Table 1 (see Section 6.1 for a description of the values in the table):

  $$r_i : 0 \qquad M : C_p + C_l$$

- If u is a call and p_3 contains no non-survive goals, then the returned value x will be passed to u in the register corresponding to its argument position, say r_k, known at analysis time. If d returns x in register r_k, the cost is zero. If it returns x in a register r_i, where $i \neq k$, the cost is that of moving a value from one register to another. If d returns x in memory, the cost is the same as in the previous case, since the load can be made into the desired register.

  $$r_k : 0 \qquad r_{i(i \neq k)} : C_x \qquad M : C_p + C_l$$

- If p_3 contains non-survive goals, then the return value must be saved over those goals, no matter where the value is returned. The incremental cost for memory over registers is that of initialization, and the cost of any register is a store (the store for memory is accounted for at the definition point).

  $$r_i : C_s \qquad M : C_p$$

These exhaust the cases where the value is used following its definition point. It is also possible that the value is "used" prior to the definition point. This case is restricted for the situation we are considering, because definedness analysis (see Section 5.2) must have determined that the actual value will not be required before it is defined. However, the variable which is being defined may have been encapsulated into a structure prior to the call, and the structure then independently returned from the procedure and referenced later (see Example 3.2). If the definition is into memory, then nothing need be done, since the appropriate location in the structure will be filled at the assignment point. However, if the definition is into a register, code must be inserted following the defining call to move that value into the appropriate memory location within the structure. As such, if the defining call is the last body goal, register return locations will prevent tail call optimization, potentially incurring an additional C_t cost in the second pass.

The final case is where the value is not used elsewhere in the body, but is returned through the head as an argument. This case corresponds to a chained definition rather than a proper use, and is ignored in the definition point analysis below, for reasons described there.

If none of the above cases occur, the computed value is not used. We assume that programs that compute inaccessible values are relatively rare, so do nothing to recognize and handle this case. However, it is possible that a value is both used and returned, or is used multiple times. The split of the first phase of the algorithm by considering use and definition points separately allows values that are both used and returned to be handled correctly: the costs for the first use are computed above, and the costs for the return are handled by the definition case below. We do not attempt to account for multiple uses of a value in a clause, taking into account only the first use; our experience suggests that this does not pose a problem in practice.

4.1.2 Cost Considerations at Definition Point

A definition point for a value looks like:

$$g(\ldots, x, \ldots) \ :- \ p_1, d(\ldots, x, \ldots), p_2. \tag{2}$$

where x is an output of g, d is an assignment to x or a procedure call with x in an output position, and the p_i are again zero or more body goals. If x is to be returned in memory, one of the inputs to g is a pointer to its storage cell. If p_1 contains any non-survive goals, this pointer must be saved across them in the activation record, so memory costs an additional $C_s + C_l$. (We assume that non-survive goals in non-tail position induce a frame allocation to save information such as return location, so there is no incremental cost for allocating the storage space).

If d defines the value by an assignment statement, then we assume the value to be assigned is computed into a register, and it doesn't matter which one, so we can arrange for it to be computed into whichever register is most useful. As such, returning a value in any register costs nothing, unless there is a non-survive goal in p_2, in which case the value must saved and reloaded, potentially deoptimizing a last call. If the value is returned in memory, the cost is that of dereferencing the variable chain and performing the store (C_a).

$$r_i : 0 \text{ or } C_s + C_l \qquad M : C_a$$

If d defines the value by a call, then the cost of each return location depends on where the defining call returns the value. Since this information isn't available yet and no alternative seems more likely than another, 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 defined all the costs that are independent of particular output value placements. We can now visit each procedure in turn, and assign to each output the location that yields the smallest incremental cost to the program as a whole.

As noted previously, fixing a choice for one procedure affects the optimal choice for another (e.g., in tail calls). One way to avoid the difficulties that arise from this is to use an iterative approach, going back to reconsider previous decisions when an assignment that might affect them is made. It is not immediately clear that such iteration will reach a fix-point. We have opted for a greedy approach, making each assignment only once and living with the consequences, but avoiding the worst of the problem by ordering the assignments to capture the biggest potential savings first.

To that end, we assign output locations to procedures in decreasing order of frequency. We start by augmenting the flags originally defined in the first pass that indicate that particular output placements will cause a tail call deoptimization. Each clause of the procedure is examined: any output defined prior to a potentially optimizable tail call will cause a loss of TCO if it is returned in a register, so register placements for all such outputs are tagged as preventing this optimization. We can also take advantage of the fact that other procedures may have already determined their output placements. Therefore, if a clause ends in a tail call to a procedure whose outputs have already been assigned, the flags are updated to indicate loss of optimization for mismatched locations that will require cleanup code to move values. When we have a tail call to an unassigned procedure, we have no way of telling which locations will eventually cause loss of optimization, so leave the flags unmodified.

The cost for assigning a particular output of the procedure to a particular location is initially set to the cost computed in the first pass. For each clause that has the potential for a tail call optimization, each location that would prevent that optimization gets an additional cost corresponding to the expense of losing the opportunity, scaled by the frequency of the clause. Furthermore, for each tail call from an already-placed procedure, placements which would force a deoptimization in the caller incur the corresponding tail call optimization loss cost. Costs corresponding to loss of TCO come in two flavors: one where the calling clause had already allocated an environment (e.g., for a previous call), and one where an environment was not already allocated but must be to preserve the caller's return address over the unoptimized tail call. The appropriate cost can be determined for a particular clause at compile time, and is scaled by the execution frequency of the caller's clause.

Within a particular procedure, there may be multiple outputs, each of which has a choice of return locations. The assignments for these outputs can also interfere, for example when an assignment of a particular register to one output at a small savings prevents use of that register for another output, incurring an overwhelming cost for the next best choice. To lessen the effect of this 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 the output is finally assigned. In the case of ties between locations, memory is chosen over registers of the same cost because memory will less-often destroy a tail call opportunity. This assignment is then set: tail calls that must be followed by cleanup code resulting from the assignment are marked as no longer potential sites for tail call optimization (so the corresponding costs are not charged multiple times), and the search and assignment is repeated until all outputs have an assigned location.

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 the second pass $O((1+p^2)(S+C))$, where S is the number of call sites in the program and C is the number of clauses. Hence, the algorithm is essentially linear in the size of the program.

4.3 Additional Optimizations

The cost model described above can be augmented in several subtle ways, to work around other aspects of the compiler that might interfere with a particular choice but weren't considered above.

Consider the following clause body:

```
:- mode p(out), q(in, out), r(in, in).
..., p(Op), q(Iq,Oq), r(Op,Oq),...
```

where the output of `p`/1 is not used in `q`/2, so must be saved across it. If the value of `Iq` is not needed after the call to `q`/2, the stack slot that held it across `p`/1 can be re-used to hold `Op`, saving the cost of allocating and initializing a stack slot for it. If `p`/1 returns `Op` in the same register as `q`/2 wants to receive `Iq`, then `Iq` must be loaded into a temporary, `Op` saved into the old stack slot, and `Iq` moved into `Op`'s old register. We can take this extra work into account in the placement scheme by adding the cost of a register-to-register move to this register when considering the best location for `p`/1; this will often cause it to chose another register that would otherwise be the same cost.

The components of the cost model can also be decomposed further, to yield a finer distinction between

programs. For example, in the case where an output value is a structure with pointer elements (such as a list), the assignment operation includes a "make-safe" operation, which ensures that the referents of the structure are in long-lived storage rather than local (stack) space, so the returned value will point to valid data. This can be accounted for by increasing the cost of assignment in those cases where it is known that the value being assigned is a structure.

Both of these optimizations have been implemented in the system described in the remainder of the paper.

5 Output Value Placement in `jc`

We have implemented the output value placement algorithm described above in the `jc` system [12], a sequential implementation of a variant of Janus [19, 12]. The algorithm assumes that information about input/output modes, suspension behavior, and relative execution frequencies has been computed separately, and also that machine-level costs for various low-level operations have been supplied. The program-dependent information is obtained from unannotated user code through several compiler analyses which are described in this section. The method used for obtaining system-specific cost parameters is described in section 6.

5.1 Suspension Analysis

The suspension analysis used—a more accurate name might be "non-suspension analysis", since we are interested primarily in identifying procedure calls that can be guaranteed to not suspend—proceeds in two phases. The first phase is a *demand analysis* that determines, for each procedure, the extent to which the input arguments have to be instantiated to guarantee that no guard will have to suspend (i.e., each guard will succeed or fail). The second phase uses this demand information to iteratively propagate information about groundness,[4] and determines which calls are transitively non-suspending. The analysis depends heavily on the fact that `jc` is a sequential implementation where the body literals in a clause are executed in their textual left-to-right order; it is easy to generalize this to the situation where for each clause in a program there is a fixed partial order over the body literals that specifies the order in which they are executed.

There are two kinds of information propagated during analysis: which calls are guaranteed to not suspend, and which variables in a clause can be guaranteed to be ground at any program point. Initially, all calls are assumed to be non-suspending. Analysis begins with the most general goals for the exported procedures. The essential idea is to proceed as follows: when analyzing a call to a procedure, the literals in the body of each clause for the called procedure are processed from left to right, propagating the set of variables in the clause that can be guaranteed to be ground at different program points. Initially, the set of variables that must be ground just before the body is executed is determined by examining the guard of the clause (since if the guard suspends, the body will not be executed). As the analysis progresses, it identifies calls that might have to suspend because not all of the actual parameters are sufficiently instantiated to meet the demand (computed during the first phase of analysis) of the called procedure. When a call is identified as being potentially suspending, our estimate of the set of ground variables at the point immediately after that call has to be updated. This now has the effect that other calls that use the values produced by this call have to be reanalyzed to determine whether they may now have to suspend (as a special case, assignment actions in the body of the caller may have to suspend because their inputs are inadequately instantiated). Proceeding in this way, the set of variables that can be guaranteed to be ground at the program point following the last

[4] The ideas underlying this analysis are to a great extent independent of the abstract domain, and it is not difficult to see how it can be adapted to other abstract domains, e.g., one of types. We restrict ourselves to groundness here in part because it is simpler to describe, and in part because that is what is currently implemented.

literal in the clause is used to determine which output arguments of the call can be guaranteed to be ground when execution returns to the caller. This information is then used to update the set of ground variables at the point immediately following the call in the caller's body. The analysis proceeds iteratively in the expected way, using extension tables to guarantee termination (see, for example, [10]), until there is no change to any of the information inferred. Details are omitted due to space constraints. It is straightforward to show, by induction on derivation lengths, that if a call to a procedure in a program can suspend at runtime, it will be identified as potentially suspending by this algorithm. It is possible that better results might be obtained using an analysis that handles suspension and resumption of goals more carefully (see, for example, [16]), but at this point the precision of our analysis seems acceptable.

5.2 Definedness Analysis

Transitive non-suspension of a particular call is not a sufficient condition for the compiler to convert that call to a version which returns values in registers. The problem is that until a value has been written to the register, the contents of that register are undefined. This makes it necessary to avoid any situation where the value of a variable is returned in a register by the procedure that generates it, but where the variable may be read—viz., the contents of the corresponding register used—before its value has been written to the register. This is handled using a dataflow analysis, called *definedness analysis*, to determine that values being returned in registers can be guaranteed to not be undefined at any use point. Let $producer(X)$ denote the goal that generates the value of a variable X in a clause and $Consumers(X)$ the set of goals that use the value of X, then X can be returned in a register if the following conditions hold:

1. neither $producer(X)$ nor any procedure called by it uses X before assigning a value to it;
2. $producer(X)$ completes execution before any of the calls in $Consumers(X)$ start executing; and
3. $producer(X)$ can be guaranteed to assign an initialized value to X (since otherwise X may contain garbage even though it may have been assigned to).

Assume that we are given a partial order on the body literals of each clause that specifies, at compile time, the order in which they will be executed: in `jc`, this is simply the textual left-to-right order. To approximate Condition 1, we compute which of the variables appearing in a call may be used during the execution of that call (this involves a simple traversal of the program) and require that neither $producer(X)$ nor any procedure called by it should use X. This is somewhat stronger than necessary, but does not seem to cause any problems in practice. It is not difficult to define a dataflow analysis so that $producer(X)$ is allowed to use X as long as a value is assigned to it beforehand, but our implementation does not currently do this. To verify that Condition 2 holds, we require that (*i*) $producer(X)$ precedes every goal in $Consumers(X)$, and (*ii*) $producer(X)$ as well as every goal called by it can be guaranteed to not suspend: the first requirement ensures that $producer(X)$ begins executing before any goal in $Consumers(X)$, while the second ensures that, given the control strategy of `jc`, $producer(X)$ will finish execution before any of the goals in $Consumers(X)$ begins executing. Information about non-suspension is obtained from the dataflow analysis described in Section 5.1. Condition 3 can be verified using a simple dataflow analysis over a two-point abstract domain {*defined, undefined*} that is very similar to that of Section 5.1.

5.3 Estimating Execution Frequencies

Static execution frequencies of procedures in a program are compile-time projections of the expected number of times that a given procedure or clause will executed in a single invocation of the program. These projections

can then be used to decide which procedures are the most likely to benefit from optimizations, and which can be "sacrificed" for a greater speedup overall. It is important to note that the purpose of these estimates is not really to predict accurately how many times a branch will be executed or a procedure called, but to guide optimization decisions by giving (possibly conservative) relative execution frequencies for different program fragments. *In fact, our algorithm is not very sensitive to execution frequency estimates: tests using our algorithm but assuming the same frequency for all clauses and procedures indicated a loss of only 2 percentage points off the overall improvements for both the tuning and timing tests listed in Section 6.* Our current implementation estimates execution frequencies by examining the call graph of the program, assuming that loops will execute some fixed number of times and branches have an equal probability of being taken, and assigning each procedure a frequency based on its loop depth and branch position. The details of our method are given in Appendix A. For a discussion of related techniques used in compilers for traditional languages, see [2, 15, 17, 25, 26]; techniques for estimating execution frequencies of logic programs from their call graph structure are discussed in [9, 20]. An alternative is to profile the program on sample inputs to estimate execution frequencies, e.g., see [11].

6 Tuning and Testing the Model

With the analysis results from the previous subsections, the algorithm of Section 4 can be implemented. As noted in Table 1, there are seven parameters to the cost model, which are system and compiler specific, on which the algorithm depends. Before using the location assignment algorithm, appropriate values must be found for these parameters on a given system. This is done by an iterative process on a small set of test programs. After parameters have been chosen that yield good behavior on the tuning programs, the model must be checked against other programs to verify that a projection of its performance is generalizable.

To avoid the tedium of implementing an assembly-level back end, and to increase portability across architectures, `jc` translates Janus programs into C [1] text, which is then compiled by the system C compiler. Tuning and performance evaluation were done experimentally, by compiling Janus programs with various location assignment policies, and measuring the time taken to execute a query with the resulting executable. Timing tests were done on a Sun IPX (40MHz SPARC) with 32Mb physical memory, running SunOS 4.1.1, 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 SPARC executable was created which ran a test query one hundred times and gave as its result the shortest measured query execution time. Queries were designed to be 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. Each experiment consisted of running each policy for each test program once, resulting in an execution time for each test and policy pair; the order of execution was random within an experiment to avoid systemic bias from disk and memory cache effects. Five of these experiments were performed. The numbers in the tables were formed by normalizing the execution times for a given benchmark to the homogeneous memory return policy result for that experiment, and taking the geometric mean of the five experiments; the global results are the geometric mean over all normalized tests. The system was tested using the Sun C compiler bundled with SunOS 4.1.1, with no optimizations and with -O4 optimizations.

6.1 Tuning the Model

Several steps are taken to tune the model for a particular system. A suite of Janus programs are translated to C code, and the C code is modified to provide a minimal pair of contrasting programs, which differ exactly by some parameter to be measured—e.g., for C_{tn}, a tail call is replaced by a call-and-return sequence. These

programs are then compiled with the C compiler and optimization levels for which we want parameters, and the difference between the pairs written to a file which can be read by the `jc` compiler when a program is being compiled using the given C compiler.

The numbers given in Table 1 were generated on the test architecture in this way. The values given are the median of five runs of each pair; the spread across the runs is generally less than $5\mu sec$, so the results are reproducible.

An alternative method of estimating the cost of a virtual machine instruction, which is initially plausible, is to count the machine-level instructions which emulate it. This becomes quite complicated when the emulation sequence involves jumps which may or may not be taken. On modern architectures the effect of pipelines, multi-level caches, and other hardware optimizations conspire to make such estimations very bad predictors of actual behavior at runtime, even when only straightline emulation code is involved. This is borne out by the results in table 1, where the cost of moving a register to memory (C_s) is about 2.7 times the cost of loading a register from memory (C_l), although both map to single machine instructions, and the actual instructions measured involve the same hardware register and memory locations. Of course, the time to execute an instruction depends strongly on the context in which it appears—hence the difference in speed between cc and cc -O4 (although the instructions corresponding to the loads and stores are essentially the same, the compiler moved the load and store with respect to surrounding instructions, presumably to avoid pipeline stalls). The vagaries of experimental analysis of systems notwithstanding, the primary evidence for the acceptability of a method is the performance achieved when using the resulting numbers: as seen below, our estimations perform very well.

After an initial set of parameters is generated, a tuning set of small benchmarks whose performance is strongly dependent on return location assignment is then run with the different assignment policies described in Section 3, along with our algorithm using these initial estimates for the costs. Since most of the tuning benchmarks are small and the procedures in them have only one or two return values, the policies tend to cover the possibilities for return locations (modulo specific registers), so it is reasonable to say that policy that results in the fastest execution time for a benchmark is close to the optimal one for that benchmark. There are also a few larger benchmarks in the tuning set to help catch situations where the performance of the small benchmarks does not extend to larger programs.

We then examine the performance of these different location assignments, and look more closely at the cases where our algorithm does not produce the empirically best assignment of output locations. If there are gross errors, this is usually due to our algorithm choosing memory over register or vice-versa, rather than choosing the wrong register. By tracing the cost assignments made by the algorithm (available through a compiler option), it is possible to find places where the wrong location was chosen, but the correct one would be chosen if one or two cost parameters were slightly adjusted. The benchmarks can then be re-run, and the process repeated until the parameters result in the best decision in as many cases as possible.

Table 2 indicates performance results for the tuning benchmarks on a Sun IPX for homogeneous memory returns, the fixed register return scheme described in Section 3.1.2, the Aquarius algorithm adumbrated in Section 3.2, and our algorithm, using the parameter estimations given in Table 1, all implemented in the `jc` compiler and available through compiler options. The schemes are normalized with respect to homogeneous memory returns, with the relative execution time of the fastest policy highlighted by bold text. Following each performance number is a letter which indicates the overall placement chosen for this benchmark: if two columns share the same letter, the algorithms resulted in the same policy (for example, for the `nr-1` benchmark our algorithm chose the same assignment as the Aquarius algorithm). The letter is "m" if the

Benchmark	Sun cc, no opt			Sun cc -O4		
Name	Our Algorithm	Fixed Reg.	Aquarius	Our Algorithm	Fixed Reg.	Aquarius
array_qs-1	**0.8941** (a)	0.8973 (b)	0.9971 (c)	0.8641 (a)	**0.8626** (b)	0.9900 (c)
array_qs-2	**0.9019** (a)	0.9028 (b)	0.9991 (m)	**0.8666** (a)	0.8715 (b)	1.0004 (m)
binomial-1	**0.6704** (a)	0.6860 (b)	1.0000 (m)	**0.7000** (a)	0.7049 (b)	0.9999 (m)
combB-1	**0.9983** (a)	1.3085 (b)	1.0000 (m)	**0.9912** (a)	1.3234 (b)	1.0006 (m)
dnf-1	**0.9917** (a)	0.9923 (a)	0.9938 (b)	**0.9865** (a)	**0.9865** (a)	1.0025 (b)
dnf-2	1.0003 (a)	**1.0000** (a)	**1.0000** (m)	0.9854 (a)	**0.9815** (a)	0.9996 (m)
dotprodB-1	1.0010 (a)	1.2271 (b)	**1.0001** (m)	1.0004 (a)	1.2226 (b)	**1.0001** (m)
fact2-1	**0.6534** (a)	0.6709 (b)	1.0003 (m)	**0.6988** (a)	0.7033 (b)	1.0056 (m)
fact3-1	**0.9979** (a)	0.9986 (a)	0.9986 (b)	**0.9998** (a)	1.0000 (a)	**0.9998** (b)
fact3-2	**0.9869** (a)	0.9953 (a)	1.0047 (b)	1.0014 (a)	**1.0000** (a)	1.0067 (b)
fib-1	**0.6003** (a)	0.6304 (b)	1.0003 (m)	**0.5578** (a)	0.5662 (b)	0.9991 (m)
hanoi-1	**0.8686** (a)	0.8814 (b)	0.9334 (c)	**0.8495** (a)	0.8585 (b)	0.9295 (c)
lclocal-1	**0.9949** (m)	1.2703 (a)	1.0000 (m)	**0.9909** (m)	1.2671 (a)	0.9993 (m)
list_qs-1	**0.9697** (a)	1.2929 (b)	**0.9697** (c)	**0.9564** (a)	1.3210 (b)	0.9569 (c)
list_qs-2	0.9714 (a)	1.2878 (b)	**0.9707** (c)	0.9575 (a)	1.3294 (b)	**0.9570** (c)
long1-1	0.5194 (a)	**0.5185** (a)	0.5748 (b)	0.5377 (a)	**0.5351** (a)	0.5551 (b)
nr-1	**1.0000** (a)	1.5594 (b)	1.0016 (a)	**1.0007** (a)	1.6142 (b)	1.0018 (a)
nr-2	**0.9991** (m)	1.5498 (a)	1.0000 (m)	**0.9986** (m)	1.6105 (a)	1.0000 (m)
pascalB-1	**0.9901** (a)	1.2571 (b)	1.0007 (m)	**0.9817** (a)	1.3158 (b)	1.0038 (m)
queen-1	**0.8760** (a)	0.9150 (b)	1.0007 (m)	**0.8942** (a)	0.9179 (b)	1.0004 (m)
short-1	0.4910 (a)	**0.4878** (a)	0.5490 (b)	0.5210 (a)	**0.5152** (a)	0.5348 (b)
short2-1	**0.5065** (a)	0.5073 (a)	0.5291 (b)	**0.5222** (a)	0.5229 (a)	0.5332 (b)
tak-1	**0.5795** (a)	0.6110 (b)	0.6115 (c)	**0.5174** (a)	0.5200 (b)	0.5236 (c)
Global	**0.8217**	0.9212	0.9004	**0.8189**	0.9205	0.8915

Table 2: Tuning test results for various policies, normalized to homogeneous memory

Benchmark	Sun cc, no opt			Sun cc -O4*		
Name	Our Algorithm	Fixed Reg.	Aquarius	Our Algorithm	Fixed Reg.	Aquarius
bessel-1	0.9147 (a)	**0.9089** (b)	0.9996 (m)	**0.8772** (a)	0.8780 (b)	1.0027 (m)
cheb-1	**0.4775** (a)	0.5042 (b)	0.9999 (m)	0.4492 (a)	**0.4396** (b)	1.0049 (m)
deriv-1	1.0271 (m)	1.1132 (a)	**1.0000** (m)	1.0212 (m)	1.1077 (a)	**1.0000** (m)
disj2-1	**0.7815** (a)	0.7889 (b)	0.8963 (c)	**0.7593** (a)	0.7837 (b)	0.8839 (c)
factsq-1	0.9891 (a)	**0.9818** (a)	0.9939 (m)	0.9924 (a)	**0.9848** (a)	0.9939 (m)
list_qs2-1	1.0002 (m)	1.1805 (a)	**0.9997** (m)	**0.9497** (a)	1.1613 (b)	0.9996 (m)
merge-1	**0.9949** (m)	1.4365 (a)	1.0000 (m)	**0.9838** (m)	1.5946 (a)	0.9964 (m)
prime1-1	**0.9984** (a)	1.2634 (b)	1.0003 (m)	1.0024 (a)	1.2718 (b)	**1.0000** (m)
queenk-1	**0.9348** (a)	0.9362 (a)	0.9648 (b)	**0.9507** (a)	0.9526 (a)	0.9739 (b)
Global	**0.8817**	0.9745	0.9833	**0.8646**	0.9672	0.9832

Table 3: Performance evaluation test results

policy coincides with homogeneous memory. Where there are two versions of the same benchmark, the first includes the time taken to construct the input (e.g., the list to be reversed, for `nr-1`); this may increase the number of opportunities to return values in registers.

Examination of the results indicate that with the initial parameter set, our algorithm picks the best tested placement in 18 or 19 of the 21 test cases, depending on the level of C compiler optimization. In the remaining cases, the Aquarius algorithm picks a better placement, but by at most 0.15% of the memory execution time. As such, there seems to be no value in tuning the parameters further for these two back-ends: the parameter generation suite yields an excellent starting point.

6.2 Testing the Tuned Model

The performance of an algorithm on simple benchmarks for which it may have been tuned is no indication of what sort of performance we can expect on other and more complicated programs. Therefore, after tuning was completed, a separate set of benchmarks which tended to be more complicated were run with the same policies, and the results analyzed.

Table 3 gives the results of various policies using the same performance model and compiler options as in Table 2 (`cc` was unable to compile `disj2` using `-O4` optimizations, due to the size of the intermediate files used by the compiler, so the results for that benchmark are using `-O`). These results show that the cost-based policy described here, with appropriate parameters, does very well on a variety of programs that it wasn't tuned to handle. Table 4 summarizes the absolute execution time of each benchmark for homogeneous memory and our algorithm, and the performance improvement that our algorithm yields over memory, along with the fraction of output values that were placed in registers by our algorithm (e.g., of 11 outputs of non-suspending procedures in `pascalB-1`, 5 were placed in registers and the rest in memory). (The overall improvement summary is an arithmetic average of the individual speedups, so is slightly lower than the geometric averages given in Tables 2 and 3). Notice that the performance improvements can be quite substantial for programs where outputs can, in fact, be returned in registers (e.g., `fib-1`, `fact2-1`, `binomial-1`); for programs that spend a large portion of their time in tail-recursive procedures constructing lists of values (`nr-1`, `list_qs-1`), returning outputs in registers is not as beneficial, and our method less-often chooses to return those values in registers. Note that, even though our algorithm may occasionally choose to give up tail call optimization in order to put output values in registers, no program does significantly worse than the traditional approach of returning outputs via memory using our scheme. Overall, we find that our algorithm for output placement produces an average performance improvement of about 12–18%, depending on compiler and benchmarks—this compares favorably with execution time reductions that have been reported for optimizations that are recognized as effective within the compiler construction community, such as procedure inlining (12% [7], 10% [4]), register allocation (20% [18]), and loop-invariant code motion (13% [18]).

7 Conclusions

While most implementations of logic programming languages return the output arguments of procedures via memory, this can be a source of unnecessary overhead and can cause a performance degradation. Returning outputs in registers is an attractive alternative, but the situation is complicated by the fact that this may lead to a loss of tail call optimization. In this paper, we examined a variety of plausible schemes for returning output arguments, and gave an algorithm for output argument placement that uses cost estimates for various alternatives, weighted by execution frequency estimates, to determine a "good" output location assignment for each procedure in a program. Our experiments indicate that for programs where the outputs are best

Benchmark	Time to Execute		Improvement	Fraction Outputs
	Memory	Our Alg.	(%)	In Registers
array_qs-1	2721.2	2351.4	13.59	8 / 8
array_qs-2	2707.8	2346.6	13.34	7 / 7
binomial-1	6220.8	4354.6	30.00	9 / 9
combB-1	974.6	966.0	0.88	2 / 4
dnf-1	563.2	555.6	1.35	3 / 3
dnf-2	560.8	552.6	1.46	2 / 2
dotprodB-1	3637.6	3639.2	−0.04	2 / 5
fact2-1	2267.4	1584.4	30.12	2 / 2
fact3-1	869.6	869.4	0.02	2 / 2
fact3-2	148.4	148.6	−0.13	2 / 2
fib-1	2161.2	1205.6	44.22	2 / 2
hanoi-1	377.4	320.6	15.05	4 / 4
lclocal-1	286.0	283.4	0.91	0 / 2
list_qs-1	793.2	758.6	4.36	3 / 5
list_qs-2	776.6	743.6	4.25	2 / 4
long1-1	687.4	369.6	46.23	3 / 3
nr-1	569.2	569.6	−0.07	1 / 3
nr-2	567.4	566.6	0.14	0 / 2
pascalB-1	633.4	621.8	1.83	5 / 11
queen-1	516.2	461.6	10.58	4 / 8
short-1	275.6	143.6	47.90	3 / 3
short2-1	270.8	141.4	47.78	2 / 2
tak-1	674.2	348.8	48.26	1 / 1
Over all tune			15.7	73.4%
bessel-1	1170.8	1027.0	12.28	13 / 13
cheb-1	2065.8	928.0	55.08	5 / 5
deriv-1	104.0	106.2	−2.12	0 / 1
disj2-1	2216.4	1682.8	24.08	16 / 19
factsq-1	132.0	131.0	0.76	2 / 2
list_qs2-1	1871.2	1777.0	5.03	1 / 4
merge-1	111.0	109.2	1.62	0 / 1
prime1-1	412.0	413.0	−0.24	2 / 7
queenk-1	1034.0	983.0	4.93	3 / 3
Over all test			11.3	76.4%

Table 4: Absolute performance (μsec) using cc -O4*

returned in memory, this algorithm usually makes the right decisions, and the performance of the resulting code is no worse than that of the traditional scheme for returning outputs via memory; for programs where registers can be used advantageously for output arguments, our algorithm produces code that is significantly faster than the traditional scheme. Although our algorithm can take advantage of additional information such as frequency estimates (see Section 5.3) to yield even better speedups, the bulk of the improvement arises from careful preservation of opportunities to return values in registers and use of cost estimates for different alternatives, resulting in code that is frequently better than any homogeneous return policies or the heterogeneous policy used in the Aquarius compiler.

A Estimating Execution Frequencies

Static execution frequencies of procedures in a program are compile-time projections of the expected number of times that a given procedure or clause will executed in a single invocation of the program. These projections can be used to decide which procedures are the most likely to benefit from optimizations, and which can be "sacrificed" for a greater speedup overall. It is important to note that the purpose of these estimates is not really to predict accurately how many times a branch will be executed or a procedure called, but to guide optimization decisions by giving (possibly conservative) relative execution frequencies for different program fragments. For a discussion of this issue in the context of compilers for traditional languages, see [2, 15, 17, 25, 26]; techniques for estimating execution frequencies of logic programs from their call graph structure are discussed in [9, 20]. An alternative is to profile the program on sample inputs to estimate execution frequencies: as the results of Gorlick and Kesselman [11] indicate, the overhead for this approach is also small.

Our implementation estimates execution frequencies by examining the call graph of the program, assuming that loops will execute some fixed number of times and branches have an equal probability of being taken, and assigning each procedure a frequency based on its loop depth and branch position. In the case of a committed-choice logic language, call graphs are in the form of an OR/AND tree, where OR-nodes represent procedures, and have as their children nodes that represent the clauses that define the procedure. Clause nodes have as children the nodes for each procedure that the clause calls. The effect of the OR/AND tree is that on entering a procedure node only one of the children (clauses) will be chosen and executed, while for the chosen clause node all the children (called procedures) will be executed in turn.

We assume for simplicity a single-rooted call graph, with a root node representing the entry point to the program, which has as its clauses nodes whose sole children are the externally visible procedures of the program. The OR/AND call graph is then computed and used to partition the procedures into *strongly connected components* (SCCs) [14]: sets of procedure nodes which are mutually reachable through the call graph.

For a given procedure node, we assume that each clause has *a priori* equal probability of being chosen. To handle the case of recursion, we assume an input of size L (some integral value), i.e., estimate that the depth of recursion is L.

We start by assigning each clause a frequency based on the assumption that its parent procedure is called once. Define the *fanout* of a clause as the number of distinct procedure-disjoint acyclic execution paths from itself to its parent procedure through the OR/AND tree without leaving its SCC. Fanout captures a measure of "recursivity" of a given clause:

Example A.1 Consider the following procedures `p/2` and `q/2`:

```
p (A, R) :- test0 (A), R is 1.
p (A, R) :- test1 (A), q (A-1, R).
p (A, R) :- test2 (A), p (A-1, R1), p (A-1, R2), R is R1+R2.
q (A, R) :- p (A, R1), R is R1 / 2.
```

`p/2` and `q/2` form a strongly connected component. The first clause of `p/2` is a base case, with fanout 0: if chosen, it cannot be reached again. The second clause has fanout 1, since it calls `q/2`, whose sole clause calls

p/2 once. The third clause has fanout 2, since if chosen, p/2 will be called twice. The single clause of q/2 has fanout 1. □

Considering each procedure in turn, let n_0 be the number of clauses with fanout 0 (the bases of recursion), n_1 be the number with fanout 1 (simple execution chains), and n_2 be the number with fanout exceeding one (execution trees); let n_c represent the total number of clauses. If a clause has fanout 0, we assign it frequency $1/n_c$, on the assumption that each clause is chosen with equal probability. If a clause has fanout 1, we assume that there is a simple loop present which will execute this clause (or another single-recursive clause in the same procedure) L times for the single external input, and hence we give it weight L/n_c (scaling the input size L by the probability n_1/n_c that some fanout 1 clause will be taken on the original entry, and distributing the result over all fanout 1 clauses). We also add $1/n_c$ to a "base weight" b, which characterizes the number of times a base clause is executed because a recursive one was chosen (by the rationale that this clause is chosen with probability $1/n_c$, and choosing this clause eventually causes one base clause to be executed to ground the recursion).

For clauses with fanout $k > 1$, we assume that the procedure is using divide-and-conquer on the size L input, and thus (for simplicity) that the execution tree will be a full k-ary tree of depth $d = \lceil \log_k L \rceil$. The interior nodes of this execution tree are recursive clauses: we assume for the sake of simplicity that they are further invocations of this clause or another with the same fanout. The leaf nodes of the tree correspond to base clauses. Therefore, for such a clause, we add $(k^{d-1} - 1)/(k - 1)n_c$ to its weight (scaling the number of interior nodes by probability of initial choice as above), and k^{d-1}/n_c to b for the base clauses.

After computing the initial weights for the clauses, b is distributed equally amongst the base clauses of the procedure. Having assigned each clause an execution frequency based on this assumption, it is necessary to propagate that information through to generate unconditional frequencies. We start by assigning each procedure a weight (initially 0) measuring the estimated number of times that procedure is called in an average execution of the program. We also give each SCC a count of the number of calls to procedures in that SCC from clauses outside it. Form a set T of nodes all of whose callers have already been visited and hence whose basic frequency has been set; initially, this is the singleton root node, whose weight is one. While T is not empty, remove from it an arbitrary node n. For each clause of n, multiply the clause frequency by the frequency of its parent procedure n, and for each procedure p called from that clause and in a different SCC, add the resulting frequency to p. Furthermore, decrement the count of external calls for p's SCC: when the count becomes 0, all external callees of nodes in the SCC have been considered. At this point, the effect of the externally-induced internal calls is propagated within the SCC—this ensures that all nodes in the SCC receive some non-zero weight even if not all of them are called from outside the SCC. Then the components of the SCC are added to T. When all clauses of a node have been updated, the weight of the node itself is redefined to be the sum of the weights of its clauses.

In the most common case, procedures rarely have clauses with fanout exceeding one, nor recursive clauses with different fanouts. Nontrivial SCCs—i.e., those with more than one procedure, such as Example A.1—are also rare. In the absence of such unusual cases, the numbers generated by the above algorithm are intuitively reasonable for the input size assumed, and in their presence the results are not unbelievable; however, it is the trends inferred by comparing relative frequencies that are most useful for analysis, rather than the raw projected frequencies.

Example A.2 Consider the following implementation of quicksort on lists:

```
:- mode q (in, out).
q(A,^B) :- list(A) | qsort(A,^B,[]).
:- mode qsort (in, out, in).
qsort([X|L],R0,R) :- split(L,X,L1,L2), qsort(L2,R1,R), qsort(L1,R0,[X|R1]).
qsort([],R1,R) :- R1=R.
:- mode split (in, in, out, out).
split([X|L],Y,L1,L2) :- X =< Y, L1 = [X|LL], split(L,Y,LL,L2).
split([X|L],Y,L1,L2) :- X  > Y, L2 = [X|LL], split(L,Y,L1,LL).
split([],X,L1,L2) :- L1 = [], L2 = [].
```

All strongly connected components are trivial, because no two procedures are mutually accessible. The fanouts, weights for SCCs considered in isolation, and weights for each procedure and clause assuming one entry into `q`/2 are in the table below.

Proc:Clause	Fanout	Base Weight	Clause Uncond. Weight	Proc Weight
`q`/2:1	0	1	1	1
`qsort`/3:1	2	$2^{d-2} - 1/2$	$2^{d-2} - 1/2$	2^{d-1}
`qsort`/3:2	0	$2^{d-2} + 1/2$	$2^{d-2} + 1/2$	
`split`/3:1	1	$L/3$	$L(2^{d-1} - 1)/6$	$(2^{d-1} - 1)(2L + 1)/6$
`split`/3:2	1	$L/3$	$L(2^{d-1} - 1)/6$	
`split`/3:3	0	1	$2^{d-2} - 1/2$	

In the first pass, the sole clause of `q`/1 has weight 1, since it is a base for its strongly connected component. The first clause of `qsort`/3 has fanout 2, and is given weight $(2^{d-1} - 1)/2$: probability 1/2 of executing it, times the number of interior nodes of a full binary tree of height $d = \lceil \log_2 L \rceil$. The base clause for `qsort`/3 gets weight $(1/2) + 2^{d-2}$: probability 1/2 of being chosen, plus probability 1/2 times the number of recursion bases resulting from choosing the first clause. `split`/4 gets $L/3$ for its first two clauses, and 1 for the last. Assuming that the sole entry point is through `q`/2, procedure and sole clause for `q`/2 get weight 1. The clauses of `qsort`/3 are multiplied by this weight. □

In other programs, such as Fibonacci, clauses have a fanout exceeding one without using a divide-and-conquer approach to the input; hence, the projected frequency is far lower than the real (exponential) frequency. However, simple analysis does not reveal this, and using an exponential frequency estimation such as L^k generally results in wildly inaccurate projections for the more common divide-and-conquer–style programs, yielding suboptimal choices in decisions based on the frequency estimation.

B Extended Example

This appendix contains an extended example of the analysis performed by our algorithm on a relatively non-trivial program: one which computes the number of solutions to the nqueens problem. The program to be analyzed is:

```
% N Queens program
:- export queen/2.

:- mode queen (in, out).
queen(N,M) :- gen(N,L), queen(L,[],[],[],A), count(A,M).

% Generate list of legitimate placements
:- mode queen (in, in, in, in, out).
queen([C|Cs],NCs,L,S2,S0) :- queen (Cs,[C|NCs],L,S2,S1), check(L,C,1,NCs,Cs,L,S1,S0).
queen([],[],L,S1,S0) :- S0 = [L|S1].
queen([],[_|_],L,S1,S0) :- S0 = S1.

% Check validity of placement
:- mode check (in, in, in, in, in, in, in, out).
check([],C,D,NCs,Cs,L,S1,S0) :- append(NCs,Cs,Ps), queen(Ps,[],[C|L],S1,S0).
check([P|_],C,D,NCs,Cs,L,S1,S0) :- P-C = D, S0 = S1.
check([P|_],C,D,NCs,Cs,L,S1,S0) :- C-P = D, S0 = S1.
check([P|Ps],C,D,NCs,Cs,L,S1,S0) :- P-C =\= D, C-P =\= D, check(Ps,C,D+1,NCs,Cs,L,S1,S0).

% Append first list to second
:- mode append (in, in, out).
append([],X,Y) :- Y=X.
append([A|B],X,Y) :-  Y=[A|Z], append(B,X,Z).

% Generate the initial list of placements
:- mode gen (in, out).
gen(N,L) :- gen(0,N,L).
:- mode gen (in, in, out).
gen(K,N,L) :- K >= N, L=[].
gen(K,N,L) :- K < N, L=[K|Ls], gen(K+1,N,Ls).

% Compute length of a list
:- mode count (in, out).
count(L,N) :- count(L,0,N).
:- mode count (in, in, out).
count([],M,N) :- N = M.
count([_|Xs],M,N) :- count(Xs,M+1,N).
```

The call graph of the program appears in Figure 1. Rectangular nodes denote procedures; their oval children are the clauses for the procedures. Edges connecting procedures to clauses (the dotted edges) are labelled with the estimated frequency of the clause, under an assumption that loops are executed 10 times ($L = 10$ in the algorithm described in Appendix A). The frequency of a particular procedure as a whole appears in parentheses in the label of the box enclosing the procedure and its clauses.

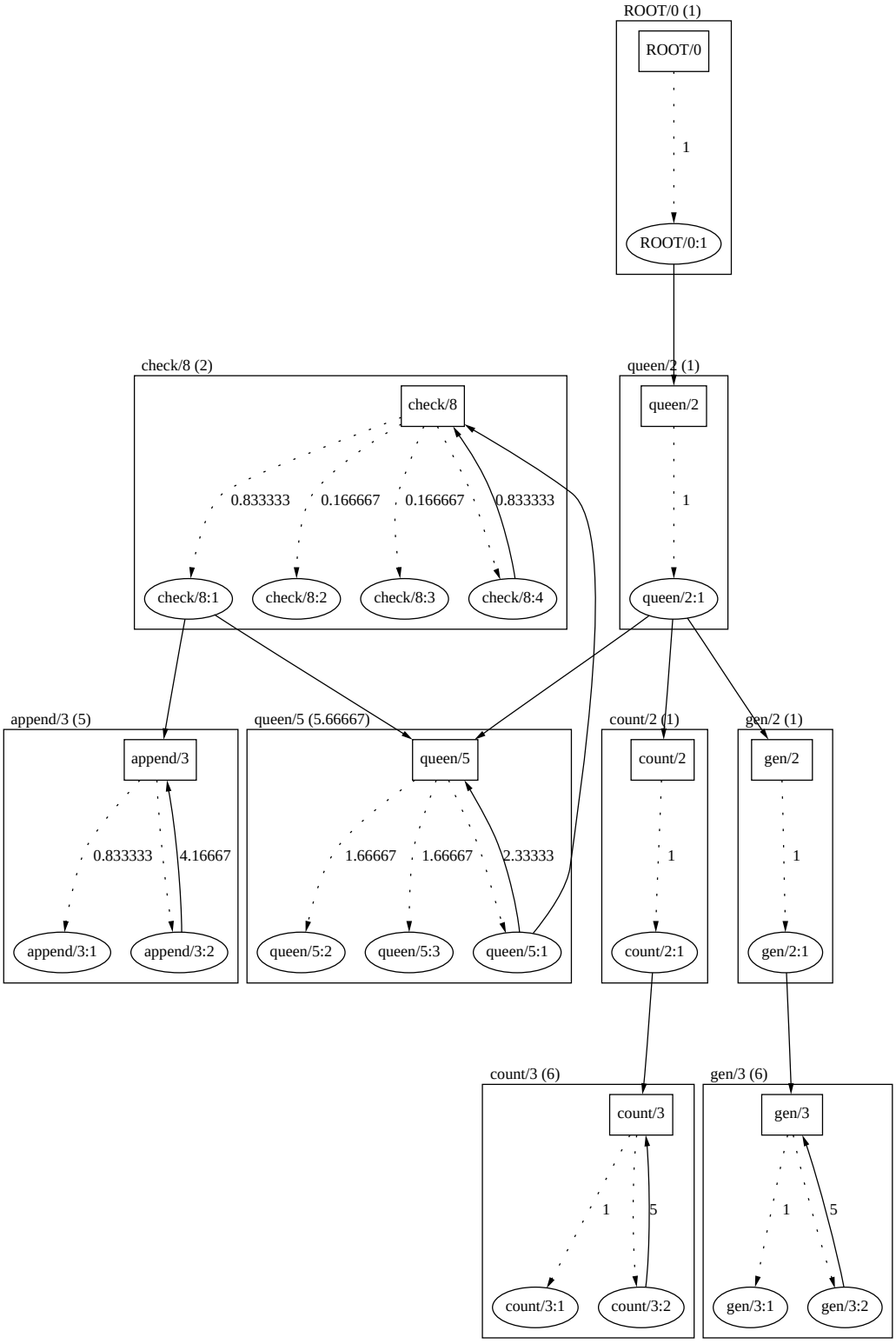

Figure 1: Call graph of queen.j

Consider the analysis for `count`/3. We start with all costs 0. As shown in the program text and the call graph, the procedure has two call sites: one in `count`/2, and one in the second clause of `count`/3. In both cases, the output argument is not used in the remainder of the clause (because the call site is a tail call), so the use-point analysis does not change the cost vector.

The definition point analysis looks at both clauses in turn. For the first clause, there is an assignment, since this clause grounds the recursion. Since this clause has weight 1, we add C_a to the cost vector entry for a memory return. In the second clause, the definition is through a call, so we don't change the cost vectors.

For `queen`/5, there are three call sites, one each in `queen`/2, `check`/8, and `queen`/5. In `queen`/2, the output of the call (`A`) will be used in the first argument to a succeeding call. Therefore, we add $C_p + C_l$ to the cost of choosing memory, and C_x to the cost of all registers except $r0$. The call site in `check`/8 is a tail call, so no changes are made for that, but the output `S1` for the site in `queen`/5 is used in the 7th argument position of the following call to `check`/8. Therefore, memory gets $2.333(C_p + C_l)$ and all registers except $r6$ get $2.333C_x$.

Each clause of `queen`/5 has a definition point for the output `S0`. For the first clause, the definition point (call to `check`/8) follows a non-survive goal, the call to `queen`/5. Therefore, $2.333(C_s + C_l)$ is added to memory to account for saving the pointer to the storage slot over the call. Again, we can make no assumptions about the return location chosen by `check`/8. The remaining two clauses ground the recursion, so add $1.667C_a$ to memory.

The last procedure we will look at in depth is `append`/3, which is called from both `check`/8 and itself. The site in `check`/8 passes its output `Ps` in $r0$ to the following call to `queen`/5: memory gets an additional $0.833(C_p + C_s)$, and registers other than $r0$ get $0.833C_x$.

The recursive call site in clause 2 of `append`/3 is more interesting, and somewhat subtle. The output `Z` of the call is assigned in a tail call, but we notice that if the output is returned in a register, the tail call must be deoptimized to store `Z` into the list which is being returned from the caller. Note that if the tail call can be optimized, this clause does not need an environment allocation. Therefore, all register locations are assigned $4.167(C_{ta} - C_p)$, to account for deoptimizing the tail call, but not needing to initialize the return location for the assignment. (The subtraction of C_p is an implementation-specific optimization: requirement of transitive non-suspension and lack of a garbage collector in `jc` ensures that the cons cell will not be accessed by anybody until the value `Z` has been computed and stored in it, so the other reasons described in Section 3.1.1 for initializing storage cells are unnecessary.)

In the first clause of `append`/3, the definition site grounds the recursion, so adds $0.833C_a$ to memory. In the second clause, the output `Y` is defined in an assignment prior to a call, so memory requires an additional $4.167C_a$. Registers must be saved over the call, so cost a store-and-load ($4.167(C_s + C_l)$), and also mark the flag indicating that register returns will force a tail call deoptimization.

The other procedures are treated similarly. The cost vectors for all outputs after the first pass of the algorithm, using the frequency estimations in Figure 1 and parameters for `cc` from Table 1, are given in Table 5.

With basic cost estimates gathered, the second pass of the algorithm can proceed to place outputs in preferred locations. Tracing through our example procedures, the highest frequency is associated with `count`/3. Although the second clause has a tail call, because it is to itself and passes the value through directly, there is no worry about deoptimizing the call to move values about. The lowest cost choice is then

Output	Mem	r0	r1	r2	r3	r4	r5	r6	r7
count/3 N	594.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
gen/3 L	3564.0	9325.0	9325.0	9325.0	9325.0	9325.0	9325.0	9325.0	9325.0
queen/5 S0	5294.3335	2.3333	3.3333	3.3333	3.3333	3.3333	3.3333	1.0	3.3333
append/3 Y	3449.166	7770.83	7771.67	7771.67	7771.67	7771.67	7771.67	7771.67	7771.67
check/8 S0	697.1667	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
gen/2 L	575.0	0.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
count/2 N	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Table 5: Cost vectors after first pass of algorithm

taken, and the output N is placed in register $r0$.

For queen/5, the first clause has a tail call to check/8, but this procedure has a lower frequency, and hasn't been assigned yet. Therefore, although we want queen/5 to return its value wherever check/8 returns its value (to preserve the tail call), we don't know what this preferred location is. The cost vector remains unmodified, and since no other tail calls interfere with the costs, the lowest cost location $r6$ is chosen.

For append/3, recall that in pass 1 we marked all non-memory locations as forcing a deoptimization of the tail call in clause 2. Therefore, all register locations receive an additional cost of $4.167C_{ta} \approx 7013$, on top of their values in Table 5. This makes the lowest cost be memory, which is appropriate for structure returns such as this.

The last interesting choice made in the second pass is in the assignment of location to S0 for check/8. First, there is a tail call in clause 1 to queen/5, which must be deoptimized unless S0 is returned in $r6$—this adds $0.833C_{tn}$ to all other locations. Also, there is a tail call *to* check/8 in queen/5, and the same agreement must be present there, adding $2.333C_{tn}$ to the other locations. As a result, $r6$ is chosen, preserving both tail calls.

The cost vectors used for the final placement are shown in Table 6.

Output	Mem	r0	r1	r2	r3	r4	r5	r6	r7
count/3 N	594	4	0	0	0	0	0	0	0
gen/3 L	3564	17740	17740	17740	17740	17740	17740	17740	17740
queen/5 S0	5294.33	2.33333	3.33333	3.33333	3.33333	3.33333	3.33333	1	3.33333
append/3 Y	3449.17	14783.3	14784.2	14784.2	14784.2	14784.2	14784.2	14784.2	14784.2
check/8 S0	3911.33	3214.17	3214.17	3214.17	3214.17	3214.17	3214.17	0	3214.17
gen/2 L	575	1683	1684	1684	1684	1684	1684	1684	1684
count/2 N	1683	1015	2698	2698	2698	2698	2698	2698	2698

Table 6: Cost vectors after final pass of algorithm

References

[1] ANSI. *ANSI X3.159-1989 (Programming Language–C).* American National Standards Institute, 1990.

[2] T. Ball and J. Larus, "Optimally Profiling and Tracing Programs", *Proc. 19th. ACM Symp. on Principles of Programming Languages*, Albuquerque, NM, Jan. 1992, pp. 59–70.

[3] 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.

[4] F. C. Chow, "A Portable Machine-Independent Global Optimizer: Design and Measurements", Ph.D. Dissertation, Stanford University, Dec. 1983. Technical Report No. 83-254.

[5] W. Clinger, "The Scheme 311 Compiler: An Exercise in Denotational Semantics", *Proc. 1984 ACM Symp. on Lisp and Functional Programming*, Austin, TX, Aug. 1984, pp. 356–364.

[6] K. Clark and S. Gregory, "PARLOG: Parallel Programming in Logic", *ACM Transactions on Programming Languages and Systems* vol. 8 no. 1, Jan. 1986, pp. 1-49.

[7] J. W. Davidson and A. M. Holler, "A Study of a C Function Inliner", *Software Practice and Experience* vol. 18, pp. 775–790, 1988.

[8] S. K. Debray, "Static Inference of Modes and Data Dependencies in Logic Programs", *ACM Transactions on Programming Languages and Systems* vol. 11, no. 3, June 1989, pp. 419-450.

[9] S. K. Debray, "A Remark on Tick's Algorithm for Compile-Time Granularity Analysis", *Logic Programming Newsletter* vol. 3 no. 1, July 1989.

[10] S. K. Debray, "Efficient Dataflow Analysis of Logic Programs", *J. ACM* vol. 39 no. 4, Oct. 1992, pp. 949–984.

[11] M. M. Gorlick and C. F. Kesselman, "Timing Prolog Programs Without Clocks", *Proc. Fourth IEEE Symp. Logic Programming*, San Francisco, CA, Sept. 1987, pp. 426-432. IEEE Press.

[12] 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.

[13] B. Hausman, "Turbo Erlang: An Efficient Implementation of a Concurrent Programming Language", manuscript, Ellemtel Telecommunications Systems Labs., Sweden, March 1993.

[14] Matthew S. Hecht. *Flow Analysis of Computer Programs.* North-Holland New York, 1977.

[15] S. McFarling, "Program Optimization for Instruction Caches", *Proc. Third Int. Symp. on Architectural Support for Programming Languages and Operating Systems*, pp. 183–191.

[16] K. Marriott, M. Garcia de la Banda, and M. Hermenegildo, "Analyzing Logic Programs with Dynamic Scheduling", *Proc. 21st. ACM Symp. on Principles of Programming Languages*, Portland, Oregon, Jan. 1994, pp. 240–253.

[17] K. Pettis and R. C. Hansen, "Profile Guided Code Positioning", *Proc. SIGPLAN-90 Conf. on Programming Language Design and Implementation*, White Plains, NY, June 1990, pp. 16–27.

[18] M. L. Powell, "A Portable Optimizing Compiler for Modula-2", *Proc. SIGPLAN-84 Symp. on Compiler Construction*, Montreal, June 1984, pp. 310–318. SIGPLAN Notices vol. 19 no. 6.

[19] V. Saraswat, K. Kahn, and J. Levy, "Janus: A step towards distributed constraint programming", in *Proc. 1990 North American Conference on Logic Programming*, Austin, TX, Oct. 1990, pp. 431-446. MIT Press.

[20] E. Tick, "Compile-time Granularity Analysis for Parallel Logic Programming Languages", *Proc. Int. Conf. on Fifth Generation Computer Systems*, Tokyo, Japan, Nov. 1988, pp. 994-1000.

[21] K. Ueda, "The Mode System of Moded Flat GHC", *Proc. ILPS-93 Post-conference Workshop on Global Compilation of Logic Programs*, Vancouver, Canada.

[22] K. Ueda, "Guarded Horn Clauses", in *Concurrent Prolog: Collected Papers*, vol. 1, ed. E. Shapiro, pp. 140-156, 1987. MIT Press.

[23] K. Ueda and M. Morita, "A New Implementation Technique for Flat GHC", *Proc. Seventh International Conference on Logic Programming*, Jerusalem, June 1990, pp. 3-17. MIT Press.

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

[25] D. W. Wall, "Global Register Allocation at Link-time", *Proc. SIGPLAN-86 Conf. on Compiler Construction*, June 1986, pp. 264–275.

[26] D. W. Wall, "Predicting Program Behavior Using Real or Estimated Profiles", *Proc. SIGPLAN-91 Conf. on Programming Language Design and Implementation*, June 1991, pp. 59–70.