Call Forwarding: A Simple Interprocedural Optimization Technique for Dynamically Typed Languages *

Koen De Bosschere,† Saumya K. Debray,‡ David Gudeman,‡ Sampath Kannan‡

† Electronics Laboratory
Rijksuniversiteit Gent
B-9000 Gent, Belgium

‡ Department of Computer Science
The University of Arizona
Tucson, AZ 85721, USA

TR 93-12

Abstract

This paper discusses *call forwarding*, a simple interprocedural optimization technique for dynamically typed languages. The basic idea behind the optimization is straightforward: find an ordering for the "entry actions" of a procedure, and generate multiple entry points for the procedure, such that the savings realized from different call sites bypassing different sets of entry actions, weighted by their estimated execution frequencies, is as large as possible. We show that the problem of computing optimal solutions to arbitrary call forwarding problems is NP-complete, and describe efficient heuristics for the problem. Experimental results indicate that (*i*) the heuristics are effective, in that the solutions produced are generally optimal or close to optimal; and (*ii*) the resulting optimization is effective, in that it leads to significant performance improvements for a number of benchmarks tested.

*The work of K. De Bosschere was supported by the National Fund for Scientific Research of Belgium and by the Belgian National incentive program for fundamental research in Artificial Intelligence, initiated by the Belgian State—Prime Minister's office—Science Policy Programming. The work of S. K. Debray and D. Gudeman was supported in part by the National Science Foundation under grant number CCR-9123520. The work of S. Kannan was supported in part by the National Science Foundation under grant number CCR-9108969. E-mail addresses: `kdb@lem.rug.ac.be` (K. De Bosschere), `debray@cs.arizona.edu` (S. K. Debray), `gudeman@cs.arizona.edu` (D. Gudeman), `kannan@cs.arizona.edu` (S. Kannan).

1 Introduction

The code generated for a function or procedure in a dynamically typed language typically has to carry out various type and range checks on its arguments before it can operate on them. These runtime tests can incur a significant performance overhead. As a very simple example, consider the following function to compute the average of a list of numbers:

```
ave(L, Sum, Count) =
     if null(L) then Sum/Count else ave(tail(L), Sum+head(L), Count+1)
```

In a straightforward implementation of this function, the code generated checks the type of each of its arguments each time around the loop: the first argument must be a (empty or non-empty) list, while the second and third arguments must be numbers.[1] Notice, however, that some of this type checking is unnecessary: the expression `Sum+head(L)` evaluates correctly only if `Sum` is a number, in which case its value is also a number; similarly, `Count+1` evaluates correctly only if `Count` is a number, and in that case it also evaluates to a number. Thus, once the types of `Sum` and `Count` have been checked at the entry to the loop, further type checks on the second and third arguments are not necessary.

The function in this example is tail recursive, making it easy to recognize the iterative nature of its computation and use some form of code motion to move the type check out of the loop. In general, however, such redundant type checks may be encountered where the definitions are not tail recursive and where the loop structure is not as easy to recognize. An alternative approach, which works in general, is to generate multiple entry points for the function `ave`, so that a particular call site can enter at the "appropriate" entry point, bypassing any code it does not need to execute. In the example above, this would give exactly the desired result: tail call optimization would compile the recursive call to `ave` into a jump instruction, and noticing that the recursive call does not need to test the types of its second and third arguments, the target of this jump would be chosen to bypass these tests.

However, notice that in the example above, even if we generate multiple entry points for `ave`, the optimization works *only if the tests are generated in the right order*: since it is necessary to test the type of the first argument each time around the loop, the tests on the second and third arguments cannot be bypassed if the type test on the first argument precedes those on the other two arguments. As this example illustrates, the order in which the tests are generated influences the amount of unnecessary code that can be bypassed at runtime, and therefore the performance of the program.

In general, functions and procedures in dynamically typed languages contain a set of actions, such as type tests and initialization actions, that are executed at entry and can be carried out in any order. There are a number of different call sites for each function, and at each call site we have some information about the actual parameters at that call site, allowing that call to skip some of these "entry actions". Moreover, each call site has a different execution frequency (estimated, for example, from profile information or from the structure of the call graph). Now an order for the entry actions that is good for one call site, in terms of the number of unnecessary entry actions that can be skipped, may not be as good for another call site, since in general, different call sites

[1] In reality, the generated code would distinguish between the numeric types `int` and `float`, e.g., using "message splitting" techniques as in [4, 5]—the distinction is not important here, and we assume a single numeric type for simplicity of exposition.

have different information available about the actual parameters. A good compiler should therefore attempt to find an ordering on the entry actions that maximizes the benefits, over all call sites, due to bypassing unnecessary code. We refer to determining such an order for the entry actions and then "forwarding" the branch instructions at different call sites so as to bypass unnecessary code as "call forwarding".

While many systems compile functions with multiple entry points, we do not know of any that attempt to order the entry actions carefully in order to exploit this to the fullest. In this paper, we address the problem of determining a "good" order for the set of tests a function or procedure has to carry out. We show that generating an optimal order is NP-complete in general, and give efficient polynomial-time heuristics for selecting an ordering. The result generalizes a number of optimizations for traditional compilers, such as jump chain collapsing and invariant code motion out of loops. Experimental results indicate that (*i*) our heuristics are good, in that the orderings they generate are usually not far from the optimal; and (*ii*) the resulting optimization is effective, in the sense that it typically leads to significant speed improvements.

The issues and optimizations discussed in this paper are primarily at the intermediate code, or virtual machine instruction, level: for this reason, we do not make many assumptions about the source language, except that it is dynamically typed. This covers a wide variety of languages, e.g., functional programming languages such as Lisp and Scheme (e.g., see [18]), logic programming languages such as Prolog [3], Strand [6], GHC [19] and Janus [11, 15], imperative languages such as Icon [10] and SETL [16], and object-oriented languages such as Smalltalk [9] and SELF [5]. The optimization we discuss is likely to be most beneficial for languages and programs where procedure calls are common, and which are therefore liable to benefit significantly from reducing the cost of procedure calls. The assumption of dynamic typing implies that the code generated for a function or procedure will contain type tests, initialization actions (especially for variadic procedures), etc. Such tests on the parameters to a procedure are typically carried out at the entry to the procedure. Moreover, they can typically be carried out in any of a number of different "legal" orders (in general, not all orderings of entry actions may be legal, since some actions may depend on the outcomes of others—for example, the type of an expression `head(x)` cannot be checked until `x` has been verified to be of type list). The code generated for a procedure therefore consists of a set of entry actions in some order, followed by code for its body. For simplicity in the discussion that follows, we assume that each entry action corresponds to a single virtual machine instruction.

2 The Call Forwarding Problem

As discussed in the previous section, the code generated for a procedure consists of a set of entry actions, which can be carried out in a number of different legal orders, followed by the code for its body. Each procedure has a number of call sites, and at each call site there is some information about the actual parameters for calls issued from that site, specifying which entry actions must be executed and which may be skipped.[2] This is modelled by associating, with each call site, a set of entry actions that must be executed by that call site. Moreover, each call site has associated with it an estimate of its execution frequency: such estimates can be obtained from profile information, or from the structure of the call graph of the program [2, 13, 14, 21]. Finally, different entry actions may require a different number of machine instructions to execute, and therefore have a different cost.

[2]The precise mechanism by which this information is obtained, e.g., dataflow analysis, user declarations, etc., is orthogonal to the issues discussed in this paper, and so is not addressed here.

The *call forwarding problem* is the problem of determining a "good" order for the entry actions of a procedure so that the savings accruing from bypassing unnecessary entry actions over all call sites for that procedure, weighted by execution frequency, is as large as possible. This problem can be formulated in the abstract as follows:

Definition 2.1 A call forwarding problem is a 4-tuple $\langle E, C, w, cost \rangle$, where:

- E is a finite set (corresponding to the entry actions);
- C is a collection of subsets of E (corresponding to the entry actions that each call site must execute);
- $w : C \longrightarrow \mathcal{N}$, where $\mathcal{N}$ is the set of natural numbers, is a function that maps each call site to its "weight", i.e., execution frequency; and
- $cost : E \longrightarrow \mathcal{N}$ represents the cost, in machine instructions, of each element of E.

∎

A *solution* to a call forwarding problem $\langle E, C, w, cost \rangle$ is a permutation π of E, i.e., a 1-1 function $\pi : E \longrightarrow \{1, \ldots, |E|\}$. The cost of a solution π is, intuitively, the total number of machine instructions executed, over all call sites, given that the entry actions are generated in the order π:

Definition 2.2 Given a call forwarding problem $\langle E, C, w, cost \rangle$ and a solution π for it, let $Not\_Skipped_\pi(c)$ denote the set of instructions that must be executed by a call site $c \in C$ under the solution π:

$$Not\_Skipped_\pi(c) = \{I \in E \mid \exists I' \in E : \pi(I') \leq \pi(I) \wedge I' \in c\}.$$

Then, the cost of the solution π is given by $cost(\pi) = \sum_{c \in C} \{w(c) \cdot cost(I) \mid I \in Not\_Skipped_\pi(c)\}$.
∎

The problem can be generalized by allowing code to be copied from a procedure to the call sites for that procedure. As an example, suppose we have a procedure with entry actions a and b, and two call sites: A, which can skip a but must execute b; and B, which can skip b but must execute a. Suppose the entry actions are generated in the order $\langle a, b \rangle$, then call site A can skip a, but B cannot skip b and therefore executes unnecessary code (a symmetric problem arises if the other possible order is chosen). A solution is to copy the entry action a at the call site B, i.e., execute the entry action at B before jumping to the callee.

If we allow arbitrarily many entry actions to be copied to call sites in this manner, then it is trivial to generate an optimal solution to any call forwarding problem: simply copy to each call site the entry actions that call site must execute, then branch into the callee bypassing all entry actions at the callee. This obviously produces an optimal solution, since each call site executes exactly those entry actions that it must execute, and can be done efficiently in polynomial time. However, it has the problem that such unrestricted copying can lead to code bloat, since there may be many call sites for a procedure, each of them getting a copy of most of the entry actions for that procedure. A reasonable solution to this problem is to impose a bound on the number of entry actions that

can be copied to any particular call site. To simplify the discussion that follows, we will restrict our attention to the case where this bound is zero, i.e., no copying of code to call sites is allowed. However, our results generalize without difficulty to the more general case where copying of code is allowed.

This generalization also presents a dual view of the problem. The discussion above has been in terms of choosing an entry point for each call site, possibly after copying some entry actions to the call site before actually jumping to the callee. In this view, each time an action is copied to a call site, the space cost of the program increases, presumably accompanied by a decrease in the time cost. A dual view would be to start with each call site executing each and only those entry actions that it needs to before branching to the callee (a situation that is optimal in terms of the number of instructions executed but which may be expensive in terms of code space), and "moving actions back" from the call sites to the callee. Intuitively, these two approaches reflect the manner in which we choose to approach the tradeoff between code size and speed.

3 Algorithmic Issues

We first consider the complexity of determining optimal solutions to call forwarding problems. The following result shows that the existence of efficient algorithms for this is unlikely:

Theorem 3.1 *The determination of an optimal solution to a call forwarding problem is NP-complete. It remains NP-complete even if every entry action has equal cost.*

Proof By reduction from the Optimal Linear Arrangement problem, which is known to be NP-complete [7, 8]. See the Appendix for details. ■

We therefore seek polynomial time heuristics for call forwarding that are efficient and achieve good solutions for common cases.

3.1 Heuristic 1: Partition-and-Sort

The first heuristic we consider is an intuitively obvious one: for each procedure, use the weights of its different call sites to compute the cumulative weight of each entry action, then sort the entry actions so that those with smaller weight—representing actions that need not be executed by the "important" call sites, i.e., the majority of the call sites or the most frequently executed ones—are generated first, and can therefore be skipped by them. There is one point that has to be taken into account: it may happen that two different entry actions have the same cumulative weight even though the call sites that need to execute them are very different. If the entry actions are simply sorted by cumulative weight, either of these instructions may be generated first. However, this can sometimes produce poor orderings, because when choosing an ordering for a group of entry actions with the same cumulative weight, this simple heuristic does not group together those actions that can all be skipped by a set of call sites. This can be rectified by grouping together sets of entry actions, as follows:

1. Partition the entry actions of a procedure so that each partition is a maximal set of instructions that have to be executed by some set of call sites, then sort these partitions by cumulative weight. Given a call forwarding problem $\langle E, C, w, cost \rangle$, the partitions S can be computed as follows: starting with $S = C$, repeatedly consider pairs of distinct elements c, c' in S that overlap, i.e., $c \cap c' \neq \emptyset$, and replace each such pair in S by $\{c \cap c', c \setminus c', c' \setminus c\}$, until no two distinct elements of S overlap. This can be implemented efficiently using bit-vector operations.

2. Compute the cumulative weight of each partition is its total execution frequency multiplied by the sum of the costs of the actions in it.

3. Sort the partitions by cumulative weight and emit the actions in the partitions in this sorted order. The actions within a partition can be generated in any order, since every action within any particular partition must be executed by the same set of call sites, so their relative ordering is not important.

While this algorithm is very simple, both conceptually and for implementation purposes, it has some drawbacks arising from the way cumulative weights are computed. These are illustrated by the following examples:

Example 3.1 Suppose we have three call sites: A, which has weight 10 and must execute the actions $\{a1, a2, a3, a4, a5\}$; B, which also has weight 10 and must execute the actions $\{a3, a4, a5, a6, a7\}$; and C, which has weight 15 and must execute the actions $\{a8, a9\}$. The partitions computed by the partition-and-sort heuristic are $\{a1, a2\}$, $\{a3, a4, a5\}$, $\{a6, a7\}$, and $\{a8, a9\}$, with cumulative weights 20, 60, 20, and 30 respectively. Sorted by cumulative weight and linearized, this produces the solution $\langle a1, a2, a6, a7, a8, a9, a3, a4, a5\rangle$. The problem with this is that the combined weight of call sites A and B cause the partition $\{a3, a4, a5\}$ to have a high cumulative weight, and as a result C is forced to execute these actions even though it does not need to. The cost of this solution is 235. A considerably better solution, with weight 190, is $\langle a1, a2, a3, a4, a5, a6, a7, a8, a9\rangle$. □

A more serious drawback, illustrating how cumulative weight computations can sometimes produce poor results, is illustrated by the following example.

Example 3.2 Suppose we have two call sites, each with weight 10: A must execute the actions $\{a, b, c, d, e\}$, while B has to execute only $\{f\}$. The partitions in this case are $\{a, b, c, d, e\}$ and $\{f\}$, with cumulative weights 50 and 10 respectively. The solution produced by the partition-and-sort heuristic, therefore, is $\langle f, a, b, c, d, e\rangle$, which has a cost of $50 + 60 = 110$. Unfortunately, this is precisely the wrong order: a much better solution is $\langle a, b, c, d, e, f\rangle$, whose cost is $60 + 10 = 70$. □

These problems are addressed by the greedy algorithm described in the next section.

3.2 Heuristic 2: Greedy Choice of Actions

The next heuristic we consider is a greedy one: the general idea is to pick actions one at a time, at each step choosing an action that minimizes the cost to be paid at that step. The algorithm maintains a list of call sites that have not been "hit" upto that point, i.e., call sites that do not need to execute any of the actions chosen upto that point—such call sites are said to be *active*. The weight of an action, at any point in the algorithm, is computed as the sum of the weights of the active call sites that need to execute that action. The algorithm is simple: it repeatedly picks an action of least weight, then updates the list of active call sites and the weights of the other actions, until all actions have been enumerated. The algorithm is described in Figure 1.

It is straightforward to extend this algorithm to the more general situation where we are willing to copy upto k instructions from the procedure to each of the call sites: we simply redefine the notion of "active call site," so that a call site remains active until k actions that it needs to execute have been chosen. After more than k actions have been encountered from a particular call site, it ceases to be active.

Input: A call forwarding problem $I = \langle E, C, w, cost \rangle$.

Output: A solution to I, i.e., a permutation π of E.

Method:

```
begin
    Active_Sites := C;
    Instrs := E;
    π := ε;                          /* the empty sequence */
    while Instrs ≠ ∅ do
        for each I ∈ Instrs do
            compute the weight of I as (∑{w(c) | c ∈ Active_Sites ∧ I ∈ c})/cost(I);
        od;
        I := an element of Instrs with the least weight so computed;
        π := append I to the end of π;
        Instrs := Instrs \ {I};
        Active_Sites := Active_Sites \ {c ∈ C | I ∈ c};
    od;
    return π;
end
```

Figure 1: The Greedy Choice of Actions Heuristic for Call Forwarding

The working of this algorithm is illustrated by the following example.

Example 3.3 Consider again the call forwarding problem of Example 3.1. There are three call sites: A, which has weight 10 and must execute the actions $\{a1, a2, a3, a4, a5\}$; B, which also has weight 10 and must execute the actions $\{a3, a4, a5, a6, a7\}$; and C, which has weight 15 and must execute the actions $\{a8, a9\}$. Assume, for the sake of simplicity, that each action has cost 1.

Initially, all call sites are active, so the weights computed for the actions are as follows: $a1 : 10$; $a2 : 10$; $a3 : 20$; $a4 : 20$; $a5 : 20$; $a6 : 10$; $a7 : 10$; $a8 : 15$; $a9 : 15$. The algorithm picks one of the actions with the smallest weight, say $a7$.

At this point, because $a7$ was picked, the call site B becomes inactive, so the set of active sites now is $\{A, C\}$. The weights for the remaining actions therefore change to the following: $a1 : 10$; $a2 : 10$; $a3 : 10$; $a4 : 10$; $a5 : 10$, $a6 : 0$; $a8 : 15$; $a9 : 15$. Note that since $a6$ does not need to be executed by either active site, its weight has gone to 0. The least weight action now is $a6$, which is picked next. This does not change the set of active sites.

At the next step, there are five actions that have least cost, namely, $\{a1, a2, a3, a4, a5\}$, and one of these—say, $a2$, is picked. Because of this, call site A becomes inactive, so the weights of the remaining actions become: $a1 : 0$; $a3 : 0$; $a4 : 0$; $a5 : 0$; $a8 : 15$; $a9 : 15$. The actions $\{a1, a3, a4, a5\}$ are therefore picked after this in some order, after which $a8$ and $a9$ are selected. The solution that is eventually produced is $\langle a7, a6, a2, a1, a3, a4, a5, a8, a9 \rangle$. The cost of this solution is 190, compared to the solution of cost 235 produced by the partition-and-sort heuristic. □

3.3 A Hybrid Strategy

Experimental results suggest that in most cases, the information that is known about the actual parameters at the different call sites for a procedure in a program tends to be similar, and that in most cases, the number of partitions obtained in the partition-and-sort heuristic is not very large.[3] A simple strategy, then, is to partition the entry actions as in the partition-and-sort heuristic to determine how many partitions there are: if this number is less than some predetermined threshold, say 5, then an optimal solution can be obtained by considering all possible permutations of these partitions (recall that the ordering of actions within a partition does not affect the cost of a solution)—because the number of partitions is small, this should not take much time. If the number of partitions exceeds the threshold, then we can use one of the heuristics discussed earlier.

4 An Example

In this section we consider in more detail the `ave` function from Section 1 to see the effect of call forwarding on the code generated. The function is defined as follows:

```
ave(L, Sum, Count) =
     if null(L) then Sum/Count else ave(tail(L), Sum+head(L), Count+1)
```

Assume that, as in many modern Lisp and Prolog implementations, parameters are passed in (virtual machine) registers, so that the first parameter is in register `Arg1`, the second parameter in register `Arg2`, and so on. Figure 2(a) gives the intermediate code that might be generated in a straightforward way. (In reality, the generated code would distinguish between the numeric types `int` and `float`, e.g., using "message splitting" techniques as in [4, 5]—the distinction is not important here, and we assume a single numeric type for simplicity of exposition.) The first three instructions of `ave` are entry actions that can be executed in any order. Moreover, at the (recursive) call site for `ave`, we know from the semantics of the `add` instruction that `Arg1` and `Arg2` are both numbers, so that the second and third entry action may be bypassed by each of these call sites. Call forwarding therefore orders the entry actions so that the tests on `Arg2` and `Arg3` come first, and can be skipped by the recursive call to `ave`, resulting in the code of Figure 2(b).

Notice that the type tests on the second and third arguments have effectively been "hoisted" out of the loop. Moreover, this has been accomplished, not by recognizing and dealing with loops in some special way, but simply by using the information available at the call sites. It is applicable, therefore, even to computations that are not iterative (i.e., tail recursive). Notice also that if we know that the second and third parameter to `ave` is a number at every call site for the function, e.g., if `ave` is an auxiliary function called from a "top level" function `average`, defined as

```
average(L) = ave(L, 0, 0)
```

then the code for the type tests on the second and third arguments is dead, and can be discarded during dead code elimination.

[3] In our experiments, the number of different partitions was rarely larger than 3 or 4. This allowed us to compute the instruction counts for optimal solutions in Table 1 in a reasonable amount of time—otherwise, considering all possible permutations of actions would have been prohibitively expensive, even for programs of modest size.

```
ave:  if !List(Arg1) goto Err
      if !Number(Arg2) goto Err
      if !Number(Arg3) goto Err
      if Arg1 == NIL goto L1
      t1 := head(Arg1)
      Arg1 := tail(Arg1)
      Arg2 := add(Arg2, t1)
      Arg3 := add(Arg3, 1)
      goto ave
L1 :  t1 := div(Arg2, Arg3)
      return t1
```

(a) Before Call Forwarding

```
ave:  if !Number(Arg2) goto Err
      if !Number(Arg3) goto Err
L0 :  if !List(Arg1) goto Err
      if Arg1 == NIL goto L1
      t1 := head(Arg1)
      Arg1 := tail(Arg1)
      Arg2 := add(Arg2, t1)
      Arg3 := add(Arg3, 1)
      goto L0
L1 :  t1 := div(Arg2, Arg3)
      return t1
```

(b) After Call Forwarding

Figure 2: The Effect of Call Forwarding on Intermediate Code for the `ave` function

5 Experimental Results

We ran experiments on a number of small benchmarks to gauge (*i*) the efficacy of our heuristics, i.e., the quality of their solutions compared to the optimal; and (*ii*) the efficacy of the optimization, i.e., the performance improvements resulting from it. The numbers presented reflect the performance of `jc` [11], an implementation of a logic programming language called Janus [15] on a Sparcstation-1.[4] This system is currently available by anonymous FTP from `cs.arizona.edu`.

Table 1 gives, for each benchmark, the number of machine instructions that would be executed over all call sites *for the entry actions in the procedures only*, using (*i*) no call forwarding; (*ii*) call forwarding using the partition-and-sort heuristic; (*iii*) call forwarding using the greedy choice of instructions heuristic; and (*iv*) optimal call forwarding. The weights for the call sites were estimated using the structure of the call graph: we assumed that on the average, each loop iterates about 10 times, and the branches of a conditional are taken with equal frequency. While the optimizations were carried out at the intermediate code level, we used counts of the number of Sparc assembly instructions for each intermediate code instruction, together with the execution frequencies estimated from the call graph structure, to estimate the runtime cost of the different solutions. The results indicate that in most cases, our heuristics produce results that are optimal or close to optimal: the partition-and-sort heuristic does poorly on some benchmarks, e.g., `tak` and `pi`, but the greedy heuristic has uniformly good performance.

Table 2 gives the improvements in speed resulting from our optimizations, and serves to evaluate the efficacy of call forwarding . The time reported for each benchmark, in milliseconds, is the time taken to execute the program once. This time was obtained by iterating the program long enough to eliminate most effects due to multiprogramming and clock granularity, then dividing the total time taken by the number of iterations. The experiments were repeated 20 times for each benchmark, and the average time taken in each case. Call forwarding accounts for improvements ranging from about 12% to over 45%. Most of this improvement comes from code motion out of inner loops: the vast majority of type tests etc. in a procedure appear as entry actions that are bypassed in recursive

[4]Our implementation uses a variant of call forwarding where entry actions are copied from the callee to the call sites as long as this will allow a later action to be skipped.

calls due to call forwarding, effectively "hoisting" such tests out of inner loops. As a result, much of the runtime overhead from dynamic type checking is optimized away. When combined with other optimizations, such as inlining of arithmetic operations, environment allocation optimization, etc., the combined optimizations give rise to speed improvements typically ranging from 27% to almost 70%.

Table 3 puts these numbers in perspective by comparing the performance of `jc` to Quintus and Sicstus Prolog, two widely used commercial Prolog systems. On comparing the performance numbers from Table 2 for `jc` before and after optimization, it can be seen that the performance of `jc` is competitive with these systems even before the application of the optimizations discussed in this paper. It is easy to take a poorly engineered system with a lot of inefficiencies and get huge performance improvements by eliminating some of these inefficiencies. The point of this table is that when evaluating the efficacy of our optimizations, we were careful to begin with a system with good performance, so as to avoid drawing overly optimistic conclusions.

Finally, Table 4 compares the performance of our Janus system with C code for some small benchmarks.[5] Again, these were run on a Sparcstation 1, with `cc` as the C compiler. The programs were written in the style one would expect of a competent C programmer: no recursion (except in `tak` and `nrev`, where it is hard to avoid), destructive updates, and the use of arrays rather than linked lists (except in `nrev`). It can be seen that even without any global dataflow analysis, `jc` is not very far from the performance of the C code, attaining approximately the same performance as unoptimized C code, and being a factor of between 3 and 4 slower than the code produced by optimizing at level `-O4`, on most benchmarks. On some benchmarks, such as `nrev` (an $O(n^2)$ "naive reverse" program for reversing a linked list of integers), `jc` outperforms unoptimized C and is not much slower than optimized C, even though the C program uses destructive assignment and does not allocate new cons cells, while Janus is a single assignment language where the program allocates new cons cells at each iteration—its performance can be attributed at least in part to the benefits of call forwarding.

6 Related Work

The optimizations described here can be seen as generalizing some optimizations for traditional imperative languages [1]. In the special case of a (conditional or unconditional) jump whose target is a (conditional or unconditional) jump instruction, call forwarding generalizes the flow-of-control optimization that collapses chains of jump instructions. Call forwarding is able to deal with conditional jumps to conditional jumps (this turns out to be an important source of performance improvement in practice), while traditional compilers for imperative languages such as C and Fortran typically deal only with jump chains where there is at most one conditional jump (see, for example, [1], p. 556).

When we consider call forwarding for the last call in a recursive clause, what we get is essentially a generalization of code motion out of loops (e.g., see Section 4). The reason it is a generalization is that the code that is bypassed due to call forwarding at a particular call site need not be invariant with respect to the entire loop, as is required in traditional algorithms for invariant code motion out of loops. The point is best illustrated by an example: consider a function

```
f(x) = if x = 0 then 1
```

[5] The Janus version of `qsort` used in this table is slightly different from that of Table 3: in this case there are explicit integer type tests in the program source, to be consistent with `int` declarations in the C program and allow a fair comparison between the two programs. The presence of these tests provides additional information to the `jc` compiler and allows some additional optimizations.

```
else if p(x) then f( g(x-1) )      /* Call Site 1 */
else f( h(x-1) )                   /* Call Site 2 */
```

Assume that the entry actions for this function include a test that its argument is an integer, and suppose that we know, from dataflow analysis, that `g()` returns an integer, but do not know anything about the return type of `h()`. From the conventional definition of a "loop" in a flow graph (see, for example, [1]), there is one loop in the flow graph of this function that includes both the tail recursive call sites for `f()`. Because of our lack of knowledge about the return type of `h()`, we cannot claim that "the argument to `f()` is an integer" is an invariant for the entire loop. However, using call forwarding we can bypass the integer test in the portion of the loop arising from call site 1. Effectively, this moves some code out of "part of" a loop. Moreover, our algorithm implements interprocedural optimization and can deal with both direct and mutual recursion, as well as non-tail-recursive code, without having to do anything special, while traditional code motion algorithms handle only the intra-procedural case.

The idea of compiling functions with multiple entry points is not new: many Lisp systems do this, Yale Haskell generates dual entry points for its functions, and Aquarius Prolog generates multiple entry points for primitive operations [20]. However, we do not know of any system that attempts to order the entry actions carefully in order to maximize the savings from bypassing entry actions.

Chambers and Ungar consider compile-time optimization techniques to reduce runtime type checking in dynamically typed object-oriented languages [4, 5]. Their approach uses type analysis to generate multiple copies of program fragments, in particular loop bodies, where each copy is specialized to a particular type and therefore can omit some type tests. Some of the effects of the optimization we discuss, e.g., "hoisting" type tests out of loops (see Section 4), are similar to effects achieved by the optimization of Chambers and Ungar. In general, however, it is essentially orthogonal to the work described here, in that it is concerned primarily with type inference and code specialization rather than with code ordering. Because of this, the two optimizations are complementary: even if the body of a procedure has been optimized using the techniques of Chambers and Ungar, it may contain type tests etc. at the entry, which are candidates for the optimization we discuss; conversely, the "message splitting" optimization of Chambers and Ungar can enhance the effects of call forwarding considerably.

7 Conclusions

This paper discusses *call forwarding*, a simple interprocedural optimization technique for dynamically typed languages. The basic idea behind the optimization is extremely straightforward: find an ordering for the "entry actions" of a procedure such that the savings realized from different call sites bypassing different sets of entry actions, weighted by their estimated execution frequencies, is as large as possible. It turns out, however, to be quite effective for improving program performance. We show that the problem of computing optimal solutions to arbitrary call forwarding problems is NP-complete, and describe efficient heuristics for the problems. Experimental results indicate that (*i*) the heuristics are effective, in that the solutions produced are generally optimal or close to optimal; and (*ii*) the resulting optimization is effective, in that it leads to significant performance improvements for a number of benchmarks tested. A variant of these ideas has been implemented in `jc`, a logic programming system that is available by anonymous FTP from `cs.arizona.edu`.

References

[1] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers – Principles, Techniques and Tools*, Addison-

Wesley, 1986.

[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] M. Carlsson and J. Widen, *SICStus Prolog User's Manual*, Swedish Institute of Computer Science, Oct. 1988.

[4] C. Chambers and D. Ungar, "Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically Typed Object-Oriented Programs", *Proc. SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990, pp. 150–164. *SIGPLAN Notices* vol. 25 no. 6.

[5] C. Chambers, D. Ungar and E. Lee, "An Efficient Implementation of SELF, A Dynamically Typed Object-Oriented Language Based on Prototypes", *Proc. OOPSLA '89*, New Orleans, LA, 1989, pp. 49–70.

[6] I. Foster and S. Taylor, "Strand: A Practical Parallel Programming Tool", *Proc. 1989 North American Conference on Logic Programming*, Cleveland, Ohio, Oct. 1989, pp. 497-512. MIT Press.

[7] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, New York, 1979.

[8] M. R. Garey, D. S. Johnson, and L. Stockmeyer, "Some Simplified NP-complete Graph Problems", *Theoretical Computer Science* vol. 1, pp. 237–267, 1976.

[9] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.

[10] R. E. Griswold and M. T. Griswold, *The Implementation of the Icon Programming Language*, Princeton University Press, Princeton, 1986.

[11] D. Gudeman, K. De Bosschere, and S. K. Debray, "`jc` : An Efficient and Portable Implementation of Janus", *Proc. Joint International Conference and Symposium on Logic Programming*, Washington DC, Nov. 1992. MIT Press.

[12] A. Houri and E. Shapiro, "A Sequential Abstract Machine for Flat Concurrent Prolog", in *Concurrent Prolog: Collected Papers*, vol. 2, ed. E. Shapiro, pp. 513-574. MIT Press, 1987.

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

[14] 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.

[15] 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.

[16] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg, *Programming with Sets: An Introduction to SETL*, Springer-Verlag, 1986.

[17] E. Shapiro, "The Family of Concurrent Logic Programming Languages", *Computing Surveys*, vol. 21 no. 3, Sept. 1989, pp. 412-510.

[18] G. L. Steele Jr., *Common Lisp: The Language*, Digital Press, 1984.

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

[20] P. Van Roy, *Can Logic Programming Execute as Fast as Imperative Programming?*, PhD Dissertation, University of California, Berkeley, Nov. 1990.

[21] 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.

Program	no optimization	partition-&-sort	greedy	optimal
`hanoi`	492	225	225	225
`tak`	574	451	172	172
`nrev`	726	360	360	360
`qsort`	1776	453	450	450
`factorial`	129	24	24	24
`merge`	720	330	330	330
`dnf`	124	25	25	25
`pi`	306	272	30	30
`binomial`	5963	1304	1304	1304

Table 1: Efficacy of different Call Forwarding heuristics (in Sparc assembly instructions)

Program	w/o forwarding (ms)	with forwarding (ms)	% improvement
`binomial`	5.95	5.14	13.6
`hanoi`	186	163	12.4
`tak`	299	207	30.8
`nrev`	1.17	0.716	38.8
`qsort`	2.31	1.87	19.0
`merge`	0.745	0.613	17.7
`dnf`	0.356	0.191	46.3

Table 2: Performance Improvement due to Call Forwarding

Program	`jc` (J) (ms)	Sicstus (S) (ms)	S/J	Quintus (Q) (ms)	Q/J
`hanoi`	163	300	1.84	690	4.2
`tak`	207	730	3.5	2200	10.6
`nrev`	0.716	1.8	2.5	7.9	11.0
`qsort`	1.87	5.1	2.7	9.4	5.0
`factorial`	0.049	0.44	8.9	0.27	5.5

Table 3: The Performance of `jc`, compared with Sicstus and Quintus Prolog

Program	`jc` (J) (ms)	C (unopt) (ms)	C (opt: `-O4`)	J/C-unopt	J/C-opt
`nrev`	0.716	0.89	0.52	0.80	1.38
`binomial`	5.14	4.76	3.17	1.08	1.62
`dnf`	0.191	0.191	0.061	1.00	3.13
`qsort`	1.33	1.25	0.34	1.06	3.91
`tak`	207	208	72	1.00	2.88
`factorial`	0.049	0.049	0.036	1.00	1.36

Table 4: The performance of `jc` compared to C

A Appendix: Proof of NP-Completeness

The following problem is useful in discussing the complexity of optimal call forwarding:

Definition A.1 The Optimal Linear Arrangement problem (OLA) is defined as follows: Given a graph $G = (V, E)$ and an integer k, find a permutation, f, from the vertices in V to $1, \ldots, n$ such that defining the length of edge (i, j) to be $|f(i) - f(j)|$, the total length of all edges is less than or equal to k. ∎

The following result is due to Garey, Johnson, and Stockmeyer [7, 8]:

Theorem A.1 *The Optimal Linear Arrangement problem is NP-complete.*

The following result gives the complexity of optimal call forwarding:

Theorem 3.1 *The determination of an optimal solution to a call forwarding problem is NP-complete. It remains NP-complete even if every entry action has equal cost.*

Proof: We first formulate optimal call forwarding as a decision problem, as follows: "Given a call forwarding problem I and an integer $K \geq 0$, is there a solution to I with cost no greater than K?" We refer to this problem as CF. The proof is by reduction from Optimal Linear Arrangement problem, which, from Theorem A.1, is NP-complete. Let $G = (V, E), k$ be a particular instance of OLA. We make the following transformation to an instance $\langle A, C, w, cost \rangle$ of CF, where:

- A is the set of vertices $1, \ldots, n$ in V along with two dummy vertices s and t;
- The elements of C are all doubleton sets:
  - corresponding to each edge $(u, v) \in E$, there is an element $\{u, v\}$ in C with weight 1: for terminological simplicity in the discussion that follows, we refer to these elements as *normal sets*;
  - let Δ be the maximum degree of any vertex in G, then corresponding to each vertex $i \in G$ of degree d_i, there is an element $\{i, s\}$ in C with weight $\frac{1}{2}(\Delta - d_i)$ (some of these sets could have zero weight, in which case they can effectively be removed): we refer to these elements as *special sets*;
  - finally, there is an element $\{s, t\}$ in C of weight M, where M is large enough to ensure that s and t have to be the last two elements in any optimal ordering of the vertices (M can be chosen to be n^3 or greater): we refer to this element as a *heavy set*.
- $cost(I) = 1$ for every $I \in A$.

We also have to define the number K that is to bound the cost of the call forwarding problem so constructed. Let $K = \frac{1}{4}n(n + 5)\Delta + 3M + k/2$. We claim that the instance of CF so defined has a solution with cost no greater than K if and only if the given instance of OLA has a solution.

Consider any proposed order of elements in a solution to the instance of CF defined above. The cost of this solution can be decomposed as follows:

As we march along the list of elements, at each point we charge $\Delta/2$ to each of the elements we have seen so far but not to either of the special elements. If vertex $i \in G$ is encountered, the charge of $\Delta/2$ on vertex i from then on can be thought of as paying $1/2$ towards each of the normal sets that contain i and paying the entire cost of the special set that contains i. Now if both elements of a normal set have been encountered, the total cost of the set will from then on be picked up by these charges to the vertices. For a normal set $\{i, j\}$, after i has been encountered and before j has been encountered the extra charge of $1/2$ at each stage will be charged to the edge (i, j). Breaking up the charges as above, one finds that for any order in which s and t finish last, the charge to the vertices is a constant independent of the order and is equal to $\frac{1}{4}(n(n+5)\Delta)$ and the charge for the heavy set is fixed at $3M$. The only variable is the charge to the edges and this charge will be exactly half the total length of the edges, since an edge gets charged only after one of its endpoints has been encountered and before the other endpoint has been encountered, i.e. for the "duration" of its length.

Thus there is a YES answer to the instance of CF created if and only if the total length of all "normal" edges is kept to k or less, or, in other words, if and only if the instance of OLA is a YES-instance. (Note that since the cost of the special sets is entirely picked up by the vertices, the lengths of the special edges do not matter.)