Compiler-directed Dynamic Linking for Mobile Programs*

Anurag Acharya Joel Saltz
UMIACS and Department of Computer Science
University of Maryland, College Park 20742
{acha,saltz}@cs.umd.edu

Abstract

In this paper, we present a compiler-directed technique for safe dynamic linking for mobile programs. Our technique guarantees that linking failures can occur only when a program arrives at a new execution site and that this failure can be delivered to the program as an error code or an exception. We use interprocedural analysis to identify the set of names that must be linked at the different sites the program executes on. We use a combination of runtime and compile-time techniques to identify the calling context and to link only the names needed in that context. Our technique is able to handle recursive programs as well as separately compiled code that may itself be able to move. We discuss language constructs for controlling the behavior of dynamic linking and the implication of some of these constructs for application structure.

1 Introduction

Mobile programs can move from host to host during execution. At migration points, the execution stack and the heap of the program are transferred from original host to the target host; execution continues at the target host. Mobile programs have been proposed as a suitable model of computation for the Internet [5, 16, 21]. To access local resources or to use site-specific operations available at individual hosts, they need to be able to name them. The mapping between program names and local procedures/operations has to be established dynamically. There are three reasons for this. First, dynamic linking allows the host to retain fine-grain control over what mobile programs can do. If a program cannot refer to the operation that opens local files, it cannot open local files. Second, dynamic linking allows the program to use the same names for identical operations for different hosts. For example, the name `open()` can be used while executing on any host to refer to the procedure used to open local files. Third, the same program could visit different hosts every time it is run.

Dynamic linking provides functionality that is necessary for secure flexible use of mobile programs but it introduces a new class of runtime errors - unbound procedure names. Dynamically linked static programs can avoid this problem by making sure that all the procedures that a program might refer to are available; only the actual linking is delayed till runtime.[1] It is still possible to have access-control/authentication errors but that is not because the program can not name the procedure for an operation; it just does not have the right to perform the operation. Linkers for mobile programs do not know where the program might be executed and, therefore, are unable to provide a similar guarantee.

There are two approaches for dealing with this problem. The first is to make sure that the execution environment on every host is exactly identical. This restriction is stronger than requiring that every host provide a well-known interface (for example, the Java^{TM}[9] API). Instead it requires that every environment provide nothing else. This would eliminate an important reason for program mobility. For example, a

*This research was supported by ARPA under contract #F19628-94-C-0057, Syracuse subcontract #353-1427

[1] Unbound procedure names can occur even on a single host, if the environment has changed since the program was last processed by the linker. This can happen, for example, after a new version of a shared library has been installed.

```
proc execute_on_A()
    cleanup_and_go("A");
    execute_A_specific_fn();
end

proc execute_on_B()
    cleanup_and_go("B");
    execute_b_specific_fn();
end
```

```
proc clean_and_go(string hostname)
    finalize_local_state();
    go(hostname);
    initialize_on_new_host();
end
```

Figure 1: A simple program that cannot be completely linked if calling context is ignored.

program that searches distributed data repositories can improve its performance by migrating to the repositories, performing the search on-site using local repository-specific procedures that can efficiently process the data in its native form and returning the result to the requesting site.

The second, and a more flexible, approach is to accept these failures as yet another class of failures possible in a distributed system and to insert checks that localize them and allow the program to take corrective action. This can be achieved by requiring that dynamic linking operations be performed only as a part of a change-execution-site operation (which we refer to as `go`). The operation succeeds iff valid bindings can be found for all names that the program could possibly refer to while executing on the target host. Else the operation is aborted and an error-code or exception is returned. The program can take corrective action by seeking an alternative site or using an alternative way of accessing the site (e.g. using an remote interface with lesser functionality).

The second approach requires identification of the set of names that the program could possibly refer to while executing on the target host. This has to be done for every call to `go` in the program. There are three ways in which this could be done. The first and the simplest alternative is to require that a valid binding for every name in the program is available on each execution site. In general, this can succeed if and only if all environments are identical. But that defeats the purpose. The second alternative is to require the user to do the identification. For simple programs, this could work well but for more complicated programs, especially programs with deep call-graphs, it could be difficult. The third alternative is to use compiler analysis.

In this paper, we present a compiler-directed technique. We use interprocedural analysis to identify the set of names that must be linked at each call to `go` (we refer to these as *linksets*). Calls to `go` embedded in procedures that are called from multiple sites pose a problem for a compile-time approach. It is not possible to distinguish between different call-sites of the enclosing procedure. This forces the analysis to be overly conservative and include names from paths corresponding to all call-sites. Since programs can contain calls to site-specific procedures, linking for fairly simple programs cannot be completed (see Figure 1 for an example). We use a combination of runtime and compile-time techniques to identify the calling context and to link only the names needed in that context. An explicit goal of our technique is to perform analyses that might be expensive only at compile-time and to defer only simple processing to run-time.

We first present a simple version of the technique that assumes all code that could contain a `go` is compiled together. That is, all the code that is to be linked in dynamically does not contain a call to `go`. We then relax this constraint and show how separately compiled mobile code can be dealt with. This also allows our technique to deal with *library sites* [20]. Library sites are an intriguing idea - these are sites that provide pre-compiled mobile code that can be picked up by mobile programs for execution on other sites. For example, an organization with multiple hosts could provide a library site which provides code needed to access data on all hosts belonging to the organization. We then discuss language constructs that can be used to control the behavior of dynamic linking and the implications of some of the constructs for application structure.

2 Linking mobile programs with non-mobile code

The basic idea of our technique is quite simple. For every call to `go`, determine the set of calls to `go` that are *visible* from it. We say program point B is visible from program point A iff there exists at least one path from A to B in the control-flow graph that has no call to `go`; calls to `go` are said to *hide* the code that they dominate (in the sense of dominators in control-flow graphs [22]). We refer to the set of `go`s visible from a particular `go` as its *departure-set*. If the program arrives at a host by this `go` and it does not terminate on this host, it will depart from the host via one of the calls in the departure-set. For every `go`, we compute the set of names referred to on any path between itself and its departure-set. If the language does not allow procedure calls, we are done. This set of names is the linkset at the `go` in question. For realistic languages, this simple scheme runs into the problem of preserving the calling context. Calls to `go` embedded in procedures that are called from multiple sites are the primary problem as it is not possible to distinguish between different call-sites of the enclosing procedure. This forces the analysis to be overly conservative and include names from paths corresponding to all call-sites. Since programs can contain calls to site-specific procedures, linking for fairly simple programs cannot be completed (see Figure 1 for an example). To get around this problem, we use compiler analysis to generate the linksets for program fragments and simple runtime support that uses information from the execution-stack to quickly construct the complete linkset for any particular `go`.

In this section, we describe our proposed technique. We first describe the program representation used for the compiler analysis. We then present the analysis algorithm and the data structure that is generated based on the information collected. This data structure is used at runtime by the dynamic linker. Finally, we describe how the complete linkset for a `go` is constructed at runtime. We use a running example to illustrate the analysis as well as the dynamic linking procedure.

2.1 Program representation

For this section, we make three assumptions about the language: (1) programs can move only by using an explicit `go`, (2) programs are first-order and (3) all the code that might contain a `go` is compiled together; none of the code that is dynamically linked has an embedded `go`. We shall remove the third assumption in the next section. Extending our analysis to programs that use higher-order functions or to programs that can move without an explicit operation is beyond the scope of this paper.

We use a modified version of the *full-program representation* (FPR) introduced by Agrawal et al [1, 2]. In our version, a program is represented by a directed multigraph; nodes correspond to program points and edges correspond to a control-flow path or paths between program points. There are five kinds of nodes: a pair of *entry* and *exit* nodes for every procedure; a pair of *call* and *resume* nodes for every call-site and a *go* node for every call to `go`. Dummy entry and exit nodes are inserted for functions that are to be linked dynamically. For every call-site, two edges are inserted: (1) an edge between the call node corresponding to the call-site and the entry node corresponding to the called procedure and (2) an edge between the exit node corresponding to the called procedure and the resume node corresponding to the call-site. For procedures that are to be linked dynamically, an edge is inserted between the corresponding entry node and the exit node. This encodes the assumption that there is no call to `go` within these procedures. In addition, an edge is inserted between a pair of nodes iff there exists at least one path in the control-flow graph which does not pass through any of the other nodes. More concretely, an edge is inserted between a pair of nodes iff there is at least one path between the two nodes that does not contain a procedure call or a call to `go`. For nodes i and j, the edge (i, j) represents the code corresponding to all paths in the control-flow graph between the corresponding program points.

We illustrate the program representation using an example. Figure 2 contains our running example. The corresponding program representation is shown in Figure 3. Note that each edge corresponds to the group of control-flow paths between the two nodes it connects. For example, the edge between *A_entry* and *call1* consists of the statement `i = 0` and the edge between *res1* and *call2* consists of code in $frag1$ as well as the loop header. Also note that each basic block can be part of more than one paths and, therefore, can be included in the code corresponding to more than one edge. For example, the header for the while loop appears in three edges: $(res1, call2)$, $(res3, call2)$, and $(res4, call2)$ corresponding to the initial entry, the loop-back from if branch and the loop-back from the else branch respectively.

In the next subsection, we present the analysis algorithm.

```
Proc A()                              Proc B()
   i = 0;                                ......   - - - - - - - - -  frag3
   B();      - - - - - - - - -  cs1      go(....); - - - - - - - - -  go1
   ........  - - - - - - - - - frag1     ......   - - - - - - - - -  frag4
   while (i > 100)                    end
     C();    - - - - - - - - - cs2
     .......  - - - - - - - - - frag2  Proc D()
     if (condition)                      B();     - - - - - - - - -  cs6
       D();  - - - - - - - - -  cs3      ......   - - - - - - - - -  frag5
     else                             end
       E();  - - - - - - - - -  cs4
     endif                            Proc E()
     i++;                                .......  - - - - - - - - -  frag6
   endwhile;                          end
   D();      - - - - - - - - -  cs5
end
```

Figure 2: Running example. The cs$i$ annotations mark the call-sites, the frag$i$ annotations mark code fragments that have no calls and no `go`s.

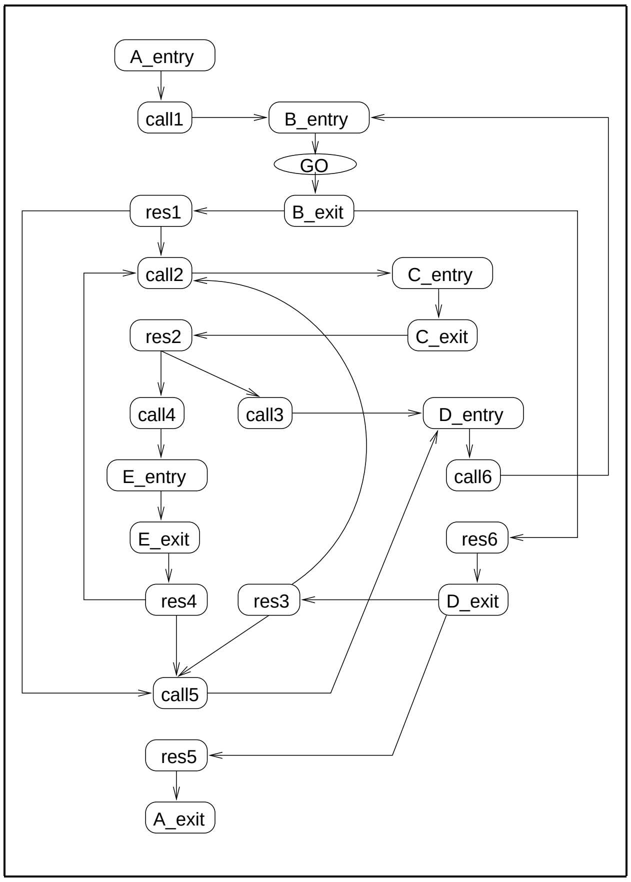

Figure 3: Program representation for the running example.

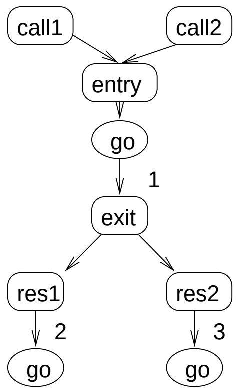

Figure 4: Simplified example to illustrate the calling context problem.

2.2 Analysis algorithm

The basic intent of the algorithm is clear enough - collect all names that occur on all paths between a `go` and its departure-set. The primary problem is preserving the calling context for calls to `go` embedded in multiply-called procedures. This problem is illustrated in Figure 4. If the calling context is not taken into consideration, the linkset for the top `go` consists of the names occurring in the blocks of code corresponding to edges 1, 2 and 3. Since the code corresponding to edges 2 and 3 may never be executed on the same host, it is possible that no host will provide the procedures referred to in both links.

To resolve this problem, we do not create complete linksets at compile-time. Instead, we partition the paths between a `go` and a member of its departure-set into a set of segments using resume nodes as separators. In Figure 4, this results in three segments, corresponding to edges 1, 2 and 3. Intuitively, each segment corresponds to the portion of the path that lies entirely within a single procedure.[2] The compiler analysis computes the linksets for individual segments. It also marks if the segment ends in a go node or a resume node. Intuitively, this indicates whether this is the final segment in a path. The complete linkset for a `go` is constructed at runtime by threading together the linksets for individual fragments. At runtime, the linker determines the calling context either by inspecting the return addresses on the runtime-stack or by inspecting a call-stack maintained separately. It walks backward over the stack. For each call-site, the associated linkset is collected. The walk continues till either it reaches the final segment of a path or it reaches the bottom of the stack. In Figure 4, if the `go` occurs when called from the left-hand-side, the linksets for edges 1 and 2 are collected; if the `go` occurs when called from the right-hand-side, the linksets for edges 1 and 3 are collected.

The algorithm to construct the linksets for path segments is presented in Figure 5. We would like to draw attention to some aspects of the algorithm. This algorithm computes a set of names for every node in the program representation graph. Only the sets corresponding to go nodes and resume nodes and root_nodes are used to create the linksets. The sets corresponding to call and entry nodes are computed for algorithmic convenience and are propagated to parent nodes (if a parent exists). The sets corresponding to exit nodes are always empty. Root_nodes are entry nodes for procedures that are not called in the code being compiled. Linksets for root_nodes are used for initial linking on the node that the program begins execution. They are also needed to handle separately compiled code. Note that the set of names corresponding to a go node are never propagated. However, the set of names corresponding to a resume node could be propagated to the corresponding *call* node. This is needed if it is possible that during the execution of the procedure called at that site, there might be no call to `go` and control might reach the resume point while the program is still at the current host. This condition is represented in the algorithm by the call to `visible(entry(p),exit(p))` which determines if the exit node of the procedure is visible from its entry. Recall that we say program point B is visible from program point A iff there exists at least one path from A to B in the control-flow graph that has no call to `go`. If there exists no such a path, the set of names for the resume node is not propagated.

[2] This intuition is *precise* only if there are no calls to procedures that do not have a `go` as the the first statement. In other cases, there is some merging of paths between procedures, but the intuition is nevertheless helpful.

Program point	Linkset
go1	nm(frag4)
res1	nm(frag1) ∪ {C} ∪ nm(frag2) ∪ nm(frag3) ∪ nm(frag6)
res3	{C} ∪ nm(frag2) ∪ nm(frag3) ∪ nm(frag6)
res5	ϕ
res6	nm(frag5)

Table 1: Linksets for the running example. The *nm* function is used for illustrative convenience. It returns the list of names in a code fragment.

The reason for this is simple: if there is no path through a called procedure P that does not contain a `go`, the program will migrate to another site before this call returns. In that case, there is no need for binding names that are visible from the resume point which implies there is no need to include the names in the linkset for this resume node in the linkset of any of the calls to `go` that precede the call under consideration.

Another point to note is that procedures that are not available during the compilation (that is, the procedures that are to be dynamically linked) are assumed to be empty for the purpose of this algorithm. These procedures are compiled separately and their entry nodes contain the set of names that need to be linked for their execution.

To illustrate the operation of the algorithm, we present the linksets for the running example (Figure 2) in Table 1. Note that two of the resume nodes, `res2` and `res4` do not need linksets as the procedures called at these sites, $C()$ and $E()$ respectively, are not mobile (their execution cannot encounter a `go`). The names in the code fragments `frag4` and `frag5` are dominated by either a call to `go` or a call to a procedure that calls `go`. This is the reason why they do not appear in the linkset for `res1`. All other code is visible from `res1`.

It is easy to see that this algorithm terminates. The linksets for all nodes are initialized to ϕ. Each iteration of the propagation phase visits every node exactly once and each iteration (except the last one) increases the linkset of at least one node by at least one name. The number of iterations cannot be more than the product of the number of nodes and the number of names. This is an extremely loose bound. For non-recursive programs, the algorithm needs exactly two iterations. For recursive programs, it needs one iteration more than the maximum *syntactic* depth of recursion: programs that have only self-recursive procedures take two iterations, programs with at most pair-wise recursive procedures take three iterations and so on. Each iteration is a depth-first traversal of the directed acyclic graph corresponding to the program. The `visited` markers ensure that no node is visited twice in the same iteration.

In the next subsection, we present the algorithm used by the linker to construct complete linksets.

2.3 Constructing complete linksets at runtime

The compiler analysis described in the previous subsection generates two data structures for the linker: an array of structures corresponding to resume nodes and an array of structures corresponding to go nodes. Each structure contains two fields: the linkset containing the names that have to be linked and a boolean indicating whether this linkset is terminal. The boolean is extracted from the `reaches_exit` field of the corresponding node (see Figure 5 for a description of how this is computed). The algorithm for the linker is given in Figure 6. The *do_link*() procedure does the actual linking. For every new procedure linked, it extracts the linkset from the entry node and checks if valid bindings for all names in `names` can be found in the execution environment available on the current host. If this check fails for some reason, it returns an error_code (or an exception if the language supports exceptions).

To illustrate the operation of the algorithm, we consider the two calls to $D()$ in the running example (Figure 2). For the `go` embedded in the call at `cs3`, the complete linkset consists of the union of the linksets for `go1`, `res6` and `res3`; the linkset for the other call at `cs5` is the union of the linksets for `go1`, `res6` and `res5`.

```
function successors(node p)
    returns the set of nodes that are successors of p
function names(node p, node q)
    returns the set of names referred to in code associated with (p,q)
function entry(node p)
    if p is a call node, returns singleton set with corr. entry node
    else returns singleton set with entry node of enclosing proc
function exit(node p)
    if p is a call node, returns singleton set with corr. exit node
    else returns singleton set with entry node of enclosing proc
function resume(node p)
    if p is a call node, returns singleton set with corr. resume node
    else returns singleton set with entry node of enclosing proc
function proc(node p)
    if p is a call or resume node, returns singleton set with name of the corr. proc.
    else returns singleton set with name of enclosing proc.
function visible(node p, node q)
    if q is visible from p returns  true
    else returns  false

function linkset(node p)
  if (p->visited ≠ 0)
      return p->linkset;
  p->visited ← 1
  prev_val = p->linkset;
  if (p->nodetype ==  entry || p->nodetype ==  go || p->nodetype ==  resume)
       succset ← successors(p);
        foreach q in succset
           p->linkset ← p->linkset ∪ linkset(q) ∪ names(p,q)
  else if (p->nodetype ==  call)
       if (visible(entry(p),exit(p))
            p->linkset ← p->linkset ∪ proc(p) ∪ entry(p)->linkset ∪ resume(p)->linkset
       else
            p->linkset ← p->linkset ∪ proc(p) ∪ entry(p)->linkset
  else /* p->nodetype ==  exit */
        p->linkset ← φ
  p->reaches_exit ← visible(p,exit(p))
  if (not_equal(prev_val,p->linkset))
       changed ← 1
  return p->linkset
end

procedure compute_linksets()
    foreach p in allnodes
      p->linkset ← φ
    changed ← 1
    while (changed == 1)
       changed ← 0
        foreach p in allnodes
            p->visited ← 0
       /* a root_node is the entry for a proc. that is not called in
        * the code being compiled */
        foreach node in root_nodes
            linkset(node)
    endwhile
end
```

Figure 5: Algorithm to compute the linksets.

```
structure {  nameset names; boolean terminal; } res_pts[numres];
structure {  nameset names; } go_pts[numgo];

/* res_seq contains the sequence of resume node identifiers.
 * the order corresponds to walking backward in the stack */
function dyn_link(sequence res_seq)
     item res;
     nameset names;

    if (res_seq == NULL) return SUCCESS
    res ← car(res_seq); res_seq ← cdr(res_seq);
    names ← φ;
    while (res->terminal ≠ 0)
        names ← names ∪ res->names;
        if (res_seq == NULL) break;
         res ← car(res_seq); res_seq ← cdr(res_seq);
    endwhile

    if (do_link(names) == SUCCESS)
        return SUCCESS
     else
        return error_code
end
```

Figure 6: Algorithm used by the dynamic linker to construct complete linksets.

3 Linking mobile programs with mobile code

In the previous section, we assumed that the procedures to be linked dynamically did not have a call to go. In this section, we relax that assumption. This allows us to deal with the possibility of *library* sites; sites that provide code to be picked up by programs for execution on other nodes. We first describe what new problem is introduced by allowing dynamically linked procedures to move. Next, we describe our solution and show that it works.

The main problem introduced by allowing dynamically linked code to move is that it is no longer possible to accurately compute the predicate $visible(p,q)$ which indicates whether there exists at least one path between nodes p and q that does not have a go. This problem does not arise for dynamically linked code that can not move – the analysis algorithm can safely and accurately model the missing code by a null statement (as mentioned in section 2.1). As a result, the compiler analysis is no longer able to determine when the linkset for a resume node should propagated to the call node and beyond.

One possible solution is to delay linking code that may be mobile to just before it is invoked. But that would violate the guarantee we would like to offer: that linking failures can happen only at calls to go. Another possible solution is ignore the fact that the linked code can contain gos and to use the analysis described in the previous section. This would result in possibly more names being linked at a go than otherwise – since linksets are possibly being propagated further than they should be. This might appear to be a conservative approximation but it is not. Consider the example in Figure 7. Assume that $B()$ is the procedure to be linked dynamically. If we ignore the fact that $B()$ might contain a go, we will propagate magic_host_specific_code to the linkset corresponding to go(ordinary_host). When go(ordinary_host) is executed, the linker will fail trying to find magic_host_specific_code on ordinary_host. We propose a third alternative and show that it preserves the fail-only-at-go guarantee and avoids the problem mentioned above.

The basic idea is simple. We cannot safely and accurately propagate linksets at compile-time; we cannot delay linking till the linked procedure is invoked; all linking must happen at gos. Therefore, the only option

Available at compile-time	**Unavailable at compile-time**

Proc A()
  go(ordinary_host);
  B();
  magic_host_specific_code()
end
	Proc B()
	  go(magic_host);
	end

Figure 7: Example to illustrate that moving names past gos is unsafe.

left is to delay the propagation of linksets till one of the gos that they are visible from. This is similar in spirit to our scheme of reconstructing complete linksets at runtime.

We modify the program representation described in section 2.1 such that *no* edge is inserted between the dummy entry and exit nodes that represent procedures not available at compile-time. By doing so, we encode the assumption that all control-flow paths in a procedure that is unavailable at compile-time have at least one call to go. As a result, linksets are never propagated past calls to unavailable procedures. We extend the compiler analysis to determine the resume nodes that are visible from each call-site of an unavailable procedure. In addition, each name in a linkset is annotated with the list of its call-sites that are visible from the corresponding program point. We extend the data structures generated by the analysis to include an array of structures corresponding to call-sites of unavailable procedures. Each structure contains the set mentioned above and a boolean that indicates whether this procedure contains at least one path from its entry to its exit that does not contain a call to go. The algorithm presented in Figure 5 already computes this predicate at each call-site; the only extension is to store the value for use at runtime. We extend the linking algorithm such that whenever a new procedure is linked in, the corresponding boolean value is checked. If it indicates that the exit of the procedure is not visible from the entry, our compile-time assumption is valid and the operations described in Figure 6 suffice. If this is not the case and there is at least one path between the entry and the exit of the procedure, the compile-time assumption is invalid and the linksets should have been propagated. To handle this case, the linker inspects the set of resume nodes associated with this call-site and adds the linksets corresponding to all of them to the set of names to be linked.

It is quite easy to see that this scheme preserves the *fail-only-at-go* guarantee and that at each go, the linkset constructed does not contain any name that it would not have contained if all mobile procedures were available at compile-time. The first part is obvious. To assure ourselves about the second part, we need to observe a couple of points about linksets and the visibility property. First, a name appears in a linkset iff a reference to the name is visible from the program point the linkset is associated with. Second, visibility is transitive. If a procedure P is to be linked at a go site G, then at least one call-site of P is visible from G. If the exit of P is visible from its entry, then all program points visible from the exit are also visible from the entry and transitively from G. Therefore, if the procedure was available during compile-time, all the names in the linksets corresponding to all resume nodes visible from the call-site would be included in the linkset for G and would be linked whenever control reaches G. This is exactly what happens with our scheme.

4 Language constructs and application structure

In the previous sections, we have described algorithms to determine what needs to be linked at particular points in a mobile program. In this section, we discuss language constructs that allow the programmer to control the behavior of the linking procedure. Our desire is to explore the space of user directives – how much control can be given to the user without forcing her to explicitly specify linking operations.

There are four flavors of dynamic linking that might be useful for mobile programs, two of which require no user directives and two that can be controlled by user directives.

- *local-only:* linking done at a site is valid at that site only. When a program departs, all bindings created on this site are voided. This guarantees that departing programs do not retain bindings to local operations. However, this works only if the code that is to be linked in does not contain a go. Also, library sites are not possible.

```
go(library_site);
while (condition)
    dynamically_linked_lib_proc();
    go(some_site);
endwhile

dynamically_linked_lib_proc()
    do_magic_operations()
    find_next_site_to_go()
    go(next_site);
end
```

```
go(library_site);
dynamically_linked_lib_proc();

dynamically_linked_lib_proc()
    while (condition)
      do_magic_operations()
       find_next_site_to_go();
       go(next_site);
    endwhile
end
```

Figure 8: Example to illustrate loop placement for mobile code linked at library sites. The version on the left might fail linking at a remote site, the version on the right will not.

- *code-on-stack-is-sticky:* this is a variation of *local-only* linking that allows code that is currently in-use (that is, an instance of the procedure is on the stack) to move when the program departs the linking site. Bindings for all other names is voided when the program departs from this site. If at a subsequent migration, the stack no longer contains the stack frame corresponding to the call to the procedure in question, the binding is eliminated and the code is garbage-collected. This allows library sites to exist. Elimination of the binding when it is no longer in use allows programs to pick up different copies of the procedures by visiting different library sites. This would be useful, for example, if library sites were associated with specific organizations and provided the code needed to access data on all execution sites belonging to the organization. The program would pick up a local version of the procedure whenever it visits hosts belonging to a new organization. This scheme also allows individual sites to control what code can be taken away by a mobile program. Note that this has implications for application structure. Once such a procedure executes a `go` statement, it can not be assured that it is executing in the same environment. Therefore, all code that is available only at the linking site and that is necessary for its execution must be inlined into this procedure. Second, repeated application of procedures linked at library sites may fail if program moves repeatedly. Therefore, loops that call such procedures should be pushed into the procedures. Figure 8 provides an example. Note that these complications can be eliminated if the unit of linking is a module instead a single procedure. In that case, inlining all locally-provided operations into each mobile procedure that might need them would not be needed.

- *user-specifies-sticky-links:* the idea here is that the user can specify which names once bound should be bound forever. This eliminates some of the complications of the *code-on-stack-is-sticky* scheme. Programs that need to pick up similar procedures from multiple library sites would have to name them differently. One difference between this scheme and user-specified linking is that in this scheme, the programmer does not need to specify which site the procedure is to be found. Second, sticky links can be easily specified by a `sticky` annotation on the interface declaration. Third, sticky links cannot be rebound, user-specified linking can rebind names.

- *user-specified linking:* in this case, the user explicitly specifies which names are to be bound to which operations on which sites. An example of this would be the `net_import()` primitive provided by Obliq [4] (as well the `NetObj.Import` primitive provided by Network Objects [3] that has been used to implement it). While this provides the greatest flexibility of all schemes, the programmer can rebind names as and when needed, it requires the user to manage all linking operations.

5 Related work

Various forms and implementations of user-specified linking have been described in the literature. At the simplest level, the *eval(env,expr)* primitive that has long been available allows the user to control the bindings for the free variables in the expression *expr*. Obliq [4] and Network Objects [3] allow the programmer to query a name-server and obtain a reference which can then be bound to a name in the program. The fragmented-objects model [17, 18] proposed by Shapiro includes a detailed interface for binding and unbinding references in a distributed system. First-class environments [8] can also be used for various scenarios in whch user-specified linking might be useful. Miller&Rozas [12] propose to use first-class environments to remove the need for a distinguished top-level interaction environment for Scheme. Jagannathan [11] proposes a *reification* operator that returns the current environment as a first-class object and a *reflection* operator that merges a set of bindings from a named environment into the current environment. Queinnec&DeRoure [15] propose a `chain-environment` function which can compose environments. This can be used to share common enviornments between different programs/users. These schemes provide varying degrees of flexibility and convenience but all of them require the user to explicit manage the linking.

There has also been considerable work on efficient (and safe) implementation of dynamic linking, particularly in the context of shared libraries and kernel extensions [6, 10, 13, 14, 19]. These schemes focus on the linking procedure and the performance of the linked code. They do not address the issue of determining what needs to be linked.

Dynamic linking issues for mobile programs, in particular the need to be able to name procedures that access resources local to an execution site, have been previously considered by Cardelli [4] and Knabe [7]. The Obliq language presented in [4] handles this problem by packaging the execution environment available at a site as an object and using its methods to access the procedures available at that site. The responsibility of determining whether the environment available on a host provides all names needed by the program resides with the user. Knabe [7] proposes that certain functions be specified to be *ubiquitous* - that is, they are available on all sites and that all remaining code should be carried by the mobile program. This does not allow for site-specific procedures.

6 Summary

In this paper, we have presented a compiler-directed technique for safe dynamic linking for mobile programs. Our technique guarantees that linking failures can occur only when a program arrives at a new execution site and that this failure can be delivered to the program as an error code or an exception. We use interprocedural analysis to identify the set of names that must be linked at the different sites the program executes on. We use a combination of runtime and compile-time techniques to identify the calling context and to link only the names needed in that context. Our technique is able to handle recursive programs as well as separately compiled code that may itself be able to move. We discuss language constructs for controlling the behavior of dynamic linking and the implication of some of these constructs for application structure.

Acknowledgments

We would like to thank Shamik Sharma for providing a sorely needed sounding board for our ideas. We would also like to thank M. Ranganathan for several discussions. The algorithms presented in Figures 5 and 6 were formatted using the `code.sty` style file written by Olin Shivers.

References

[1] G. Agrawal, A. Acharya, and J. Saltz. An interprocedural framework for placement of asynchronous I/O operations. In *Proceedings of the 1996 International Conference on Supercomputing*, pages 358–65, May 1996.

[2] G. Agrawal, J. Saltz, and R. Das. Interprocedural partial redundancy elimination and its application to distributed memory compilation. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 258–69, Jun 1995.

[3] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network objects. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages 217–30, Dec 1993.

[4] L. Cardelli. A language with distributed scope. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, Jan. 1995.

[5] L. Cardelli. Mobile computation. Position Paper, 1995. *http://www.research.digital.com/SRC/personal-/Luca_Cardelli/Papers/MobileComputationPosition.ps.*

[6] C. Cowan, T. Autrey, C. Krasic, C. Pu, and J. Walpole. Fast concurrent dynamic linking for an adaptive operating system. In *Proceedings of the Third International Conference on Configurable Distributed Systems*, pages 108–15, May 1996.

[7] F.C.Knabe. Language and compiler support for mobile agents. PhD Thesis, Carnegie Mellon University, Nov. 1995.

[8] D. Gelernter and S. Jagannathan. Environments as first class objects. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 98–110, Jan 1987.

[9] J. Gosling and H. McGilton. The Java language environment white paper, 1995.

[10] E. Ho, C. Wei-Chau, and L. Leung. Optimizing the performance of dynamically-linked programs. In *Proceedings of the 1995 USENIX Technical Conference*, pages 225–33, Jan 1995.

[11] S. Jagannathan. Dynamic modules in higher-order languages. In *Proceedings of the 1994 International Conference on Computer Languages*, pages 74–87, May 1994.

[12] J. Miller and G. Rozas. Free variables and first-class environments. *Lisp and Symbolic Computation*, 4(2):107–41, Apr 1991.

[13] M. Nelson and G. Hamilton. High performance dynamic linking through caching. In *Proceedings of the Summer 1993 USENIX Conference*, pages 253–65, 1993.

[14] D. Orr, J. Bonn, J. Lepreau, and R. Mecklenburg. Fast and flexible shared libraries. In *Proceedings of the Summer 1993 USENIX Conference*, pages 237–51, 1993.

[15] C. Queinnec and D. D. Roure. Sharing code through first-class environments. In *Proceedings of the 1996 International Conference on Functional Programming*, May 1996.

[16] M. Ranganathan, A. Acharya, S. Sharma, and J. Saltz. Network-aware mobile programs. In *Proceedings of the 1997 USENIX Annual Technical Conference*, Jan 1997. To appear.

[17] M. Shapiro. Flexible bindings for fine-grain distributed objects. Technical Report 2007, Institut National de Recherche et en Automatique, August 1993.

[18] M. Shapiro. A binding protocol for distributed shared objects. In *Proceedings of the 14th International Conference on Distributed Systems*, Jun 1994.

[19] E. Sirer, M. Fiucynski, P. Pardyak, and B. Bershad. Safe dynamic linking in an extensible operating system. In *The First Workshop on Compiler Support for System Software*, Feb 1996.

[20] J. White. Talk at the DAGS'96 Workshop on Transportable Agents, Sep 1996.

[21] J. White. Telescript Technology: Mobile Agents, 1996. *http://www.genmagic.com/Telescript/Whitepapers.*

[22] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1995.