Dynamic Reconfiguration in Distributed Systems: Adapting Software Modules for Replacement

Christine Hofmeister James Purtilo
Computer Science Department and UMIACS
University of Maryland, College Park, MD 20742 USA

Abstract

Dynamic reconfiguration of a distributed application is the act of changing the configuration of the application as it executes. Examples of configuration changes are replacing a software component (module), moving a module to another machine, and adding or removing a module from the application. Existing dynamic reconfiguration environments support these basic configuration changes, but require the developer to determine how to capture and restore a module's state information, and to add this functionality manually. We present a machine-independent method for automatically installing this functionality in the application, given a set of reconfiguration points designated by the programmer. Our focus in this paper is on the difficult problem of capturing and restoring the state of a module during a procedure call, when the activation record stack contains crucial parts of the process state.

This research is supported by the National Science Foundation under contract NSF CCR-9021222.

1 Introduction

Dynamic reconfiguration allows developers to change the configuration of an application while it is executing, by replacing an executing software component (*module*) of the application, moving a module to another machine, adding or removing a module, and changing the interactions (*bindings*) between modules. Dynamic reconfiguration is needed in order to make changes to very long-running applications or those that must be continuously available. Some reasons for dynamically reconfiguring an application are to perform software maintenance, or to migrate a process. Dynamic reconfiguration is also needed to support applications where changing the configuration is an integral part of the application's semantics.

A platform supporting dynamic reconfiguration must provide *reconfiguration primitives*: operations for the application-level reconfiguration activities of adding and deleting modules and the bindings between them. Because these activities occur while modules are executing, the platform must also provide support for divulging and installing a module's process state. Divulging and installing process state is a module-level activity, requiring that modules *participate* in the reconfiguration. In addition, the platform must provide a mechanism for combining these two aspects, the reconfiguration primitives and the module participation, into a *reconfiguration script* that performs the desired reconfiguration.

Existing dynamic reconfiguration environments support the application-level reconfiguration activities of adding or deleting modules and the bindings between them, but these environments require the programmer to manually adapt a module to participate during reconfiguration. [3] [6] [9] In this paper we present a machine-independent technique for automatically preparing a module to participate during reconfiguration, given a set of reconfiguration points designated by the programmer. Our main focus is on the difficult problem of capturing the state of a module during a procedure call, when the activation record stack contains crucial parts of the process state. We assume that a module is written in a statically-scoped language and has a single thread of control.

The remainder of the Introduction describes the reconfiguration platform and the general issues involved in capturing process state information and installing it in a dynamically created module. Section 2 presents an example of a distributed application and shows how it can be reconfigured by moving a module to another machine. In order to demonstrate capturing and restoring the activation record stack, the module is moved while it is performing a series of recursive calls. Section 3 describes the algorithm for automatically preparing a module for reconfiguration, and the final section discusses this approach and compares it to other work.

0-8186-3770-6/93 $3.00 © 1993 IEEE

1.1 The Reconfiguration Platform

A heterogeneous distributed software application consists of software modules and bindings between them, where a *module* is a software process with its own memory and its own thread of control. Modules can communicate with each other via named *interfaces*, which are logical communication ports designated as incoming, outgoing, or bi-directional. Messages are sent on outgoing interfaces and received on incoming interfaces, and message passing is asynchronous. *Bindings* connect the interfaces of modules in an application. We use POLYLITH [8] as a platform for distributed software applications. POLYLITH provides both a language for describing the application and a software bus for managing the runtime activities. The language is used to provide information to the software bus, which in turn initiates the execution of each module and establishes communication channels between modules in the running application. POLYLITH provides basic operations for sending and receiving messages, and for obtaining the current configuration of the application. To send or receive messages on a module's interfaces, the programmer invokes a POLYLITH communication primitive defined for the language that the module is written in, naming the interface and the variable containing the message. The POLYLITH bus handles any data transformation needed to communicate across heterogeneous hosts.

To perform reconfiguration, the environment must provide reconfiguration primitives and support for module participation, and provide a mechanism for integrating these into a higher-level reconfiguration activity such as replacing or replicating modules. The reconfiguration primitives added to POLYLITH are described in [9]. They include operations for adding and deleting modules and bindings, and an operation that signals a module to divulge state information on a particular interface, then moves that state information to an interface of another module. The modules involved must be provided with the capability of divulging state information and installing it in a dynamically created module.

The work described in [5] explores dynamic reconfiguration without module participation, and describes how an application can be automatically prepared for certain reconfiguration activities, namely replacement and replication of modules.

The extension to the reconfiguration platform described in this paper automatically prepares a module for participation in reconfiguration. The programmer must define reconfiguration points within the module, indicating for each where reconfiguration is safe and what the program state is. Then the source program for the module is transformed, by adding the statements needed to:

- delay reconfiguration until appropriate
- package up and send state
- install the state, restoring the activation record stack as necessary
- resume execution at the appropriate place

The approach we take is similar to the technique proposed in [10] for heterogeneous process migration, where by compiling a special program that restores the process state, they force the compiler to manage the machine-specific details of restoring the activation record stack. We use the compiler to restore *and capture* the activation record stack, without making any changes to the compiler or operating system.

1.2 Abstract Process State

To support dynamic reconfiguration activities, the characterization of the process state must be in an abstract, not machine-specific, format. On a multiprogrammed computer, programs are continually swapped into and out of main memory, and with each swap a process state must be saved and another restored. This capture/restoration of process state is machine-specific, but it serves as a model for an abstract characterization of the process state. Assuming a static-scoped language and single-threaded modules, the items comprising a process state are:

- static data– in data area
- dynamic data– in activation record (AR) stack
- user-allocated data– in heap
- file descriptors, process status information– accessible only to kernel
- temporary values– in registers or AR stack
- program counter
- procedure call/return information– in AR stack
  - return address
  - control link to restore context upon return
  - register values to restore upon return

The temporary or intermediate values used when a statement in a high-level language is compiled into multiple machine language statements may not be compatible with the values used when compiling into a different machine language. Thus the abstract program state must be captured between high-level statements. In addition, optimizing compilers sometimes cache variables in registers to reduce the number of writes to memory. Our approach is to let the compiler handle referencing these variables correctly.

Because the process status information is machine-specific, we do not attempt to restore this information in the new process. File descriptors are an essential part of the process state, but this information is usually accessible only to the kernel of the operating system, so we do not automatically capture them at this time. The data stored in the heap is dynamically allocated by the programmer. At the present time, the programmer must write code to capture and restore heap data structures and to regain access to files.

When a process executes a procedure call, the process state makes a logical "context switch" by pushing a new frame onto the activation record stack. This new activation record contains the procedure parameters, variables local to the procedure, and various pointers for accessing non-local variables and for restoring the old context after the procedure completes. Upon returning from the procedure, the topmost frame is popped from the activation record stack. Although all static-scoped languages follow this approach, the exact format of the activation records depends on the compiler and the architecture of the machine.

The obvious approach to capturing and restoring state is to write machine-specific programs that translate the run-time stack and data areas into an abstract format, then back to another machine-specific format. Our solution add a layer of abstraction: these translation programs are written in a high-level language, namely the same language used to write the module. Thus all machine-specific details are generated by the standard compilers provided with the machine. The example is Section 2 illustrates how this activation record capture and restoration work.

2 The Monitor Example

The Monitor example is a distributed application containing three modules, each of which can be distributed to a different machine. The reconfiguration performed in this example is to move one of the modules to another machine while the application executes. The starting configuration is shown in Figure 1 (left): module **sensor** produces temperature values at regular intervals, module **display** requests a value then displays it, and upon request module **compute** performs a computation on a group of temperature values and returns the result. In the ending configuration shown in Figure 1 (right), the **compute** module has been relocated to another machine.

Although the computation performed in the **compute** module is merely to average a group of temperature values, in order to best illustrate the mechanism used to capture the activation record stack, we have used a recursive algorithm to perform this computation, and placed the reconfiguration point within the recursive procedure. Thus moving the **compute** module during execution requires capturing the state of the activation record stack in the midst of these recursive calls.

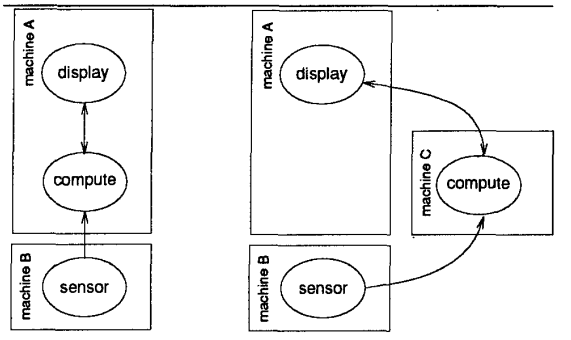

Figure 1: The Monitor Example Before Reconfiguration (left); After Reconfiguration (right)

The POLYLITH configuration specification for this application is shown in Figure 2. In the configuration specification, each of the three modules is described by a *module specification*, which defines the interfaces of the module, where the executable resides, and other attributes of the module. The *application specification* lists the modules used in the application and the bindings between interfaces. In order to make this application reconfigurable, the only change made to the configuration specification was to define the reconfiguration point **R** in the module specification for **compute**, and list the variables comprising the process state at that reconfiguration point. This reconfiguration point corresponds to a label **R** inserted by the programmer into the source code for module **compute**.

The source code for the **compute** module is shown in Figure 3. It loops forever, checking for requests on the **"display"** interface. If one arrives, it recursively computes the average of **n** temperature values read from the **"sensor"** interface. When no requests are pending, it discards any available temperature values by trivially computing the average of one value.

When reconfiguration is requested, the **compute** module will continue execution until it reaches reconfiguration point **R**. Because the reconfiguration point is inside the recursive procedure, at reconfiguration time there will be one or more activation records for

```
module display {
  source="./display.out" ::
  client interface temper
     pattern = {integer} accepts{^float} :: }

module compute {
  source="./compute.exe" ::
  server interface display
     pattern = {^integer} returns{float} ::
  use interface sensor pattern = {^integer} ::
  reconfiguration point = {R} :: }

module sensor {
  source="./sensor.exe" ::
  define interface out pattern = {integer} :: }

module monitor {
  instance display
  instance compute
  instance sensor
  bind "display temper" "compute display"
  bind "sensor out" "compute sensor" }
```

Figure 2: Application Configuration Specification

this procedure at the top of the stack. Since the recursive procedure could have been called from one of three places within the module, the activation record just below the top-most one in the stack can correspond to any one of these three calls.

2.1 Module Participation

During reconfiguration, it is the responsibility of the module to delay reconfiguration until the appropriate point, package up its state, and install state in a dynamically created module. In the monitor example, module **compute** is moved to another machine, so this module must be prepared to participate during reconfiguration. Our method of automatically providing this module participation is to pre-process the source program for the module, adding code to capture and restore the process state at the reconfiguration point specified by the programmer. When the reconfiguration point is located in a procedure other than the main procedure, the state of each procedure in the activation record stack must be captured and restored.

Figure 4 shows the source code for module **compute** after the module participation statements (in slanted typeface) have been automatically inserted. When a reconfiguration signal arrives, the flag **mh_reconfig** is turned on, and the module continues executing until it reaches the block of code just above the reconfiguration point R. Inside this capture block, the **mh_reconfig** flag is turned off, the **mh_capture_stack**

```
main(argc, argv)
int argc; char **argv;
{  int n; double response; void compute();

   mh_init (&argc, &argv, NULL, NULL);
   while (1) {
      /* handle requests for temp */
      while (mh_query_ifmsgs("display")) {
         mh_read("display","i",NULL,NULL,&n);
         /* compute avg of n temps */
         compute (n, n, &response);
         mh_write("display","F",NULL,NULL,response);
      }
      /* keep sensor buffer clear */
      if (mh_query_ifmsgs("sensor")) {
         compute (1, 1, &response);
      }
      sleep(2);
   }
}

void
compute(num, n, rp)
int num, n; double *rp;
{  int temper;

   if (n<=0) { *rp = 0.0; return; }
   compute (num, n-1, rp);
R: mh_read ("sensor","i",NULL,NULL,&temper);
   *rp = *rp + ((double)temper / (double)num);
}
```

Figure 3: Original Compute Module

flag is set, the state specified by the programmer is captured, and the procedure returns, thus completing the capture and pop of the top-most activation record.

The procedure containing the reconfiguration point could have been called from one of three statements, labeled here by **L1, L2, or L3**. Immediately following each of these three statements is another capture block. This block is executed when the **mh_capture_stack** flag is set: it simply captures the local state and returns. The differences between the capture blocks in the main and those in the procedure are that the local state is different, and that the main contains an **mh_encode()** to send the captured state outside the module. The capture block just after **L3** will execute once for each recursive call on the activation record stack. The bottom-most activation record is captured and popped by one of the capture blocks in the main.

In each of the calls to **mh_capture**, in addition to the local variables, an integer **1, 2, 3,** or **4** is captured. This integer value corresponds to a label marking the statement where execution should resume dur-

```
main(argc, argv) int argc; char **argv;
{ int n; double response; void compute();
   mh_init (&argc,&argv,NULL,NULL);
/* ------------------------------------------------------------------ begin restore -------- */
   if (strcmp(mh_get_status(),"clone")==0) mh_restoring=1; else mh_restoring=0;
   if (mh_restoring) {
      mh_decode(); mh_restore("iif",&mh_location,&n,&response);
      if (mh_location==1) goto L1;
      if (mh_location==2) goto L2;
   }
   signal(SIGHUP,mh_catch_reconfig);
/* ------------------------------------------------------------------ end restore --------- */
   while (1) { /* handle requests for updated temperature */
      while (mh_query_ifmsgs ("display")) {
         mh_read("display","i",NULL,NULL,&n);
L1:      compute (n, n, &response);
/* ------------------------------------------------------------------ begin capture -------- */
         if (mh_capture_stack) {
            mh_capture("IIF",1,n,response); mh_encode();
            return; }
/* ------------------------------------------------------------------ end capture --------- */
         mh_write("display","F",NULL,NULL,response);
      }
      if (mh_query_ifmsgs ("sensor")) {
L2:      compute (1, 1, &response);
/* ------------------------------------------------------------------ begin capture -------- */
         if (mh_capture_stack) {
            mh_capture("IIF",2,n,response); mh_encode();
            return; }
/* ------------------------------------------------------------------ end capture --------- */
      }
      sleep(2);
   }
}
void compute(num, n, rp) int num, n; double *rp;
{ int temper;
/* ------------------------------------------------------------------ begin restore -------- */
   if (mh_restoring) {
      mh_restore ("iiif",&mh_location,&num,&n,rp);
      if (mh_location==3) goto L3;
      if (mh_location==4) {
         mh_restoring=0; signal(SIGHUP,mh_catch_reconfig);
         goto R; }
} /* ------------------------------------------------------------------ end restore --------- */
   if (n<=0) { *rp=0.0; return; }
L3:compute (num, n-1, rp);
/* ------------------------------------------------------------------ begin capture -------- */
   if (mh_capture_stack) {
      mh_capture("IIIF",3,num,n,*rp);
      return; }
/* ------------------------------------------------------------------ end capture --------- */
/* ------------------------------------------------------------------ begin capture -------- */
   if (mh_reconfig) {
      mh_reconfig=0; mh_capture_stack=1;
      mh_capture("IIIF",4,num,n,*rp);
      return; }
/* ------------------------------------------------------------------ end capture ----------*/
R: mh_read ("sensor","i",NULL,NULL,&temper);
   *rp = *rp + ((double)temper / (double)num);
}
void mh_catch_reconfig() { mh_reconfig=1; }
```

Figure 4: Module "compute" Prepared for Reconfiguration

ing restoration.

During restoration, the same code is executed for module **compute**, but it is assigned a status such that the flag **mh_restoring** is turned on. This flag remains on, triggering the restore blocks, until the activation record stack has been completely rebuilt. Each restore block restores the local state and jumps to the statement where execution should resume. When the final activation record is restored, the **mh_restoring** flag is turned off, and execution resumes at the reconfiguration point.

Since the code for divulging and installing program state is part of the source program, the compiler takes care of dereferencing variables from the activation record stack and of rebuilding the activation record stack during restoration. Thus the module thread is captured and restored without explicit reference to the program counter or to any of the call/return information stored in the activation record stack.

Because the state is captured and restored between statements in the high-level language, temporary or intermediate values used in a computation are never part of the process state. Variables cached in registers will be correctly captured and restored, because the capture and restore statements are embedded in the source program, and the compiler takes care of referencing these variables correctly.

2.2 Application-level Changes

To perform dynamic reconfiguration, application-level changes and module participation are coordinated in a reconfiguration script, which is a procedural description of the events occuring during reconfiguration. The reconfiguration script for the monitor example appears in Figure 5; this script performs replacement of the **compute** module.

First the script obtains the specification for the existing module, in order to determine its bindings and eventually delete the module and bindings. This module specification contains the same items as those supplied in the original configuration specification (Figure 2), but it corresponds to the *current* configuration, which could have been changed dynamically. The new module uses the same specification with a new **MACHINE** attribute, and a **STATUS** value that indicates the module is a restoration.

Next the rebinding commands are prepared. Bindings to the old module's interfaces are replaced by bindings to the new module's interfaces of the same name. The rebinding commands are applied all at once, after the old module has divulged its state. The **mh_objstate_move** operation signals the old module to

```
mh_obj_cap(&old,"compute");     /* access old module */
mh_obj_cap(&new,"compute");     /* create new module */
mh_edit_objattr(&new,"add","MACHINE",new_site);
mh_edit_objattr(&new,"add","STATUS","clone");

mh_bind_cap(&b);          /* prepare binding commands */
mh_struct_objnames(&old,if,&num_if);
for (i=0; i<num_if; i++) {
                                  /* rebind outgoing */
    mh_struct_ifdest(&old,if[i],bind,&num_bind);
    for (j=0; j<num_bind; j++) {
        mh_edit_bind(&b,"del",&old,if[i],bind[j],NULL);
        mh_edit_bind(&b,"add",&new,if[i],bind[j],NULL);
    }
                                  /* rebind incoming */
    mh_struct_ifsources(&old,if[i],bind,&num_bind);
    for (j=0; j<num_bind; j++) {
        mh_edit_bind(&b,"del",bind[j],NULL,&old,if[i]);
        mh_edit_bind(&b,"add",bind[j],NULL,&new,if[i]);
        mh_edit_bind(&b,"cpq",&old,if[i],&new,if[i]);
        mh_edit_bind(&b,"rmq",&old,if[i],NULL,NULL);
    }
}
       /* get state from old module, send it to new */
mh_objstate_move(&old,"encode",&new,"decode");

mh_rebind(&b);              /* apply binding commands */
mh_chg_obj(&new,"add");       /* start up new module */
mh_chg_obj(&old,"del");         /* remove old module */
```

Figure 5: Reconfiguration Script for Replacement

divulge its state, waits until the old module has complied, and sends this state to the new module. The final step is to start up the new module and remove the old.

This reconfiguration script is easily parameterized to accept a module name and attributes. The parameterized reconfiguration script could be used to replace a module in any application, provided the module had been prepared to participate during reconfiguration.

3 Source Code Transformation

The monitor example described in Section 2 illustrates how the activation record stack is captured and restored. This section describes the general technique for transforming a source program with reconfiguration points specified by the programmer into a reconfigurable source program. Several issues that must be resolved for the general case did not arise in the monitor example. A program may have more than one reconfiguration point; in such a case, the question is whether each reconfiguration point must have its own capture and restore blocks, or all reconfigu-

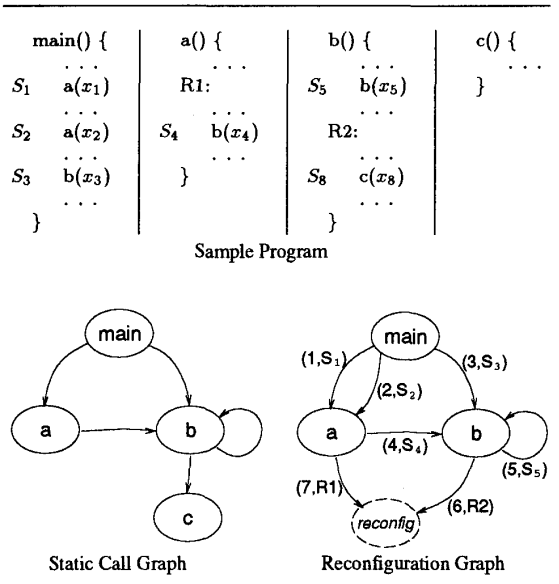

Figure 6: Example of Static Call Graph and Reconfiguration Graph

ration points can share the same capture and restore blocks. A more subtle problem is the potential discrepancy between the local state used to restore the activation record stack and the local state when the original procedure call was made.

The static call graph of a program contains a node for each procedure/function in the program, and a directed edge from node *a* to node *b* if and only if the source code for procedure *a* contains a call to procedure *b*. All nodes in this graph have one or more incoming edges except for the node corresponding to the *main* procedure, which has only outgoing edges. Figure 6 shows an example program and its corresponding static call graph. The static call graph is determined by examining the source program, not by analyzing the run-time behavior. At any particular time during program execution, the frames contained in the activation record stack correspond to a path in the static call graph originating at node *main*. Thus the static call graph defines all possible activation record stacks.

Because we allow reconfiguration to occur only at reconfiguration points and not at any arbitrary point in the program execution, when reconfiguration occurs, only procedures containing a reconfiguration point can be at the top of the activation record stack. Thus only procedures which could be below these on the activation record stack need be instrumented for reconfiguration. In terms of the static call graph, only nodes on paths starting at *main* and ending at a procedure containing a reconfiguration point are of concern; these nodes and edges define a subgraph of the original static call graph. Each of the procedures in this subgraph, including the main and the procedure containing the reconfiguration point, must be prepared for reconfiguration.

The first step in preparing a program for reconfiguration is to augment this subgraph of the static call graph. The augmented subgraph, called the *reconfiguration graph*, contains an edge for each procedure call, and each edge is labeled with the line number of the call. Thus if procedure *main* calls *a* in two different statements, there are two edges from *main* to *a*. The reconfiguration graph also contains a new node, named *reconfig*, and an edge from each reconfiguration point to the *reconfig* node, annotated with the line number of the reconfiguration label. In addition, the edges in the reconfiguration graph are numbered consecutively, so each edge is labeled (i, S_i), where i is an integer and S_i is a line number. These edges define the places where the program state is potentially captured and restored. Figure 6 shows the reconfiguration graph and its corresponding program.

The second step is to install code to capture and restore the program state. Each node in the reconfiguration graph will receive a single restore block and one or more capture blocks. For each edge originating at that node, restore code is inserted into the restore block, and a capture block is installed at the line number associated with that edge.

There are two kinds of capture blocks that must be inserted in order to reconfigure. The first is used at a reconfiguration point, and the second is used to capture the state of the activation record stack. The only difference between these two types of blocks is that they are triggered by different flags (Figure 7). The flag set in the reconfiguration signal handler triggers the blocks installed at reconfiguration points, and these blocks set the flag which triggers the blocks installed for activation record capture.

For each edge terminating at node *reconfig*, a capture block for that reconfiguration point is installed immediately preceding the reconfiguration label installed by the programmer. For each remaining edge i associated with statement S_i in the reconfiguration graph, a label `Li` is inserted at the statement immediately following S_i, and a capture block is installed immediately preceding label `Li`.

Notice that a single capture block is installed after

Capture Block for Reconfiguration Edge (j,R):

```
    if (mh_reconfig) {
        mh_reconfig = 0;
        mh_capture_stack = 1;
        mh_capture (j, local vars);
        return;
    }
R:
```

Capture Block for Edge (i,S_i):

```
Si: f(x);
    if (mh_capture_stack) {
        mh_capture (i, local vars);
        return;
    }
Li:
```

Figure 7: Capture Blocks

```
if (mh_restoring) {
    mh_restore (&mh_location, local vars);
    restore code for each edge
}
```

Restore Code for Edge (i,S_i):

```
    if (mh_location==i) {
        f(x);
        goto Li;
    }
```

Restore Code for Reconfiguration Edge (j,R):

```
    if (mh_location==j) {
        mh_restoring = 0;
        install reconfiguration signal handler
        goto R;
    }
```

Figure 8: Restore Block

each procedure call that could be interrupted by a reconfiguration. In our example, *main*'s call to *a* in line S_1 could be interrupted by a reconfiguration at R1 (in procedure *a*) or at R2 (in procedure *b*). Capturing the state of *main* at S_1 does not depend on which reconfiguration point triggered the capture, so reconfiguration points can share capture blocks.

We have not yet discussed how to automatically determine which variables should be captured in these capture blocks. At the present time, the programmer includes this information as part of the specification of a reconfiguration point. For capturing the state of the activation record stack, the relevant variables are the parameters and local variables of a procedure. At a reconfiguration point, data-flow analysis could be used to determine the set of live variables. The primary difficulty in automatically determining which variables to capture arises with pointer variables. Since pointers are addresses, they must be translated into an abstract format for capture and restoration. For example, a pointer variable containing an explicit address would be translated into a variable that points to the n^{th} character of a string located at some symbolic address.

A restore block is inserted at the top of each procedure present in the reconfiguration graph. Inside this restore block the local state is restored, then there is restore code for each edge originating at that node (Figure 8). Again, the restore code for a reconfiguration point differs slightly from the code for restoring the activation record stack. If the state capture was triggered by reconfiguration point R, then the activation record stack has been completely restored and after jumping to R, reconfiguration is complete. If the state capture was triggered by a return from a procedure call interrupted by reconfiguration, then the restore code repeats the procedure call and jumps to the statement immediately following the original procedure call.

Here arises the question of whether it is acceptable to simply repeat the original procedure calls when restoring. The local state at the time of the original procedure call is not guaranteed to be the same as the local state at the time it is captured, because the called procedure may change the value of variables that are visible to the callee. Thus the values of the arguments when the procedure is originally called can be different from their values when the procedure is reinvoked during restoration. On the surface this discrepancy appears to be acceptable, since during restoration the first action of the called procedure is to restore its own local state, including all parameter values, so the values of arguments passed during restoration are inconsequential. However, a problem can arise when the arguments are not scalars but are expressions. When the original procedure call is repeated during restoration, these expressions are evaluated with the restored state, and their evaluation can cause a run-time error that did not arise when they were evaluated with the original state. The solution to this problem is to not repeat the original procedure call, but to modify the call by substituting dummy arguments for expressions whose evaluation could result in a run-time error. The

data types of these dummy arguments are determined by the types declared in the parameter list of the procedure.

4 Discussion

Our approach does not use checkpointing, in which the entire state of the process is saved periodically, and execution is rolled back to the most recent checkpoint in order to restore the process. Instead, when a reconfiguration is requested, the process continues executing until it reaches the next reconfiguration point. Thus the run-time cost is merely that of periodically testing the flags installed for reconfiguration. The cost of capturing the process state is paid only when a reconfiguration is performed, instead of at regular intervals during execution.

The frequency of flag testing depends on how many reconfiguration points are inserted, and where they are placed. In order for a module to quickly respond to a reconfiguration request, the reconfiguration points must be located within the most frequently executed code. However, for applications with an execution time on the order of days rather than seconds, placing reconfiguration points where they will be checked *regularly* is more important than placing them where they will be checked frequently. Reconfiguration points located in deeply-nested procedures or procedures that are called from many places increases the occurrence of reconfiguration flags in the source code, but the execution cost due to testing these flags depends on how often the procedure is actually invoked.

By virtue of where a reconfiguration point is placed, it could prohibit certain compiler optimizations such as code motion. Because these reconfiguration techniques are intended for long-running or continuously available applications, a reconfiguration delay measured in seconds rather than micro-seconds may be perfectly acceptable. If so, it is preferable to place reconfiguration points outside of computationally intensive loops or procedures, so that the code executed most often can be optimized as much as possible.

A reconfiguration platform supports dynamic updates at a particular level of atomicity: updates can be atomic at the module level, at the procedure level, or at the statement level. If the reconfiguration is atomic at the module level, it means that modules execute atomically with respect to reconfiguration; a module cannot be updated while it is executing. Platforms providing this level of support are those that reconfigure without module participation, such as [9].

A system that supports updates with procedure-level atomicity is described in [4]. This system is restricted to updating a program without moving it from the original machine. The program is updated by replacing each procedure when it is not executing. To maintain consistency between the old version and the new during the replacement, they perform the update from the bottom up, by allowing a procedure to be replaced only after all the procedures it invokes have been replaced. The implications of this update strategy are that programs written in a top-down style will be updated more successfully than those that are not modularized. When changes to the program are restricted to the lower-level procedures, updates can be performed quickly, but when the higher-level procedures have changed, the update cannot complete until these procedures are inactive. For example, when the `main` procedure has changed, the update cannot complete until the program terminates. When a procedure has some cached state that must be installed in the new version, the programmer must write a special routine to do this.

A platform that preserves atomicity at the statement level could support reconfiguration either at every statement, or at certain reconfiguration points specified by the programmer. We take the latter approach, as does the reconfiguration framework of Conic [6]. In Conic, reconfiguration activities are separated into configuration level concerns and application level concerns. Configuration level activities are independent of the algorithms, protocols, and states of the application, and are guaranteed to leave the system in a consistent state, where consistency is defined in terms of the application. To ensure consistency after reconfiguration, the programmer writes code for the modules to respond to the configuration level commands `passivate`, `unlink`, and `link`. These commands correspond to the module-level activities of moving to a compatible state, capturing process state, and restoring process state.

The method proposed in [10] supports heterogeneous process migration (moving a module) between every possible statement. These *migration points* are the places where the abstract state defined by the high-level source program and the state in the binary correspond. To capture the process state at one of these migration points, they propose using a procedural interface to an existing source-level debugger. At migration time, a machine-independent *migration program* would be generated, compiled, and executed on the target machine. The migration program first reconstructs global and heap data, then rebuilds the

activation record stack by executing a sequence of calls to special procedures, which are modified versions of the procedures in the activation record stack at migration time. The modified procedures initialize local variables, call the next modified procedure in the call stack, and arrange to resume execution in the original procedure. Thus the details of the code and data translation are hidden in the compilers for each machine.

One of differences between our work and the technique described in [10] is that their goal is to support heterogeneous process migration, requiring that migration points be transparent to the programmer and that they be available at as many places in the source code as possible. Our goal is to support dynamic reconfiguration: we expect the programmer to be involved in selecting a small set of reconfiguration points. Because the number of reconfiguration points is relatively small, we can prepare the program for *all possible* reconfigurations when the original program is compiled, whereas they prepare a migration program for only the specific migration requested, thus must prepare it at migration time. For the same reason, we can prepare the program to capture the process state in addition to restoring it, instead of using one technique for capturing process state and a different technique for restoring it.

5 Conclusion

Using the approach described in this paper, we are able to automatically generate the code needed to support module participation, while other dynamic reconfiguration environments rely on the programmer to write this code. Given reconfiguration points specified by the programmer, we augment the semantics of the module, giving it the capability of divulging and installing its state at these reconfiguration points. This module state is expressed in an abstract format, thus permitting executing modules to be moved to different architectures or to be updated for maintenance purposes. Because the statements that capture and restore a module's state, including the state of its activation record stack, are part of the module's source code, the compiler handles the machine-specific translation details, and no changes are made to the compiler nor to the underlying operating system.

References

[1] A. Aho, R. Sethi, J. Ullman, "Run-Time Environments," **Compilers: Principles, Techniques, and Tools**, Addison-Welsey, Chapter 7, pp. 389-462, 1986.

[2] M. Bach, **The Design of the Unix Operating System**, Prentice-Hall, Chapters 6-7, pp. 146-246, 1986.

[3] M. Barbacci, D. Doubleday, C. Weinstock, "Application-Level Programming," *Proceedings of the 10th International Conference on Distributed Computing Systems*, pp. 458-465, 1990.

[4] O. Frieder, M. Segal, "On Dynamically Updating a Computer Program: From Concept to Prototype," *Journal of Systems and Software*,vol. 14, pp. 111-128, 1991.

[5] C. Hofmeister, E. White, J. Purtilo, "SURGEON: A Packager for Dynamically Reconfigurable Distributed Applications," *IEE Software Engineering Journal*, to appear March 1993.

[6] J. Kramer, J. Magee, "The Evolving Philosophers Problem: Dynamic Change Management," *IEEE Transactions on Software Engineering*, vol. 16, no. 11, pp. 1293-1306, 1990.

[7] T. Pratt, **Programming Languages: Design and Implementation**, Prentice-Hall, Chapters 6-8, pp. 149-302, 1984.

[8] J. Purtilo, "The Polylith Software Toolbus," To appear, *ACM TOPLAS*. Currently available as *University of Maryland CSD Technical Report 2469*, 1990.

[9] J. Purtilo, C. Hofmeister, "Dynamic Reconfiguration of Distributed Programs," *Proceedings of the 11th International Conference on Distributed Computing Systems*, pp. 560-571, 1991.

[10] M. Theimer, B. Hayes, "Heterogeneous Process Migration by Recompilation," *Proceedings of the 11th International Conference on Distributed Computing Systems*, pp. 18-25, 1991.