SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2002; **32**:1–27 (DOI: 10.1002/spe.464)

Data collection and restoration for heterogeneous process migration

Kasidit Chanchio and Xian-He Sun*,†

Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616, U.S.A.

SUMMARY

This study presents a practical solution for data collection and restoration to migrate a process written in high-level stack-based languages such as C and Fortran over a network of heterogeneous computers. We first introduce a logical data model, namely the Memory Space Representation (MSR) model, to recognize complex data structures in process address space. Then, novel methods are developed to incorporate the MSR model into a process, and to collect and restore data efficiently. We have implemented prototype software and performed experiments on different programs. Experimental and analytical results show that: (1) a user-level process can be migrated across different computing platforms; (2) semantic information of data structures in the process's memory space can be correctly collected and restored; (3) costs of data collection and restoration depend on the complexity of the MSR graph in the memory space and the amount of data involved; and (4) the implantation of the MSR model into the process is not a decisive factor of incurring execution overheads. With appropriate program analysis, we can practically achieve low overhead. Copyright © 2002 John Wiley & Sons, Ltd.

KEY WORDS:

1. INTRODUCTION

As network computing becomes an increasingly popular choice for computing, network process migration has recently received unprecedented attention. One driving force behind process migration is its support for fault tolerance (fault-driven): to migrate processes from the faulted machine to other machines when a fault is detected. Checkpointing is commonly used for fault tolerance. Checkpointing enables the execution of the code to be resumed from a previously saved state (checkpoint) rather

*Correspondence to: Xian-He Sun, Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616, U.S.A.
†E-mail: sun@cs.iit.edu

Contract/grant sponsor: National Science Foundation; contract/grant number: ANI-0123930, ACI-0130458
Contract/grant sponsor: IIT ERIF program

Received 31 March 2000
Revised 26 February 2002
Accepted 26 February 2002

than its beginning; thus, the damage caused by the fault can be limited to a tolerable degree. Since a checkpointed process can be restarted at other machines in a distributed environment, fault tolerance can be achieved either by resuming the process at the same machine after the fault is recovered or resuming at a new machine via network transfer. Another driving force of process migration is performance (performance-driven): to migrate processes from one machine to another for better performance. Load balance is one of the motivations of performance-driven migrations. Recent research shows process migration is necessary for load balancing in non-dedicated distributed environments [1–3]. Other reasons for performance-driven migration include data access locality, migrating processes towards the source of the data, and resource sharing, migrating processes toward the source of appropriate hardware and software. Emerging new applications such as collaborative computing, ubiquitous immersion computing, and network grid computing [4], require the support of process migration. Migration provides the mobility of computing. Efficient process migration is recognized as a critical issue for next generation network environments [5]. Although a few approaches for heterogeneous process migration have been presented [6–10], due to its complexity, no wildly accepted or standard solutions currently exist for efficient heterogeneous process migration for traditional stack-based languages such as C or Fortran. A current success in mobile computing is the creation of Java. Java is a good choice for many applications. However, Java's status as a scientific programming language has been continuously under question due to its poor performance on computationally intensive applications and its simplified data structures.

In this study, we present a new methodology of data collection and restoration that enables sophisticated data structures such as pointers to be migrated appropriately in a heterogeneous environment. This methodology analyzes the pre-stored or current program state for heterogeneous process migration and can be used for both fault-driven and performance-driven purposes. Our approach is highly independent of computer hardware, operating system and compilation tools, and can reduce the costs of process migration by transferring only live data, the data needed for further computation beyond the point where process migration take place, during process migration. It has also been applied and implemented to migrate applications written in the C language. It is an important step toward the search for a general solution to network process migration. This study may also be of benefit to strengthening the performance of mobile languages, such as Java.

Fundamentally, there are three steps that enable process migration in a heterogeneous environment on existing code.

1. Identify the subset of language features which are migration-safe, i.e. features that theoretically can be carried across a network.
2. Choose a methodology to perform heterogeneous process migration. We define processes which can be migrated based on the chosen methodology as being 'migratable'.
3. Develop mechanisms to migrate the 'migratable' process reliably and efficiently.

Smith and Hutchinson [8] have identified the migration-unsafe features of the C language. With the help of a compiler, most of the migration-unsafe features can be detected and avoided. Different approaches have been proposed during the last two decades for heterogeneous process migration, including our recently proposed *migration-point*. In the migration-point approach, procedures and data structures are given to transform a high-level program into a migratable format automatically via a pre-compiler. The transformation process includes migration-point analysis, data analysis, and the insertion of migration macros [10,11]. The basic ideas of the migration-point methodology are discussed in

Section 2. The focus of this study is on the last step, mechanisms for carrying out migration correctly and efficiently. Our mechanisms include:

1. recognizing the complex data structures of a migrating process for heterogeneous process migration;
2. encoding the data structures into a machine-independent format;
3. transmitting the encoded information stream to a new process on the destination machine; and
4. decoding the transmitted information stream and rebuild the data structures in the memory space of the new process on the destination machine.

Other design alternatives for encoding and decoding the state information may also be possible. The need for different conversion mechanisms for different computing platforms could be avoided by using a common machine-independent format. A machine-independent format would be more appropriate for general enterprise network environments.

We have designed and implemented the data collection and restoration mechanisms to support process migration of applications written in any stack-based programming languages with the presence of pointers and dynamic data structures. A prototype runtime library has been developed to support process migration of migration-safe C code [8] in a heterogeneous environment. Experimental measurements of C programs with different dynamic data structures and execution behaviors have been performed. Experimental results are very encouraging. They confirm that: (1) a user-level process can be migrated across different computing platforms; (2) semantic information of data structures in the process' memory space can be correctly collected and restored; (3) the costs of data collection and restoration depend on the complexity of the data structures involved; and (4) with appropriate program analysis, we can practically achieve low overhead throughout the program execution. Although runtime overheads may occur due to the insertion of data collection and restoration macros, we have made a number of observations on the sources of the overheads and how they might be avoided.

This paper is organized as follows. Section 2 gives background discussions on how a process is migrated and what the difficulties of migration are in the presence of pointer-based data structures. Section 3 introduces a logical model to describe data elements and pointers at a snapshot of the program memory space. We present data collection and restoration mechanisms in Section 4. Section 5 shows the implementation and experimental results of three programs with different execution behaviors. We discuss related works in Section 6. Finally, Section 7 gives conclusions and a discussion of future work.

2. BACKGROUNDS

In the C language, some language features and programming practices can be unsafe for process migration. Some of them can prohibit process migration, while others require additional supportive mechanisms. Some language features, especially those that generate machine-specific intermediate outputs, may prohibit heterogeneous process migration. For example, if integer values are converted from pointer addresses, they may be meaningless or misleading when the process migrates to a new machine.

Nevertheless, the presence of these features in a process alone does not mean process migration is impossible. In fact, the consideration is rather application specific. For the pointer conversion example, process migration would not be possible if the computational logic of a process depends highly on the memory addresses of the source machine. On the other hand, the migration is allowed when the computation does not involve or depend on machine-specific data. In our design, we employ a program analysis technique to notify users of the existence of these features and their use in source code and let users decide whether to apply the migration mechanisms.

There are also other language features that do not prohibit heterogeneous process migration, but need supportive mechanisms to make it possible. In many cases, problems arise when specific properties that are needed for data collection and restoration during the migration cannot be determined at compile time. As a result, we need to apply certain rules to handle these features during a migration.

For example, the uses of `union` can cause a memory block to hold data values whose type cannot be determined at compile-time. As a result, we do not know how to encode and decode the data values into an appropriate machine-independent format during a heterogeneous process migration. In our current solution, we assume that `union` is used for saving memory space rather than data conversion. Thus, we simply replace `union` with `struct` to avoid type ambiguity.

Another example of such features is dynamic memory allocation. Programmers may use `malloc()` to allocate a memory block of arbitrary length and later cast the memory block addresses to pointers of arbitrary type. The casting can cause the type of data in the memory block to change at runtime. In our solution, we put additional constraints to the uses of dynamic memory allocation. We require all dynamic memory allocation to use our wrapper function which requires the programmer to specify a data type to the created memory block. During process migration, data values in memory blocks are encoded and decoded based on this type information.

Another feature of interest is the function pointer, which can cause a change of execution flow that may be hard to determine at compile time. We will discuss our solution in the next section.

2.1. Source code annotation

In our design, a program must be transformed into a ‘migratable’ format. As introduced in our previous work [11], we apply source code annotation to insure the program is migration capable. In the annotation process, we first select a number of locations in the source code on which process migration can be performed. We call such a location a ‘poll point’. At each poll point, a label statement and a specific macro containing migration operations are inserted. Every time the process execution reaches the poll point, the macro will check whether a migration request has been sent to the process. If so, the migration operation is executed. Otherwise, the process continues normal execution. We refer to the poll point where the migration occurs as the ‘migration point’. The migration operations include the operations to collect the execution state and live data of the migrating process, and the operations to restore them on the memory space of a process on another machine. In our design, the selection of poll points and the insertion of macros are performed automatically by the source code transformation software (or pre-compiler). Users can also select their preferred poll points if they know suitable migration locations in their source codes.

The pre-compiler consists of different phases of program analysis and source code annotation mechanisms. The program analysis techniques consist of poll-point analysis and live variable analysis. The poll-point analysis identifies poll-point locations in source code. The pre-compiler investigates function body one by one. If poll points are *selected* (manually or automatically) in a function, the poll-point analysis will insert *associated* poll points at every function call statement made to the function. The reason for this is to identify a sequence of function calls when a migration is performed at any selected poll point.

In case of function pointers, we assume that the pointers could invoke any function including those that have poll points. As a result, we insert an associated poll point at every function call statement to a function pointer in the source code.

After selected poll points have been identified, we apply live variable analysis on them to define a set of variables whose values are needed for future computation. For associated poll points, a slightly different technique is used. We define live variables at the location right after the function call statement rather than at the location where the call statement is made. The reasons for this will be given in the next section.

Finally, the pre-compiler annotates a set of macros into various locations in the source code. For a selected poll point, we insert macros to collect and restore data values of live variables. For an associated poll point, we insert two macros immediately before and after the function call statement, respectively. The former macro keeps a record of the function calling. The latter macro performs the data collection and restoration of the live variables as being identified as the output of live variable analysis of the associated poll point. These macros work together during process migration to perform a data-transfer mechanism to be discussed later.

In our process migration environment, we assume that the source program has been pre-distributed and compiled on potential destination machines. We model a distributed environment to have a scheduler, which performs process management and sends a migration request to a process. The scheduler conducts process migration directly via a remote invocation and network data transfers. First, the process on the destination machine is invoked to wait for the execution state and the live data of the migrating process. Then, the migrating process collects this information and sends it to the waiting process. After successful transmission, the migrating process terminates. Concurrently, the new process receives the state information, restores it on appropriate memory locations, and resumes execution from the point where process migration occurred.

2.2. Data-transfer mechanism

The data-transfer mechanism is a mechanism that governs the collection, transfer, and restoration of execution status and memory contents of a process when multiple or nested function call are made. A migration point can be inside a nested function call or a recursion during a migration situation. In our design, we collect and restore live data of the innermost function first and those of its callers subsequently. For example, the function call sequence, `main` → `f1` → `f2`, means the function `main` calls `f1` and `f1` calls `f2`. Figure 1 shows an example process migration from a source to a destination computer. Suppose a migration occurs within `f2`. The live data are then collected in the order of function `f2`, `f1`, and `main` accordingly. From the figure, when program execution reaches the migration point in `f2()`, the annotated migration operations recognize the sequence of function calls as the execution state and then collect the live data of `f2` before returning to its caller function, `f1`. Note that the rest of the execution in `f2` is abandoned. In `f1`, live data is collected and the execution returns to the `main` function. Finally, the migration operations collect live data in `main`, send the execution state and the collected live data to the destination computer, and terminate the migrating process.

After receiving the execution state and live data from the migrating process, the new process at the destination machine restores the execution state by executing a series of ‘`goto`’ and function calls to reconstruct the nested function calls identical to that of the migrating process. Then, it restores

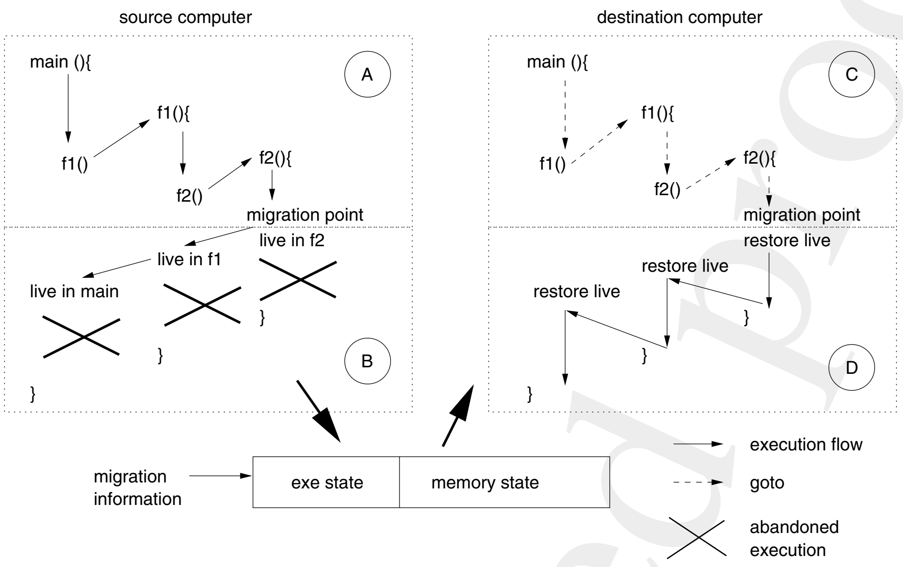


Figure 1. An example process migration.

the live data for the innermost function and resumes execution there until the function finishes. Upon returning to the caller function, its live data would be immediately restored before the function resumes execution. Note, that these repeating operations are the result of the macros associated with the associated poll points previously discussed.

From the `main` → `f1` → `f2` example in Figure 1, live data of `f2` will be restored first. Then, the function `f2` continues execution until the computation of that function ends. After the execution flow returns to the caller function `f1`, the live variables at the associated poll points are restored. Then, the function `f1` continues execution until `f1` finishes. The live variables of the associated poll points in the main function are restored afterward. After that, the main function continues computation until the program finishes.

To collect and restore live data, special purpose interfaces are applied to collect and restore the values of live variables. Live variables at every poll point are defined based on live variable analysis performed by the pre-compiler. We have developed the `Save_memory` and `Restore_memory` functions to collect and restore variable contents, respectively. In the augmented macros, the functions are applied to live variables in the same order of data collection and restoration.

2.3. Process memory space

Generally, in a process memory space, every variable occupies a piece of memory containing data values of a certain type. We call this piece of memory a *memory block*. A memory block may reside in any part of a program memory space: global, stack, or heap segments. It consists of an array of data values with the same type. The data type could be primitive data types such as character, integer, and pointer, or compositional types such as array or structure. A pointer is a memory address. It is considered as a primitive data type which allows referential relationships to be created among memory blocks.

Despite the common uses of pointers and their well-known benefits, the referential capability of pointer poses difficulties in data collection and restoration. It is easy to transfer non-pointer data values across heterogeneous computers. The data values can be converted to network format such as XDR [12] for data transmission, or they can be converted directly from the source to the destination computers' formats. On the other hand, transferring pointer values is non-trivial. Since a pointer is a platform-specific memory address, the pointer values in one machine do not make sense to other machines. Moreover, pointers could represent a referential relationship among memory blocks crossing different memory segments (e.g. from memory blocks in a stack segment to a heap segment and *vice versa*). The pointer variables can also have their pointer values changed dynamically at runtime (by pointer assignment or arithmetic). In complex program data structures, the circular referential relationships among memory blocks can also be created. A scheme to migrate data in memory space with the presence of pointers must be able to handle these complexities.

As an overview to our solution, the process migration mechanism is strongly based on a logical memory model, namely the Memory Space Representation (MSR), machine-independently representing memory blocks and pointers in process memory space. Based on the model, the transfer of memory space contents can be described in four stages, consequently corresponding to areas `A`, `B`, `C`, and `D` in Figure 1. First, during normal operation before migration, the process records the properties of memory blocks created in its memory space in a data structure called the *MSR Lookup Table* (MSRLT). This data structure works as a mapping table between the conceptual MSR model and the physical representation of the process memory space. It also gives a logical addressing mechanism which allows the memory blocks to be machine-independently accessed. Second, during process migration, operations in migration macros invoke `Save_memory` to collect data from live variables. The routines scan the logical memory model to collect the contents of memory blocks as well as their logical addresses into a machine-independent format. Third, after the migration information is transmitted to the destination computer, the new process rebuilds the mapping and MSRLT data structures while executing a series of `gotos` to resume the execution state. Finally, data restoration operations are performed by invoking `Restore_memory` to extract memory contents and a logical address out of the transmitted migration information. Then, the data contents are assigned to the memory space of the destination process at the physical locations corresponding to given logical addresses. Note that such physical locations can be derived from the logical addresses using the MSRLT data structures previously rebuilt.

3. MEMORY SPACE REPRESENTATIONS

This section describes the logical model, the MSR, and its associated operations for data collection and restoration for process migration in a heterogeneous environment. We model a *snapshot* of a program memory space as a graph G, defined by $G = (V, E)$ where V and E are the set of vertices and edges, respectively. It is called the MSR graph. Each vertex in the graph represents a memory block, whereas each edge represents a relationship between two memory blocks when one of them contains a pointer, which is an address that refers to a memory location of any memory block node in V.

3.1. Representation of memory blocks

A memory block is a piece of memory allocated during program execution. It contains an object or an array of objects of a particular type. In case of the array, every object element has the same data type. Each memory block is represented by a vertex v in the MSR graph. The following terminologies are used in our study:

- *head(v)*, the starting address of the memory block v;
- *type(v)*, type of the object stored in the memory block;
- *elem(v)*, Number of objects of type $type(v)$ in the memory block.

When we refer to the *address of a memory block*, we mean any address within the memory block. The predicate $Address\_of(x, v)$ is true if and only if $head(v) \leq x \leq head(v) + (unit\_size \times elem(v))$ where *unit_size* is the size of an object of type *type*(v). For the sake of brevity, we use an example to illustrate the MSR graph concept.

Given a sample program in Figure 2(a), Figure 2(b) shows all the memory blocks in the snapshot of the program memory space right *before* the execution of the memory allocation instruction at line 20 of function `foo`, assuming that the *for* loop at line 13 in the `main` function had been executed *four* times before the snapshot was taken. Each memory block in Figure 2(b) is represented by a vertex v_i, where $1 \leq i \leq 12$. The associated variable name for each memory block is shown in parenthesis.

For local and global variables, a memory block is assigned separately for each variable. It only contains an array of objects if and only if it corresponds to an array declared in the source code. At line 5 of the program in Figure 2(a), although `first` and `last` are declared next to each other, we considered them to occupy separate memory blocks (`v1` and `v2`) in memory space instead of an array of size two. In our design, a memory block contains an array of objects iff it associates with an array variable declaration.

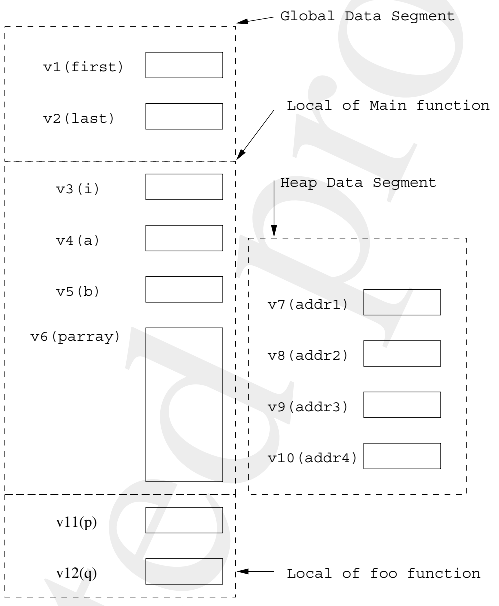
For memory blocks in the heap segment, we assign a memory block containing an array of objects when the process executes a dynamic memory allocation instruction. Let *addr* be an address in the program memory space, $addr_i$, where $1 \leq i \leq 4$, are addresses of dynamically allocated memory blocks created at runtime. We also use $addr_i$ as a memory block's name in this example.

The memory blocks can reside in different areas of the program memory space. If a memory block is created in the global data segment, it is called a *global memory block*. If it is created in the heap segment by dynamic memory allocation, we name it the *heap memory block*. In the case where the memory block resides in the activation area for a function f in the stack segment, it is called the *local memory block of function* f. Let *Gm*, *Hm*, Lm_{main}, and Lm_{foo} represent sets of global memory blocks, heap memory blocks, local memory blocks of function `main`, and the local memory block of function `foo`, respectively. From Figure 2, we can define $Gm = \{v_1, v_2\}$, $Hm = \{v_7, v_8, v_9, v_{10}\}$, $Lm_{\text{main}} = \{v_3, v_4, v_5, v_6\}$, and $Lm_{\text{foo}} = \{v_{11}, v_{12}\}$.

Figure 3 shows the MSR graph representing a snapshot of memory space of the sample program in Figure 2(a). The edges e_i, where $1 \leq i \leq 12$, represent the relationships between the pointer variables and the addresses of their reference memory blocks. General representations of pointers will be discussed in Section 3.4.

```
struct node {
   float data;
   struct node * link;
};
struct node *first, *last;
main()
{
   int i;
   int a, *b;
   struct node *parray[10];
   a = 1;
   b = &a ;
   first = parray[0];
   for ( i = 0; i < 10; i++ ){
      foo ( parray + i, &b );
      last  = parray[i];
      first->link = last;
      if( i > 0 )
         parray[i]->link = parray[i-1];
   }
}
foo( struct node **p, int **q ){
   *p = (struct node *)
      malloc ( sizeof( struct node ) );
   (*p)->data = 10.0 ;
   (**q) ++ ;
}
```

(a)



(b)

Figure 2. An example program and its memory blocks.

3.2. Data type of memory blocks

A data type describes a set of properties for the same kind of data and associates a set of functions to manipulate data objects. To support data collection and restoration, additional information and operations have to be provided to recognize and manipulate the data value of memory blocks.

3.2.1. Type information table

At compile time, we assign a unique number, namely the *Type Identification* (Tid) number, to every data type. The *Type Information* (TI) table is a data structure used to store information of every data type. The TI is generated and annotated to source code by the pre-compiler. The TI table contains platform-specific information of every data type declared in the source code. The information is generated when we compile the annotated source code to produce an executable for a particular computing platform. Most importantly, the TI table contains a number of type-specific functions to encode and decode

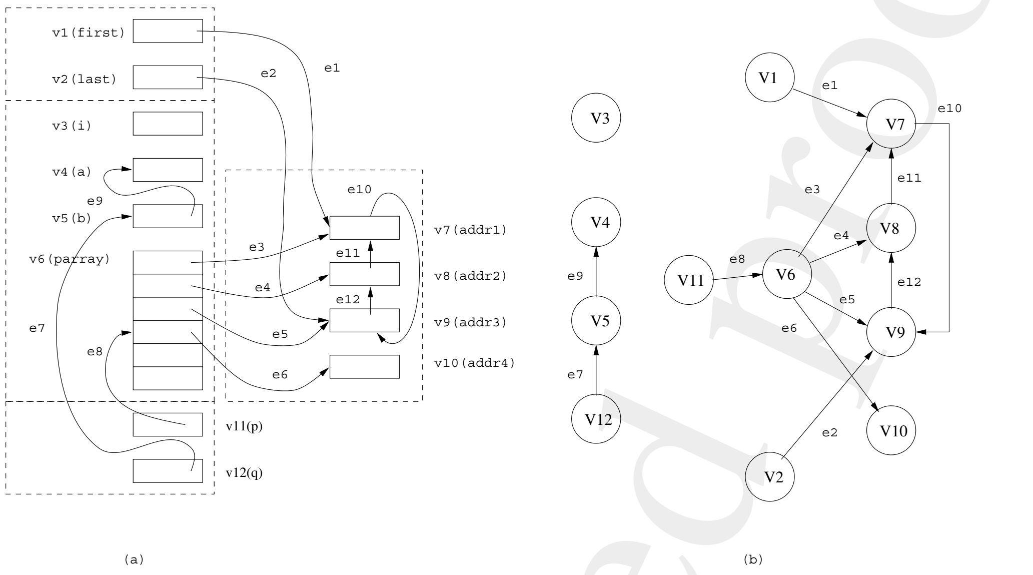


Figure 3. An example of the MSR graph.

data and to transform data between machine-specific and machine-independent formats. We define the TI table globally so that process migration operations can access it from anywhere during program execution. The TI table is indexed by the Tid number and contains the following fields.

1. *Unit size*. Platform-specific size in bytes of an object of a particular type. An object of a particular type can occupy different sizes (in bytes) on different platforms. In the case of an array, the unit size is the size of a unit member of the array.
2. *Number of elements*. Number of array elements.
3. *Object Tid number*. The Tid number of the contents of an array or the Tid of an object being referenced by a pointer.
4. *Number of components*. Number of components of a structure or record data structures.
5. *A pointer to* `component_layout` *table*. The `component_layout` table contains information about the format of a structure type. Each record in the table supplies information for a component of the structure. This table is used to translate the offset of any pointer address relative to the beginning address of the memory block to the machine-independent format and *vice versa*. Each record in the `component_layout` table contains the following information:

   (a) the beginning offset (in bytes) of the corresponding component relative to the beginning position of the structure;
   (b) the Tid number of a component unit; and
   (c) the number of elements of an array-typed component.

6. *Saving function*. A function to save the contents of a memory block into a stream of machine-independent information. It is used during the data collection process.
7. *Restoring function*: A function to extract memory block contents from the transmitted information stream and rebuild the memory block in memory space. It is a basic method for data restoration.

These fields are defined differently based on the characteristics of the data type of interest. If the data type is an array of basic indivisible data values such as integers, we define a record for this type in the fields 1, 2, 6 and 7. For instance, we define a Tid table record for the `int abc[10]` to have *unit size* = `sizeof(int)`, *number of elements* = `10`, and *saving and restoring functions* = methods to save and restore an `int` data value. On the other hand, we define fields 1, 2, 4, 5, 6 and 7 to represent an array of compositional data values such as an array of a structure. For example, taking `struct node` type in Figure 2(a), `struct node arr[10]` could be represented by having *unit size* = `sizeof(struct node)`, *number of elements* = `10`, *number of components* = 2, and the *component layout* as well as *saving and restoring* functions to their corresponding entities for `struct node` data type. In the case of an array of pointers, we define fields as 1, 2, 3, 6 and 7. For examples, we can describe verb—struct node * parray[10]— in the Tid table using *unit size* = `sizeof(struct node *)`, *number of elements* = `10`, *object Tid number* = Tid number of the `struct node` data type.

3.2.2. *Memory block saving and restoring functions*

To be able to collect and restore the data values of a memory block, the saving and restoring functions are created based on the structure of its data type. The pre-compiler generates these functions as a part of the TI table.

The saving and restoring functions are the most important functions for data collection and restoration mechanisms. During process migration, the data collection mechanism is performed to collect the memory blocks in a process. When a memory block is encountered, the saving function is invoked according to the type of data values stored in the memory block. Then, the memory block's contents are encoded into a machine-independent format and saved to an output buffer.

After the buffer is transmitted to a new machine, the data restoration mechanism is operated. In turn, the mechanism invokes the restoring function to extract the contents of memory blocks, decode them to a machine-specific format, and store the decoded information to the appropriate memory locations.

Figure 4 shows the structures of the saving and restoring functions for the `struct node` data type. We refer to the saving and restoring function with the prefixes `pack_` and `unpack_`, respectively, followed by the type name. Two different methods are applied to save and restore the contents of a data type. First, to save a non-pointer value such as the floating-point value of `node.data`, XDR techniques [12] can be readily employed. From Figure 4, the functions `pack_float` and `unpack_float` are generated to save and restore values of `node.data`, respectively. On the other hand, to save and restore pointer values, the function `Save_memory` and `Restore_memory` are used. The component `node.link`, a pointer variable as seen in Figure 4 is saved and restored in this way. Section 4 discusses both functions in details.

Structure of a data type

```
struct node {
   float   data ;
   struct node * link ;
}
```

Algorithm for the saving function

```
pack_float (...) {
  Use XDR to encode and save values
}
pack_struct_node_type (...){
        1. identify a memory block ;
        2. Call  pack_float to save
           values of  node.data
        3. Call  Save_memory()  to
           collect    node.link
}
```

Algorithm for the restoring function

```
unpack_float (...) {
  Use XDR to decode and restore values
}
unpack_struct_node_type (...){
        1. identify a memory block ;
        2. Call  unpack_float to restore
           values of  node.data
        3. Call  Restore_memory() to
           restore   node.link
}
```

Figure 4. Algorithms of saving and restoring functions for the *struct node* data type.

3.3. MSRLT data structure

The MSRLT data structure is introduced to provide the following benefits.

1. To keep the properties of memory blocks to support data conversion during process migration. These properties include the Tid, unit size, and number of elements.
2. To support memory block search during data collection. As mentioned in Section 3.2, the saving functions traverse nodes of the MSR graph and collect their contents in a depth-first manner. Every MSR node has a corresponding variable in the MSRLT data structure, which will be marked when the node is visited. Since a visited node will never be saved again, duplication in collecting and transferring of a memory block is prevented.
3. To provide machine-independent identifications of memory blocks. Memory blocks are usually identified by their address in a process. However, because different memory addressing schemes are used on different computer architectures, a technique to logically identify the memory blocks is developed for data collection and restoration between heterogeneous computers. Figure 5 shows the one-to-one mapping of memory addresses between two heterogeneous computers (A and B) using the MSRLT data structures. A memory address on computer A is converted to a logical address using MSRLT on A. Then, on computer B, we convert the logical address back to a native memory address using B's MSRLT structures.

3.3.1. Significant and trivial memory block nodes

In our design, the memory blocks recorded in the MSRLT data structure are those that need to have their status recorded for the data collection and restoration operations. In practice, keeping track of all the memory blocks is quite expensive and unnecessary. Only the memory blocks that are visible to multiple functions and those that are (or may be) pointed to by multiple pointers are recorded for process migration. Since such memory blocks could be visited multiple times during data collection,

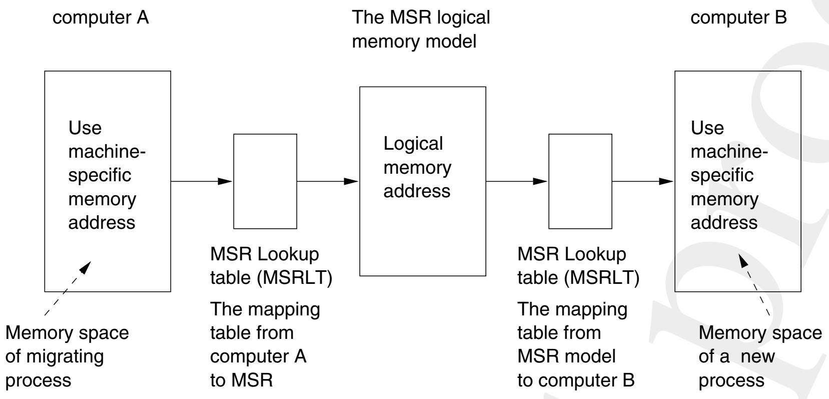

computer A
The MSR logical memory model
computer B
Use machine-specific memory address
Logical memory address
Use machine-specific memory address
MSR Lookup table (MSRLT)
MSR Lookup table (MSRLT)
Memory space of migrating process
The mapping table from computer A to MSR
The mapping table from MSR model to computer B
Memory space of a new process

Figure 5. The MSRLT data structures allow mappings of memory addresses among heterogeneous computers.

their visiting status has to be recorded in the MSRLT. In the MSR graph, these vertices are called *significant* nodes, whereas the other vertices are called *trivial* nodes. The significant nodes and their properties will be recorded in the MSRLT data structures. We classify nodes in the MSR graph into two types, because during process migration the significant nodes might be visited multiple times due to their multiple references, while the trivial nodes are collected and restored only once via their variable names or memory addresses. For the sample program given in Figure 2, let Gs, Hs, Ls_{main}, and Ls_{foo} be the sets of the significant nodes of global, heap, local data of function `main`, and local data of function `foo`, respectively. We get $Gs = \{\}$, $Hs = \{v_7, v_8, v_9, v_{10}\}$, $Ls_{\mathrm{main}} = \{v_4, v_5, v_6\}$, and $Ls_{\mathrm{foo}} = \{\}$.

At compile-time, a variable is registered to the MSRLT data structure when the following occur.

1. The variable is a global variable referred by multiple functions or in a function with possibility of direct or indirect recursion. Since the variable's memory block can be accessed from multiple functions in the activation record, they could be significant memory blocks.
2. The memory block address of a variable is or may be assigned to a pointer or used in the right-hand side of any pointer arithmetic expression. In C, addresses and information of variables that apply '`&`' are registered to the MSRLT data structure. For array variables, their names represent the starting addresses of memory blocks. Therefore, if an array name is used in the right-hand side of a pointer arithmetic statement, its memory block could have multiple references and thus could be a significant memory block.

Special macros are inserted at the beginning of the body of the main function to register information of significant global variables to the MSRLT data structure, while those of significant local variables are inserted at the beginning of the function in which they are declared. In case the local variables belong to the main function, the MSRLT registration macros are inserted right after those of the global variables.

In the case where a memory block is dynamically allocated in a heap segment, we know that its starting address is assigned to a pointer variable. Therefore, we record the address and other information to the MSRLT data structure. In our design, we create a function that wraps up the original memory allocation statement. The registration of an allocated memory space is performed right after the memory allocation inside the wrapper function. In the C language, we replace the original `malloc` function by our `MSR_ALLOC` function. For example, the statement `... = (struct tree *) malloc( sizeof(struct tree) )` will be replaced by `... = (struct tree *) MSR_ALLOC( tid, sizeof(struct tree) )` where `tid` is the Tid number of `struct tree`. The `MSR_ALLOC` function calls `malloc` and then registers information of the allocated memory block to the MSRLT data structure. Likewise, the `MSR_FREE` is used instead of the original `free` operation. It deletes the memory block's address and information from the MSRLT data structure and then calls `free`.

3.3.2. Data structure

The MSRLT data structure is used to keep information of significant memory blocks. It works like a mapping table between the physical memory space and the MSR model. It also provides each memory block with a logical identification that is used for reference for heterogeneous process migration. Figure 6 shows the structure of the MSRLT data structure consisting of two tables: the `mem_stack` and `mem_set` tables. The `mem_stack` table is a table that keeps track of the activated functions. Each record in the table represents a particular data segment of the program. The first record denoted by `mem_stack[0]` is used for keeping information of the set of significant memory blocks of the global variables. The second record, `mem_stack[1]`, contains information of the set of significant memory blocks in the heap segment. The third one is used for the set of significant memory blocks of local variables of the main function. The rest are used for the significant memory blocks of local variables of each activated function. A record of the `mem_stack` table consists of two fields: a pointer to a `mem_set` table and the number of records in the pointed `mem_set` table.

The `mem_set` table is used to store information of every significant memory block of a data segment of the program represented by a record in the `mem_stack` table. The `mem_set` table is separated from the `mem_stack` to facilitate the implementation. Since the size of each `mem_set` table could be arbitrary, the separation makes it easier to allocate the table's memory space. Each record in the `mem_set` table consists of the following fields:

1. `Tid`, type identification number of the object contained in the memory block;
2. `unit_size`, size in bytes of each object in the memory block;
3. `element_num`, the number of objects stored in the memory block;
4. `mem_block_pointer`, a pointer to the starting address of the memory block; and
5. `marking_flag`, a marking flag used to check if the memory block has been visited during the memory collection operation.

When the program starts its execution, the first three records of the `mem_stack` table will be created. Then, whenever a function call is made, a `mem_stack` record will be added to the `mem_stack` table in a stack-like manner. If there are significant local variables in the function, they will be added to the `mem_set` table of the last `mem_stack` record. Note that significant variables are registered to the MSRLT data structures using special macros inserted into the program source code. After the function finishes its execution, the `mem_stack` record as well as its `mem_set` table will be destroyed. In the case of memory blocks in a heap segment, the information of the memory

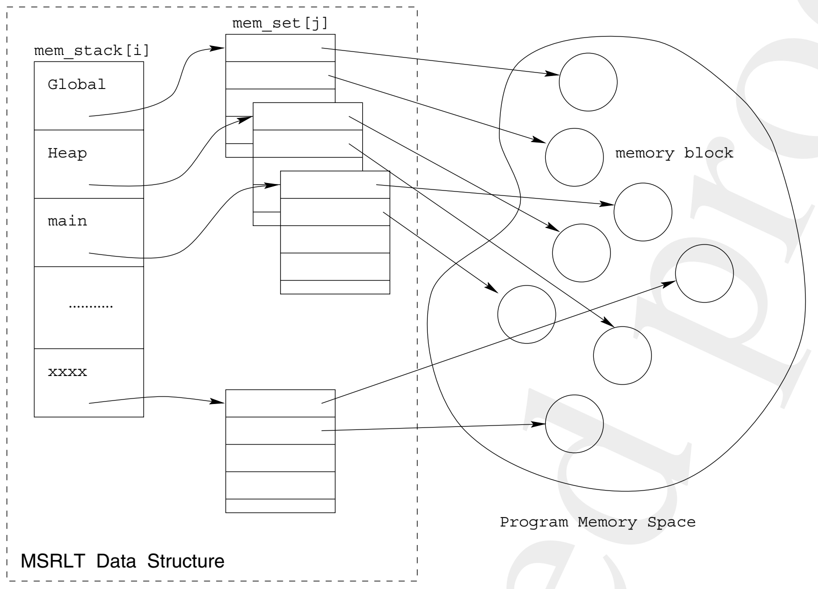


Figure 6. The MSRLT data structures.

block allocated by the function `malloc` will be added to the `mem_set` table of the `mem_stack[1]` record. They will be deleted from the `mem_set` table when the `free` operation is called.

Every significant memory block can be identified by a pair of indices of its `mem_stack` and `mem_set` records. This identification scheme will be used as a logical identification of the significant memory blocks across different machines. Let v, $stack\_index(v)$, and $set\_index(v)$ be a significant MSR node, the index of its `mem_stack` record, and the index of its `mem_set` record, respectively. The logical representation of v is given by $(stack\_index(v), set\_index(v))$.

3.4. Representation of pointer

As stated at the beginning of this section, each edge in the MSR graph represents a pair of memory addresses: the memory address of a pointer and the memory address of the object to which the pointer refers. The format is shown in Figure 7. Note that examples in this section are not related to those given previously. There are three edges between nodes v_1 and v_2 in Figure 7. For example, edge e_1 is represented in the form of $(addr_1, addr_4)$ where $addr_1$ and $addr_4$ are addresses that satisfy the predicate *Address_of* for node v_1 and v_2, respectively. The symbol $addr_1$ is a pointer object that contains the address $addr_4$ in its memory space. Therefore, given $addr_1$ we always get $addr_4$ as its content. By taking a closer look at e_1, we can also write it in the form of $(addr_1, head(v_2)+(addr_4-head(v_2)))$. The address $head(v_2)$ is called the *pointer head*, and the number $(addr_4 - head(v_2))$ is called the *absolute pointer offset*.

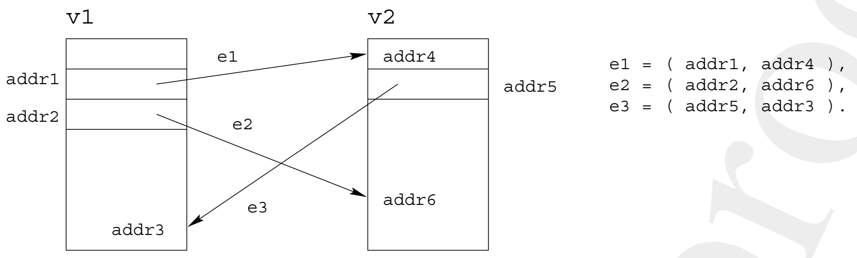
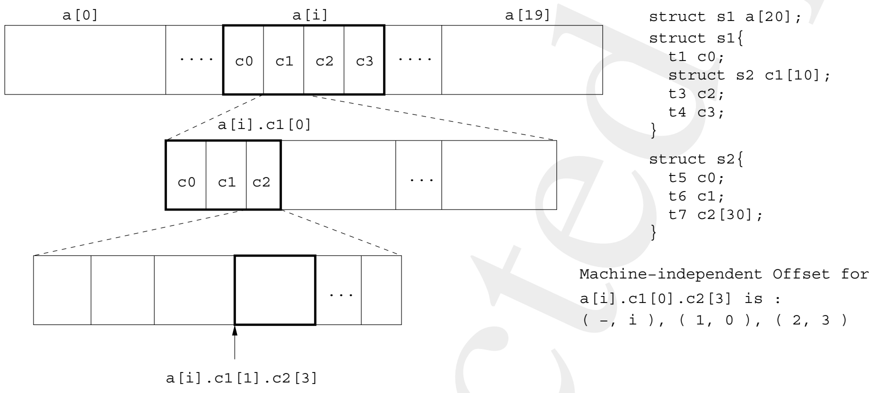


Figure 7. A representation of a pointer between two nodes of the MSR graph.



```
struct s1 a[20];
struct s1{
  t1 c0;
  struct s2 c1[10];
  t3 c2;
  t4 c3;
}
struct s2{
  t5 c0;
  t6 c1;
  t7 c2[30];
}
```

Figure 8. An example of the machine-independent format of the pointer offset.

The representation of a pointer in machine-independent format consists of the machine-independent representations of the pointer head and the pointer offset. According to the definition of the significant memory block, the node that is pointed to is always a significant node. Thus, its properties are stored in the MSRLT data structure. From the example in Figure 7, the logical identification of v_2 can be represented by ($stack\_index(v_2)$, $set\_index(v_2)$). We use this logical identification to represent the pointer head in the machine-independent information stream for process migration.

To represent the offset of the pointer in machine-independent format, we have to transform the absolute pointer offset into a sequence of (component position, array element position) pairs. The component position is the order of the components in the memory space of a structure to which the pointer refers, whereas the array element position is the index to the array element that has the pointer pointing to its memory space. The sequence of machine-independent pointer offsets can be generated from a given absolute pointer offset using the information provided in the `component_layout` tables mentioned in Section 3.2. The absolute pointer offset can also be derived from the machine-independent offset using the same information.

For example, the absolute pointer offset (`&a[i].c1[0].c2[3]` − `a`) of the memory block of the variable `a` in Figure 8 can be translated into a sequence $\langle (0, i), (1, 0), (2, 3)\rangle$. Note that we start indexing from zero. From the first pair, the component position is zero because the memory block has only itself as a component. The array position tells us that the pointer points to the $(i + 1)$th element of the array `a`. The component position part of the second pair $(1, 0)$ means that the pointer points to the second component of the structure stored in `a[i]`, which is `a[i].c1`. The array element position part of $(1, 0)$ tells us that `a[i].c1` is the array and that the pointer points to the first element of that array, `a[i].c1[0]`. Finally, the component position of the pair $(2, 3)$ means that `a[i].c1[0]` is the structure and the pointer falls on the third component of `a[i].c1[0]`, which is `a[i].c1[0].c2`. The array element position of the pair $(2, 3)$ indicates that the component `a[i].c1[0].c2` is the array and that the pointer points to the fourth element of `a[i].c1[0].c2`, which is `a[i].c1[0].c2[3]`.

4. DATA COLLECTION AND RESTORATION MECHANISMS

Figure 9 shows software components that support data collection and restoration mechanisms. The components are the MSRLT data structure, the TI table, and the data collection and restoration library.

In our design, the data collection and restoration mechanisms are implemented in the data collection and restoration library. The mechanisms are performed upon the invocation of the `Save_memory` and `Restore_memory` functions. Both functions are in the following formats:

```
Save_memory( mem_block_address, Tid )
```

and

```
Restore_memory( mem_block_address, Tid )
```

where the `mem_block_address` is a memory address of a memory block and `Tid` is a type identification number of the data value of the memory block. We apply `Save_memory` and `Restore_memory` in two different situations.

- They are used to collect and restore live variables in source codes as mentioned in the data transfer mechanism in Section 2.
- They are employed to collect and restore pointer contents in the saving and restoring functions as previously discussed in Section 3.2.2.

When process migration occurs, `Save_memory` is applied to collect the contents of a live variable at a migration point. The function searches for a record of a memory block in the MSRLT data structure based on the memory address given as an input parameter. If the memory block is found and it has never been visited before, `Save_memory` will save the given memory address (`mem_block_address`) in the machine-independent pointer format (as discussed in Section 3.4) and proceed to save the data contents of the memory block by invoking the type-specific saving function. Recall that the saving function already handles XDR encoding and makes recursive calls to `Save_memory` if there are any pointers contained in the memory block. On the other hand, if the memory block has already been

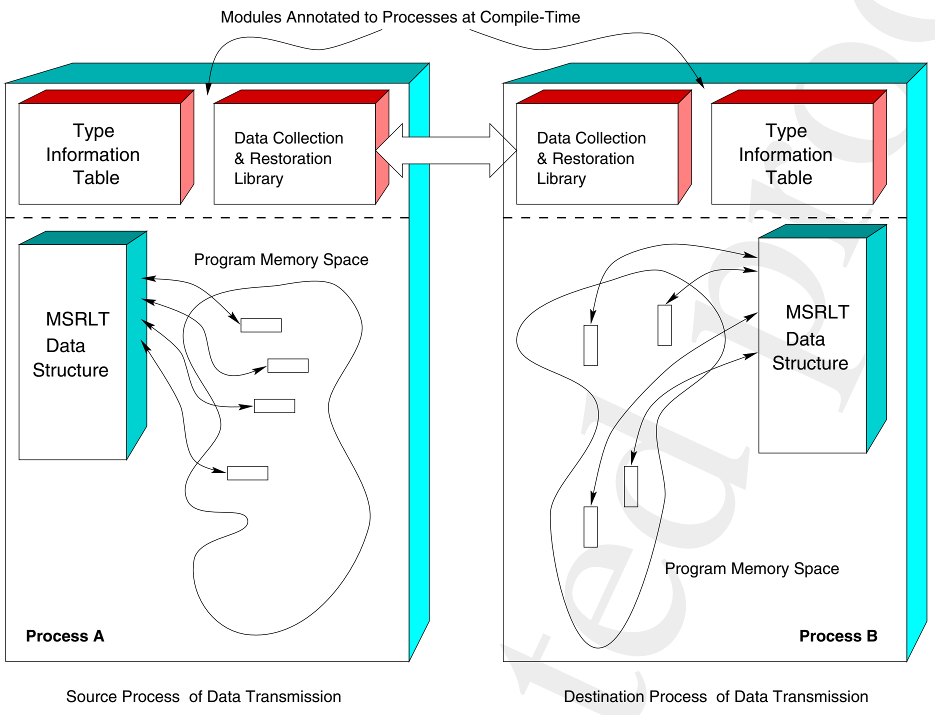

Modules Annotated to Processes at Compile-Time
Type Information Table
Data Collection & Restoration Library
Data Collection & Restoration Library
Type Information Table
MSRLT Data Structure
Program Memory Space
Process A
MSRLT Data Structure
Program Memory Space
Process B
Source Process of Data Transmission
Destination Process of Data Transmission

Figure 9. Software components for data collection and restoration mechanisms.

visited, `Save_memory` will only save the machine-independent pointer to the output buffer and then return. As a result, `Save_memory` collects information of MSR nodes and edges of any directed cyclic or acyclic MSR graph in a depth-first-search manner.

After the buffer is transmitted to a new machine, the restoring function extracts the information of the memory blocks, decodes them to a machine-specific format, and stores them in appropriate memory locations. We should recall that (according to the data transfer mechanism) the MSRLT data structure has already been rebuilt in the memory space of the new process before the restoration of memory block contents. Therefore, the mapping mechanism between machine-independent and machine-specific memory block identifications is provided. To extract memory blocks' information from the buffer, the function `Restore_memory` is applied. It first extracts the type and logical location information from the buffer. Then, `Restore_memory` restores information of the memory block to the MSRLT data structures and then uses the MSRLT to map the logical location to an appropriate physical memory address. After that, it invokes a type-specific restoring function to convert the data contents of the memory block to a machine-specific format and restore the data contents there. Note that if the contents contain pointers, `Restore_memory` would be recursively invoked to restore them. After the contents of memory blocks are restored, `Restore_memory` restores pointer data values by converting the machine-independent pointer information from the buffer to appropriate memory addresses.

4.1. An illustrative example

This section gives an illustrative example of the MSR model and explains how the data collection and restoration are performed on the MSR nodes and links. The emphasis is on mechanisms that work with the MSR at a high level.

Based on the example given in Figure 2, we describe the data collection and restoration as follows. Suppose that the migration point is set right before the execution of the instruction at line 20 when the *for* loop at line 12 had been executed four times previously. According to the data transfer mechanism mentioned in Section 2, the live data of the function `foo` have to be saved, followed by that of the function `main`. For brevity, we only discuss the collection and restoration of the variables `p` (or v_{11}) in `foo` and `first` (or v_1) in `main`. In data collection, the statement `Save_memory( p, .. )` would be executed at the migration point in `foo`, and the statement `Save_memory( &first, .. )` would be called at a location in `main` where `foo` returns. Likewise, the statements `Restore_memory( &p, .. )` and `Restore_memory( &first, .. )` are operated in the same locations in `foo` and `main`, respectively, for data restoration.

Due to the depth-first traversal, the collection of v_{11} would result in the values of v_{11}, e_8, v_6, e_6 and v_{10} being saved first. Then, the algorithm would backtrack to collect e_5, v_9, e_{12}, v_8, e_{11}, v_7 and e_{10} before backtracking again to save e_4 and e_3. After the collection process finishes in `foo`, the data collection operation in `main` will start. Taking v_1 as an example, only the values of v_1 and e_1 are collected for the `first` variable. This is because the node v_7 and its subsequent links and nodes have already been visited.

In the data restoration process, the variables in function `foo` and `main` are restored in the same sequence in which they were collected. The restoration functions will be invoked recursively on the destination process. The functions use the MSRLT data structure to translate the graphical notations (nodes and links), as well as their values, back to the machine-specific format.

5. IMPLEMENTATION AND EXPERIMENTAL RESULTS

Software for heterogeneous data transfer can be classified into four layers, as illustrated in Figure 10. The first layer relies on basic data communication utilities. Migration information can be sent to the destination machine using TCP protocol, shared file systems, or remote file transfer. In the second layer, XDR routines [12] are used to translate primitive data values such as 'char', 'int', and 'float' of a specific architecture into a machine-independent format. In the third layer, the *MSR Manipulation* (MSRM) library routines are employed to translate complex data structures, such as user-defined types and pointers, into a stream of machine-independent migration information. The MSRM library provides routines such as `Save_pointer` and `Restore_pointer` and those for manipulating the MSRLT data structures. These routines are called by macros annotated to source programs to support a migration event. Finally, the annotated source code is linked to the TI table, as well as the saving and restoring functions, to generate a migratable process in the application layer. The TI table, as well as the saving and restoring functions, is also used by the MSRM library routines.

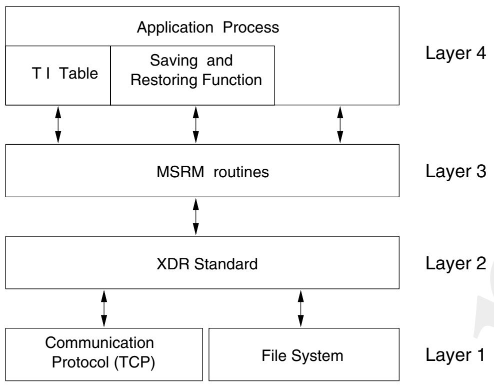


Figure 10. The layered structure of software for data collection and restoration.

5.1. Heterogeneity

We have conducted experimental testing on various programs to verify our heterogeneous process migration model. The experimental results of three programs, namely *test_pointer*, *linpack benchmark*, and *bitonic sort* program, which represent different classes of applications, have been selected to be presented here.

The test_pointer is a synthesis program which contains various data structures including a pointer to integer, a pointer to an array of 10 integers, a pointer to array of 10 pointers to integers, and a tree-like data structure. The MSR graph of the tree-like data structure is shown in Figure 11(a). The root pointer of the tree is a global variable. All tree nodes are generated with dynamic memory allocations. The program works in two steps. It first generates and initializes every data structure and then traverses the tree-like structure. In our experiment, we perform process migration in the *main()* function after all data structures have been generated and initialized. After the migration, the tree-like data structure is traversed on the migrated machine.

The linpack benchmark from the netlib repository at ORNL [13] is a computationally intensive program with arrays of double and arrays of integer data structures. Their MSR graphs are as simple as unconnected memory block nodes. Pointers are only used to pass parameters between functions. The benchmark solves a system of linear equations, $Ax = b$. Most variables in this program are declared in the main function. We have performed two experiments on this program with two different problem sizes. First, the program solves a matrix problem of order 200. The size of the matrix A in this case is 200×200. In the second test, the order of matrix is increased to 1000. At runtime, we force the program to migrate when the function DAXPY is executing with a function call sequence $main() \rightarrow DGEFA() \rightarrow DAXPY()$, which means that the function *main()* calls the function *DGEFA()* which in turn calls the function *DAXPY()*.

Finally, the bitonic sort program [14,15] was tested. In this program, a binary tree, illustrated in Figure 11(a), is used to store randomly generated integer numbers. The program will manipulate the

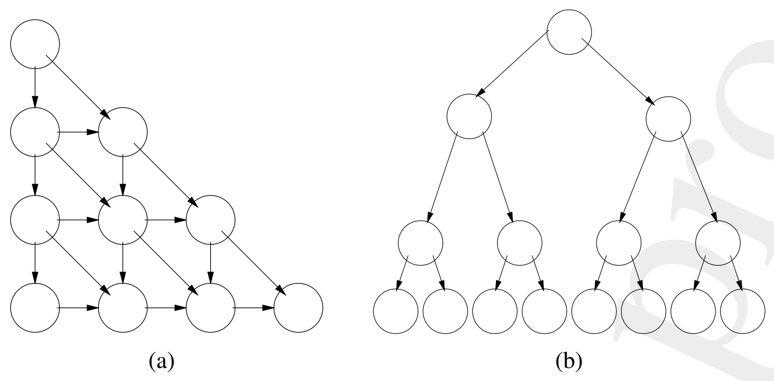


Figure 11. MSR graph for the test_pointer (a) and the bitonic sort (b) programs.

tree so that the numbers are sorted when the tree is traversed. The root of the tree is defined in the main function. Dynamic memory allocation operations and recursions are used extensively in this program. Two different problem sizes are again tested for the experiments. One is a tree with 1024 nodes, the other is a tree with 4096 nodes. Process migration is conducted in the function *bimerge()* with a sequence of function recursive calls, *main*() → *bisort*() → *bisort*() → *bisort*() → *bisort*() → *bimerge*() → *bimerge*() → *bimerge*() → *bimerge*().

In each experiment, we originally run the test program on a DEC 5000/120 workstation running Ultrix and then migrate the processes to a SUN Sparc 20 workstation running Solaris 2.5, so the migration is truly heterogeneous. The DEC 5000/120 is a 32-bit machine, while the SUN Sparc 20 is a 64-bit machine. The two systems use different endianness. They are connected via a 10 Mbit/s Ethernet network. Each machine has its own file system. All the test programs are compiled with optimization using *gcc* on the Sparc 20 workstation and using *cc* on the DEC workstation. As listed in Table I, we test all the programs on two different data sizes, which create different sizes of data transmission (Tx Size) (in bytes) during process migration. The total cost of process migration (Migrate) can be split into three parts: the cost of collecting data structure of a migrating process (Collect), the cost of transmitting those data (Tx), and the cost of restoring them on a destination machine (Restore). Note that the data collection and restoration in all test cases appear to be significantly different due to the unparallel performance between the source and destination machine.

Output results indicate that all applications run correctly under different testing circumstances. We inspected all the data structures and their contents and found them to be consistent before and after process migration. In the test_pointer program, despite multiple references to MSR's significant nodes, all memory blocks and pointers are collected and restored without duplication. Other data structures such as the pointer to an array of 10 pointers to integers also contain correct values under C de-referencing semantics. For the linpack benchmark, large floating-point data are correctly transferred. The data collection and restoration process preserves the high-order floating-point accuracy. The linpack benchmark also indicates the correctness of migration in the presence of multiple function calls. Finally, the bitonic sort program tests the capabilities of our process migration

Table I. Timing results (in seconds) of migration.

Program	Test_pointer		Linpack		Bitonic	
Tx Size	1 165 680	3 242 480	325 232	8 021 232	46 704	182 248
Collect	2.678	14.296	0.303	5.591	0.150	0.419
Tx	1.200	4.296	0.357	9.815	0.053	0.191
Restore	2.271	4.563	0.095	2.962	0.077	0.278
Migrate	6.150	23.181	0.756	18.368	0.280	0.889

Table II. Timing results (in seconds) of migration.

Programs	Linpack 1000 × 1000	Bitonic 8192
Collect	0.657	0.275
Tx	0.790	0.037
Restore	0.649	0.214
Migrate	2.096	0.526

technique under dynamic memory allocation and execution behaviors. During the bitonic execution, the size of heap and stack data segments change dynamically. The migration results indicate correct data transfer in all program segments.

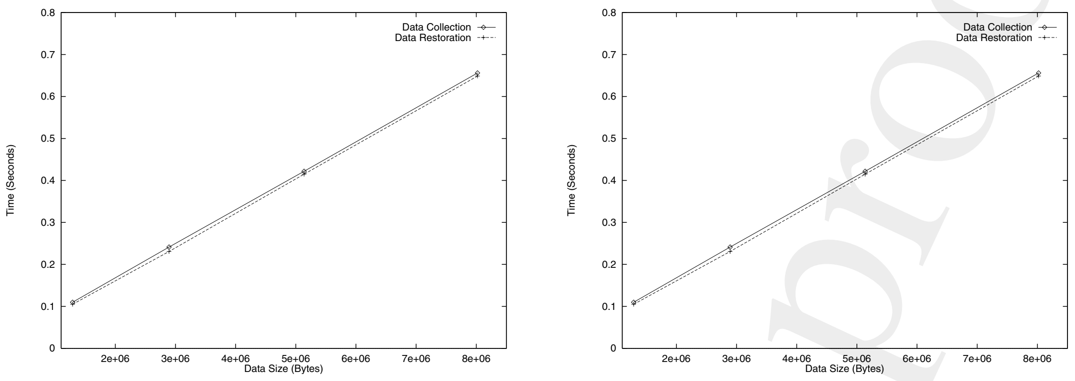
5.2. Data collection and restoration complexity

Table II shows the performance of process migration in a homogeneous environment where the linpack benchmark and the bitonic sort program are migrated from an Ultra 5 machine to another via a 100Mb/s Ethernet. The table shows that different applications can demonstrate different portions of time spent for data collection, transfer, and restoration. We define process migration time as

$$\texttt{Migration} = \texttt{Data Collection} + \texttt{Transmission} + \texttt{Data Restoration}$$

By Table II the migration time of the linpack and bitonic programs are 2.096 and 0.526 s, respectively.

The complexity of data collection and restoration is application dependent. For example, the linpack program has a small number of MSR nodes, yet each node occupies a substantial amount of memory space. Therefore, most of the data collection time is spent on encoding data in memory blocks and copying data to an output buffer. Likewise, the data restoration would mostly involve decoding the transmitted data and copying the results to the memory space. On the other hand, the bitonic sort program contains a large number of small memory blocks. Thus, in data collection, we not only encode and copy data to the output buffer, but also search for live memory blocks in memory space. For data



(a) (b)

Figure 12. (a) Data collection and restoration time of the linpack program. (b) Data collection and restoration time of the bitonic sort program.

restoration, we do not have to search memory blocks, but a large number of memory allocations has to be considered.

Based on the data collection algorithm, we can define the collection complexity as

$$\texttt{Collect} = \texttt{MSRLT Search} + \texttt{Encoding and Copying}$$

where the `MSRLT Search` time is the time for searching the MSRLT data structure. Suppose that there are n fully-connected MSR nodes in the program memory space and each node has D_i bytes, where $1 \leq i \leq n$, then the `MSRLT Search` time depends on n and has the upper bound complexity of $O(n \log n)$, while the encoding and copying time depends on the size of live data to be migrated, $\sum_{i=1}^{n} D_i$. The complexity is $O(\sum_{i=1}^{n} D_i)$. For the data restoration algorithm, we define

$$\texttt{Restore} = \texttt{MSRLT update} + \texttt{Decoding and Copying}$$

Since the logical location of every migrated memory block is attached to its data, the data restoration algorithm only spends constant time to restore the items according to the MSRLT data structure. Thus, the *MSRLT update* takes $O(n)$ time complexity, and the `Decoding and Copying` takes $O(\sum_{i=1}^{n} D_i)$.

Figure 12(a) compares the data collection and restoration times of the linpack program. In this experiment, we measure the performance of matrices with sizes 400×400, 600×600, 800×800, and 1000×1000. All experiments were performed on two Ultra 5 SUN workstations connected via a 100Mb/s network. Data collection and restoration times are shown together as a function of migration data ($\sum_{i=1}^{n} D_i$). In the linpack benchmark, memory spaces for matrices are allocated as local variables at the beginning of the *main()* function and are referenced by other functions throughout program lifetime. The program is computationally intensive and contains no dynamic memory allocation. The larger the problem size, the bigger the size of memory blocks that hold the input matrices. Since the

Table III. Overhead of migratable programs on different poll-point placements.

Programs	Original	One-level	Two-level
Linpack 1000 × 1000	764.64	765.21 (0.57)	777.61 (12.97)
Bitonic 8192	0.82	0.94 (0.12)	1.02 (0.2)

number of MSR nodes does not increase when the problem size scales up, the `MSRLT search` time and `MSRLT update` time are held constant. As the results show, the data collection and restoration complexities scale linearly with the size of live data to be transmitted during a migration. The timing differences between data collection and restoration are also constant for all transmitted data sizes.

The bitonic sort program exhibits a different behavior. Figure 12(b) shows the data collection and restoration performance of the bitonic program for different data sizes. Let n be the number of the MSR nodes in the program and $\sum_{i=1}^{n} D_i$ be the size of all the data. As the input data of the bitonic sort program scales, both n and $\sum_{i=1}^{n} D_i$ also increase. As a result, the effect of `MSRLT search` time ($O(n \log n)$) contributes to a noticeably higher collection time than that of the `MSRLT update` time ($O(n)$) for data restoration, when the number of data to be sorted scales up.

5.3. Execution overhead

Source code annotation may remove certain code optimizations and bring some overhead to the execution. The overhead is application specific and may come from many factors. Without considering the external factors such as interaction with the operating system or I/O contention, experiences show that the overhead of process migration depends mostly on two factors: the placement of poll points and the number of memory allocations. Table III reports the execution overhead when poll points are placed on different levels of a nested function call. Numbers inside the parentheses indicate the overheads, the average wallclock timing differences between the migratable processes (with different poll-point placements) and the original benchmark. In the linpack benchmark with 1000×1000 problem size, the execution overhead of 0.57 s is detected when poll points are put to the *main*() function (or `1-level` of nested function call). To make the process migratable inside the function *DGEFA*(), the *main*() and *DGEFA*() both have to be annotated. The overhead appears to increase 12.97 s in the second level nested call. When we go further and modify the *DAXPY*() function in the third level nested function call, a significant increase of 491.85 s in execution overhead occurs. The reason for this is that the *DAXPY*() is called by many functions over 13 million times during execution, while *DGEFA*() is called 26 times and *main*() is invoked only once. Moreover, since the amount of work for each call to the *DAXPY*() is small, the inserted migration macro will cause a significant proportion of the overhead. The overhead which occurred is reasonable and can be mostly avoided. In a practical situation, there is no need to insert a poll point inside of a small kernel.

In case of the bitonic sort, a similar phenomenon to that of the linpack benchmark is noticed. As we try to make the inner function of the call sequence migratable, a significant amount of execution overhead occurs. This overhead is also caused by the high frequency of invocation of the inner function and the low ratio of work over the overhead of source code annotation.

The performance of the bitonic program also suffers from dynamic memory allocation. As stated earlier, the wrapper function must be applied to the original `malloc` and `free` functions, high execution overhead can occur when a large number of small memory blocks are allocated during program execution. To solve this problem, good poll-point placement mechanisms need to be developed. Some work has been done in developing heuristic algorithms [10,16]. Smart memory allocation policies may also be employed to avoid the memory overheads. More investigation into this matter is needed.

Since the data structure of MSRLT depends on where poll points are placed, the execution overhead of maintaining these data structures is a function of the location of poll points and the number of runtime memory allocations.

6. RELATED WORKS

Early works of Theimer and Hayes [7] propose the use of a debugger to access runtime data, integrate them to program source code, and recompile the program. This design pioneers the source code manipulation to achieve machine-independent state transfer. However, the data integration and recompilation could be time consuming and create unacceptable migration overhead, especially for large-scale applications.

An alternative approach for heterogeneous process migration is based on the interpretive language models. Attadi *et al.* [17] emulates an abstract common machine on different computing platforms and uses abstract interpretive instructions to execute applications. More recently, mobile agent systems such as the Java-based IBM Aglet [18] and Telescript [19] also use the interpretive models to handle heterogeneous process migration. However, this approach has a drawback in slow execution due to interpretation overheads.

The work of von Bank *et al.* [20] has defined a theoretical framework for language systems to support heterogeneous migration. Their work indicates that a program can transfer its states among different machines only at locations where its machine codes have an equivalent process state. We define such equivalent locations using poll points.

The approach presented in Dubach *et al.* [21] and Shub [6] uses a modified compiler to generate code along with migration information for heterogeneous machines. Their system is implemented on top of the V operating system [22] and extends V's migration mechanisms to handle heterogeneity. Their design requires that each primitive data item occupies the same memory address space in all architectures. In the case of a structure or record, an item must be found at the same offset. These requirements are not practical for heterogeneous process migration in general. Their design also relies on the implementation of the V system.

Recent work on the TUI system [8] by Smith and Hutchinson, investigated features of high-level languages that are compatible with process migration and developed a prototype to migrate C applications. Their definition of 'migratable' features in C helps identify a class of C applications that can be migrated in a heterogeneous environment. In their prototype implementation, process migration is controlled by the external agents, `migrout` and `migin`, for data collection and restoration, respectively. The compiler must be modified to provide debugging information which contains locations of pre-emption points and call points inside the intermediate code to capture and restore process state. Their work has a number of design aspects that comply with the foundation given in [20].

Our data collection and restoration techniques are different. Instead of using external agents to access process memory space, we annotate the source code to systematically keep track of program data structure in form of a MSR graph. The annotation also results in process migration to be incorporated as a part of user programs. There is a tradeoff between TUI's design and ours. Although TUI's design exhibits no execution overheads, the need to modify the front-end and back-end of the compiler limits portability to various computer platforms. In TUI's design, the compiler, assembler, and linker have to be modified for every computing platform in the environment. Such modification may not be possible for commercialized compilers and is non-trivial for sophisticated opened-source compilers such as `gcc` [8]. We believe our design is more portable and less dependent on external processes. We also argue that with appropriate poll-point selection and memory management, our approach can practically achieve low execution overheads.

The other research using program annotation techniques to support process migration is the Process Introspection (PI) approach proposed by Ferrari *et al.* [9]. The PI performs source code annotation but collects and restores all global, heap, and stack data during process migration, while only live data are transferred by our technique. PI is a general approach for checkpointing, which is similar to the migration-point approach proposed in [10,11].

7. CONCLUSION AND FUTURE WORK

In this study, a fundamental technique for heterogeneous process migration, low-level mechanisms for data collection and restoration, is presented. A graph model, namely the MSR graph, is proposed to identify required data in memory space. Memory blocks, as well as relationships among memory addresses indicated by pointers, are represented in the form of nodes and edges, respectively, in the MSR graph. The TI table is constructed to store properties of every data type to be used during program execution. The development of functions to save and restore contents of memory blocks is also a part of the TI table construction. Then, the MSRLT data structure is employed to keep track of memory blocks in the program memory space. The MSRLT data structures also provide a logical scheme to identify memory blocks during process migration. In addition, the representation of pointers in a machine-independent format is described. Finally, the proposed design is implemented in the form of C library routines. Three C programs with different data structures and execution behaviors, including numerically intensive and pointer intensive, with recursions have been transformed to the migratable format and tested in our experiments. Analytical and experimental results show that the proposed data collection and restoration method is correct, efficient, and general. While our software is developed for C programs, the poll-point concept, memory representation model, and data collection and transfer techniques introduced in this study are independent of C and can be extended to other languages as well.

Work remains to be done to develop a distributed system which can support network process migration dynamically, transparently, and efficiently. This includes the development of a scheduler which can make optimal decisions on when and where to migrate, the requirement of a pre-compiler which can make migration transparent to the user, and the establishment of a well-defined virtual machine environment which can be seamlessly integrated with the scheduler, the pre-compiler, and the proposed basic migration mechanism.

ACKNOWLEDGEMENTS

This work was supported in part by National Science Foundation under NSF grants ANI-0123930 and ACI-0130458, and by the IIT ERIF program.

REFERENCES

1. Leutenegger S, Sun X-H. Limitations of cycle stealing of parallel processing on a network of homogeneous workstations. *Journal of Parallel and Distributed Computing* 1997; **43**(3):169–178.
2. Harchol-Balter M, Downey A. Exploiting process lifetime distribution for dynamic load balancing. *ACM Transactions on Computer Systems* 1997; **15**.
3. Krueger P, Livny M. A comparison of preemptive and non-preemptive load balancing. *Proceedings of the 8th International Conference on Distributed Computing Systems*, 1988; 336–343.
4. Foster I, Kesselman C. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann: San Mateo, CA, 1999.
5. Grimshaw AS, Wulf WA, and the Legion team. The Legion vision of a worldwide virtual computer. *Communications of the ACM* 1997; **40**(1):39–45.
6. Shub CM. Native code process-originated migration in a heterogeneous environment. *ACM Conference on Computer Science*. ACM: New York, 1990; 266–270.
7. Theimer MH, Hayes B. Heterogeneous process migration by recompilation. *Proceedings 11th IEEE International Conference on Distributed Computing Systems*, June 1991; 18–25.
8. Smith P, Hutchinson N. Heterogeneous process migration: The TUI system. *Technical Report 96-04*, University of British Columbia, Department of Computer Science, February 1996.
9. Ferrari AJ, Chapin SJ, Grimshaw AS. Process introspection: A checkpoint mechanism for high performance heterogeneous distributed systems. *Technical Report CS-96-15*, University of Virginia, Department of Computer Science, October 1996.
10. Chanchio K, Sun X-H. MpPVM: A software system for non-dedicated heterogeneous computing. *Proceedings of the 1996 International Conference on Parallel Processing*, 1996.
11. Chanchio K, Sun X-H. Efficient process migration for parallel processing on non-dedicated network of workstations. *Technical Report 96-74*, NASA Langley Research Center, ICASE, 1996.
12. Corbin J. *The Art of Distributed Applications*. Springer: Berlin, 1990.
13. Available at: http://www.netlib.org.
14. Rogers A, Carlisle MC, Reppy J, Hendren L. Supporting dynamic data structures on distributed memory machines. *ACM Transactions on Programming Languages and Systems* 1995; **17**:233–263.
15. Bilardi G, Nicolau A. Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines. *SIAM Journal of Computing* 1989; **18**:216–228.
16. Sun X-H, Niak VK, Chanchio K. A coordinated approach for process migration in heterogeneous environments. *1999 SIAM Parallel Processing Conference*, 1999.
17. Attardi G *et al*. Technique for dynamic software migration. *Proceedings of the 5th Annual Esprit Conference*, 1988; 475–491.
18. Lange D, Oshima M. *Programming Mobile Agents in Java—with the Java Aglet API*. Addison-Wesley Longman: New York, 1998.
19. White J. Telescript technology: An introduction to the language. *White Paper*, General Magic, Inc., Sunnyvale, CA.
20. von Bank D, Shub CM, Sebesta RW. A unified model of pointwise equivalence of procedural computations. *ACM Transactions on Programming Languages and Systems* 1994; **16**.
21. Dubach F, Rutherford RM, Shub CM. Process-originated migration in a heterogeneous environment. *ACM Conference on Computer Science*. ACM: New York, 1989.
22. Cheriton DR. The V distributed system. *Communications of the ACM* 1988; **31**(3):314–333.

Annotations from spe464.pdf

Page 1

Annotation 1;
Au:
Please supply keywords.

Page 21

Annotation 1;
Au:
Alteration to Figure 11(b) OK?

Annotation 2;
Au: line 10
Please clarify `endianness'.

Page 27

Annotation 1;
Au:
Please supply publisher details for [7,10,16,17].
Please supply town of publication for [4,12,18].
Any further details for reference [13]?