Data Collection and Restoration for Heterogeneous Process Migration *

Kasidit Chanchio Xian-He Sun
Department of Computer Science
Illinois Institute of Technology
Chicago, IL 60616
{kasidit,sun}@cs.iit.edu

Abstract

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 study a logical data model which recognizes complex data structures in process address space. Then, novel methods are developed to incorporate the model into a process and to collect and restore data efficiently. We have implemented a 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) the costs of data collection and restoration depend on the complexity of the logical model representing the process's data structures and the amount of data involved, and (4) the implantation of the data collection and restoration mechanisms into the process is not a decisive factor of incurring execution overheads; with appropriate program analysis, we can achieve practically low overhead.

1 Introduction

With recent popularity of network and internet computing, process migration has become a subject of great interest. It provides the mobility of computing and is useful for many situations such as load balancing, data access locality, reconfigurable computing, and system administration [4, 5, 6]. The employment of process migration can lead to the improvement in application performance and environmental-wide efficiency. Efficient process migration is recognized as a critical issue for next generation network environments [3]. Heterogeneous process migration is the migration of a process between computers that have different computation platforms. It adds more flexibility to process mobility and increases the applicability of process migration in a distributed environment. However, due to its complexity, currently no satisfactory solutions exist for efficient heterogeneous process migration for traditional stack-based languages such as C or FORTRAN. Although few research projects for heterogeneous process migration are presented [4, 7, 5, 8, 6], none is widely accepted in engineering practice due to their inherent limitations and immaturity. Heterogeneous process migration is still in its infancy. 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 computational intensive applications and its simplified data structures. Due to space limitation, more discussions on related works can be found in [1].

To be able to migrate processes written in C or FORTRAN in heterogeneous environment, a number of fundamental issues have to be addressed. First, we have to identify the subset of language features which do not prevent process migration. Smith and Hutchinson [5] 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. Then, we have to define mechanisms to capture, transfer, and restore process state. In our previous works [6], we have proposed a novel methodology in which procedures and data structures are given for transforming a high-level program into a format capable of heterogeneous process migration. Another important mechanism in heterogeneous process migration is the mechanism to transfer memory state of a process. The memory state basically contains data structures in process memory space. Difficulties in transferring the memory state arise due to the

* This work was supported in part by National Science Foundation under NSF grant ASC-9720215 and CCR-9972251, and by IIT under the ERIF award.

presence of pointer-based data structures and dynamic memory allocation. In our design, basic steps in transferring memory state of a process includes collecting program data structures, encoding them to a machine-independent format, and restoring them on a destination machine.

Data collection and restoration in a heterogeneous environment is a challenging issue. We introduced the basic idea of the Memory Space Representation (MSR) model, a logical model representing the process memory space in 1998 [2]. In this study, we present the extended model of the memory representation and provide a full exploration of implementation design and performance analysis. We have designed and implemented the data collection and restoration mechanisms for heterogeneous process migration, and developed a prototype run-time library to support process migration of migration-safe C code in a heterogeneous environment. Experimental measurements of C programs with different dynamic data structures and execution behaviors are 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's 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 achieve practically low overhead throughout the program execution. Although runtime overheads occur due to the implantation of data collection and restoration mechanisms to the original source codes, we have made a number of observations on the sources of the overheads and how they might be avoided.

2 Process Migration Environment

In our design, a program must be transformed into a "migratable" format. As introduced in our previous work [6], 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 when 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 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. Poll-points can be inserted to various functions in the source code so that process migration can occur in a nested function call. The selection of poll-points as well as the macro insertion are performed automatically by a source-to-source transformation software (or a pre-compiler). Users can also select their preferred poll-points if they know suitable migration locations in their source codes.

At every poll-point, the pre-compiler defines *live* variables whose data values are needed for computation beyond the poll-point. To collect and restore live data, special purpose interfaces are applied to collect and restore values of live variables. We have developed the following four interface routines. The $Save_pointer$ and $Restore_pointer$ routines are developed to collect and restore contents of pointer variables, while the $Save_variable$ and $Restore_variable$ routines are employed for non-pointer cases. These routines are placed inside the inserted migration macros.

In process migration environment, we assume that the source program (in the "migratable" format) 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 execution and memory states of the migrating process. Then, the migrating process collects those information and sends them to the waiting process. After successful transmission, the migrating process terminates. At the same time, the new process restores the transmitted execution and memory states, and resumes execution from the point where process migration occurred.

3 Memory Space Representation

The data collection and restoration mechanisms are based on the MSR model. In the MSR, 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. We use it as a logical representation of program data structures. 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. More details about the MSR model are provided in [2].

3.1 Data Collection and Restoration

In our design, the software components which support data collection and restoration mechanisms are the MSRLT data structure, the Type Information (TI) table, and the data collection and restoration library.

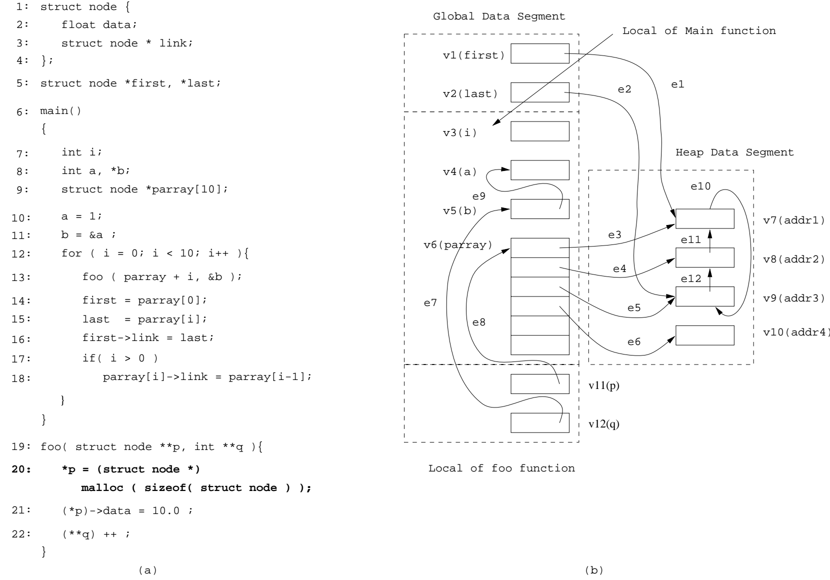

Figure 1. An example program and MSR model.

At runtime, the MSRLT data structure is created in process memory space to keep track of memory blocks. It also provides machine-independent identification to the memory blocks and supports memory block search during data collection and restoration operations. The MSRLT works as a mapping table which supports address translation between the machine-specific and machine-independent memory address.

The TI and data collection and restoration library are linked to the process when the executable is generated. The TI contains type information of every memory block in a process including type-specific functions to transform data of each type between machine-specific and machine-independent formats. We call these functions the memory block saving and restoring functions.

The saving and restoring functions have different structures for different kinds of memory blocks. For a memory block that does not contain any pointers, we can apply XDR techniques to save and restore. For a memory block contains pointers, the function $Save_pointer$ and $Restore_pointer$ are used to collect and restore the pointer values, respectively. $Save_pointer$ initiates a depth-first traversal through connected components of the MSR graph. It examines memory blocks that are referred to by pointers and then invokes type-specific saving functions to save their contents. During the traversal, visited memory blocks are marked so that they are not saved again. At the destination machine, the function $Restore_pointer$ is called recursively to rebuild memory blocks in memory space from the output of $Save_pointer$.

The $Save_pointer$, $Restore_pointer$, $Save_variable$, and $Restore_variable$ functions are implemented in our data collection and restoration library. There are two different situations for the functions, especially the $Save_pointer$ and $Restore_pointer$ to be applied. First, they are used to collect and restore live variables in source codes. Another application is that they are used in the saving and restoring functions as discussed above.

3.2 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 in high-level. More details on mechanisms to convert the MSR components as well as memory blocks' data values between machine-specific and machine-independent formats can be found in [2].

Given a sample program in Figure 1(a), Figure 1(b) shows all 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. We assume that the *for* loop at line 12 in the $main$ function had been executed *four* times before the snapshot was taken. Each memory block in Figure 1(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. Let $addr$ be an address in the program memory space, $addr_i$ where $1 \leq i \leq 4$ are addresses of a dynamically allocated memory blocks created at runtime. We also use $addr_i$ as a memory block's name in this example. As shown in Figure 1(b), the memory blocks

can reside in different segments (global, heap, and stack) in program memory space.

Figure 1(b) also shows the MSR graph. 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. To save a pointer value for heterogeneous process migration, the value have to be converted into the machine-independent format in form of pointer header and offset [2]. Supposed that a pointer value refers to a data element inside a memory block. The pointer header is the logical identification (in the MSRLT) of the memory block where the pointer value refers to. The offset is the ordering number of the data elements inside the memory block.

Based on the example, we describe the data collection and restoration as follows. Supposed 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 for four times. According to the mechanism to transfer execution state in [6], the live data of function foo have to be saved, following by that of 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 $Save_pointer(p)$ would be executed at the migration point in foo, and the statement $Save_variable(\&first)$ would be called at a location in $main$ where foo returns. Likewise, the statements $p = Restore_pointer()$ and $Restore_variable(\&first)$ are operated the same locations in foo and $main$, respectively, for data restoration.

Because 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} to be 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. Taken 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. The collected data is encoded and put into a buffer before being transmitted to a destination machine.

In the restoration process, the MSRLT data structure is rebuilt to provide the mapping mechanism between machine-independent and machine-specific memory block identifications. To extract memory blocks' information from the buffer, functions $Restore_pointer$ or $Restore_variable$ are applied. These functions extract the type information and contents of the memory block from the buffer and invoke an appropriate type-specific restoring function to convert memory block contents to a machine-specific format. The functions consult the MSRLT data structures for appropriate memory locations and restore the memory block contents there. Note that if the contents contain pointers, $Restore_pointer$ would be recursively invoked to restore them.

For the given example, the variables in function foo and $main$ are restored in the same sequence they are 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.

4 Implementation and Experimental Results

Software for heterogeneous data transfer can be classified into four layers. The first layer relies on basic data communication utilities. Migration information can be sent to the destination machine using either TCP protocol, shared file systems, or remote file transfer. In the second layer, XDR routines are used to translate primitive data values such as 'char', 'int', '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 are also used by the MSRM library routines.

4.1 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, are selected to be presented here. The test_pointer is a synthesis program which contains various data structures, including a tree structure, a pointer to integer, a pointer to array of 10 integers, a pointer to array of 10 pointers to integers, and a tree-like data structure. The linpack benchmark from netlib repository at ORNL is a computational intensive program with arrays of double and arrays of integer data structures. The benchmark solves a system of linear equations, $Ax = b$. Finally, the bitonic sort program was tested. In this program, a binary tree is used to store randomly generated integer numbers. The program manipulates the tree so that the numbers are sorted when the tree is traversed. The program demonstrates extensive memory allocations and recursions.

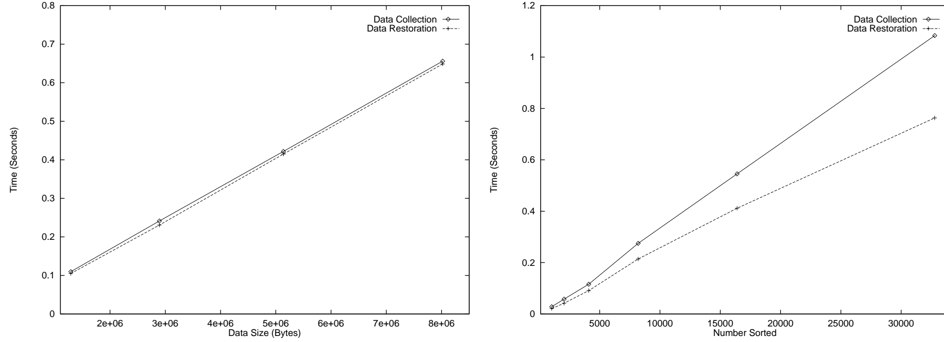

Figure 2. Data collection and restoration time of the (a) linpack and (b) bitonic sort programs.

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. It is truly heterogeneous because both systems use different endianness. Both machines are connected via a 10 Mbit/s Ethernet network. All the test programs are compiled with optimization using *gcc* on the Sparc 20 workstation and using *cc* on the DEC workstation. Output results indicate all applications run correctly under different testing circumstances. We inspected all data structures and their contents and found them to be consistent before and after process migration. In the bitonic sort and test_pointer programs, despite multiple references to MSR's significant nodes, all memory blocks and pointers are collected and restored without duplication. For the linpack benchmark, large floating-point data are correctly transferred. The data collection and restoration process preserves the high-order floating point accuracy.

4.2 Complexity

Table 1 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/Sec Ethernet. We define process migration time as the total of data collection (Collect), transmission (Tx), and restoration (Restore) time. By Table 1 the migration time of the linpack and bitonic programs are 2.096 and 0.526 second, respectively.

Programs	Collect	Tx	Restore
Linpack 1000x1000	0.657	0.790	0.649
bitonic 8192	0.275	0.037	0.214

Table 1. Timing results (in seconds)

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 substantial amount of memory space. Therefore, most of data collection time is spent on encoding data in memory blocks and copying data to a 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 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 $Collect = MSRLT\ Search + Encode\ and\ Copy$, 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 $O(nlogn)$, 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 $Restore = MSRLT\ update + Decode\ and\ Copy$. 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 $MSRLT\ update$ time takes $O(n)$ time complexity, and the $Decode\ and\ Copy$ time takes $O(\sum_{i=1}^{n} D_i)$.

Figure 2(a) compares data collection and restoration time of the linpack program. In this experiment, we measure the performance of matrices with size 400×400, 600×600, 800×800, and 1000×1000, respectively. All experiments are performed on two Ultra 5 SUN workstations connected via a 100Mb/Sec network. Data collection and restoration time 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 computation intensive and contains no dynamic memory allocation. The increase of problem size effects directly to the size of memory blocks that hold the input matrices. Since the 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 shown, 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 a constant for all transmitted data sizes.

The bitonic sort program exhibits a different behavior. Figure 2(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 ($O(nlogn)$) contributes noticeable 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.

4.3 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 migration points and the number of memory allocations. The overhead could be high if poll-points are placed in a kernel function which performs only few operations but being invoked so many times. For memory allocation, the overhead could be high if many small memory blocks are repeatedly allocated, causing large MSRLT. However, the overhead occurred is reasonable and mostly can be avoided. In a practical situation, there is no need to insert poll-points inside of a small kernel. Smart memory allocation policies may be employed to the applications to avoid the memory overheads.

5 Conclusion and Future Works

In this study, a fundamental technique for heterogeneous process migration, low-level mechanisms for data collection and restoration, is presented. A graph model, the Memory Space Representation (MSR) model, is used to identify needed data in memory space. The Type Information (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. The MSR Lookup Table (MSRLT) data structure is also proposed to keep track of memory blocks in the program memory space. The MSRLT data structures also provides a logical scheme to identify memory blocks during process migration. An illustrative example is given to explain the data collection and restoration mechanisms. Finally, the proposed design is implemented in the form of C library routines. Three C programs with different data structures and execution behaviors have been transformed to the migratable format and tested. 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 migration point concept, memory allocation analysis, and migration technique introduced in this study are independent of C and can be extended to other languages as well.

Data collection and restoration is a basic component of network process migration. 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 precompiler which can make migration transparent to the user, and the establishment of a well defined virtual machine environment which can be integrated with the scheduler, the precompiler, and the proposed basic migration mechanism seemlessly.

References

[1] K. Chanchio and X.-H. Sun. Data collection and restoration for heterogeneous network process migration,. Tech rep 97-17, Louisiana State University, 1997.

[2] K. Chanchio and X.-H. Sun. Memory space representation for heterogeneous networked process migration. In *12th International Parallel Processing Symposium*, Mar. 1998.

[3] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[4] D. S. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process Migration. Tech rep, Hewlett-Packard, Dec. 1998.

[5] P. Smith and N. Hutchinson. Heterogeneous process migration : The TUI system. Tech rep 96-04, University of British Columbia, Feb. 1996.

[6] X.-H. Sun, V. K. Niak, and K. Chanchio. A Coordinated Approach for Process Migration in Heterogeneous Environments. In *1999 SIAM Parallel Processing Conference*,1999.

[7] M. H. Theimer and B. Hayes. Heterogeneous process migration by recompilation. In *Proceeding of the 11th IEEE International Conference on Distributed Computing Systems*, pages 18–25, June 1991.

[8] D. von Bank, C. M. Shub, and R. W. Sebesta. A unified model of pointwise equivalence of procedural computations. *ACM Transactions on Programming Languages and Systems*, 16, Nov. 1994.