Memory Space Representation for Heterogeneous Network Process Migration

Kasidit Chanchio Xian-He Sun
Department of Computer Science
Louisiana State University
Baton Rouge, LA 70803-4020
sun@bit.csc.lsu.edu http://www.csc.lsu.edu/~scs/

Abstract

A major difficulty of heterogeneous process migration is how to collect advanced dynamic data-structures, transform them into machine independent form, and restor them appropriately in a different hardware and software environment. In this study we introduce a data model, the Memory Space Representation (MSR) model, to recognize complex data structures in program address spaces. Supporting mechanisms of the MSR model are also developed for collecting program data structures and restoring them in a heterogeneous environment. The MSR design has been implemented under a prototype heterogeneous process migration environment. Pointer-intensive programs with function and recursion calls are tested. Experimental results confirm that the newly proposed design is feasible and effective for heterogeneous network process migration.

1. Introduction

As network computing becomes an increasingly popular choice for computing, network process migration has received unprecedented attention recently. 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. 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. It is known that process migration is a necessity for achieving a guaranteed high performance in a non-dedicated environment [1]. More recent research shows process migration is also efficient for utilizing idle machines for parallel processing. Given a one-to-two ratio of needed and existing machines on the network, there are always enough idle machines available for parallel processing. Moreover, the mean idle-time of the machines is sufficiently long for harvesting parallel processing and compensating for the migration overhead [2]. Efficient process migration is recognized as a critical issue for next generation network environments [3], however, due to its complexity, currently there are no solutions for efficient heterogeneous process migration.

Fundamentally, there are three steps to making existing code migratable in a heterogeneous environment.

1. Identify the subset of language features which is migration-safe, *i.e.* features which theoretically can be carried across a network
2. Invent a methodology to transform migration-safe code into a “migratable” format so that it can be migrated at run-time
3. Develop mechanisms to migrate the “migratable” process reliably and efficiently

Smith and Hutchinson [4] 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. Chanchio and Sun have given the procedures and data structures for transforming a high-level program into a migratable format via a precompiler, which includes the migration-point analysis, data analysis, and the insertion of migration macros, etc. [5]. The focus of this study is on the last step, mechanisms for carrying out migration correctly and efficiently. We have developed the Memory Space Representation model and its associated mechanisms to support heterogeneous process migration. This concept can be employed with applications written in any stack-based programming languages with the presence of pointers and dynamic data structures.

2. Memory Space Representations

In our design, we model a *snapshot* of the program memory space as a graph G, defined by $G = (V, E)$ where V and E is the set of vertices and edges respectively. It is called the *Memory Space Representation (MSR)* graph. Each vertex in the graph represents a memory block; whereas, each edge represents a relationship between two memory blocks when a memory block contains pointers, defined as addresses that point to memory locations of any memory block node in the MSR graph.

2.1. Representation of Memory Blocks

A memory block is a piece of memory allocated during the execution of a program. It contains an object or an array of objects of a particular type. Each memory block is represented by a vertex v in the MSR graph. The following terminology is 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 the memory block, we mean any address within the memory block. Let $addr$ be an address in the program memory space. 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 $type(v)$.

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 instruction, we name it the *heap memory block*. In case the memory block resides in the activation area for a function f in the stack segment of the program, it is called the *local memory block of function f*.

2.1.1. Data Type of Memory Blocks

A memory block consists of one or more objects of a particular type. Each type describes a set of properties for the same kind of data and associates a set of functions to manipulate the data objects. To support the data collection and restoration during process migration, we need to provide certain information as well as additional operations to manipulate data of a specific type stored in the memory block.

At compile-time, we assign a unique number, namely the *Type Identification (Tid)* number, to every type to be used during the execution of the program. The information of every type in the program will be stored in the *Type Information (TI)* table. The saving and restoring functions will be created as parts of the TI table. They are used to transform the contents of the memory block to and from the machine-independent information stream. The TI table along with these functions are attached to the global declaration section of the program source code so that they can be globally accessed during the execution of the program. The TI table is indexed by the Tid number. Detail description of the TI table can be found in [6].

The saving and restoring functions are the type-specific to be used during the memory block collection and restoration. Once process migration is started on the source machine, the memory collecting macros will identify the memory blocks to be collected [6]. The saving function will be invoked according to the type of the memory block. It will encode the contents of the memory blocks to machine-independent format and make them a part of the machine-independent information stream for process migration. After transmitting the information to the new machine, the restoring function will extract the information for the memory block from the stream, decode it, and store the results in the appropriate place within the memory space of the new process.

In case the memory block does not contain any pointers, we can apply techniques to encode and decode contents of a memory block to and from the XDR information stream as described in [7] to construct the saving and restoring functions. On the other hand, in case the memory block contains pointers, we have to use functions `save_pointer( pointer_content, tid )` and `restore_pointer( tid )`, where `pointer_content` represents the memory address stored in a pointer and `tid` is the Tid number of the pointer, to save and restore the pointers, respectively. `save_pointer` will initiate the traversal through the connected components of the MSR graph in a depth-first search manner. It will examine the memory block that the pointer points to and then invoke an appropriate saving function stored in the TI table to save the contents of the memory block. The visited memory blocks will be marked so that they are not saved again. On the other hand, `restore_pointer` will recursively rebuild the memory blocks on the memory space of the destination machine according to the information saved by `save_pointer`.

2.1.2. Significant and Trivial Memory Block Nodes

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 any pointers during the execution

of the program are needed to be recorded. In the MSR graph these recorded vertices are called *significant* nodes; whereas, the others are called *trivial* nodes. The significant nodes and their properties will be recorded in the MSR Lookup Table (MSRLT) data structures. We classify nodes in the MSR graph into two types because during process migration the significant nodes can be collected and restored multiple times due to their multiple references; while the trivial nodes will be collected and restored only once via their variable names or memory addresses. To prevent multiple copies of significant memory blocks from being transmitted during process migration, we need to keep track of the significant nodes so that the status of the nodes can be checked. In case the memory block is a global or local variable, we can verify whether it is significant or trivial at compile time. The compiler will insert special instructions to the source code to register the significant memory blocks to the MSRLT data structures. In case of the dynamically allocated memory block, we replace the memory allocation instruction by the wrapper function which will register information of the newly allocated memory block to the MSRLT data structure at run-time. More details can be found in [6].

2.1.3. MSRLT Data Structure

The MSRLT data structure is, first, used to keep information of every significant memory block in the program memory space. Second, it is used as the search index for the saving function during data collection operation. Finally, it provides each memory block a logical identification that can be used for reference between two machines during process migration. Figure 1 shows the structure of the MSRLT Data Structure. The structure consists of two tables: the

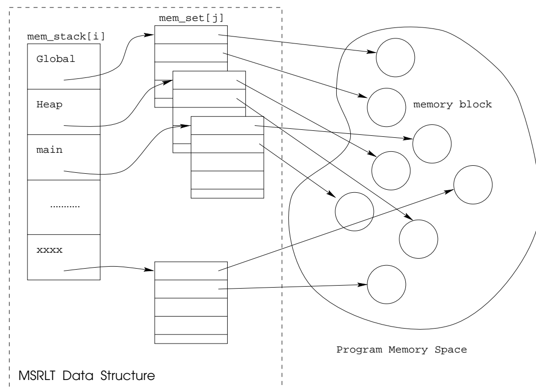

Figure 1. A Diagram shows the MSRLT Data Structures.

`mem_stack` and `mem_set` tables. The `mem_stack` table is a table that keeps track of the function calls in the program activation record. 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. Each record in the `mem_set` table consists of important information including a pointer to the starting address of the memory block and a *marking flag* used to check if the memory block has been visited during the memory collecting operation.

When the program starts its execution, the first three records of the `mem_stack` table will be created to keep track of significant memory blocks of global variables, heap memory, and local variables of the main function, respectively. Then, whenever a function call is made, a `mem_stack` record will be added to the `mem_stack` table in stack-like manner. If there are any significant local variables in the function, they will be added to the `mem_set` table of the last `mem_stack` record. After the function finishes its execution, the `mem_stack` record as well as its `mem_set` table will be destroyed. In case of memory blocks in heap segment, the information of the memory 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 index 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 $(stack_index(v), set_index(v))$.

2.2. Representation of Pointer

As stated in the beginning of this section, in our design, each edge in the MSR graph can be represented by a pair of source 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 2. There are three edges between nodes v_1 and v_2 in Figure 2. For example, edge e_1 can be 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. $addr_1$ 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

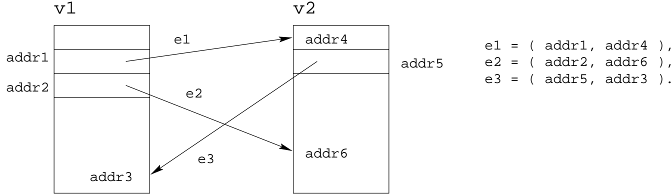

Figure 2. A representation of pointer between two nodes of the MSR graph.

number $(addr_4 - head(v_2))$ is called the *absolute 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 2, 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. More details are given in [6].

3. Implementation and Experimental Results

We have developed the *MSR Manipulation (MSRM) library* on top of the XDR routines [7] to translate complex data structures such as user-defined types and pointers into a stream of machine-independent migration information and vise versa. To verify the correctness of the proposed model and appropriate algorithms, we have conducted process migrations on three experimental programs with different kinds of data structures and execution behaviors. These programs are the *test_pointer*, the *linpack benchmark*, and the *bitonic sort* programs. They are migration-safe. The program analysis and annotated migration operations [5, 6] are applied them migratable in a heterogeneous environment. The *test_pointer* is a program which contains various data structures, including pointer to integer, pointer to array of 10 integers, pointer to array of 10 pointers to integers, and a tree-like data structure. The linpack benchmark from netlib repository at ORNL [8] is a computational intensive program with arrays of double and arrays of integer data structures. Pointers are used to pass parameters between functions. Finally, the bitonic sort program [9] was tested. In this program, a binary tree is used to store randomly generated integer numbers. Dynamic memory allocation operations and recursions are used extensively in this program.

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. Both machines are connected via a 10 Mbit/s Ethernet network. Each machine has its own file system. All the test programs were compiled with optimization using *gcc* on the Sparc 20 workstation and using *cc* on the DEC workstation.

Every experiment is performed in two modes: the direct network process migration and the process migration through file systems. In case of network migration, migration operations will scan program's data structures and keep the machine-independent migration information in a buffer. After that, the buffer will be sent over the network via the TCP transmission protocol. On the destination machine, migration operations will use the received information to restore the execution state and data structures of the program. Thus, we estimate the migration time (Migrate) by summation of memory scan time (Scan), data transmission time (Tx), and memory restoration time (Restore). On the other hand, in the situation of migration through file systems, migration operations will scan the program's memory and write the migration information to a file. After that, we will remotely copy the file to the destination machine of process migration using the `rcp` utility. Then, the new process will read migration information from the file and restore the data structure on the destination machine. The migration time (Migrate) is the summation of memory scan and write time (Scan&Write), remote copy time (Rcp), and the file read and memory restoration time (Read&Restore). The results of our experiments are shown in Table 1.

We have found that the outputs of the implementation with migration and the implementation without migration are identical, for any testing input and testing program. The design and implementation are correct. By the diversity of the test programs, the process migration approach should be feasible for any migration-safe C code. Apparently, direct network process migration is significant faster than migration through file systems as shown by the difference of their migration time (Diff) in Table 1.

We have observed that, in the pointer-intensive program, costs of saving and restoring the program's data structure can be significant. The more the number of memory blocks and pointers, the more the overheads needed for saving and restoring operations. During the saving operation the MSRM routines have to search every memory block, thus

Network						
Program	test_pointer		Linpack		bitonic	
Tx Size	1,165,680	3,242,480	325,232	8,021,232	46,704	182,248
Scan	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
File						
Scan&Write	18.533	69.032	0.997	45.243	0.803	4.896
Rcp	15.4	20.3	11.9	39.1	10.6	11.5
Read&Restore	8.693	17.602	0.124	3.654	0.303	1.234
Migrate	42.626	106.934	13.220	87.998	11.707	17.631
Comparison						
Diff	36.48	83.75	12.46	69.63	11.43	16.74
Speed Up	6.93	4.61	17.48	4.79	41.72	19.82

Table 1. Timing results of heterogeneous process migration in seconds.

the cost of memory scanning is high for pointer-intensive programs. Also, during restoration many memory allocation operations are used to recreate the data structure. The searching and dynamic allocation costs depend mostly on the number of memory blocks and pointers among them. On the other hand, the array-based program such as linpack only needs a trivial amount of time to search for memory blocks during the memory scanning operations since most data are stored in local or global variables. At the receiving end of the migration, the migration information will be copied into the reserved memory space of the new process. We do not need dynamic memory allocation for the restoration process of the linpack program.

4. Conclusion and Future Works

In this study, a memory space representation and its associated mechanisms are presented for data collection and restoration under a heterogeneous environment. A graph model, namely the Memory Space Representation (MSR) graph, is proposed to identify needed data in memory spaces. Memory blocks and relationships among memory addresses indicated by pointers are represented in the form of nodes and edges in the MSR graph, respectively. Analytical and experimental results show that the proposed data collection and restoration method is correct, efficient, and general. While the current implementation is for C code only, the proposed memory space representation and memory allocation mechanisms are independent of C and can be extended to other languages as well.

References

[1] S. Leutenegger and X.-H. Sun, "Distributed computing feasibility in a non-dedicated homogeneous distributed system," in *Proceedings of Supercomputing'93*, pp. 143–152, 1993.

[2] A. Acharya, G. Edjlali, and J. Saltz, "The utility of exploiting idel workstations for parallel computing," in *Proc. of SIGMETRICS/Performance Conf.*, pp. 225–236, May 1996.

[3] A. S. Grimshaw, W. A. Wulf, and the Legion team, "The Legion vision of a worldwide virtual computer," *Communications ACM*, vol. 40, no. 1, pp. 39–45, 1997.

[4] P. Smith and N. Hutchinson, "Heterogeneous process migration : The TUI system," Tech. Rep. 96-04, University of British Columbia, Department of Computer Science, Feb. 1996.

[5] K. Chanchio and X.-H. Sun, "MpPVM: A software system for non–dedicated heterogeneous computing," in *Proceeding of 1996 International Conference on Parallel Processing*, Aug. 1996.

[6] K. Chanchio and X.-H. Sun, "Data collection and restoration for heterogeneous network process migration," Tech. Rep. 97-017, Louisiana State University, Department of Computer Science, 1997.

[7] J. Corbin, *The Art of Distributed Applications*. Springer-Verlag, 1990.

[8] Available at http:://www.netlib.org.

[9] J. R. Anne Rogers, Martin C. Carlisle and L. Hendren, "Supporting dynamic data structures on distributed memory machines," *ACM Transactions on Programming Languages and Systems*, vol. 17, pp. 233–263, Mar. 1995.