A Coordinated Approach for Process Migration in Heterogeneous Environments

Xian-He Sun* Vijay K. Naik† Kasidit Chanchio*

Abstract

We consider the problem of process migration across heterogeneous platforms. We introduce a software environment, called *Scalable Networks Of Workstations* (SNOW) environment, that supports process migration for applications written in "C" language with MPI or PVM communication interface. The SNOW envirionment includes a pre-compiler for analyzing the source code for migration points insertion and for augmenting the code to extract and create the information necessary for migration at run-time. The SNOW environment includes a run-time system that captures the process state at run-time and migrates the process across platforms. We briefly describe our design methodology and structure of the SNOW environment, discuss the key implementation considerations, and present some experimental results.

1 Introduction

Process migration is the act of transferring a live process from one system to another. The two systems may differ in hardware and/or the operating environment, in which case the migration is across heterogeneous platforms. Applications of process migration include *load (re)distribution*, *fault tolerance*, *resource sharing*, *data access locality*, *mobile computing*, *gypsy servers* (non-machine-specific processes that may carry out a range of system tasks), and so on. With the increasing penetration of Unix and NT based systems in commercial world, the need for efficient process migration is becoming important in mission critical enterprise network computing environments, as well. While process migration is relatively easy in a homogeneous environment, it is much more challenging in heterogeneous environments and requires a coordinated approach starting from the application development phase through the compiling and execution stages. In this paper, we consider this problem and describe one approach to deal with some of the issues.

To realize its full potentials, process migration (1) must preserve its internal state across migration, (2) must be efficient, and (3) must be as transparent as possible to the external environment. Preserving internal state means the process must be able to continue its execution past the migration point exactly in the same manner as it would have, had it not migrated. Efficient accomplishment of this task means minimum or no re-execution of any of the code that preceded the migration point, in addition to performing the actual actions efficiently. External transparency implies maintaining transparency with respect to the networks, the file systems, and any external clients or servers or other entities the process may be coordinating its activities with. Thus, process migration requires (1) capturing and restoring internal process state (represented in the process stack and heap, shared data

*Department of Computer Science, Louisiana State University, Baton Rouge, LA 70803-4020.
†IBM T.J. Watson Research Center, P. O. Box 704, Yorktown Heights, NY 10598.

segments, and any shared libraries being used by the process); (2) capturing and restoring externally exposed states including the network state (represented by the messages in the network, state of the communication sub-system, and state of the connections to clients and servers); (3) signals; and (4) capturing and restoring the state of the I/O sub-systems.

Clearly, differences across heterogeneous platforms, in terms of instruction sets, process representation and structure, memory organization, system/kernel interfaces, all tend to limit the scope of efficient process migration. In some cases, especially those involving complex external states, capturing and restoring the states efficiently may be almost impossible. To address some of these issues, we have developed a system which we describe in the rest of the paper. In the following section, we briefly describe various components of this framework. In Section 3, we describe a precompiler that analyzes and annotates application source code so that the application processes are capable of migration at run-time. The run-time mechanisms used to bring about the actual process migration at run-time are described in Section 4. We present some preliminary performance results in Section 5 and conclude the paper in Section 6.

2 The SNOW Framework

We have developed a framework called the Scalable Network Of Workstations (SNOW) system, that addresses some the issues discussed in the introductory section. SNOW architecture consists of four components – Network Of Workstations (NOW), Process Migration and Re-creation Environment (PMRE), migration enabled applications, and a resource locator and migration coordination system.

NOW is the fundamental resource of the SNOW system and forms the first layer of the environment. This layer may consist of multiple heterogeneous platforms connected to one another by means of a network. We do not restrict this layer to be a single cluster, but it could be a cluster of multiple clusters.

PMRE is the primary layer that provides the basic plumbing for process migration across platforms. This layer performs three functions: (1) performing the necessary actions for capturing, transferring, and restoring the internal state of a process, (2) maintaining consistency in the communication state before and after migration, and (3) capturing and restoring other external states such as the process state with respect to a file system. If any of these functions are performed by the native environment that is more efficient, PMRE takes advantage of those facilities. For example, if the communication subsystem can capture the network state for the process in a reliable manner, PMRE bypasses its own generic mechanisms and exploits those that are provided by the native subsystem.

To be able to migrate under the SNOW environment, an application must be first enabled for migration. Under SNOW, a process can be migrated only at certain safe points during the course of its execution. Moreover, it must incorporate systematic and efficient techniques for reconstructing the stack and the heap when migrated across platforms. For this we apply a combination of compile-time and run-time analysis and support mechanisms. A pre-compiler called migration supporting pre-compiler or MSP (described in detail, in Section 3), is used to determine migration points and to add some auxiliary instructions, in the application source code. These instructions generate state related information when migration is to be taken and is then used by PRME for capturing and restoring the process state. PMRE consists five primary components: an extended virtual machine, migration protocol, naming abstraction, data collection and restoration, and reliable communication.

To capture the external state on the network, we currently depend on the communication sub-system. The mechanism used is that of draining any messages that are already in the network and blocking any subsequent connections to the process until migration is complete [1]. If the underlying communication sub-system is capable of providing these facilities, then we make use of those. Otherwise, the concept of PVM style virtual machine, which we refer to as the extended Virtual Machine (eVM) is used to encapsulate the message passing layer which achieves the desired result. In this case, eVM makes use of the reliable communication protocol provided by TCP/IP. Only the communication with the migrating process is affected while the rest of the application processes, that are not being migrated, can continue computing and communicating with one another.

The external state in the communication sub-system also includes address maps that translate application-level process addresses to network addresses. Since process migration affects its network address, it is necessary to modify the address maps in the communication sub-system (local and remote). Here again we make use of any hooks provided in the underlying communication sub-system, if available. The hooks could be in the form of dynamic update of the communication library and/or of the low-level communication layer. In the absence of such hooks, we rely on the extended virtual machine that coordinates with the virtual machines of other communicating processes for systematically updating the address maps.

3 Migration Supporting Precompiler

One goal of the Migration Supporting Precompiler (MSP) is to make a PVM/MPI source program written in high-level programming language such as C migratable in a heterogeneous environment. For this, MSP analyzes the input source program and produces the MODF (MODiFied) file, which is migration capable with the help of the run-time system in PMRE. There are three major functionalities in MSP: migration point analysis, necessary data analysis, insertion of migration macros with data transfer algorithms. Migration operations can be generated automatically by the precompiler. For efficiency and debugging purposes, however, user interfaces are also provided so that experienced users can customize migration operations in their applications.

3.1 Migration Point Analysis

A migration point is a location in the body of a program where the process can be migrated safely and correctly. Finding efficient migration points is non-trivial. The migration cost and transparency may be affected based on where the migration point is chosen. Clearly, if the programmer can specify these points in the source code, then that is the most straightforward method for identifying migration points in a program. For experienced programmers this may be relatively easy since they usually know the structure and workload in various parts in the application. However, in general, this can be undue burden on the programmer. This can be especially difficult in the following cases:

- legacy applications,
- large applications where many programmers are involved in the development, and
- complex applications where any systematic program analysis is a challenging task.

MSP adopts a new approach for migration point analysis. The placement of migration points is based on the following considerations: (1) the proximity of two adjacent migration points, (2) the size of the live data and control structure that defines the internal state, and

(3) the ability to capture the external state. Heuristics are developed to take into account each of these factors. The proximity of two adjacent migration points, or the frequency of migration points in a program, is governed by the tolerance to the actual process migration cost. In general, this cost (usually in terms of time) should not exceed a certain fraction of the time spent on useful work done by the process since the last migration. For this, work performed by the program is estimated by compile time analysis. MSP views each statement in the program as an *instruction*. Each instruction is weighted by the number of floating-point operations involved in its execution. The number of floating-point operations under all possible branches of execution as specified by the program structure is used as an approximation for the *computation workload* of a portion of code. Further, in our analysis currently users have to specify the maximum number of process migrations allowed per unit of work. This is referred to as the *estimated frequency of process migration*, which is defined as an inverse of the estimated floating-point operations between any two consecutive migration points.

We note here that, since process migration is allowed only at a migration point, the frequency of process migration gives an upper bound on the response time for the process to begin migration after a migration signal is issued to the process. Thus, the frequency serves the dual purpose of (1) limiting the ability of resource owners or schedulers to interrupt the running process for migration, as well as, (2) controlling the responsiveness of a process for migration.

To limit the size of the live data to be transferred during a process migration, MSP is biased towards choosing locations within the main function or those invoked earlier in a function calling sequence and locations within the body of the outer loop in a nested loop. The rational is that such points tend to involve small number of variables.

Moreover, the migration point insertion should also take into account the extent to which the state external to the process would be affected by migration. Typically, this is determined by the interactions of the process to be migrated with I/O and communication subsystems, as well as its interactions with the kernal and shared system resources. Although the PMRE data communication protocol can guarantee reliability, choosing the location with less pending I/O or message passing operations can help reduce costs of process coordination and message forwarding of the process migration protocol. Currently, MSP does not make any special provisions for handling these interactions.

In our current solution, the following observations are made as a guideline in finding appropriate migration points.

1. In a sequence of instructions excluding the I/O, message passing, and control flow instructions, MSP counts number of floating point operations and inserts a migration point according to the estimated frequency of process migration. Once a migration point is inserted, the counting is restarted. A sequence of instructions is the simplest component in the program structure; it may appear as a part of a branch instruction, a loop instruction, or a function.

2. When MSP encounters a branch instruction, it assigns different counters to each branch. MSP assigns its accumulated number of instructions to each counter and continues its counting on each branch separately. The criteria for migration point insertion for sequential code is applied to each counter. At the end of the branch instruction, the counter with maximum value among branches is used in future analysis.

3. In the case of a nested loop, we consider the innermost loop first. If the loop body can have at least a migration point according to the calculation on total amount of work inside the loop and the frequency of process migration, we select the migration points where the number of live variable is smallest. Otherwise, MSP tries to estimate the number of iterations and total amount of works in executing the loop, and use them as a part of the migration point calculation for the outer loop. If there is no outer loop, MSP accumulate the amount of works as a part of sequential code. Recent researches in Symbolic Compiler Analysis [2, 3] can be employed to estimate the number of iteration.

4. For a subroutine call instruction, if there is no migration points inside the subroutine, the total amount of works of the subroutine is accumulated to the current work counter. Otherwise, MSP assumes that at least one process migration may occur during the call and, thus, the counting is restarted immediately after the subroutine call.

5. In the course of program execution, I/O and communication latency costs can often be significant in the overall complexity of the application. An estimation on their cost is not easy due to the dependency on the amount of involving data and external factors such as network contention and resource availability.

   In our model, these instructions are clustered according their locational proximity. Then, we put migration points to separate those clusters from each other. To avoid message forwarding overheads during process migration, we choose not to insert migration points among these instructions when they are closely located.

3.2 Data Analysis and Source Code Annotation

The goal of data analysis is to minimize data transfer time during process migration. To do so, the *Necessary Variable (NV)* set–defined as the minimum set of data needed to be migrated at the migration point–is identified using live variable analysis [4, 5]. In general, such data analysis can be complicated. Complex user-defined data types and pointers may be used in the applications. Also, memory can be dynamically allocated during program execution. To handle these problems, we employ both compile-time and run-time systems in our solution.

After MSP determines a set of live variables at every migration point, MSP analyzes the variables and their data types at compile-time. It also creates a data structure containing information about data types and type-specific functions to save and restore memory objects for a run-time library created to support transmission of complex data structures.

In the runtime system, we have developed a novel concept of Memory Space Representation (MSR) graph model to recognize complex data structure in the memory space of a process [6, 7, 8]. A run-time library based on the MSR model is developed to provide mechanisms to save and restore the application data structure.

After defining migration points and their necessary variables, MSP inserts special global variables and macros to create its output (the MODF file). Important global variables are Control Buffer (CB), Data Buffer (DB), Execution Flag (EF), and other variables such as those for communication protocol at the top of the file. CB keeps track of function calls before the migration, whereas DB is used to store live variable of functions in CB. The EF contains the execution status of the process at a certain point of program execution. Its values such as NOR, MIG, and RES represents the execution in normal, process migration, and process restoration, respectively. Finally, macros are inserted to carry on the migration of a process from the original machine, and its resumption on the destination machine.

4 Extended Virtual Machine

The extended virtual machine (eVM) consists of a set of daemon processes. Each daemon resides on a workstation and represents the workstation as a computing node. Major responsibilities of a daemon are:

- to maintain process identification,
- to provide mechanisms for dynamic process reconfiguration, and
- to handle interprocess communication in a reliable manner.

Identification is given to a process in three levels. At the workstation level a process is identified by the operating system. At the virtual machine level a process identification is defined by the workstation rank number in the workstation pool and the process rank inside each machine. Finally, the application process rank number such as that employed by MPI is given to every process. Mappings among these are maintained by a table, namely the Process Table (PT), constructed inside memory space of every application process.

To bring about process migration effectively, a coordination among application processes, the virtual machine daemon, and the resource scheduler is required. In our design, the migration procedure consists mainly of two parts. First, a mechanism to pause a process at certain points of execution and to maintain global consistency over any period of time desired. Second, once the process is frozen, existing network connections are shut down and the execution state and data of the migrating process are transferred to the new machine. Daemons on the two platforms coordinate with one another according to a set protocol during this entire procedure. The destination daemon also controls restoration of data as well as network status of the destination process.

In providing the reliable connection-oriented data communication in process migration environment, we consider two cases. First, if there is no established connection between the peer and the migrating process and the peer wants to send data, the peer has to first send a connection request to the migrating process to establish a connection. Under the protocol we are using, if the receiving process is undergoing migration, it does not grant any connection request and, as a result, the sender process has to wait until the migration is complete. Once the receiver process completes migration, eVM notifies the sender that the receiver no longer exists at the previously requested destination. The sender then has to contact the resource locator/scheduler to see whether the receiver process terminated due to completion or has migrated elsewhere. In case of migration, the sender gets information about the new location of the migrated process, updates its PT, and resends a connection request to the the new process.

The second case covers the situation where there is a connection between two processes already exists and one of these processes, decides to migrate. In this case, all messages sent to the migrating process must be received to a buffer before closing connections. This part is accomplished by the process migration protocol. The buffered messages are forwarded to the initialized process during the migration operation. After migration, future direct communications between the sender and the new process follow as describe above.

The virtual machine also provides interfaces to its services in form of a programming language library, which contains subroutines for both scheduler and application processes.

Scheduler Interface is a set of subroutines to monitor and control work loads of the computing environment. These subroutines are used only by the scheduler. They include dynamic configuration functionalities such as adding hosts and deleting hosts, and process migration supports such as process initialization, pre-initialization and migration. The interface also contains subroutines to collect load informations of the system [1]. *Application (or User) Interface* is a collection of subroutines that support efficient distributed computing of the application processes in the migration environment. Their functionalities include process controls such as process creation and termination, and message passing services such as sending, receiving, and multicast. Since the process control subroutines can effect work loads of the environment, every request according to these subroutines from the applications must be supervised by the scheduler.

5 Experiments

A prototype SNOW system has been developed. We have conducted three experiments on three different applications on the prototype. In the first two experiments, we show heterogeneous process migration of sequential processes from a DEC 5000/120 workstation running Ultrix to a Sun Sparc 20 workstation running Solaris 2.5. The migration is conducted via the 10 Mbits/s Ethernet network. The experimental programs have two different types of data structures and execution behaviors. First, the C version of the linpack benchmark is a numerical intensive application with array-based data structure. Second, the C implementation of the bitonic sort program which contain intensive dynamic memory allocation and recursion.

Program	Linpack		bitonic	
Tx Size (in bytes)	325,232	8,021,232	46,704	182,248
Scan	0.303	5.591	0.150	0.419
Tx	0.357	9.815	0.053	0.191
Restore	0.095	2.962	0.077	0.278
Migrate (in seconds)	0.756	18.368	0.280	0.889

Table 1

Timing results of heterogeneous process migration of the linpack and bitonic sort programs.

As shown in Table 1, we have tested both programs with two different data sizes, which cause different size of data transmission (Tx Size) during process migration. The total costs of process migration can be split into three parts: the cost of scanning data structure of a migrating process (Scan), the cost of transmitting those data (Tx), and the cost of restoring them on a destination machine (Restore).

In the migration of the linpack benchmark, the cost of memory scanning and restoration are smaller than the data transmission because of the simplicity of the array-based data structure used in the benchmark. The data collection function of SNOW runtime library does not spend a lot of time searching for data in the program memory space. On the other hand, the migration performance of the bitonic sort program spend higher amount of time scanning and restoring memory space. Since the bitonic sort contain tree data structure and have recursion behavior, the data collection and restoration mechanisms of SNOW must spend more time searching data in the process memory space.

Finally, in the last experiment we tested our prototype process migration mechanism, process migration protocol, and reliable direct data transmission with an NAS parallel numerical kernel MG benchmark [9]. The kernel MG is a parallel program to execute four iterations of the V-cycle multigrid algorithm to get an approximate solution to a discrete Poisson problem with periodic boundary conditions on a $128 \times 128 \times 128$ grid. The C implementation using PVM from [9] is modified for migratability. The program contain extensive interprocess communications and complicate data structures. We originally run the benchmark on a cluster of 8 Sun Ultra 5 workstations; each contain an application process. Then, after two iterations, we migrate a process to an idle Ultra 5. All machines are connected via 100Mbit/s Ethernet.

Total	original	modified	migration
Execution time	16.130	16.379	18.833
Communication	4.051	4.205	6.647

TABLE 2

Timing results (in seconds) of overhaed of the migratable kernel MG program in comparison to the original benchmark.

Table 2 shows the measured turnaround time of the parallel MG benchmark. We can see, compared with the total execution time, the cost to carry a process migration is relative small. The overhead of running the modified code is about one percent in average and the overhead of execution time with single process migration is approximately 16 percents. In the comparison of communication time of our modified program to that of the original benchmark, the results shows the overhead of our reliable data communication protocol to be very small. In the parallel kernel MG benchmark, extensive message passing and data communication occurs; over 48 Mbytes of data on the total of 1472 messages are transmitted during its execution. In case of process migration, over 7.5 Mbytes of live data are transmitted with the average of 0.7662, 0.73, and 0.6794 seconds for transmission time, data collection time, and data restoration time, respectively. The process migration protocol spends 0.1166 seconds to coordinate with other communicating processes. The total process migration time on average is 2.2922 seconds. The process migration overhead increases the turnaround time and communication time because some parallel processes have to wait for process migration to complete before resuming data communication with the migrated process.

6 Conclusions

In this paper, we have described a framework that facilitates process migration in heterogeneous environments. The framework is based on a coordinated approach, where migration enabled processes and an environment external to the processes cooperate and coordinate with one another to bring about migration across platforms. The migration enabled processes are instrumented to capture and restore their internal states at migration points that are defined at application compile time. The external environment is responsible for signaling the process for migration at the next migration point, for capturing and restoring the state external to the migrated process including the I/O, communication, and client/server connections. In this paper, we have discussed the salient points of the pre-compiler and the run-time system. We have presented experimental results from three test cases to show the viability of our approach.

Despite its high potential, certain limitations of heterogeneous process migration exist. In binding our method to a language which allows high adaptability to its code and data structure such as C, we must limit the uses of some language features and operations that can corrupt data transformation between the machine-specific and machine-independent formats [10, 7]. Another limitation is on the interoperation between the migratable process and the machine-optimized dynamic libraries. Since most of the libraries are specifically developed and optimized for a particular platform and their code may not be publicly available, it is quite difficult to analyze the equivalency of functionalities during the execution of the same library routines on two different computing platforms. Also, since process migration can occur only at migration points, which cannot be inserted inside the dynamic library routines, our process migration model automatically limits process migration only to be performed within the executable code generated from application modules.

The framework we have discussed is currently prototypical and we are working on improving robustness and adding more functionality. Due to high availability of underutilized computing resources in the networked environment and the availability of high speed networks, process migration is becoming increasingly important for increasing overall resource utilization, load balance, fault tolerance, and quality-of-service in mobile computing. This research is an attempt to provide a coordinate solution to meet the demand.

References

[1] V. K. Naik, S. P. Midkiff, and J. E. Moreira, " A Checkpointing Strategy for Scalable Recovery on Distributed Parallel Systems," in *the Proceedings of SC97:High Performance Networking and Computing*, (San Jose, CA), Nov 1997.

[2] T. Fahringer and B. Scholz, "Symbolic Evaluation for Parallelizing Compilers," in *Proc. of the 11th ACM International Conference on Supercomputing*, (Vienna, Austria), July 1997.

[3] W. Blume and R. Eigenmann, "Demand Driven Symbolic Range Propagation," in *Proc. of the 8th Workshop on Languages and Compilers for Parallel Computing*, (Columbus, OH), Aug. 1995.

[4] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[5] C. N. Fischer and J. Richard J. LeBlanc, *Crafting A Compiler.* Benjamin/Cummings, 1988.

[6] K. Chanchio and X.-H. Sun, "Memory space representation for heterogeneous network process migration," in *12th International Parallel Processing Symposium*, Mar. 1998.

[7] 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.

[8] 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.

[9] S. White, A. Alund, and V. S. Sunderam, "Performance of the nas parallel benchmarks on pvm based networks," Tech. Rep. RNR-94-008, Emory University, Department of Mathematics and Computer Science, May 1994.

[10] 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.