EFFICIENT CHECKPOINTING FOR HETEROGENEOUS COLLABORATIVE ENVIRONMENTS: REPRESENTATION, COORDINATION, AND AUTOMATION

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Kasidit Chanchio
B.S., Thammasat University, Thailand, 1990
M.S., Louisiana State University, 1996
December, 2000

To My Parents

Acknowledgments

First of all, I would like to thank my major professor, Prof. Xian-He Sun for his trust on me and his invaluable guidances on my research. In the past five years, he has taught me not only technical skills, but also how to work professionally. His constructive criticism, careful training, cheerful encouragement, and confidence in my abilities have helped me building up skills for my research. He is not only caring for his students, but also teaching them to find their ways to success. I also appreciate his kind support throughout my study. Without his help, I would not have reached this proud "point" in my career and my life. I could never thank him enough for what he has done for me.

I would like to thank the co-chairman of my doctoral committee, Prof. Donald H. Kraft and the committee members, Prof. Jagannathan Ramanujam, Prof. Aiichiro Nakano, Prof. J. Bush Jones, and Prof. Brian Wolshon for their comments and suggestions. I particularly thank Prof. Ramanujam and Prof. Nakano for their kindness and encouragement during my doctoral study.

My special thanks go to Kirk Cameron, Steve O'neal, Guillermo Tonsmann, and Adisak Srinakarin and Y. H. Hwang for their brotherly friendships and encouragements. Dr. Cameron is the best friend whom I can always count on. I thank Steve O'neal for teaching me all about American culture and computer games. For me, Guillermo Tonsmann is my elder brother. He always has great ideas to borrow. I would like to thank Elias Khalaf for his professional and personal helps during my study. I am grateful to Carrol Landry and Andy Kolchin for their warm friendship and hiking trips. I am thankful to Kunya Artson, Khemsuda Kaewkantha and the Scollard, Prof. Witoon & Ms. Ratana Prinyawiwatkul, Pee Aub and Pee Tum, Orachos Napasintuwong, Jay Rangsibramanakul and his parents, Pee Pey, and Note for their wonderful friendship in Baton Rouge. I can never thank enough to my wonderful friends–Dr. Xingfu Wu and his family, Dr. Wu Zhang, and Mr. Shao Shanrong–whom I regard as my own family in Chicago. They are always there when I faced hard time in life. I appreciate warm friendships from Pasakorn and Piboon Apirakvorapinit, Patima Klinsong, Witty Sri-saan, Xiaoshan He, Cong Du, Surendra Byna, Hareesh Achi, Narin, and the Thai soccer players at IIT during my stay in Chicago.

It has been said that "friendship is forever", I thank my lifelong friends Ponchai Lertchoo mongkol, Saravut Wansanoh, Somchai Lertusawawiwat and Varaluk Chalitapanukul for their kindness, encouragement, and helps to me and my family during my long journey away from home. I also thank my close friends from Thammasat including Ky and Pom Panachet, Thanwadee Thanitsukarn, Pawana Thongratana, Worawut Sorasermsombat, Pornpan Tongprasit, Kaiwan Hongladarom, and the rest of the TUCS1, TUCS2, and TUCS3.

Most importantly, I am very grateful to my parents for providing me their supports, encouragement, and love. They are the most important persons in my life. I am also thankful to my two younger brothers, Chaichan and Thannop, for taking care of my family while I am away from home. Now it is time for our family to move on to a bright future.

Table of Contents

List of Tables

List of Figures

Abstract

Checkpointing can be used to adapt resource utilization in heterogeneous distributed environments. In checkpointing, the state of a process is captured and later restored on a computer to restart execution from the point where the state capturing had occurred. Such capability can be applied to process migration for which resource utilization is adapted toward high-performance by moving a running process from one computer to another.

For a heterogeneous environment, problems in checkpointing can be categorized into three domains regarding mechanisms to capture and restore the execution state, memory state, and communication state of a process. Although a few solutions have been proposed, a well-defined solution is not yet exist.

This thesis presents a practical solution to capture and restore the process state in heterogeneous distributed environments. The solution is based on three novel mechanisms: the data transfer mechanism, the memory space representation model and its associated data collection and restoration mechanisms, and the reliable communication and process migration protocols. These mechanisms define the machine-independent *representations* of the execution state, the memory state, and the communication state. They work in *coordination* to perform process migration in a heterogeneous environment. A software system is designed and implemented to *automatically* migrate a process. A number of process migration experiments are tested on sequential and collaborative processes. Experimental results advocate correctness and practicability of our solution.

Chapter 1 Introduction

The construction of national high-speed communication networks, or the so-called "information highway", marks the beginning of a new era of computing in which communication plays a major role. Until recently, computing resources on networks remain separate units. Now, while the World Wide Web and electronic mail are changing the way people do business, heterogeneous networks of computers are becoming commonplace in high-performance computing.

The emergence of world-wide internetworking and high-speed network allow distributed applications to collaborate large number of computers efficiently for their execution. For years, enormous power of large-scale collective computation resources has been noticed. National efforts such as the Globus [18] and Legion projects [21] have been funded to create large-scale distributed computation environment, or *computational grids*. The grids provide efficient and reliable accesses to the computation resources in the global-wide networks of computers.

Although physical high-speed networks provide fast access to resources, the development of a software infrastructure to govern the utilization of resources in the computational grids is a significant challenge. Heterogeneity poses an initial difficulty in such attempt. Resources in the computational grids are heterogeneous by nature. Moreover, network interconnections on different parts of the grids, can also be of different types. Despite diversity, the software infrastructure of the computational grids is required to identify, access, and utilize the resources in efficient manner. Another difficulty lies on the dynamic nature of computational activities. The bigger the grid size, the more dynamic the activities. Due to numerous participations of users in the environment, the creation and termination of activities are unpredictable. Overtime, this can cause imbalance in resource utilization. To alleviate this problem, adaptability in resource utilization is needed in the grids' infrastructure. Although the abilities to handle heterogeneity and adaptability are key points toward high-performance, the construction of the software infrastructure to support them is quite a challenging task.

1.1 Research Contributions

Checkpointing is a mechanism to capture a process state and resume the process execution using the state previously captured. Although checkpointing is commonly used for fault tolerance, another important application of checkpointing is for acquiring high performance: to adapt resource utilization in a distributed environment by means of process migration.

Most research in process migration is focused on homogeneous environment where a process can only migrate between the same kind of computers. On the other hand, process migration in heterogeneous environment (or *heterogeneous process migration*) where the migration is conducted between different kinds of computers is more complicated. Many questions on how to carry a process's functionalities across machines arise almost immediately with the research issue. At the current time, little research has been attempted and there is no well-defined solution to the problems.

Problems of checkpointing in heterogeneous environment arise due to possible differences of computing platforms when a process is checkpointed and resumed. Difficulties in the design and implementation of checkpointing mechanisms for heterogeneous distributed environment occur in three domains. First, differences in hardware such as machine instruction sets and data representation make it impossible to reuse binary code across incompatible machines. Second, most operating systems provide different memory management. This makes it impossible to transfer data structures of a process simply by copying memory images across machines. Finally, process migration makes data communication in a distributed system more complicated. Since processes can be relocated while passing messages among each other, failure of distributed computations may occur when the transmitted messages cannot find their destinations.

To overcome these problems, a new perception on the process state as well as practical mechanisms for checkpointing are presented. There are three major contributions in this work.

1. **Data transfer mechanism**

   The data transfer mechanism is the mechanism to capture execution location and runtime stack content of a process and represent them in a machine-independent format. The mechanism takes a major role in managing the sequence of data collection and restoration of functions in a process. As a result, the checkpoint is created in a highly structural manner which open ways for sophisticated manipulation including the process state collection and restoration in heterogeneous environment.

2. **Memory space representation and its associated mechanisms**

   The Memory Space Representation (MSR) model is a graph model to logically represent data in a snapshot of process memory space. Memory blocks and pointers are represented in the form of nodes and edges in the MSR graph, respectively. We have invented the associated data structures and algorithms to the MSR model to machine-independently capture and restore complex data structures in the process memory space.

3. **Reliable data communication and process migration protocols**

   The reliable data communication and process migration protocols work in cooperation to guarantee correctness of point-to-point message passing in distributed environments. Algorithms are introduced to perform reliable send and receive operations, which guarantee no loss of messages and preserve the correctness of communication semantics. To capture a process communication state, we invent a process migration protocol to work in cooperation with the data communication algorithms. The migration protocol also defines an algorithm to perform overall process state transfer (execution, memory, and communication states) and algorithms to schedule process migration in distributed environments.

As a result, a software system for heterogeneous process migration is designed and implemented. The system involves the compile-time and runtime systems which work in coordination. In the compilation system, we augment the data transfer mechanism into source codes. The process generated from such source codes is said to be "migratable". The runtime systems consist of libraries to access process memory space and to perform message passing. The process is linked with these libraries which will capture and restore its memory and communication states on an event of process migration.

There are a number of issues that are not the current focus of this work. First, the mechanisms to capture and restore the I/O state of a process in heterogeneous environment is an interesting research issue. The problems on how to maintain correct and reliable accesses to local or remote data in a migration environment is an open research topic. Another interesting issue is the effects of compiler optimization on the "migratable" process. Since we augment the data transfer mechanism to the program source code, our future research will consider the problem of how to prevent the augmented mechanism from being eliminated by compiler optimization when the binary code is generated. There are other research issues such as accesses to dynamic libraries and OS system calls which have been considered extensively in homogeneous process migration. The problems on how to implement them on heterogeneous environment, however, need further investigation.

1.2 Motivation

Our motivation in employing checkpointing for high performance computation includes the following:

1. **Dynamic changes in resources utilization**

   In the national-wide distributed computing environment such as the computational grids [19], computation resources may join or leave the environment at any moment. Since applications running in such large-scale environment are usually of highly importance, process migration is necessary for adaptability of the applications to the constant changes in resource utilization.

2. **Load balancing**

   When load-imbalance occurs in a distributed system, processes on overloaded computers can be migrated to under-loaded computers for load balancing. In a non-dedicated environment, computers tend to be heterogeneous and privately owned. This means the privately owned machines may only be used for collaborative processing on an available basis. Competition for computing resources does not lead to guaranteed high performance. "Stealing" of computing cycles, instead of competing for computing cycles, is a more reasonable way to achieve parallel processing in a non-dedicated parallel and distributed environment. Previous work [24] shows that process migration is a promising solution to the cycle "stealing" concept. Recent research shows process migration is also efficient for utilizing idle machines for collaborative processing [22].

3. **Locality accesses to data**

   Process migration allows a process to move closer to sources of data or to aquire a specific device. In many cases, moving data to the process is not cost-effective when the data size is very large. Moving the process to the sources of data could be a better alternative in such circumstances. For example, in a large enterprise or even the internet, data is highly distributed. It could be more cost-effective to have a process migrate to data rather than having all the data transmitted for processing on one computer.

4. **Accesses more powerful resources**

   Processes can be migrated to more powerful computers to seek faster computation or higher quality communication services. For example, a process can migrate from a personal computer to a workstation server with higher CPU speed and physical memory to solve computation intensive problems.

5. **Mobile computing**

   In a mobile computing environment, a process in a mobile computer can move to a more powerful server computer at a cellular hub. Another example is that a mobile computer may issue a request to a server computer at the hub while it is moving to another cellular area. Instead of spending more time in communication between the previous server and the mobile client due to indirect message routing, the server process may migrate to the server machine at the new hub covering the area where the client currently resides.

Checkpointing is also an important mechanism in other areas. Mechanisms to checkpoint a process in heterogeneous environment could also be beneficial to the following applications.

Fault-tolerant computing is a common application of checkpointing. Checkpointing enables the execution of the code to be resumed from a previously saved process state (i.e., execution state, memory state, and external state) rather 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 by either resuming the process at the same machine after the fault is recovered or resuming at a new machine via network. The ability to checkpoint and restart process among heterogeneous machines can significantly expand resource utilizations.

Other applications of checkpointing which are not for performance are fault-resilience and system administration. Checkpointing is applied for fault-resilient when the current host of a process has partial failure and then the process is checkpointed and migrated to somewhere else. In case of system administration, system administrators can migrate processes on a computer to other computers when they want to upgrade or reconfigure the computer the processes is residing on.

1.3 Problems Domains

Efficient process migration in a heterogeneous distributed environment is a subject of great challenge. Problems in the design and implementation of such mechanisms lay in three problem domains: mechanisms to migrate execution state, memory state, and communication state of a process. In a heterogeneous environment, the execution state of a process cannot simply be transferred by copying program counter registers across machines. Machine-independent mechanisms are required for transferring the execution state. For the memory state, problems arise due to different in memory configuration and memory management among heterogeneous computers. Memory storages on different computers could have different data formats, addressing scheme, etc. Mechanisms to capture information in a process memory space as well as to migrate them to a new computer are needed. Finally, correctness in data communication should be maintained during process migration. In internet and enterprise network computing, processes are most likely communicating with one another. Correct data communication must be guaranteed. Mechanisms to migrate the communication state have to be carefully developed to warrant the correctness and, in the meantime, minimize the overhead.

To implement mechanisms for heterogeneous process migration, we have to deal with the following problems:

1. **How to capture and restore execution state of a process?**

   Since the execution state of a process cannot simply be transferred by copying the program counter register across machines, we need a mechanism that can represent the "point" of execution of a process in a machine-independent format. We should note that the process of our interests are those written in imperative, stack-based languages such as C, Pascal, and FORTRAN. The mechanism must be able to keep tracks of the current location of program execution and its function call behavior.

2. **How to capture and restore memory state of a process?**

   Although data marshalling technology such as XDR have been used to collect and restore data for remote procedure call [11], it is not general enough to recognize complex data structures in process memory space. Most of data complexity are results of the uses of pointer. Moreover, data in a process can reside on various memory segments which, in turn, effect their lifetime in many ways. Efficient mechanisms to collect and restore data in heterogeneous environment are needed.

3. **How to capture and restore communication state of a process?**

   The ability to capture and restore communication state of interaction between a process is crucial for distributed computation. In a message passing environment, such mechanism must guarantee no lost of messages and preserve communication semantics such as message ordering. It must not corrupt the logic of distributed computation or cause deadlock.

4. **How to transfer process state efficiently during a migration?**

   After the process state is captured, it must be transferred to a destination computer for restoration. Since the process state is usually very large, mechanisms for efficient process state transfer must be considered.

5. **How to maintain scalability in distributed environments?**

   When a process is migrated in a distributed environment, we have to consider its effect to other processes in the environment. For examples, how to update new location information of the migrating process to other processes in the environment? Broadcasting such information is not a good solution because the environment could be vary large. A scalable technique must be applied for process management.

1.4 Design Principles

Based on the problem statements, we designed our solution based on the following principles:

1. **Heterogeneity**

   Process migration is required to perform in heterogeneous environment with correctness despite complex execution behavior and data structures of the migrating process. In a heterogeneous environment, the execution state of a process cannot simply be transferred by copying program counter registers across machines. Machine-independent mechanisms are required for transferring the execution state. For the memory state, problems arise due to different in memory configuration and memory management among heterogeneous computers. Memory storages on different computers could have different data formats, addressing scheme, etc. Mechanisms to capture information in a process memory space as well as to migrate them to a new computer are needed. Moreover, precisions of data values should be maintained at a minimum level offered by data representations among the source computer, the communication media, and the destination computer. Finally, in term of data communication, if a message is transmitted between heterogeneous computers, the data communication protocol has to convert message contents into a data format of their destination computers.

2. **Responsiveness to process migration requests**

   In our design, process migration is performed within a certain amount of computation after a migration request have been intercepted by the migrating process. Instead of allowing a process to perform process migration at any time, we believe that the process should be allowed to migrate only after every certain amount of works have been performed. We also believe that a process should be allowed to migrate itself under a particular circumstance.

3. **Efficiency**

   In non-migration situations, the augmentation of process migration capability should not incur high execution overheads to a process. When the process migrates, the transfers of execution state, memory state, and communication state of a process must also be performed in an efficient manner.

4. **Correctness of distributed computation**

   Most importantly, the migration software system must preserve the logic of distributed computation. Additional activities which may occur due to process migration must not corrupt the execution of programs that have correct computation logics. These activities include mechanisms to transfer the process state and manage process migration in heterogeneous distributed environments. For collaborative processes, the migration software system must guarantee no loss of messages and does not introduce deadlocks.

5. **Scalability**

   Scalability in process migration can be measured in many respects [27]. First, process migration should operate regardless of the number of computers in the environment. Second, during a process's lifetime, there should be no limit the number of migrations. Third, process migration must be scalable in heterogeneity ,i.e., the complexity in migrating a process does not depend on the number of different computation platforms in the environment. Fourth, process migration should not require global synchronization to update location information of a process. Finally, it should *not* block other processes from sending messages to the migrating process or other processes in the environment during process migration.

6. **No residual dependency**

   After process migration, a process should be independent from its previous host; no futhur access to resources or communication from the previous machine are needed.

1.5 Solution Overview

We propose mechanisms and a software system for carrying process migration between computers in a heterogeneous distributed environment. The target applications of our process migration mechanisms are sequential and parallel programs written in stack-based languages such as C and FORTRAN. They are computation intensive applications requiring high performance computation resources. We assume processes communicate each other via message passing.

Mechanisms have been proposed to solve the three fundamental problems on how to migrate execution state, memory state, and communication state in heterogeneous process migration. They have been implemented into our prototypical compilation and runtime systems.

Execution State

In order to transfer execution state of a process in a heterogeneous environment, we have developed a software, namely the *pre-compiler*, to analyze program source codes and augment process migration operations into the source codes in the form of programming macros. At the program analysis stage, the pre-compiler first checks whether the source code is valid for process migration. In languages such as C, certain language features are highly dependent on underlying computation platforms and may not be safe for heterogeneous process migration [32]. These language features will be transformed to equivalent features during the pre-compilation process.

After the source code is verified, mechanisms to capture, transfer, and restore execution state of a process are implanted into the source code. In doing so, the pre-compiler first determines appropriate locations in the source code, namely the "poll-points", which allow a process to migrate. The selection of such locations can either base on automatic program analysis or user-directive. We call a poll-point where process migration occurs at runtime as the "migration point". The mechanism for automatic selection of poll-points is called the "migration point analysis". Then, at each migration point, the pre-compiler applies "live data analysis" to determine a minimal set of variables whose values need to be collected for future computation after a migration completes. Finally, the pre-compiler inserts macros at the poll-points in the source code. These macros contain algorithms which work collectively to collect, transfer, and restore process state during process migration. We name such cooperation the "data transfer mechanism". A "buffered data transfer" (BDT) mechanism is designed to carry the data transfer effectively and will be discussed in Chapter 4.

Migration Environment

In a process migration environment, we assume that the augmented source code is distributed to every computer which may involve process migration, and executables are generated by native compilers on different platforms. During the compilation, the source code is linked with libraries to handle memory state transfer and reliable data communication, which will be discussed in Chapter 5 and 6, respectively.

In our design, a runtime system for process migration consists of a virtual machine, runtime libraries, and a scheduler. The virtual machine consists of daemon processes cooperating each other for process management in PVM's style [20]. The runtime libraries contain routines to perform process state transfer and message passing. The scheduler is a process which monitors resource utilization and assigns application processes to computers in the environment.

At runtime, after determining the migrating process, source, and destination computers for process migration, the scheduler sends a signal to invoke an executable (on a destination computer), which is generated from the same annotated source code as that of the migrating process, to wait for process state transfer. Then, it sends a migration request to the migrating process which, after intercepting the request, will continue execution until the nearest poll-point is reached. At the migration point, the annotated macros embark the data transfer mechanism to coordinate process state transfer between the two machines. Finally, program execution is resumed on the destination machine after the migration is completed.

Memory State

Generally, in a process memory space, every variable occupies a piece of memory containing data values of a certain type. We refer to this piece of memory as a "memory block". A memory block may reside in any part of a program memory space: global, stack, or heap segments. The data type could be primitive data types such as character, integer, and pointer, or it could be compositional types such as array or structure. In case of pointer, the data value is a memory address. The pointer is considered as a primitive data type which allows a referential relationship among memory blocks to be created.

To migrate a process across heterogeneous platforms, data structures in the process memory space must be correctly collected, transferred, and restored. Data values in memory blocks as well as the referential relationships among them must be preserved in the migration. In our approach, we perform the followings:

1. We define a logical memory model, namely the "Memory Space Representation" (MSR) model, to represent process data structures machine-independently. The model is based on graphical notations in which a memory block is represented by a vertex and a referential relationship between two memory blocks (generated by a pointer) is represented by a graph edge. A diagram in Figure 1.1 shows the basic idea behind our mechanisms where the machine-specific data structures are mapped into the MSR model and then converted to the physical memory format of the destination computer.

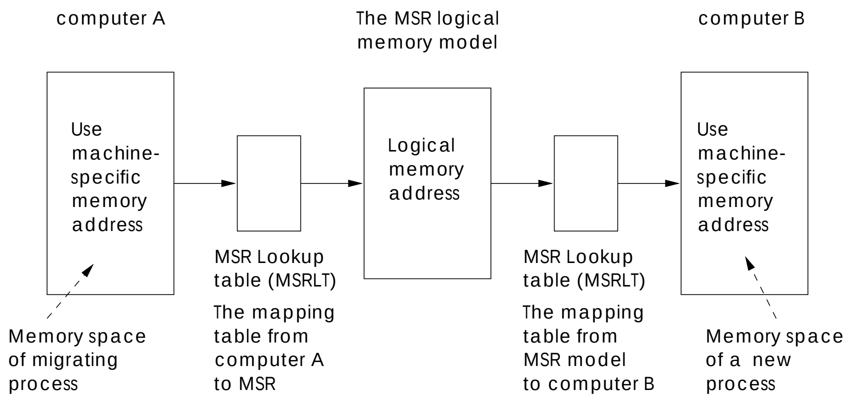

computer A
The MSR logical memory model
computer B
Use machine-specific memory address
Logical memory address
Use machine-specific memory address
MSR Lookup table (MSRLT)
The mapping table from computer A to MSR
MSR Lookup table (MSRLT)
The mapping table from MSR model to computer B
Memory space of migrating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

Figure 1.1: Relationships between logical and physical memory models

In our implementation, data structures and a runtime library that provide such mapping mechanisms are created. The MSR Look up table (MSRLT) data structures are built to match physical and logical representations of memory blocks in process memory space. On the other hand, the "MSR graph Manipulation" (MSRM) runtime library contains routines for data collection and restoration. The library is linked to the annotated source code during the platform-specific code generation for different computing platforms.

2. When a migration occurs, the data collection operation on the migrating process starts collecting values of live variables at the migration point. The annotated macro at the migration point contains calls to routines in the MSRM library that perform data collection and restoration.

   Since the variables' storages are associated to the MSR graph nodes, the data collection operation traverses the MSR graph in a depth-first-search manner and collects information in memory blocks. Finally, the collected data from visited graph nodes and edges is converted into a machine-independent format and sent to the destination machine.

3. After receiving the machine-independent migration information, the new process on the destination machine employs the MSRLT data structures and the MSRM library routines to map graph nodes in the MSR model back to physical memory blocks in memory space of the destination computer.

4. Then, the restoration operation extracts data values from the transmitted migration information, converts them to machine-specific format, and stores the output data values at appropriate memory block locations in the memory space of a process on the destination computer. We discuss detailed mechanisms in Chapter 5.

Communication State

In a process migration environment, messages may be in transit while their sender or receiver processes migrate. Mechanisms to guarantee no message loss and correct message ordering must be developed for correct data communication.

In our design, we have developed a data communication and process migration protocols for such purposes. The communication protocol involves algorithms for sending and receiving messages, while the process migration protocol concerns algorithms to coordinate a migrating process and its communication peers. Both protocols work collectively to support correct data communication. The protocols is discussed in details in Chapter 6.

Our protocols are suitable for process migration in large-scale distributed environments due to the following properties. First, they make process migration scalable. During the migration, the protocols only coordinate a minimal set of peer processes, only those directly connected with the migrating process. Second, the protocol is nonblocking i.e., they allow other processes to send messages to the migrating process simultaneously during a migration. Third, the protocols do not create deadlock. The process migration protocol prevents circular wait while coordinating a migrating process and its peers for process migration. Finally, the protocols can be implemented for heterogeneous environment. We have implemented them in a prototype message passing library running on top of the PVM communication system.

1.6 Organization

The rest of this dissertation is organized as follows. Chapter 2 discusses related work. We give the overview of software systems for process migration in Chapter 3. Chapter 4 presents the compilation system which annotates mechanisms for heterogeneous process migration into source code. We address techniques of migration point analysis, live variable analysis, and source code annotation in details. Chapter 5 presents the MSR concept and software system to collect and restore process memory state. Chapter 6 discusses communication state transfer in process migration and presents the reliable data communication and process migration protocols. We discuss experimental results in Chapter 7. Finally, Chapter 8 discusses conclusions and future work.

Chapter 2
Related Work

Various approaches for process migration have been proposed. We consider them in four categories: OS kernel support, user-level checkpointing, application embedded, and mobile agent. Our approach can be classified as a user-level checkpointing technique and has its detailed discussion in section 2.1. For process migration at OS level, many projects such as MOSIX, V, Sprite, and Accent [27] have been implemented. These systems are highly dependent on OS implementation and underlying hardware. They are hardly possible to extend to support process migration in heterogeneous environments. In the application embedded approach, programmers have to integrate process migration operations as a part of their codes. Although this approach is simple and can support heterogeneous migration, it is application-specific and not transparent to users. Process migration under the mobile agent approach, on the other hand, can support heterogeneity and user transparency. Their migration mechanisms are based on interpretive languages such as Java and Telescript. However, while providing a satisfactory solution for many internet applications, mobile agent may not be an appropriate choice for computation intensive applications.

2.1 User-level Checkpointing

User-level checkpointing is a mechanism to capture process state using tools available for users without acquiring special privilege from OS. Condor [25] is well-known software that uses the checkpointrestart mechanism to implement process migration. The Condor library is linked to the object code of a process so that checkpointing is performed when the process intercepts a migration request. Since Condor checkpoints binary images of a process, it only supports process migration between homogeneous computers. Condor does not support reliable inter-process communication during process migration; it only provides process migration services for sequential processes. Although our approach and Condor are both implemented in user space, they are different in the followings:

1. Like Condor, we employ the checkpointrestart technique to capture and restore process state information in user space. However, the checkpoint generated by our approach is different from that of Condor in that it is machine-independent and contains communication state of the migrating process.
2. In our approach, process migration mechanisms are augmented to user applications during software development. Unlike Condor where the migration mechanisms are linked to applications' object code, our technique annotates the mechanisms to source code before passing it to a compiler.
3. Our approach allows heterogeneous migration to perform at migration points. Without a concern for heterogeneity, Condor, on the other hand, can perform process migration immediately after the migration request signal is intercepted.

Although the process migration mechanisms are parts of the executable, they are user transparent and do not depend on a specific knowledge about the application. As mentioned earlier, program analysis and augmentation can be performed automatically by the compilation system, which, in turn, distinguishes our techniques from the application level process migration where specific knowledge about an application program is required [28].

2.2 Heterogeneity

Heterogeneity has increasingly been addressed in distributed computation with the growth of the network community. Heterogeneous process migration is an important functionality that can enable users applications to exploit enormous amount of resources on the network. Early work of Theirmer and Hayes [36] proposes the use of 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 may create unacceptable migration overheads especially for large-scaled applications.

The work of von Bank, Shub, and Sebesta [37] defined the theoretical framework for language system to support heterogeneous migration. Their work indicates that a program can transfer its state among different machines only at locations where its machine codes has equivalent process state. Our approach also follows their concepts. We define the equivalent locations for process state transfer using poll-points.

Recent work of the TUI system [32] by Smith and Hutchinson investigated features of high-level languages that is 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 monitoring agents, `migrout` and `migin`, for data collection and restoration, respectively. The compiler must be modified to provide debugging information and to insert preemption points and call points to intermediate code to capture and restore process state. Our work differs from the TUI approach in three aspects.

1. Instead of employing external agents and a debugger to perform process migration, we augment the migration operations into source code and let a process conducts process migration by itself. We believe our design is more portable and less dependent to external processes.
2. In TUI approach, the needs to modify front-end and back-end of a compiler may limit portability to various computer platforms since the compiler has to be modified for every computing platform in the environment. Our approach eliminates such limitation by means of program analysis and source code augmentation.

3. TUI transfers all data in process memory space during migration, while our approach only transfer live data during process migration.

The other research using program annotation technique to support process migration is the Process Introspection (PI) approach proposed by Ferrari, Chapin, and Grimshaw [14]. The PI annotates source code so that the migrating program can monitor, collect, and restore its execution state and memory state. In terms of software design and software development, their approach is similar to ours. However, the PI and our work are quite different in their fundamental mechanisms to migrate execution state and memory state of a process, which may lead to different process migration performance.

In the PI's design, the mechanisms to capture and restore execution state is similar to our early work [8]. On the other hand, our current work has improved the previous design for better efficiency. Due to source code annotation, macros are inserted to various functions in source code. At runtime, these macros work collectively and perform the data transfer mechanism (introduced in Chapter 1).

According to the PI and the work in [8], their data transfer mechanism work in stack-like manner. The mechanism starts collecting process state from the innermost function and retreats to the main function, while it restores the state information in reverse order. As a result, the state restoration can only start after the collection finishes.

On the other hand, our new design accommodates process migration via network by performing the collection and restoration of the process state in the same order. This design makes it possible to perform process state collection and restoration in a pipelining manner. Nevertheless, performance consideration is subject to future investigation.

In terms of memory state, the PI collects and restores all data in global, heap, and stack segments of a process, while our technique only transfers live data at a migration point. We should note here that neither TUI or PI support communication state transfer in process migration.

2.3 Communication Support

Mechanisms to support correct data communication can be classified into two different approaches. The first approach is using the existing fault-tolerant, consistent checkpointing technique. To migrate a process, users can "crash" a process intensionally and restart the process from its last checkpoint on a new machine. Since global consistency is provided by the checkpointing protocol, safe data communication is guaranteed. Projects such as CoCheck [33] follow this approach. On the other hand, mechanisms to maintain safe data communication during process migration can be implemented directly into the data communication protocol. When a process migrates, process migration operations coordinate with data communication operations on other processes for reliability. Our work and the MPVM project [6] are along the second direction. We choose the second direction because the latter is more scalable and less costly than that of the former. Process migration is important enough to receive an efficient mechanism on its own right.

Theoretically, our protocol designs are based on the classic work of Lamport [23], which considers process execution as a sequence of events, and on the distributed snapshot algorithm of Chandy and Lamport [10], which coordinates processes in distributed computation for consistent checkpointing. A survey on consistent checkpointing can be found in [12, 31].

As mentioned earlier, we do not intend to make process migration a by product of any fault-tolerant mechanisms but its own. Therefore, we customized the classic algorithms in [23, 10] and integrated them to the design of existing message passing software such as PVM [20] and MPI [17]. As a result, we have designed the data communication and process migration protocols containing algorithms for reliable message delivery and process coordination during a migration event. In our current implementation, the two protocols are built on top of the PVM system. An implementation for MPI is our further consideration.

There are a few user-level process migration systems that support communication state transfer in a distributed environment. By our knowledge, however, none supports heterogeneity. MPVM [6] is designed to support transparent process migration for PVM applications. It supports both connection-oriented and connectionless communication based on the PVM direct and indirect communication modes, respectively. A major difference between MPVM and our work is in the connection establishment issue. The MPVM design does *not* maintain automatic connection establishment in point-to-point communications. After the migration, some messages can only be routed via PVM indirect communication, which can severely degrade communication performance. More importantly, process migration under the MPVM system is homogeneous. In the MPVM design, the process is migrated by invoking a new process on a destination computer and transferring runtime information in data and stack segments of the migrating process to appropriate memory addresses of the new process.

Another system, called CoCheck [33], is designed to support coordinate checkpointing for fault-tolerance in the PVM environment. It implements a consistent checkpointing protocol based on Chandy and Lamport's algorithms [10]. CoCheck can also be used to support process migration. However, we should note forcefully here that the main purpose of CoCheck is to support fault-tolerant not process migration. As a result, the CoCheck design suffers two drawbacks which can be alleviated by the specifically optimized systems for migration purpose. First, it requires coordination of all processes directly and indirectly connected to the migrating process. Second, some processes must be blocked from sending messages during checkpointing to maintain consistency. Since CoCheck uses the checkpointing ability of Condor [26] to save the state of a process, it only supports process migration in homogeneous environment. Both MPVM and CoCheck do not consider the situation when an unconnected process initiates communication with the migrating process during the migration.

Our protocols are designed to support dynamic, heterogeneous distribute environments and remove these known drawbacks. In our protocols, the automatic communication establishment is maintained throughout program execution. During a migration, only processes directly connected to the migrating process are coordinated. Our protocols do not block other processes in the system from sending messages to the migrating process. They also allow unconnected processes to make connections and send their data to the migrating process transparently to a migration occurrence.

Chapter 3
Software Structure

Based on our design principles, the software system for heterogeneous process migration involves both compilation and runtime systems.

3.1 Compilation System

In our design, users have to pass their source code to a *pre-compiler* to generate the migration-supported code. We define the pre-compiler as a source-to-source transformation software which performs program analysis and augmentation. Figure 3.1 illustrates basic steps for software development in the migration environment. The pre-compiler gives two output files, a map file (MAP) and a MODified file (MOD). The MAP file shows locations of poll-point, while the MOD file is an annotated source code for process migration. If the users do not like the selected poll-points, they can reconfigure the MAP file and let the precompiler generate a new MOD file. After the users satisfy with the poll-point selection, we assume that the MOD file is distributed to every possible destination computers of process migration. Then, it is passed to a native compiler to generate a migration-supported executable. At this step, the executable is also linked to a checkpointing and a message passing libraries.

3.2 Runtime System

The runtime system supports process management, process state transfer, and reliable message passing in process migration environments. It consists of the following software components:

1. **The MSRM library**

   The MSRM library contains novel mechanisms to transfer the memory state of a process in heterogeneous environments. During process migration, the augmented

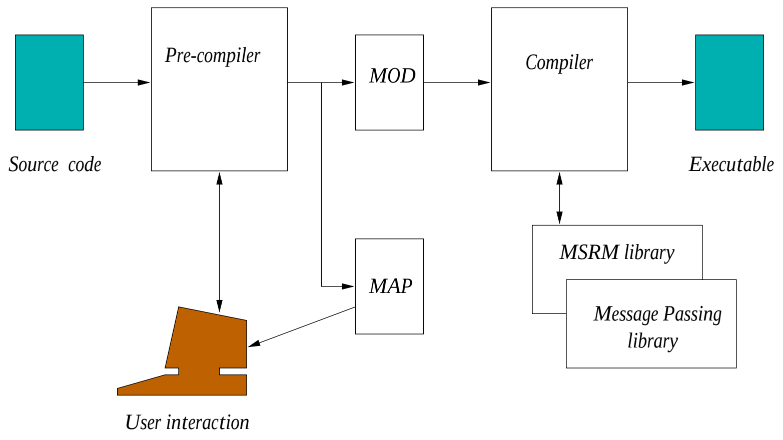


Figure 3.1: Software development for heterogeneous process migration.

code invoke routines in this library to collect data values of live variables at a migration point.

In programming languages with presence of pointers such as C, data collection and restoration are not trivial. In our solution, we invent the MSR model and its associate library routines to collect and restore complicate data structures in the process memory space. The collected state information is converted to a machine-independent format and then transmitted to be restored on a destination computer.

In the restoration process, a new process on the destination computer first transfers program execution to the migration point, converts the machine-independent state information to a machine-specific data format, and restores the data values (including their associated data structures) of the live variables before resuming execution. This library is linked to the migration-supported process during the executable generation as shown in Figure 3.1.

2. **Message passing library**

The message passing library contains routines to perform reliable data communication and process migration protocols. The reliable data communication protocol includes send and receive operations which guarantee correct message passing in process migration environments.

Due to unpredictable communication behaviors and process relocation in migration environment, the process migration protocol have to coordinate with the communication protocol to preserve semantics of point-to-point communication including no message lost and message ordering. The process migration protocol also controls the transfers of execution state and memory state and makes sure that new location information of the migrating process is known by communication peer processes after the relocation. The message passing library is linked to the migration-supported executable as shown in Figure 3.1.

3. **Virtual machine daemons**

   The daemons cooperate one another on top of a network of workstations and comprise a *virtual machine*. Figure 3.2 shows the virtual machine environment. The virtual machine provides process identification and process management functionalities including process creation, migration, and termination. We also use the virtual machine for carrying control messages and interrupt signals among processes in the environment. From the figure, the virtual machine functions as a middle software layer providing methods for schedulers, application processes, and other software components in the environment to communicate one another.

4. **Scheduler**

   The scheduler is a process or a number of processes that control environmental-wide resource utilization. Its functionalities include bookkeeping utilization information of resources and managing resource allocation in a distributed environment. From

Figure 3.2, the scheduler uses tools provided by the virtual machine to monitor and perform resource and process management.

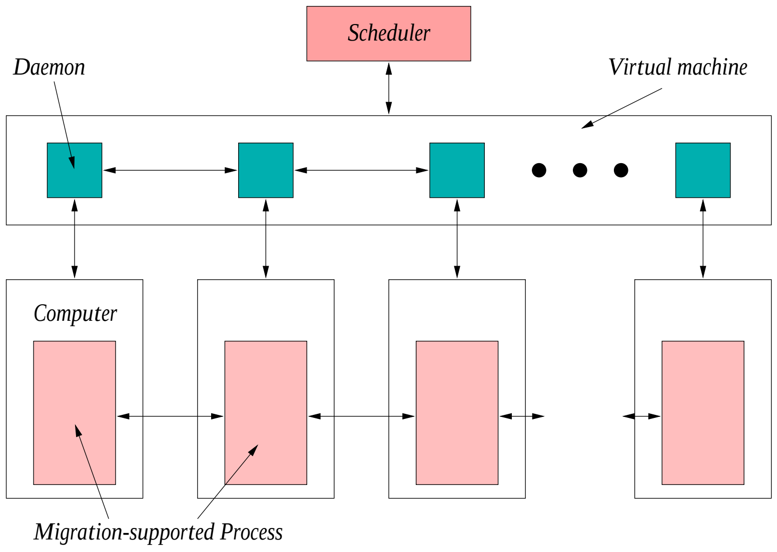


Figure 3.2: The virtual machine environment.

Giving an example process migration situation, when load imbalance occurs, process migration may be employed to solve the problem. To migrate a process, the scheduler determines a migrating process in a distributed environment and choose one of the idle or lightly loaded machines to be a destination computer. Then, the scheduler remotely invokes an executable which is generated from the same MOD file as that of the migrating process. In a heterogeneous environment, these executables exhibit identical functionalities since they are generated from the same source code. The remote invocation service is one of the tools provided by the virtual machine. Due to its specific purposes, we call the invocation for process migration as *process initialization*. Following the augmented macros in the MOD file, the initialized process wait for a connection from the migrating process.

Usually, the initialization is performed when the scheduler want to migrate a process. In our model, the process can also be initialized at anytime before the migration is needed. This situation is called *pre-initialization* which may help reduce process migration overheads because the executable is already loaded to memory of the destination computer by the time process migration is needed. The strategies to manage pre-initialization depend on the design of the scheduler. Although scheduling policies are not the focus of this work, pre-initialization is recommended.

After the initialization, the scheduler sends information of the initialized process to the migrating process. Then, the migrating process performs state collection and send its state information to the destination machine. Details of mechanisms to migrate the execution state, memory state, and communication state are to be discussed in the following chapters. Finally, the migrating process terminates, while the initialized process resumes computation on the destination machine.

Chapter 4
Pre-Compilation System

This chapter presents the design of the compilation system to support heterogeneous process migration. We define the pre-compiler as a source-to-source transformation system which perform program analysis and augmentation. The objective of the pre-compiler is to transform a source program written in high-level programming language such as C migratable in a heterogeneous environment. There are three major functionalities in the pre-compiler design: migration point analysis, data analysis, and insertion of migration macros based on our data transfer mechanism. Figure 4.1 shows structure of the pre-compilation system. Migration operations could be generated automatically by the pre-compiler. For efficiency and debugging purposes, however, user interfaces are also provided so that experienced users can customize migration operations in their applications.

4.1 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.

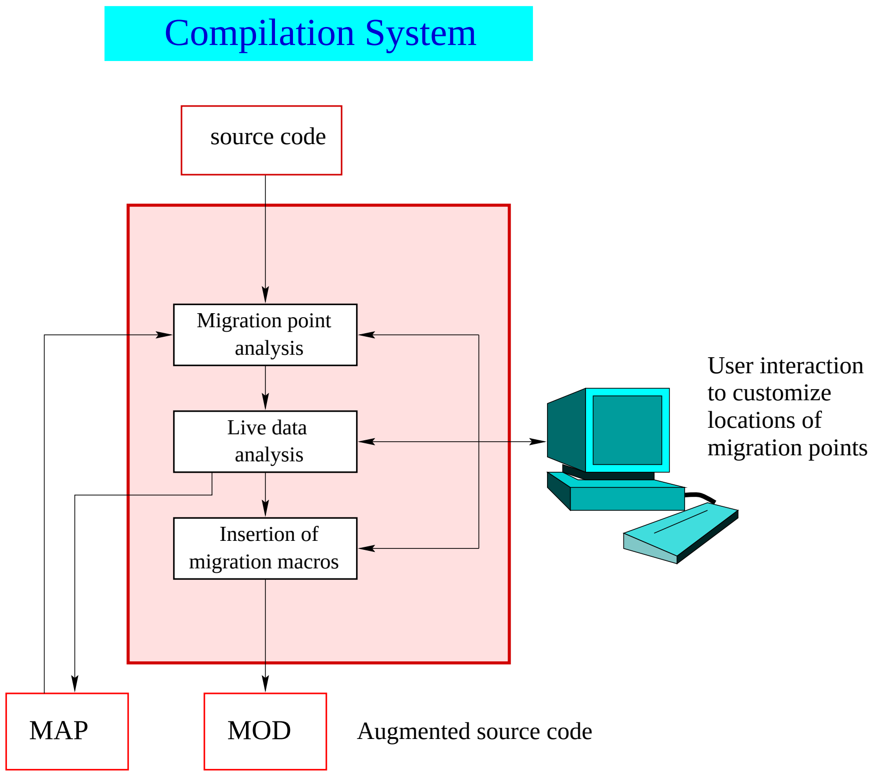


Figure 4.1: The pre-compiler for heterogeneous process migration.

The pre-compiler 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. The pre-compiler 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, the pre-compiler 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 kernel and shared system resources. Although the 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, the pre-compiler 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, the pre-compiler 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 the pre-compiler encounters a branch instruction, it assigns different counters to each branch. The pre-compiler 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, the pre-compiler 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, the pre-compiler accumulate the amount of works as a part of sequential code. Recent researches in Symbolic Compiler Analysis [13, 5] 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, the pre-compiler 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.

4.2 Data Analysis

The goal of data analysis is to minimize the data transfer time during process migration. In Figure 4.2, a migration point (Mp) and its data analysis are given. In this example the sets would be {a, b, x} and {x, b, c, y}, respectively. The intersection of these two sets, {x, b}, gives us the minimal set of data needed to be migrated at a migration point Mp.

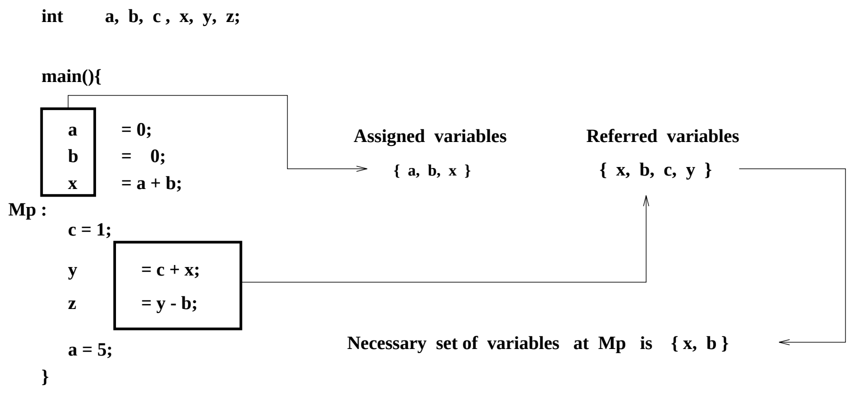


Figure 4.2: A simple example of live data analysis.

However, in real applications data analysis is much more complicate than in the given example. Complex user-defined data type and pointers are always used in the applications. Also, memory can be dynamically allocated during program execution. In dealing with these problem, we employ both compile-time and run-time systems in our solution.

At compile-time, the pre-compiler determines a set of live variables and their data types at each migration point. Then, it annotates function calls to a runtime library for data collection and restoration. We will discuss this runtime library in the next chapter.

4.2.1 Live Variable Analysis

The key idea of data analysis is to find a minimal set of variable for which their value need to be transfer during process migration. *Live variable analysis* can be applied for this purpose. To do so, the pre-compiler first generates a function call graph and a control flow graph for every function in the program. Then, it defines *def* and *use* sets of instructions and apply the following data flow equation to find a set of live variables at every migration point.

Supposed that B is a basic block, we define:

- $in[B]$ as the set of live variables at the point immediately before the basic block B, the migration point of our interest
- $out[B]$ is the set of live variables immediately at the end of the block B
- $def[B]$ is the set of variables that are assigned values before any uses of them in B, and
- $use[B]$ is the set of variables whose values may be used in B before any assignment to the variables.

According to algorithms given in [1, 16], the backward analysis on the data flow equation

$$in[B] = use[B] \cup (out[B] - def[B])$$

$$out[B] = \cup_{S \text{ a successor of } B} in[S]$$

can be applied for our purpose.

4.2.2 The Def and Use Sets

In the language such as C where the uses of pointers are common, we have to address the definition of def and use set. We classify variables in such language into two types: the *non-pointer* and *pointer* variables. For examples, the non-pointer variables are those whose reference are made to single memory storage throughout its lifetime. They are, for example, those declared at lines 1, 2, 3, 7, 10, and 11 in Table 4.1. On the other hand, the variables whose reference can dynamically change to different memory storages at run-time are pointer variables. Since the pointer stores the address of a memory storage, its address content can be changed during execution. Their examples are those at lines 4, 5, and 9 of the Table 4.1.

Table 4.1: Examples of C language statements and their *def* and *use* sets.

No.	Variables	Instructions (I)	def(I)	use(I)
1	int a, b, c;	$a = b + c;$	$\{a\}$	$\{b, c\}$
2	int d[10], f[10];	$d[i] = a + f[j];$	$\{d\}$	$\{i, j, a, f\}$
3	struct... g, h;	$g.x = h.y + 2;$	$\{g\}$	$\{h\}$
4	int *p;	$p = \&d[0] + 5;$	$\{p\}$	$\{d\}$
5	struct... *q;	$*(p + 1) = *(q- > x);$	$\{p\}$	$\{p, q\}$
6		$*(q- > x) = *(p + 1);$	$\{q\}$	$\{p, q\}$
7	int *r[10];	$r[0] = d;$	$\{r\}$	$\{d\}$
8		$*r[0] = d[1];$	$\{r\}$	$\{r, d\}$
9	int (*s)[10];	$s = t;$	$\{s\}$	$\{t\}$
10	int t[2][10];	$s[0][0] = a;$	$\{s\}$	$\{s, a\}$
11	int u[2][10];	$s = u;$	$\{s\}$	$\{u\}$
12		$s[0][0] = b;$	$\{s\}$	$\{s, b\}$

Each kind of variable has different definition to the *def* and *use* sets. We apply the following rules. For an assignment statement, if pointer dereferences occur on left-hand side of the operation, the variable name is defined in both the *def* and *use* sets. Otherwise, the variable name on the left-hand side of the operation is added to the *def* set only. In case of expression every variable name involved is assigned to the *use* set.

We show the application of those rules in the following examples. From Table 4.1, the assignment statement at line 1 involves non-pointer variable a, b, and c. The array and structure variables used at lines 2 and 3 are also non-pointer. Therefore, no pointer dereference would occur to the left-hand side of these example instructions. On the other hand, variables p and q declared at lines 4 and 5, respectively, are pointer variables. Due to pointer dereferencing on the left-hand side of the instruction at line 5, p is assigned to both the *def* and *use* sets. Similarly, q is assigned to both sets at line 6. At line 7, r is declared as an array of pointers. Although, there is an array (or subscript) dereference in this instruction, it is not pointer dereferencing. Thus, we assign r to the *def* set. On the other hand, the instruction at line 8 shows a different scenario. There is a pointer dereference occurred to one of the element of r. Because of the dereference, the pointer content on the left-habd side of the instruction must be evaluated before the assignment operation. Therefore, r is defined in both *def* and *use* sets. From line 9 to 12, s is declared as a pointer to an integer array of size 10. Since s is a pointer variable its address content could be assigned to various storages of the same type as shown in instructions at lines 9 and 11. Therefore, s is assigned to both the *def* and *use* sets of instructions in line 10 and 12 due to pointer dereferences.

In case of subroutine calls, we use parameter passing evaluation and interprocedural data-flow analysis to determine the *use* and *def* set. In case of parameter passing evaluation, we apply the following rules. If a memory address is passed as an actual parameter, we define the actual parameter variable in both *def* and *use* sets of the instruction. This rule covers the call-by-pointer and call-by-reference parameter passings [15]. On the other hand, if the value is not a memory address, the parameter variables is assigned to the put to the *use* set only.

Interprocedural data-flow analysis is used to determine a set of global variables whose value may be changed during the subroutine call. Well-established algorithms for this problem are given in [1, 40]. The results of the analysis are included to the *def* set of the subroutine call instruction. Similar analysis can also be used to obtain a set of all global variables used during a function call. The results are put to the *use* set.

4.3 Source Code Annotation

After defining migration points and their live variables, the pre-compiler annotates global variables and macros to source code to create a MOD file. These global variables and macros are reserved only for source code annotation and must be different from those defined by a user. The main purpose of the source code annotation is to implement a mechanism to transfer execution state and live data across machines.

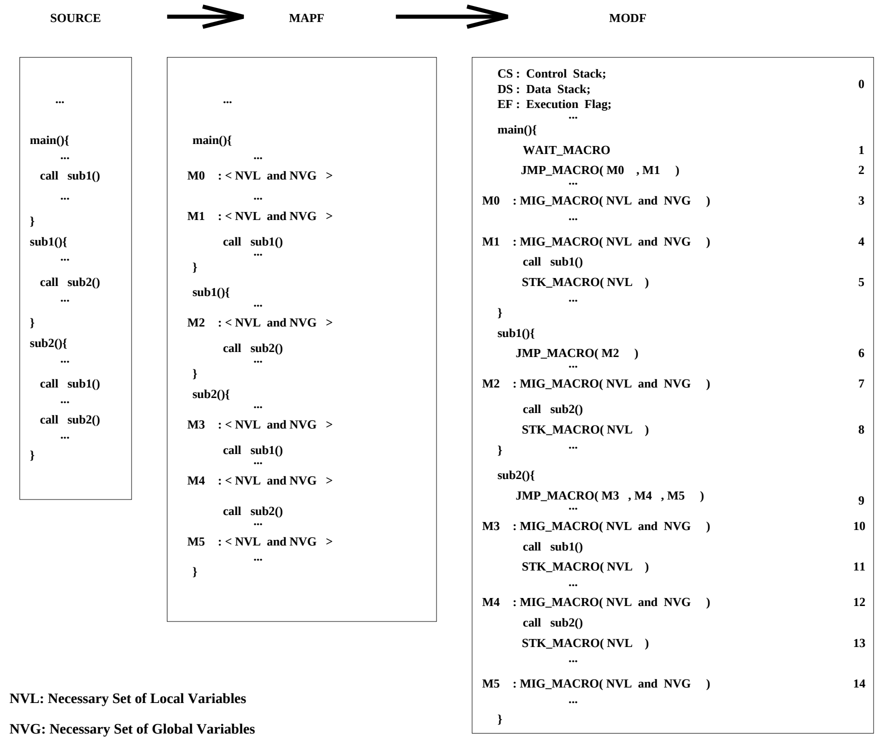
Data transfer mechanism is the cooperation of macros to manage the data collection, transmission, and restoration in a process migration. In our early work [7], we have introduced a data transfer mechanism, namely the *Stack-like Data Transfer (SDT)* mechanism, applying stack-based operations (i.e., push and pop) to collect and restore the execution state and memory state of a process. The SDT mechanism is discussed in Section 4.3.1.

However, in our current design, a novel technique called the *Buffered Data Transfer (BDT)* mechanism has been invented to accommodate efficient network process migration. The BDT mechanism allows the collection and restoration of the process state on the source and destination computers to be performed in parallel, while having the process state information transmitted between them in pipelining fashion. Detailed discussion of the BDT mechanism is given in Section 4.3.2.

4.3.1 Stack-like Data Transfer Mechanism

Under the SDT mechanism, Figure 4.3 shows example source code and its corresponding MAP and MOD files. After generating a MAP file, the pre-compiler inserts global variables and macros to generates a MOD file. The global variables include a Control Stack (CS), a Data Stack (DS), an Execution Flag (EF), and other variables such as those for data communication. The CS is used to keep track of function calls before process migration. During a migration, the name of the poll-point where the migration occurs and the names of poll-points associated to every function in a function calling sequence are pushed into the CS. Live data associated to every poll-point in the CS are stored in the DS.

The EF variable indicates execution status of a process. Its values are normal (NOR), restoring the process (RES), migrating the process (MIG), and collectingrestoring data in



```
SOURCE

    ...

main(){
    ...
  call  sub1()
    ...
}
sub1(){
    ...
  call  sub2()
    ...
}
sub2(){
    ...
  call  sub1()
    ...
  call  sub2()
    ...
}
```

```
MAPF

      ...

  main(){
        ...
M0   : < NVL  and NVG  >
        ...
M1   : < NVL  and NVG  >
      call  sub1()
        ...
  }
  sub1(){
        ...
M2   : < NVL  and NVG  >
      call  sub2()
        ...
  }
  sub2(){
        ...
M3   : < NVL  and NVG  >
      call  sub1()
        ...
M4   : < NVL  and NVG  >
      call  sub2()
        ...
M5   : < NVL  and NVG  >
        ...
  }
```

```
MODF

  CS :  Control  Stack;                          0
  DS :  Data  Stack;
  EF :  Execution  Flag;
          ...
  main(){
      WAIT_MACRO                                 1
      JMP_MACRO( M0   , M1   )                   2
          ...
M0   : MIG_MACRO( NVL  and  NVG   )              3
          ...
M1   : MIG_MACRO( NVL  and  NVG   )              4
      call  sub1()
      STK_MACRO( NVL   )                         5
          ...
  }
  sub1(){
     JMP_MACRO( M2   )                           6
          ...
M2   : MIG_MACRO( NVL  and  NVG   )              7
      call  sub2()
      STK_MACRO( NVL   )                         8
          ...
  }
  sub2(){
     JMP_MACRO( M3   , M4   , M5   )             9
          ...
M3   : MIG_MACRO( NVL  and  NVG   )              10
      call  sub1()
      STK_MACRO( NVL   )                         11
          ...
M4   : MIG_MACRO( NVL  and  NVG   )              12
      call  sub2()
      STK_MACRO( NVL   )                         13
          ...
M5   : MIG_MACRO( NVL  and  NVG   )              14
          ...
  }
```

NVL: Necessary Set of Local Variables

NVG: Necessary Set of Global Variables

Figure 4.3: An example source code annotation under SDT mechanism.

the stack segment (STK). NOR is a default indicating normal execution. RES is set for the initialized process upon creation. It tell the initialized process to wait for the connection from the migrating process. MIG, on the other hand, notifies the migrating process to start the migration at a migration point. Finally, STK is used for collection and restoration of data in the stack segment in case a nested function call is made during process migration.

After inserting variables, the pre-compiler augments macros at various locations in the source code. During a process migration, these macros collect CS and DS stacks, transfer data across machines, and restore data to the appropriate variables in the destination process. These macros are WAIT_MACRO, JMP_MACRO, MIG_MACRO, and STK_MACRO as shown in Figure 4.4. The pre-compiler puts WAIT_MACRO at the beginning of main function to wait for the connection and the contents of CS and DS from the migrating process. JMP_MACRO is put right after WAIT_MACRO in the main function and at the

WAIT_MACRO

```
- get EF value from Scheduler;
if (EF == RES){
    - accept TCP connection and
      receive CS and DS from the
      migrating process;
}
```

JMP_MACRO

```
if (EF == RES){
    - pop Mp name from CS;
    - Jump to that Label in
       the function;
}
```

MIG_MACRO

```
if (EF == MIG){
    - push current Mp name to CS;
    - push NV to DS;
    - if ( the Mp label is in main() ){
            - request TCP connection and
              send CS and DS to the new
              process.
            - exit() the program;
      }
      else{
            - set EF to STK;
            - return() to the caller function;
      }
}
else if( EF == RES ){
    - pop NV set;
    - if( size of CS is zero ){
            - set EF from RES to NOR;
      }
}
```

STK_MACRO

```
if ( EF == STK){
    - push current Mp name to CS;
    - push NV to DS;
    - if ( the Mp label is in main() ){
        - request TCP connection and
          send CS and DS to the new
          process;
        - exit() the program;
      }
      else{
        - return() to the caller function;
      }
}
```

*** Note that NV stands for Necessary Variable set.

Figure 4.4: Macros for the SDT mechanism.

beginning of the body of other functions. MIG_MACRO is inserted at every migration point. In case the migration point was inserted before the function call to keep track of subroutine calling instructions, a STK_MACRO is inserted right after the function call associated to those migration points.

At a migration event, the scheduler initializes a process on an idle machine. The EF flag of the new process is set to RES. The new process waits for transmission of process state from the migrating process. Then, the scheduler sends a signal to the migrating process to start process migration.

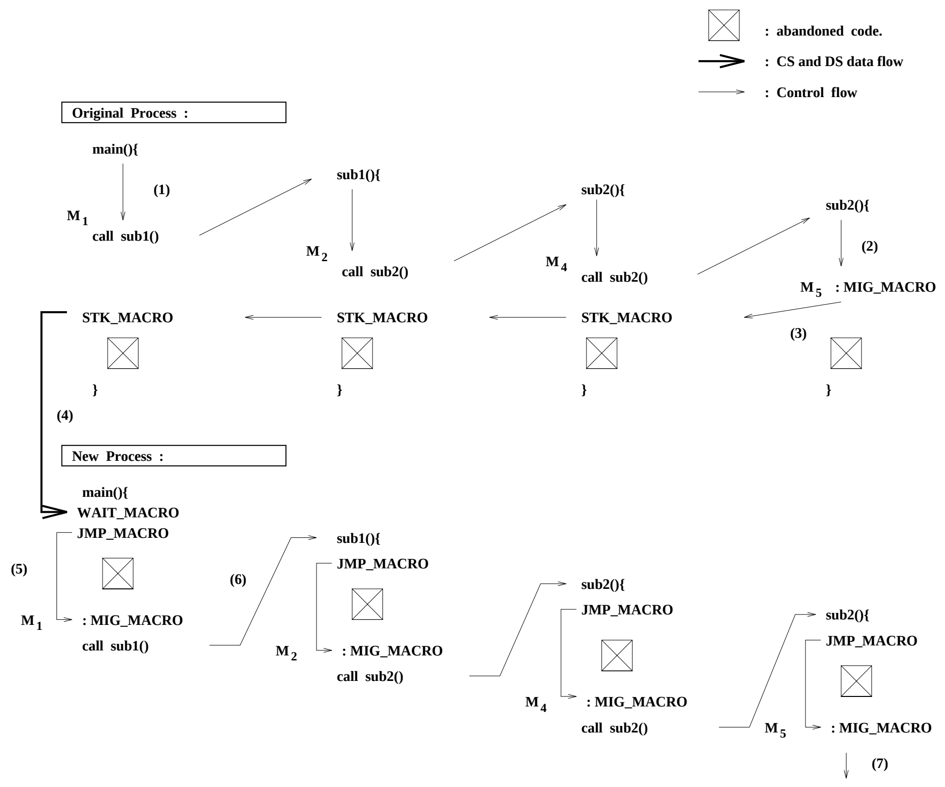


Figure 4.5: An example process migration.

To illustrate the migration operation, we refer to the MOD file in Figure 4.3 and consider a process migration situation in Figure 4.5. At (1) the process starts its execution from main() and then invokes sub1(), sub2(), and sub2() again. While executing instructions between M_4 and M_5, the scheduler signals the migrating process to change the EF flag from NOR to MIG (2). The process continues its execution until reaching M_5. The execution enters MIG_MACRO there. Because the EF flag is MIG, the process push the label M_5 to the CS stack and push live data at M_5 to the DS stack. Then, it changes the EF flag to STK and abandons the rest of sub2() by returning to its caller function (3). At the caller function, the process enters STK_MACRO and saves M_4 and associate live data to CS and DS, respectively. The execution then returns to sub1(), the caller function. The process perform similarly until the execution reaches main() function. In main(), instructions in STK_MACRO are executed. At this point, after the label M_1 and its live data are collected to CS and DS, the information in both stacks are transmitted to the destination machine (4).

On the destination computer, the new process first enter WAIT_MACRO to wait for process state transfer. After receiving the CS and DS contents, the new process enter JMP_MACRO. Then, the label name (M_1) is popped from the CS stack. The execution flow is then transferred to the location of M_1 by executing a *goto* statement (5). At the migration point M_1, the process enters MIG_MACRO with the EF value of RES. Then the process pops appropriate values and assigns them to live variables at the migration point location. After that, the process calls sub1() as in (6). In sub1(), the instruction in JMP_MACRO pop M_2 from the CS stack, and then jump to the M_2 label. At M_2, the data values of live variables are popped from DS. Then, the process calls sub2(). The JMP_MACRO in sub2() transfers the execution to M_4. After that, the process pop DS stack and then call sub2() again. In the new sub2(), the execution flow pop the migration point from CS, jump to M_5, and then pop values of live variables from DS. Finally, the EF flag is set to NOR and normal execution resumes(7).

4.3.2 Buffer Data Transfer Mechanism

In our current work, we have developed a more efficient mechanism, namely the *Buffer Data Transfer (BDT)* mechanism, whose design is better supportive to the data collection and restoration software in a network environment. The BDT mechanism has an advantage over the SDT mechanism in that its design allows the data collection, transmission, and restoration operations to be overlapped during process migration. With a small number of modifications to the macros used in the SDT mechanism, the BDT mechanism can be implemented under the program analysis framework presented in [8, 7]. Detailed program analysis and migration macros are given later in this section.

Basic Idea

First, we want to show basic ideas of how the BDT mechanism is employed for data collection and restoration. Figure 4.6 shows an example of the BDT mechanism when a sequence of function calls starting from function `main()` to `fi()` are made prior to process migration. The following functionalities of the BDT mechanism are illustrated in areas 1, 2, and 3 in Figure 4.6, respectively.

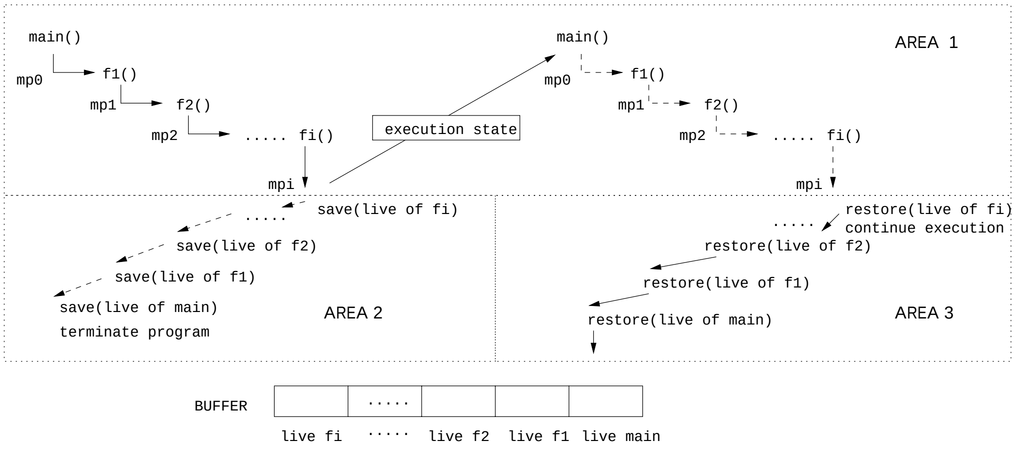


Figure 4.6: An overview example of the BDT mechanism.

1. When process migration occurs at the migration point `mpi`, the BDT mechanism will send information about the execution state of the migrating process to a new process on the destination machine of process migration. The information will be used by the new process to create a sequence of function calls identical to that of the migrating process and jump to the migration point `mpi`. Then, the BDT mechanism of the new process will initiate data restoration operation to wait for live data of the function `fi()` that will be transmitted from the migrating process.
2. After sending the information about the execution state to the new process, the migrating process will start saving live data of the innermost function in the calling sequence to a buffer and return to its caller function. The same operation will be performed until the main function is reached. The data collection operation will give us the result buffer as shown in Figure 4.6. Before terminating the migrating process the BDT mechanism will send the stream of information stored in the buffer to the new process on the destination machine.
3. At the destination machine of process migration, the new process will read and restore live data from the transmitted information stream. The BDT mechanism will restore live data of the function `fi()` and let the function execute until the end of its execution. After the function return to its caller, the BDT mechanism will again read the content of the information stream and restore live data of the caller function. After the restoration, the BDT mechanism will again let the function work until the end of its execution. The BDT mechanism will manage the order of data restoration in this manner until the main function is reached and its live data are restored. After that, the new process will continue execution till its termination or another process migration is met.

In terms of implementation, the BDT mechanism allows performance improvement for both performance-driven and fault-driven process migration. In case of performance-driven where the buffered information stream is sent via network, a portion of saved live data of the migrating process can be transmitted to the new process first. The new process can restore its memory space simultaneously while the migrating process is saving the next portion of data from its memory space. In other words, the operations illustrated in areas 2 and 3 in Figure 4.6 can be performed simultaneously. The high potential of the BDT mechanism in the aspect of concurrence can be realized by new technologies of network interfaces and protocols such as Active Messages [38] and Fast Messages [30] that allow efficient overlapping of communication and computation. For fault-driven process migration, assuming that the source and destination machines of process migration are sharing the same network file system, the benefit of the BDT mechanism is that the new process on the destination machine can start reading the checkpoint file and restore its memory space while the migrating process writes information to the checkpoint file during the data collection operation. The BDT mechanism seems to have real potential. In this study, we focus on basic data collection and restoration mechanisms and their data structures. Overlapping of data collection, transmission, and restoration is not performed in our implementation.

Detailed Description

We illustrate, in Figure 4.7, the annotated variables and macros of the BDT and their functions. Figure 4.7 shows examples of a source file, and its MAP and MOD files under the BDT mechanism. In the Figure, G denotes the global declaration segment of the source code. Px where $x \in \{0, 1, 2\}$ is a set of formal parameters of a function. Likewise, Lx where $x \in \{0, 1, 2\}$ represents the local variable declaration section of a function. In the body of

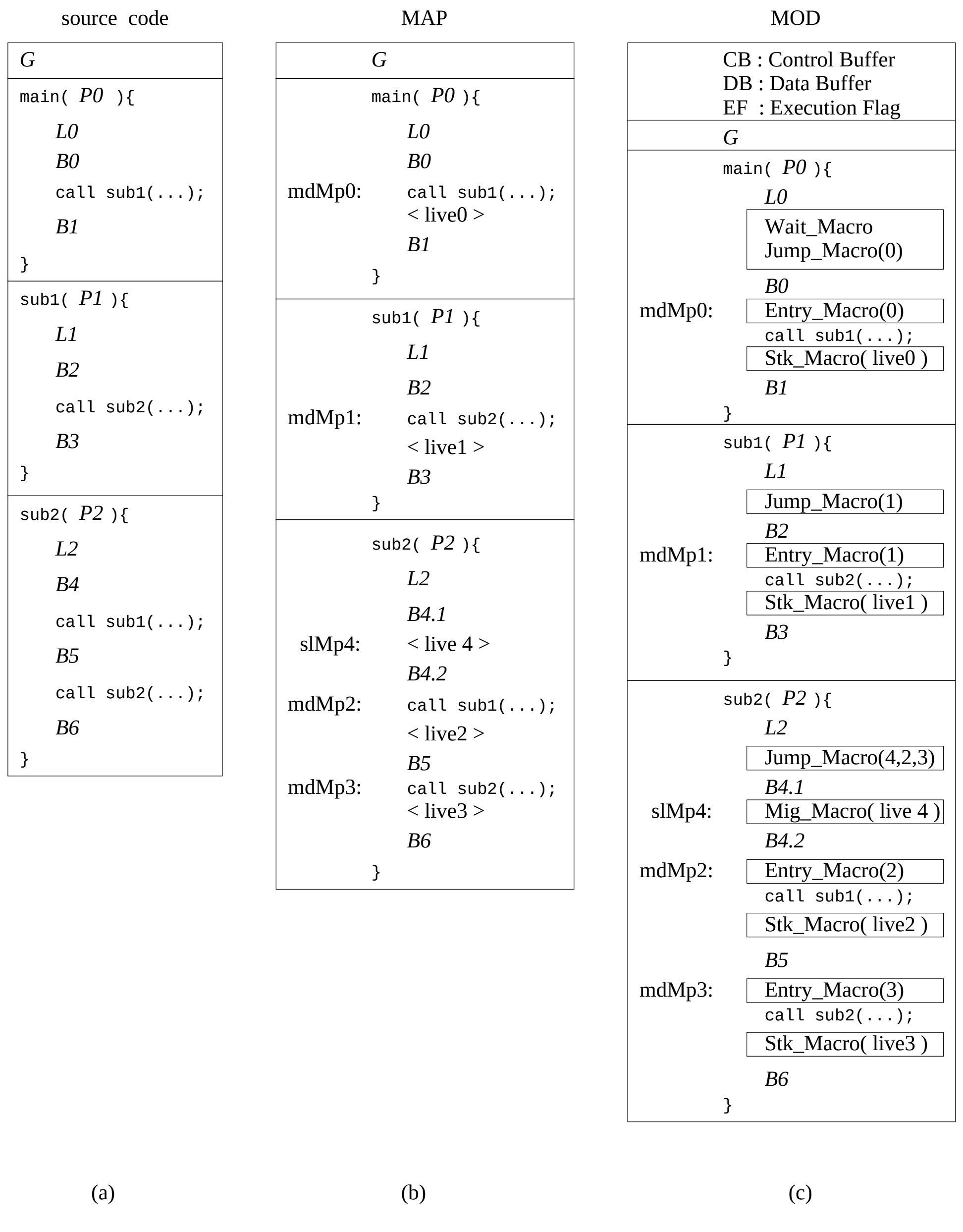


Figure 4.7: An example of source code annotation for the BDT mechanism.

the functions, Bx where $x \in \{0, 1, \cdot, 6\}$ denotes a sequence of instructions. Note that the sequence of instructions $B4$ can be split into two parts, $B4.1$ and $B4.2$.

The pre-compiler employs the migration point and live variable analysis to the source code. The MAP file is generated as shown in Figure 4.7(b). In this example, we assume that the pre-compiler has selected the migration point *slMp4* in the first phase of the migration point analysis. Then, the mandatory migration points *mdMp0*, *mdMp1*, *mdMp2*, and *mdMp3* are inserted in the second phase. The insertion of a mandatory migration point is different from the selected one in that the migration point is inserted before its corresponding subroutine call but live variable analysis is performed at the point immediately after the call. For instance, the migration point *mdMp0* is inserted before the subroutine call to `sub2(...)` but its corresponding set of live variables *live0* is defined after the call. For the selected migration point, live variable analysis is performed at the the insertion location. In the Figure, the live variable set *live4* is defined at the insertion of *slMp4*.

To generate a MOD file, global variables and macros are annotated to the source code as shown in Figure 4.7(c). The global variables include a Control Buffer (CB), a Data Buffer (DB), an Execution Flag (EF), and other variables such as those for reliable data communications at the top of the file. The CB keep track of function calls before the migration. During a migration, a name of the migration point where the migration occurs and names of migration points associated to every function called before the migration will be added into the buffer. We also need to maintain the DB to store live data at the point where migration occurs as well as those of the migration points stored in CB.

Every macro annotated to the source code, as illustrated in Figure 4.9 and 4.8, works according to value of the EF variable. The EF represents the execution status of the process at a certain point of program execution. Its alternation is performed by signaling between the process and the scheduler and by operations inside the BDT macros. Eight types of EF values, normal (NOR), waiting (WAIT), migration (MIG), migration of activation record stack (STK_MIG), jump (JUMP), restoration (RES), and restoration of activation record stack (STK_RES) are defined.

1. The NOR flag, the default value, represents normal execution of the process.
2. The WAIT flag is assigned to the EF of the initialized process by the scheduler to wait for a communication connection from the migrating process.
3. The MIG flag tells the process to start its migration at the nearest coming selected migration point.
4. In case nested function calls occur at a migration, the EF is set to STK_MIG during the data collection operation of the caller functions.
5. The JUMP flag is set in the initialized process after CB and DB are transmitted. It causes the process to transfer its execution to a particular migration point using a sequence of `goto` statements.
6. The RES flag is set when the execution of the initialized process is transferred to the migration point that cause process migration. This triggers the restoration of live data of the function that the migration point belong.
7. The STK_RES is set in the initialized process when live data of the caller functions is restored in presence of nested function calls during the migration.

After inserting variables, the pre-compiler inserts migration macros at various locations over a source program. During a process migration, these macros will collect the CB and DB, transfer them across machines, and restore them to appropriate variables in the destination process.

Figures 4.9 and 4.8 show macros used by the BDT mechanism. They are Entry_Macro, Mig_Macro, Stk_Macro, Wait_Macro, and Jump_Macro. The pre-compiler inserts Wait_Macro at the beginning of the main function to wait for the connection and the contents of CB and DB from the migrating process. Jump_Macro is put right after Wait_Macro in the main function and at the beginning of the body of other functions. Mig_Macro is inserted at every migration point. In case of the mandatory migration point, the Entry_Macro is inserted immeduately before the function call associated to the migration point and the Stk_Macro is inserted right after the function call.

```
Entry_Macro( Mp )

1.  Put Mp to CB based on function call sequence.
```

```
Stk_Macro( Live Data )

1.  if ( EF = STK_MIG ){
2.      Save Live Data in DB.
3.      if ( The Macro is in main function ){
4.          Request connection from the initialized process.
5.          Write CB and DB to the communication link.
6.          Exit  the  program.
        }
7.      else   Return  to the caller function.
    }
8.  else if ( EF = STK_RES ){
9.      if ( Local_Restore_flag = FALSE ){
10.         Restore Live Data  from DB.
11.         Set Local_Restore_flag to TRUE.
12.         if ( size of  CB is  zero )
13.             Set  EF  to  NOR.
        }
    }
```

```
Mig_Macro( Live Data )

1.  Put Mp to CB based on function call sequence.
2.  if ( EF = MIG ){
3.      Save Live Data in DB.
4.      if ( The Macro is in main function ){
5.          Request connection from the initialized process.
6.          Write CB and DB to the communication link.
7.          Exit  the  program.
        }
8.      else{
9.          Set  EF  to STK_MIG.
10.         Return  to the caller function.
        }
    }
11. else if ( EF = RES ){
12.     if ( Local_Restore_flag = FALSE ){
13.         Restore Live Data  from DB.
14.         Set Local_Restore_flag to TRUE.
15.         if ( size of  CB is  zero )
16.             Set  EF  to  NOR.
17.         else  Set  EF to  STK_RES.
        }
    }
```

Figure 4.8: The Entry_Macro, Mig_Macro, and Stk_Macro macros.

An Illustrative Example of the BDT Mechanism

To better understand the BDT mechanism, we give an example circumstance of a process migration. The annotated code in Figure 4.7 are used in this illustration.

At a migration event, the scheduler initialize a process on an idle machine. The EF flag of the new process is set to WAIT. The initialized process waits for a connection from the migrating process. Then, the scheduler send a signal to the migrating process to start migration operations.

Wait_Macro

```
1.  Get  the EF value from the Scheduler.
2.  if ( EF = WAIT ){
3.      Accept connection from the migrating process.
4.      Read CB and DB from the communication link.
5.      Set  Call_Depth to  length of CB.
6.      Set  Call_Count to  zero.
7.      Set  EF to JUMP.
    }
```

Jump_Macro(Mp ...)

```
1.  if ( EF =  JUMP ){
2.  Set  Local_Restore_flag to FALSE.
3.      Increase Call_Count by 1.
4.      if ( Call_Count = Call_Depth )
5.          Set  EF to RES.
6.      else  if ( Call_Count  <  Call_Depth )
7.          Set  EF  to JUMP.
8.      Extract  the front  Mp from  CB.
9.      Jump to that Mp.
    }
```

Figure 4.9: The Wait_Macro and Jump_Macro macros.

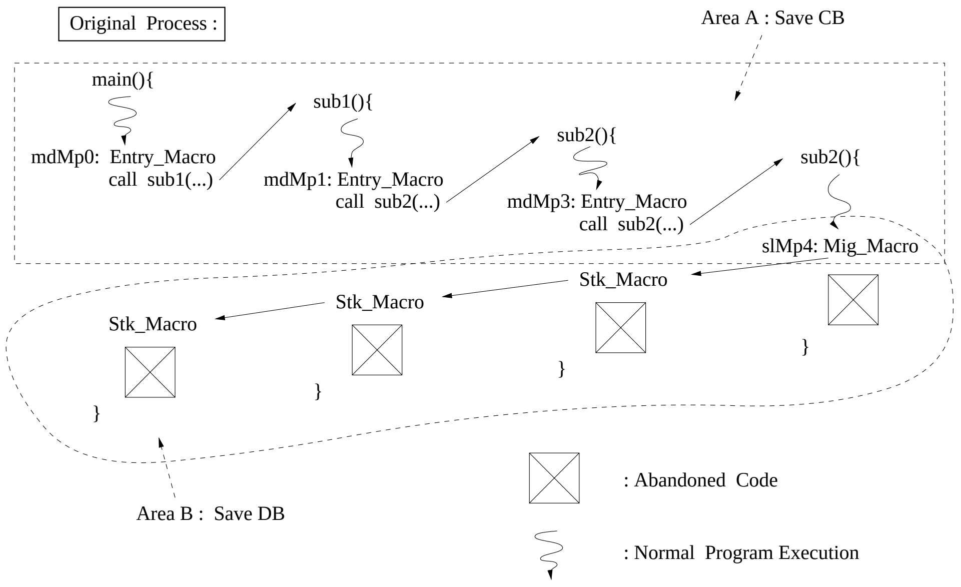


Figure 4.10: An example BDT mechanism.

Figure 4.10 shows the BDT mechanism on the migrating process. In the *Area A* of the Figure, the process execute the main function and enter subroutines `sub1(...)`, `sub2(...)`, and `sub2(...)` respectively. At each entrance the Entry_Macro associated to those subroutine call instructions are performed. The macro put identification of each mandatory migration points for each function call to the CB. While executing the innermost function `sub2(...)`, the original process get a migrating signal from the scheduler. The signal handler change EF to MIG meaning the embark of process migration upon reaching the nearest selected migration point. In our example, the migration point `slMp4` is reached and the BDT operations in the Mig_Macro are performed. In the Mig_Macro, since the EF is MIG, live data of `sub2(...)` at `slMp4` is saved to the DB. Because the execution is not in the main function, the EF is set to STK_MIG and the execution is returned to the caller function.

In the Area B of Figure 4.10, the execution flows from the function `sub2(...)` to its caller and enter the Stk_Macro. According to the functionality of the macro, live data of the caller function is saved and execution is returned to a caller function in the calling sequence. This same operations are performed until the main function is reached. A connection request is sent to the initialized process. The contents of CB and DB are sent to the initialized process once the communication link is established. Then, the process is free to terminate.

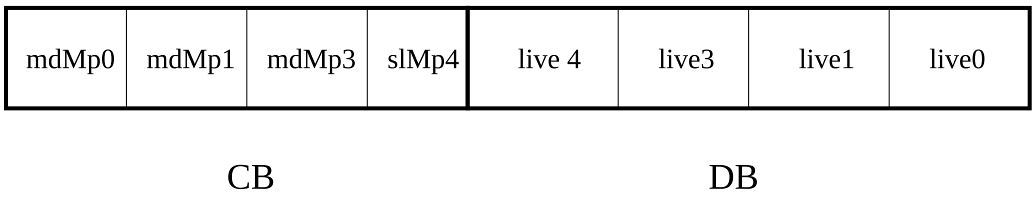


Figure 4.11: The output of the BDT mechanism.

The output of the BDT operation on the migrating process is given in Figure 4.11. This output is sent to the new process. It stores contents of CB following by that of DB. The CB contains a sequence of identification of migration points representing a sequence of function calls at a migration event. The CB is a production of the subsequent calling of the Entry_Macros in the migrating process. On the other hand, the content of the DB is the result of the consecutive performance of the Stk_Macro.

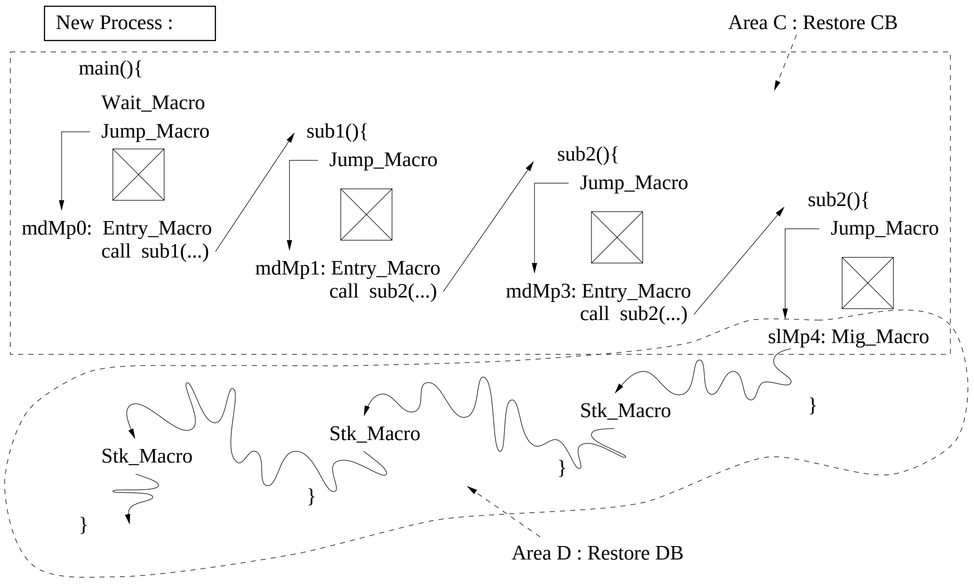


Figure 4.12: The BDT mechanism on the initialized process.

Figure 4.12 displays the BDT restoration operation on the initialized process. Note that the EF flag of the initialized process is initially set to WAIT, waiting for a connection from a migrating process. As described in the Wait_Macro in Figure 4.9, upon the acceptance of connection the CB and DB are read from the established communication. The EF flag is set to JUMP. The Area C in Figure 4.12 shows the restoration of execution state following the content of CB. The Jump_Macro keeps jumping to the migration points stored in the CB. During the jumps, the EF is set to JUMP. As illustrated in the Figure, consecutive operations of the Jump_Macro create a sequence of function calls similar to that previously occurred in the migrating process. At the last jump to the migration point `slMp4` in the innermost function `sub2(...)`, the EF is set to RES. As the result, the execution enter the Mig_Macro where the live data at the migration point `slMp4` is restored and the EF is set to STK_RES.

The Area D in Figure 4.12 illustrates the BDT operation to restore live data of the process. After exiting the Mig_Macro, we let the process continue its execution on the instruction immediately after the migration point. After returning from `sub2(...)`, the execution flow into Stk_Macro and restore live data of the caller function. Likewise, the instruction right after the macro is executed normally until the function finish. The restoration sequence goes on like this until live data of the main function is restored. Then, the EF is set to NOR.

In case of multiple process migration where the new process need to be migrated again while the EF is STK_RES, the BDT model can also accommodate the circumstance efficiently. Because of the design of the Entry_Macro to keep track of CB during execution, old and new contents of CB and DB can be manipulated for later process migration with shorter time.

Chapter 5
Memory Space Representation

In this chapter, we present a new methodology of data collection and restoration, enabling 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 costs of process migration by transferring only live data (i.e. 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 C language. It is an important step toward the search for a general solution to network process migration. This study may also benefit to strengthening 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 is migration-safe, *i.e.*, features which theoretically can be carried across a network
2. Choose a methodology to perform heterogeneous process migration. We define a process which can be migrated using the chosen methodology as the "migratable" process
3. Develop mechanisms to migrate the "migratable" process reliably and efficiently

Smith and Hutchinson [32] 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. Although there exist a few approaches for heterogeneous process migration, we have invented a new methodology in which procedures and data structures are given for transforming a high-level program into a migratable format via a precompiler. The transformation process includes migration-point analysis, data analysis, and the insertion of migration macros, etc. [8, 7]. The basic ideas of this methodology are discussed in Section 5.1.

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

1. Recognize the complex data structures of a migrating process for heterogeneous process migration
2. Encode the data structures into a machine-independent format
3. Transmit the encoded information to a new process on the destination machine
4. Decode the transmitted information stream and rebuild the data structures in the memory space of the new process on the destination machine

As the results, we have designed and implement 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 run-time library is developed to support process migration of migration-safe C code in a heterogeneous environment. The prototype and experimental measurements will be presented in Chapter 7.

This chapter is organized as follows. Section 5.1 gives discussions on process migration environment including the overview of mechanisms for execution and memory state transfer. Section 5.2 discusses a logical model representing data elements in process memory space and operations associated to the memory model. In Section 5.4, an illustrative example is given to describe the data collection and restoration mechanisms.

5.1 Overview

In our design, a program must be transformed into a "migratable" format. As introduced in the previous chapter, 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. Everytime when the process execution reaches the poll-point, the macro will first 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. The selection of poll-points as well as the macro insertion are performed automatically by a source-to-source transformation software (a pre-compiler). Users can also select their preferred poll-points if they know suitable migration locations in their source codes.

In 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 execution state and live data 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 state and live data, and resumes execution from the point where process migration occurred.

5.1.1 Data Collection and Restoration Sequence

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 calls are made. The collection and restoration of variables in each function are the starting points of the collection and restoration of the overall complex data structures of the process, which will be discussed in the next section. A migration point can be inside a nested function calls or a recursion during a migration situation. Based on the BDT mechanism, we collect and restore live data of the innermost function first and those of its callers subsequently. For example, a function call sequence `main` $\rightarrow$ `f1` $\rightarrow$ `f2` means the function `main` calls `f1` and `f1` calls `f2`. Figure 5.1 shows the example process migration from a source to a destination computers. 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 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 return 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 the live data for the innermost function and subse-

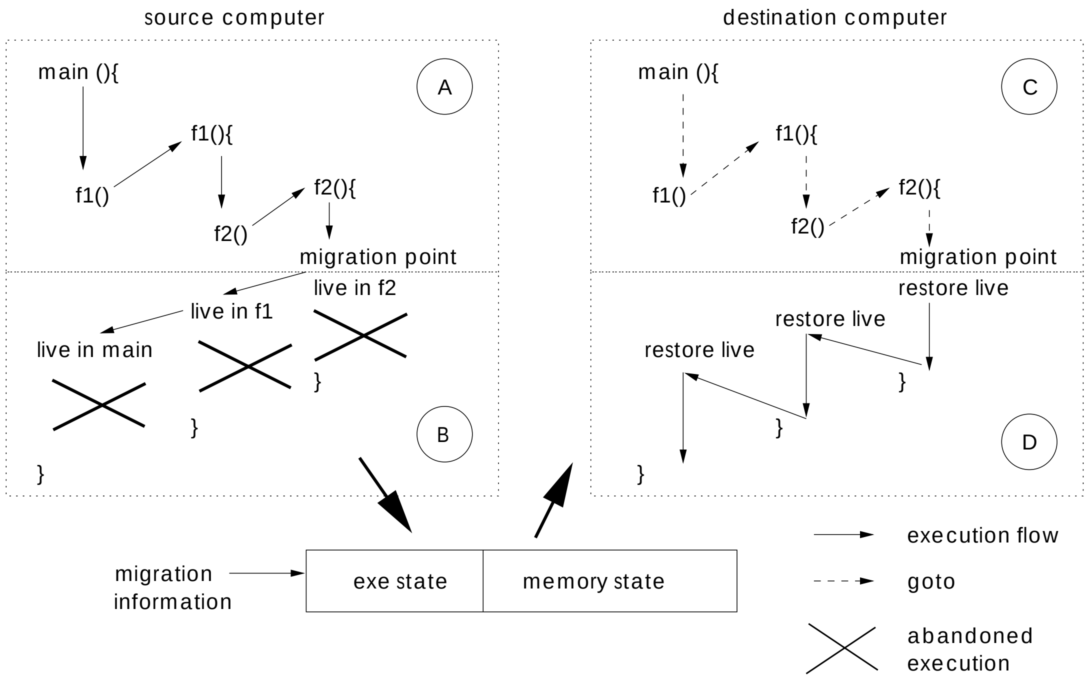


Figure 5.1: An example process migration.

quently for its callers after the innermost function terminates. From the `main` → `f1` → `f2` example, live data of `f2` will be restored first, following by `f1`, and then `main`. Note that after restoring live data on each function, the process continues normal execution until the function returns to its caller. The restoration of live data on the caller function is then performed as depicted in Figure 5.1.

To collect and restore live data, special purpose interfaces are applied to collect and restore values of live variables. Live variables at every poll-point are defined based on live variable analysis performed by the pre-compiler. We have been 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 such a way that live variables at each poll-point are collected and restored in the same order.

5.1.2 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 parts 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. Pointer is a memory address. It is considered as a primitive data type which allow referential relationships to be created among memory blocks.

Despite the common uses of pointers and their well-known benefits, the referential capability of pointer posts 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 [11] for data transmission, or they can be converted directly from source to 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 others. Moreover, pointers could represent referential relationship among memory blocks crossing different memory segments (e.g. from memory blocks in stack segment to 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 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 5.1. First, during normal execution, the migrating process bookkeeping the memory blocks' properties in special data structures. This data structures work as a mapping table between the conceptual MSR model and physical representation of the process memory space. They also give a logical addressing mechanism which allows the memory blocks to be machine-independently accessed. Second, during process migration, operations in migration macros invoke `Save_pointer` and `Save_variable` routines to collect data from live variables. The routines scan the logical memory model to collect 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 rebuild the mapping data structures while executing series of `goto`'s to resume the execution state. Finally, data restoration operations are performed by invoking `Restore_pointer` and `Restore_variable` routines to extract memory blocks' contents and the attached logical address out of the transmitted migration information. Then, the data contents are assigned to 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 mapping table data structures.

5.2 Memory Space Representation

This section describes the logical model, the Memory Space Representation (MSR), and its associated operations for data collection and restoration for process migration in 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 *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 one of them contains a pointer, which is an address that refer to a memory location of any memory block node in V.

We first describe properties of memory block nodes in the MSR graph. One of the most important properties of the memory block node is its data type. Based on the type information, we construct the *Type Information (TI)* table containing necessary information including functions to save and restore the memory block nodes. Then, we present special data structures, namely *MSR Lookup Table (MSRLT)* data structures, to keep track of memory block nodes of the MSR graph and to generate the memory blocks' logical addresses. The MSRLT data structures surve as the mapping table between the physical and logical memory addresses of a process on each computing platform. After that, we discuss the representation of pointers in a machine-independent format. And finally, we illustrate the data collection and restoration operations with an example.

5.2.1 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. Every object element in a memory block must have the same type. Each memory block is represented by a vertex v in the MSR graph. The following terminologies are used in our study.

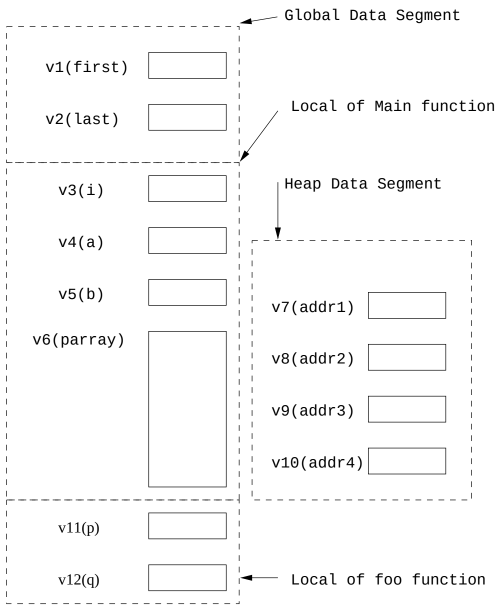
- $head(v)$: the starting address of the memory block v
- $type(v)$: type of the object stored in the memory block
- $elem(v)$: Number of objects of type $type(v)$ in the memory block

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 bravity, we use an example to illustrate the MSR graph concept.

Given a sample program in Figure 5.2(a), Figure 5.2(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`, assuming 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 5.2(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. 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.

```
 1: struct node {
 2:    float data;
 3:    struct node * link;
 4: };
 5: struct node *first, *last;
 6: main()
    {
 7:    int i;
 8:    int a, *b;
 9:    struct node *parray[10];
10:    a = 1;
11:    b = &a ;
12:    for ( i = 0; i < 10; i++ ){
13:       foo ( parray + i, &b );
14:       first = parray[0];
15:       last  = parray[i];
16:       first->link = last;
17:       if( i > 0 )
18:          parray[i]->link = parray[i-1];
       }
    }
19: foo( struct node **p, int **q ){
20:    *p = (struct node *)
          malloc ( sizeof( struct node ) );
21:    (*p)->data = 10.0 ;
22:    (**q) ++ ;
    }
```

(a)



(b)

Figure 5.2: An example program and its memory blocks.

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 case 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 5.2, we can define $Gm = \{v_1, V_2\}$, $Hm = \{v_7, v_8, v_9, v_{10}\}$, $Lm_{main} = \{v_3, v_4, v_5, v_6\}$, and $Lm_{foo} = \{v_{11}, v_{12}\}$.

Figure 5.3 shows the MSR graph representing a snapshot of memory space of the sample program in Figure 5.2(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 representation of pointer will be discussed in Section 5.2.4.

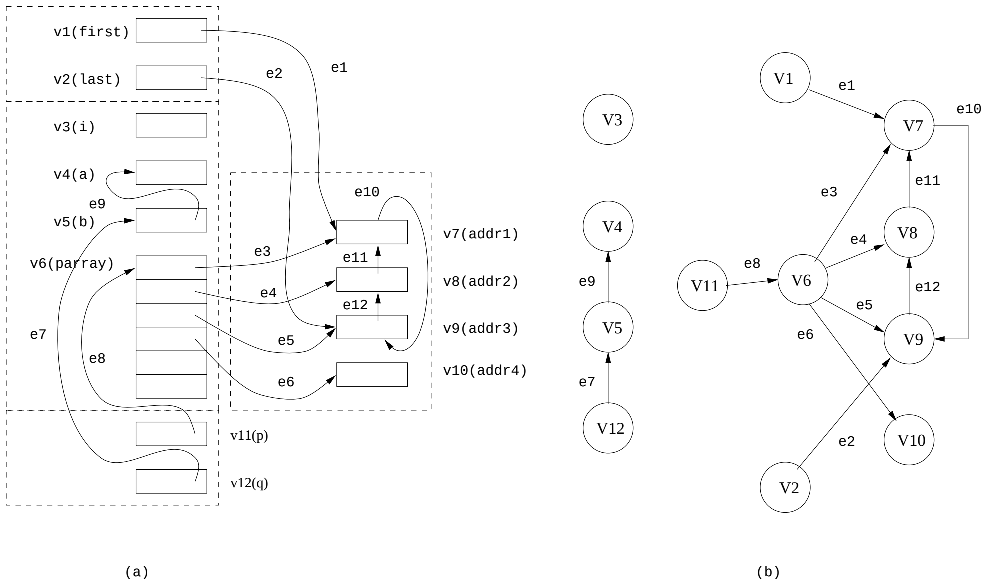


Figure 5.3: An example of the MSR graph.

5.2.2 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 data collection and restoration, additional information and operations have to be provided to recognize and manipulate data value of memory blocks.

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 to store information of every data type. It also includes a number of type-specific functions to encode and decode 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.** Size in bytes of an object of a particular type. In case of array, the unit size is the size of a unit member of the array.
2. **Number of elements.** Number of array elements.
3. **Number of components.** Number of components of structure or record data structures.
4. **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.
   (c) Number of elements of an array-typed component.

5. **Object Tid number.** The Tid number of contents of an array or the Tid of an object being referenced by a pointer.
6. **Saving function.** A function to save 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.

Memory Block Saving and Restoring Functions

The saving and restoring function are the most important functions for data collection and restoration. Pointers to these functions are included in the TI table. Once we start migrating a process, the data collection function will search for memory blocks to be collected. Then, the saving function is invoked according to types of the memory blocks to encode their contents to machine-independent format, and save them to an output buffer.

After the buffer is transmitted to a new machine, the restoring function extracts contents of memory blocks, decode them to a machine-specific format, and store them into appropriate memory locations.

For a memory block that does not contain any pointers, we can apply XDR techniques [11] to save and restore. For a memory block contains pointers, the function `Save_pointer( pointer_content, tid )` and `Restore_pointer( tid )` will be used, where `pointer_content` is a memory address stored in a pointer variable and `tid` is its Tid number. `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 invoke 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`. All the functions, `Save_pointer`, `Restore_pointer` and others, are implemented and included in our data collection and restoration library. An illustrative example of these functions will be given in Section 5.4.

5.2.3 MSR Lookup Table Data Structure

The MSR Lookup Table (MSRLT) data structure is introduced to:

1. Keep information for the conversion of machine-independent format. Properties of the memory blocks such as Tid number, unit size of its data value, and number of elements are needed for data conversion.
2. Support memory block search during data collection. As mentioned in Section 5.2.2, 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. 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 logical technique to identify the memory blocks are developed for data collection and restoration between heterogeneous computers.

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 some pointers are needed to be recorded for process migration. 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 5.2, 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 $Gs = \{\}$, $Hs = \{v_7, v_8, v_9, v_{10}\}$, $Ls_{main} = \{v_4, v_5, v_6\}$, and $Ls_{foo} = \{\}$.

At compile-time, a variable is registered to the MSRLT data structure when:

1. It is a global variable referred to in 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 are significant.
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 already represent starting addresses of memory blocks. Therefore, if an array name is used in the right-hand-side of a pointer arithmetic statement, its address and properties information are registered to the MSRLT data structure.

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 case a memory block is dynamically allocated in 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 wrap 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 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 register 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`.

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 a logical identification that is used for reference for heterogeneous process migration. Figure 5.4 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 function calls in the program activation record. 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 function call in the 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.

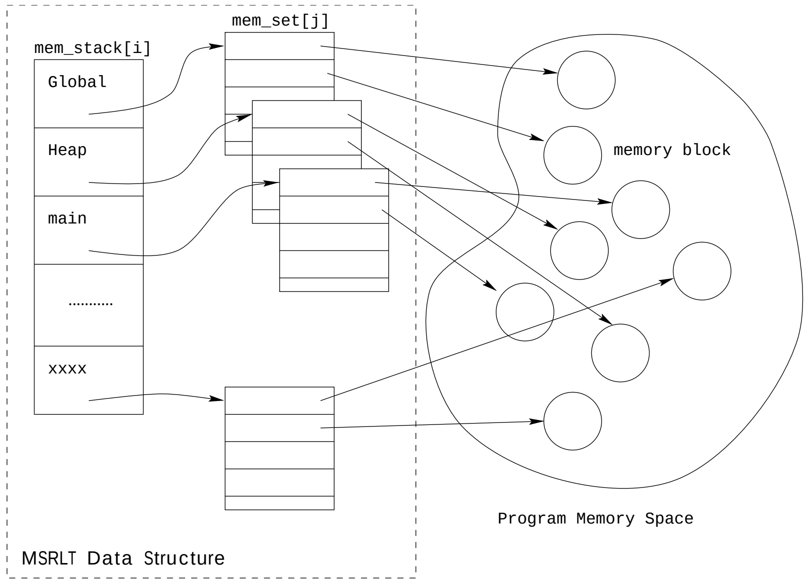


Figure 5.4: A Diagram shows the MSRLT Data Structures.

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 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.
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 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 to program source code. 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 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. Thus, the logical representation of v is $(stack\_index(v), set\_index(v))$.

5.2.4 Representation of Pointer

As stated in 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 5.5. There are three edges between nodes v_1 and v_2 in Figure 5.5. 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. $addr_1$ 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*.

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 5.5, the logical identification of v_2 is $(stack\_index(v_2), set\_index(v_2))$. We use this logical identification to represent the pointer head in the machine-independent information stream for process migration.

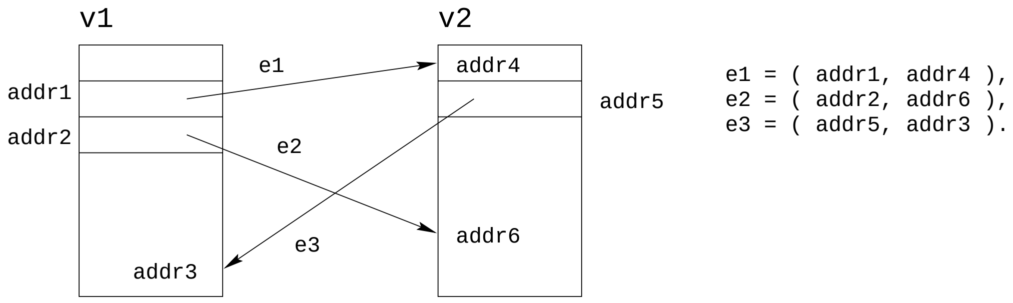
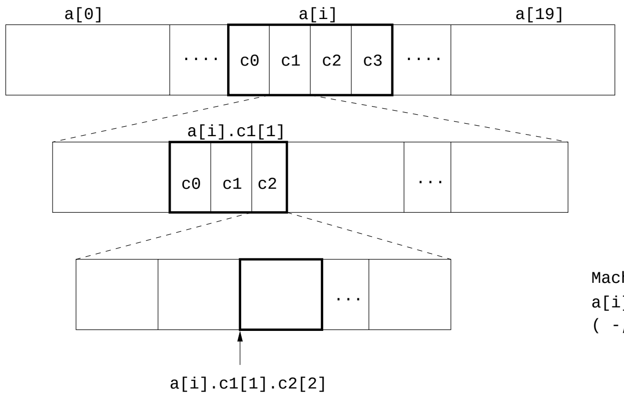


Figure 5.5: A representation of pointer between two nodes of the MSR graph.

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 5.2.2. 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[1].c2[2]` $-$ `a`) of the memory block of the variable `a` in Figure 5.6 can be translated into a sequence $< (0, i), (1, 1), (2, 2) >$. 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 $(1, 1)$ 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 $(1, 1)$ tells us that `a[i].c1` is the array and that the pointer points to the second element of that array, `a[i].c1[1]`. Finally, the component position of the pair $(2, 2)$ means that `a[i].c1[1]` is the structure and the pointer falls on the third component of `a[i].c1[1]`, which is `a[i].c1[1].c2`. The array element position of the pair $(2, 2)$ indicates that the component `a[i].c1[1].c2` is the array and that the pointer points to the third element of `a[i].c1[1].c2`, which is `a[i].c1[1].c2[2]`.



```
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];
}
```

```
Machine-independent Offset for
a[i].c1[1].c2[2] is :
( -, i ), ( 1, 1 ), ( 2, 2 )
```

Figure 5.6: An example of the machine-independent format of the pointer offset.

5.3 Data Collection and Restoration Mechanisms

Figure 5.7 shows software components which support data collection and restoration mechanisms. The components are the MSRLT data structure, the Type Information (TI) table, and the data collection and restoration library.

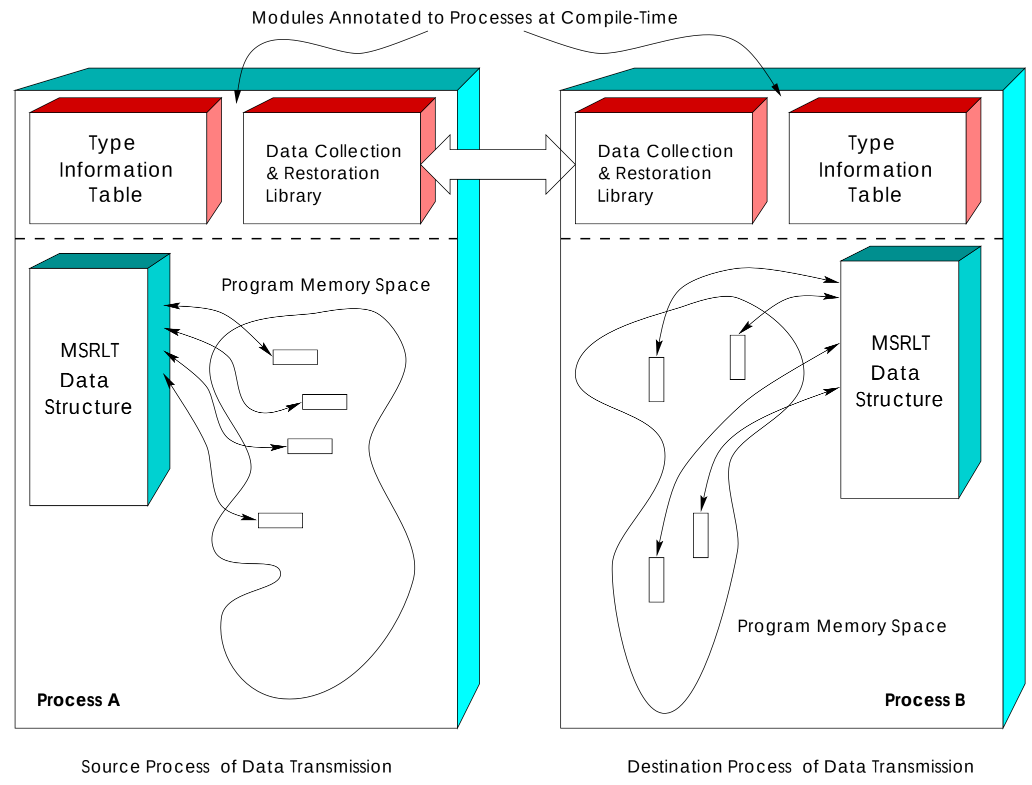

Modules Annotated to Processes at Compile-Time
Type Information Table
Data Collection & Restoration Library
Data Collection & Restoration Library
Type Information Table
MSRLT Data Structure
Program Memory Space
Program Memory Space
MSRLT Data Structure
Process A
Process B
Source Process of Data Transmission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

Figure 5.7: Software components for data collection and restoration mechanisms.

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 [11] 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 and included in our data collection and restoration library. The application of `Save_pointer` and `Restore_pointer` have two folds. First, they are used to collect and restore live variables in source codes as mentioned in the data transfer mechanism (Section 5.1). Another application is that they are used in the saving and restoring functions as discussed above.

When process migration occurs, `Save_pointer` or `Save_variable` are applied to collect live variables at a migration point. The functions search for records of memory block in the MSRLT data structure based on the memory addresses given to them as input parameters. The machine-independent identifications of memory blocks as well as their type information are returned as outputs. Then, the saving function is invoked according to types of the memory blocks to encode memory block contents to a machine-independent format, and save them to an output buffer. Note that if the memory block contains pointers, `Save_pointer` would be recursively invoked until all components derived from depth-first traversal are visited.

After the buffer is transmitted to a new machine, the restoring function extracts information of memory blocks, decodes them to a machine-specific format, and stores them into appropriate memory locations. We should remind 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, 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 also 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.

5.4 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 [9].

Given a sample program in Figure 5.2(a), Figure 5.2(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 5.2(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. 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.

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 case 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 5.2, we can define $Gm = \{v_1, v_2\}$, $Hm = \{v_7, v_8, v_9, v_{10}\}$, $Lm_{main} = \{v_3, v_4, v_5, v_6\}$, and $Lm_{foo} = \{v_{11}, v_{12}\}$.

Figure 5.3 shows the MSR graph representing a snapshot of memory space of the sample program in Figure 5.2(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. 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 [9]. 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 given in Figure 5.2, 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 data transfer mechanism mentioned earlier in Section 5.1, 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.

In the data restoration process, 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.

Chapter 6 Protocols

Activities in a large-scale distributed environment are dynamic in nature. Adding process migration functionality makes data communication even more challenging. A number of fundamental problems have to be addressed. First, a process should be able to communicate to others from anywhere and at anytime. Process migration could occur during a communication. Mechanisms need to be developed to guarantee correct message arrivals. Second, the problem of updating location information of a migrating process has to be solved. After a process migrates, other processes have to know its new location for future communications. The updating technique should be scalable enough to apply to a large network environment. Third, if a sequence of messages are sent to a migrating process, correct message ordering must be maintained. Finally, the communication state transfer needs to be integrated into the execution and memory state transfer seamlessly to form a process migration enabled environment.

Mechanisms to support correct data communication can be classified into two different approaches. The first approach is using the existing fault-tolerant, consistent checkpointing technique. To migrate a process, users can "crash" a process intensionally and restart the process from its last checkpoint on a new machine. Since global consistency is provided by the checkpointing protocol, safe data communication is guaranteed. Projects such as CoCheck [33] follow this approach. On the other hand, mechanisms to maintain safe data communication during process migration can be implemented directly into the data communication protocol. When a process migrates, process migration operations coordinate with data communication operations on other processes for reliability. Our work and the MPVM project [6] are along the second direction. We choose the second direction because the latter is more scalable and less costly than that of the former. Process migration is important enough to receive an efficient mechanism on its own right.

In this chapter, we have developed data communication and process migration protocols working cooperatively to solve the aforementioned problems. Our protocol design is based on the concept of point-to-point connection-oriented communication. It is aimed to provide a robust and general solution for communication state transfer. Mechanisms to handle process state transfer are implanted to a number of communication operations which could occur at data communication end points. These operations include send and receive operations in the data communication protocol and migration operation in the process migration protocol. They coordinate one another during the migration to guarantee correct message passing. The protocols are naturally suitable for large-scale distributed environment due to their inherited properties. First, they are scalable. During a migration, the protocols coordinate only those processes directly connected to the migrating process. The operations in process migration are performed mostly at the migrating process, while communication peer processes are only interrupted shortly for the coordination. Second, the process migration protocol is non-blocking i.e., it allows other processes to send messages to the migrating process during the migration. These two properties are quite beneficial for large environments where the number of participating processes is high. Third, the protocols do not create deadlock. They prevent circular wait, while coordinating a migrating process and its peers for migration. Finally, the protocols are simple in implementation and are practical for heterogeneous environments. They can be implemented on top of existing connection-oriented communication protocols such as PVM (direct communication mode) and TCP. We conduct empirical studies based on a prototype implementation on PVM.

The rest of this chapter is organized as follows. Section 6.1 gives the basic assumptions on the distributed computation model and communication semantics. In Section 6.2, we discuss the basic idea about communication state transfer and the communication and process migration algorithms. Section 6.4 discusses correctness of our algorithms when process migration occurs in different communication situations.

6.1 Backgrounds

We consider a distributed computation as a set of collaborative processes $\{P_0, P_1, \cdots, P_N\}$ executing under a virtual machine environment. Each process is a user-level process which occupies a separate memory space. The processes communicate through message passing.

A virtual machine environment is a collection of software and hardware to support the distributed computations. It has three basic components. First, a network of workstations is the basic resource for process execution. The workstations as well as their physical interconnection can be heterogeneous. Second, a number of daemon processes residing on workstations comprise a virtual machine. These daemons work collectively to provide resource accesses and management. A process can access the virtual machine's services via programming interfaces provided in forms of library routines. Finally, the third component is the scheduler, a process or a number of processes that control environmental-wide resource utilization. Its functionalities include bookkeeping and decision making. Unlike in static distributed environments such as that supported by PVM and MPI, a scheduler is a necessary component of a dynamic distributed environment such as the Grid [19]. For simplicity, we assume that the scheduler migrates one process at a time.

We identify processes in the environment in two level of abstractions: *application-level* and *virtual-machine-level*. In the application-level, a process is identified by a *rank* number, a non-negative integer assigned in sequence to every process in a distributed computation.

The rank number allows us to make references to a process transparently to its whereabouts. On the other hand, the virtual machine includes location information of a process in its naming scheme. A *virtual-machine-level process identification* (*vmid*) is a coupling of workstation and process identification numbers. They both are non-negative integers assigned sequentially to workstations and processes created on each workstation, respectively. The mappings between rank and *vmid* are maintained in a process location (PL) table, where the PL table is stored inside the memory spaces of every process and the scheduler. While the rank numbers are given only to application-level processes, the *vmid* is assigned to every process in the environment including the scheduler and virtual machine daemons. We assume that both the scheduler and the daemon do not migrate.

6.1.1 Communication Characteristics

In our design, processes interact by means of message passing. Every message complies to a single format, a coupling of a header and a body. We define the header by a triple of source and destination processes' ranks, and a *tag*, an integer used to distinguish a message. The body is an empty or finite sequence of values.

We require that message passing among processes in application-level follows blocking point-to-point communication in buffered modes. Assuming a message content is stored in a memory buffer, the send operation blocks until the buffer can be reclaimed, and the receive operation blocks until the transmitted message is stored in the receiver's memory. The sender process do not coordinate with the receiver for data transmission. Once the message is copied into internal buffers of an underlying communication protocol, the sender process can continue execution.

Figure 6.1 shows the protocol stack layout of the communication system for process migration environment. The lowest layer is the OS-supported data communication protocols between computers. The second layer lies communication protocols provided by the virtual machine built on top of the first communication layer. The virtual machine provides three basic communication services. They are the connection-oriented communication utilities, the connectionless communication utilities, and signaling across machines in a distributed environment. We assume the connection-oriented communication to create a bi-directional FIFO communication channel between two processes.

The third layer is the focus of this work. It consists of the migration-supported data communication and process migration protocols which are discussed in the next section in more details. The protocols in this layer provide primitives for the fourth layer, application-level process layer.

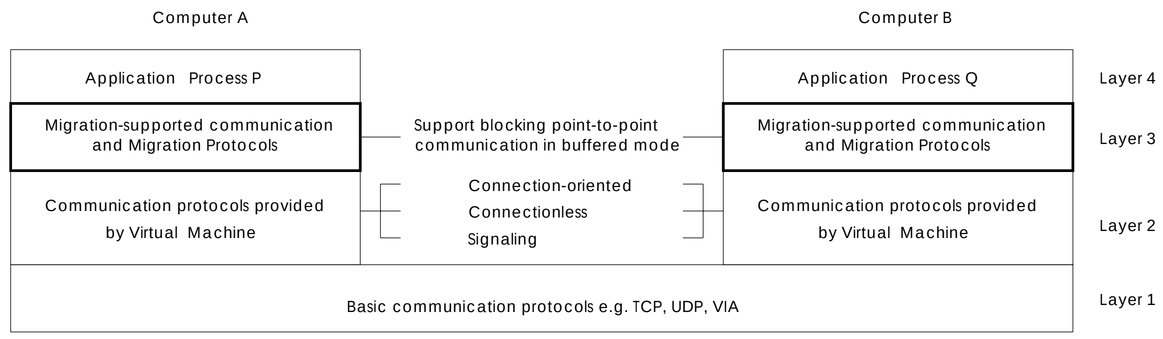

Computer A
Computer B
Application Process P
Application Process Q
Layer 4
Migration-supported communication and Migration Protocols
Support blocking point-to-point communication in buffered mode
Migration-supported communication and Migration Protocols
Layer 3
Communication protocols provided by Virtual Machine
Connection-oriented
Connectionless
Signaling
Communication protocols provided by Virtual Machine
Layer 2
Basic communication protocols e.g. TCP, UDP, VIA
Layer 1

Figure 6.1: The protocol stack layout.

6.1.2 Process Migration Environment

Based on our previous works [8, 9, 34], the process migration environment consists of the compilation and runtime systems. The compilation system annotates process migration mechanisms to source codes. 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*. The poll-point where process migration takes place is called the *migration point*. At each poll-point, a label statement and a specific macro containing migration operations are inserted. The macro includes operations to collect, transfer, and restore process states between computers. 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 are knowledgeable of their source codes.

Under the migration environment, the annotated source codes are pre-distributed to potential destination computers of process migration. Then, they are compiled (using compilation tools available on their host machines) and linked to three library modules which handle execution, memory, and communication state migration, respectively. A migration-supported executable, thus, is generated on each potential destination machine.

At runtime, a process migration is conducted directly via remote invocation and network data transfers. The scheduler manages process migration. When a participant in the environment want to migrate a process, it sends a request to the scheduler, which, in turn, decides the destination computer and remotely invokes the migration-supported executable to wait for process state transfer. We call the operations performed by the scheduler as the *process initialization* mechanisms. During the migration, the migrating process coordinates the initialized process to transfer its state information. Finally, while the migrating process terminates, the initialized process resumes execution from the point where process migration occurred.

6.2 Design Concepts

This section presents the design concepts of mechanisms to migrate the communication state. They give conceptual foundations to data communication and process migration algorithms in the following section. Since the data communication at the application-level is performed on top of the connection-oriented communication protocol, we define the communication state of a process to include all communication connections and messages in transit at any moment in the process's execution. To migrate the communication state, one has to capture the state information, transfer it to a destination computer, and restore it successfully.

Migrating a communication state is non-trivial since various communication situations can occur during process migration. To better describe activities in a distributed computation, we define each application-level process to generate a sequence of events [23]. An event is defined as an encapsulation of operations to perform certain function. We are interested in the computation, send, receive, and migration events. Supposed that a *migration point* is a location in the space-time diagram where a processor executes process migration operations. Figure 6.2 gives example communication situations when process $P2$ migrates at a migration point *MP*. There are four concurrent processes in the example. Based on the connection-oriented concept, a connection is established between $P1$ and $P2$ when $m1$ is sent. As a result, the same communication channel will be used to transmit $m2$ and $m3$. When the migration occurs at *MP*, there are three communication situations to be considered as shown by the example.

1. We need mechanisms to handle messages in transit during a process migration. From the figure, this is the case for the transmission of $m2$ and $m3$.
2. Mechanisms to handle message transmission from unconnected processes during a process migration must be considered (see the transmission of $m4$).
3. Finally, we need mechanisms to handle message transmission after process migration complete. This situation covers the cases of transmission of $m5$ and $m6$.

We describe the solution mechanisms to these situations based on observations on how the elements of the communication state of a process are captured, transferred, and restored to support process migration.

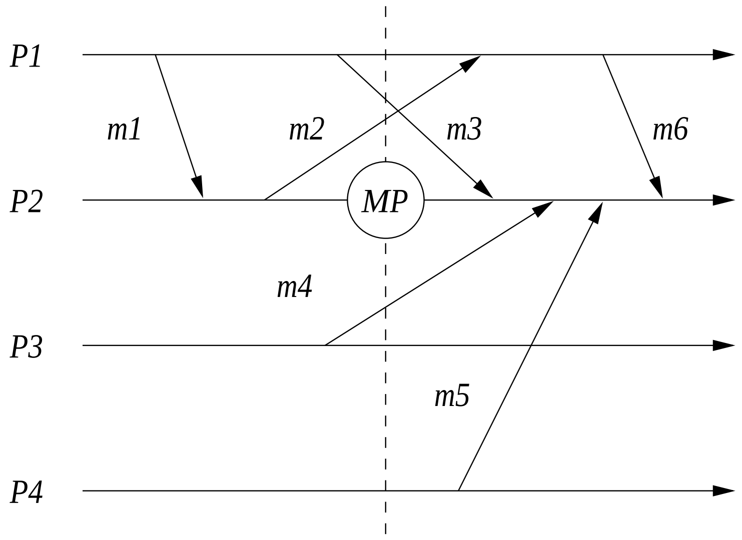


Figure 6.2: Communication situations.

6.2.1 Capturing and Transferring Messages in Transit

In the first case, to capture transmitting messages in a communication channel, processes on both ends can coordinate each other to receive the messages until there is no message left in the communication channel. The empty channel is then free to be closed for a process to relocate from one host to another. To do so, we present the following schemes:

1. A process coordination mechanism is needed as a part of the process migration algorithm. The migrating process has to initiate the coordination with every connected peer. From the example, $P2$ would initiate such coordination with $P1$. Our message coordination technique is based on the classical work of Chandy and Lamport [10] and will be discussed more in Section 6.3.2.
2. As the result of the coordination, messages are drained from the channel as soon as possible during a migration and are stored in the process memory space for future uses. A memory storage, namely the *received-message-list*, in user space of every process is needed for such purpose. From the examples, $m2$ and $m3$ are kept in the received-message-list of $P1$ and $P2$ respectively, as the result of the coordination.

3. Because messages may be stored in the receive-message-list before needed, the receive operation have to search for a wanted message from the list before taking a new message from a communication channel.

After the coordination, messages in transit are captured and existing communication connections are closed down. One may consider the messages stored in the received-message-list of the migrating process as a part of the process's communication state which have to be transferred to the destination computer. Based on the example in Figure 6.2, Figure 6.3 illustrates a possible situation where a migration event M is operated at the migration point *MP*. In order to capture the transmitting messages, $P2$ coordinates $P1$ and receives $m3$ into the received-message-list which would be forwarded (as "*comm state*") to the process *New P2* on a destination computer. Note that *New P2* is a migration-supported executable of $P2$ that is initialized on the destination computer.

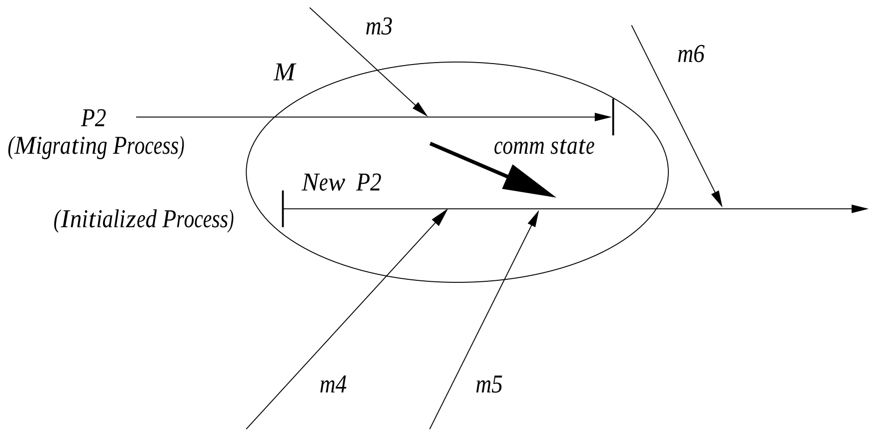


Figure 6.3: Activities in the migration event M.

6.2.2 Migration-aware Connection Establishment

To handle data communication from an unconnected process, the connection establishment mechanisms have to be able to detect process migration on one of its endpoint processes and automatically resolve the problem. The following schemes are employed:

1. Since our message passing operations only employ send and receive primitives and does not support explicit commands for connection establishment, the establishment mechanisms are installed inside the send and receive operations hidden from the application process.
2. In connection establishment, we employ the sender-initiated technique where the sender send a connection request to its intended receiver process. Having process migration into the picture, the establishment mechanisms must be able to detect the migration (or the past occurrence of the migration) and inform the sender process. For the examples in Figure 6.2, the mechanisms are applied to the sending of $m4$, $m5$, and $m6$. In the case of $m6$, we assume that the connection between processes $P1$ and $P2$ has been closed down before the transmission of $m6$. Therefore, at the moment $m6$ is sent, $P1$ is regarded as an unconnected process.

Supposed that the sender process defines the *vmid* of processes which participate connection establishment in a *participant* set. The *vmid* is used because identities of host computers are required by lower-level protocol to establish a physical communication connection. If the receiver process is detected to be migrating during the communication establishment, its *vmid* in the participant set must be changed to the *vmid* of the new process on the destination computer so that the sender process can retry the connection establishment again. Based on this scheme, in Figure 6.3, when $P3$ tries to send $m4$, the connection establishment mechanisms would detect the migration of $P2$ and redirect the establishment attempt to the process *New P2* instead. Similar situation occurs in the sending of $m5$. On the other hand, the connection establishment from $P1$ detects the past occurrence of process migration on the sending of $m6$ before redirecting its connection attempt to *New P2*.

Observation 1: Because of the process coordination and migration-aware connection establishment mechanisms, a sender process is not blocked while sending messages to a migrating process.

Observation 2: The update of location of the migrating process is always performed by a sender process as a part of connection establishment. Since the establishment is performed "on demand" and does not need global synchronization, the updating mechanism is scalable.

6.2.3 Communication State Restoration

The scheme for the restoration of communication state on a new process can be discussed in two parts. First, contents of the receive-message-list forwarded from the migrating process are inserted to the front of the receive-message-list of the new process. This scheme restores the messages which are in transit during the migration on the new process. Second, messages sent from a newly connected process to the new process are appended to the receive-message-list next to the message contents inserted by the first part.

We use this scheme to ensure message ordering in a point-to-point communication. For Figure 6.3, the received-message-list of $P2$ would store $m3$ as the result of process coordination and then being forwarded to *New P2* as a part of the *comm state*. Then, $m3$ would be stored as a part of the received-message-list of *New P2*. After the coordination, the communication connection between $P1$ and $P2$ is closed. $m6$ is considered as being sent from an unconnected process and would be appended to the received-message-list of *New P2* accordingly.

6.3 Algorithms

Based on the scheme for communication state transfer, algorithms are invented to support process migration. The data communication algorithms consists of send and receive algorithms which take care of the connection establishment and the receive-message-list, while

```
send ($m$, $dest$)
1: if($dest \notin Connected$) then
2:         $cc[dest]$ = connect($dest$);
3: end if
4: send $m$ along the $cc[dest]$ communication channel;
```

Figure 6.4: The send algorithm.

the process migration algorithms consists of two algorithms which run concurrently on the migrating and new processes to carry process migration. The algorithms use the following global variables. *Connected* is a set of rank numbers of connected peer processes. An array *pl* represents the process location table. The *vmid* of process P_i is stored in $pl[i]$. The *exe_flag* is a global variable indicating execution status of a process.

6.3.1 Data Communication Algorithms

In our design, the send algorithm initiates data communication between processes by sending a request for connection establishment to the receivers. In case the receiver cannot be found due to process migration, the send algorithm will consult the scheduler, a process which manages process allocation in the environment, to locate the receiver. Once the receiver's location is known, the sender establishes connection and sends messages to the receiver process. Figure 6.4 shows the send algorithm where a communication connection must be created before message transmission. The connection establishment mechanisms are described in the connect() function in Figure 6.5. The function starts by sending the connection request `con_req` to a receiver process. If the receiver is migrating, it will reject the request and send `conn_nack` back. Then, it will consult the scheduler as mentioned earlier.

```
connect($dest$)
1:  while ($dest \notin Connected$) do
2:      send conn_req to pl[dest];
3:      if (receive conn_ack from pl[dest]) then
4:          $cid := make\_connection\_with(pl[dest])$;
5:          $Connected := \{dest\} \cup Connected$;
6:      else if (receive conn_req from any process p) then
7:          grant_connection_to(p);
8:      else if (receive conn_nack from $pl[dest]$) then
9:          consult $scheduler$ for exe $\underline{status}$ and $\underline{new\_vmid}$ of $P_{dest}$
10:         if ($status$ = migrate) then
11:             $pl[dest] := \underline{new\_vmid}$;
12:         else report "error: destination terminated";
13:             return error; end if
14:         end if
15:     end if
16: end while
17: return $cid$;
```

Figure 6.5: Functions *connect*().

```
recv ($src$, $m$, $tag$)
 1: While ($m$ is not found) do
 2:     if ($m$ is found in received_message_list) then
 3:         return $m$, delete it from the list, and return to a caller function;
 4:     end if
 5:     get a new data or control message, $n$;
 6:     if ($n$ is data message) then
 7:         append $n$ to received_message_list;
 8:     else (handle control messages)
 9:         if $n$ is con_req then
10:             grant_connection_to(sender of $n$);
11:         else if $n$ is peer_migrating then
12:             close down the connection with the sender of peer_migrating;
13:     end if
14: end while
```

Figure 6.6: The recv algorithm.

The receive algorithm, as shown in Figure 6.6, is designed to collect messages in an orderly manner in process migration environment. The algorithm stores every messages arrived at a process in the received-message-list in the receiver's memory space.

The receive algorithm also has functionalities to help migrating its peer processes. In case that a process is running the receive event while one of its connected peer process migrates, the receive event may receive a control message from the migrating process. The `peer_migrating` control message is a special message sent from a migrating peer. The message indicates the last message sent from the peer and instructs the closure of communication channel. The reception of this message implies all the messages sent from the migrating process in the communication connection have already been received.

6.3.2 Process Migration Algorithms

The process migration protocol involves algorithms to transfer process state across machines. Two main algorithms in the protocol are the algorithm to collect the process state on a migrating process and the algorithm to restore the process state on a destination computer.

The former algorithm is shown in Figure 6.7. On the migrating process, the migration algorithm first checks whether a `migration_request` signal has been sent from the scheduler and is intercepted by the migrating process. If so, the algorithm rejects further communication connection so that it can coordinate with existing communication peers to receive messages in transit into the receive-message-list. In making the rejection, the algorithm first informs the scheduler that the migration has started and then changes the *exe_flag* variable which indicates execution status of the process. This step is important because from this point beyond, all `con_req` messages from unconnected processes will be rejected with `con_nack`. Thus, according to the send algorithm, the sender will have to consult the scheduler which already has information about the migration in hand.

In process coordination, the migrating process sends `disconnection` signal and `peer_migrating` control messages out to all of its connected peers. The `disconnection` signal will invoke an interrupt handler on the peer process if the peer is running a computation event. The handler keeps receiving messages from the communication connection until the `peer_migrating` message is found and then closes the connection. In case the peer process is running a receive event, the receive algorithm may detect `peer_migrating` while waiting for a data message. The peer process then will close down the communication connection by the receive algorithm (see statement 12 of Figure 6.6).

The last message from a closing peer connection is the end-of-message symbol. The migrating process receives messages from all communication connections to its receive-message-list until all the end-of-message's are received. The migrating process, then, closes the existing communication connections and collects the execution state and memory state.

```
Process: $P_i$

migrate()
 1: if($exe\_flag$ = migrate_request)then
 2:     inform the $scheduler$ migration_start;
 3:     get new_vmid of $P_i$ from $scheduler$;
 4:     $exe\_flag$ := migrate;
 5:     All the con_req messages arrive
            beyond this point will be rejected;
 6:     Send disconnection signal and peer_migrating
            control message to all connected peers;
 7:     Receive incoming messages to receive-message-list until getting all
            responses saying that the channels are closed by peers
 8:     close all existing connections;
 9:     Send received-message-list (comm state) to the new process;
10:     perform exe and memory state collection;
11:     Send the exe and memory state to the new process;
12:     wait for migration_commit msg from $scheduler$;
13:     cooperate with the virtual machine daemon to make sure that no more
            con_req control messages left to reject;
14:     terminate;
15: end if
```

Figure 6.7: The migrate() algorithm on the migrating process.

Then, it sends content of the receive-message-list as well as the execution and memory state information to the destination machine. Note that, for heterogeneity, the execution and memory state transfers are based on the techniques presented in our previous works [8, 9] and the XDR encoding/decoding operations are performed to data transmission.

On the destination computer, a new process is created to wait for the process state transfer from the migrating process. The algorithm for this operation is given in Figure 6.8. According to the migration algorithm, while waiting, the new process grants connection establishment to any peer processes who want to send data to it. We should note here that this granting operation occurs simultaneously to the rejection of connection requests on the migrating process. Therefore, according to the send algorithm, the sender processes

```
Process: $P_i$

initialized()
 1: $exe\_flag$ := wait;
 2: All the con_req messages are accepted beyond this point;
 3: Receive received_message_list of the migrating process;
 4: insert it to the front of the original received_message_list;
 5: $exe\_flag$ := restore;
 6: Receive "exe and mem state" of the migrating process;
 7: Restore process state;
 8: inform the scheduler restore_complete;
 9: wait for contents of the PL table and old_vmid from the scheduler;
10: inform the scheduler migration_commit;
11: $exe\_flag$ := compute;
```

Figure 6.8: The initialize() algorithm on the initialized process.

whose connection request are rejected will consult the schedule and automatically redirect their requests to the new process at the destination of process migration. After receiving all the process state, the algorithm on the new process will restore the state information and resume execution.

6.4 Protocol Analysis

Since activities generated by process migration are additional to what generated in non-migration situation, the distributed computation logic must be preserved. This section discusses the correctness of our process migration algorithm in two aspects.

1. Despite the migration, the message ordering semantics of point-to-point communication is preserved.
2. The process migration does not introduce deadlock, other than what could be produced by normal operations of the distributed computation.

6.4.1 Point-to-Point Communication

Figure 6.9 shows three possible "crossing-migration" communication situations between two processes. We show that the mechanisms used to capture the communication state also preserve message ordering semantics and maintain correct message delivery in different communication modes. Let S and M represent the send and migration events, respectively, while X could be a send, receive, or compute event. We assume that there are two processes $P1$ and $P2$ in the environment and the migration event always occurs at $P2$. The dashed line from events X to M indicates a timeline when a migration occurs.

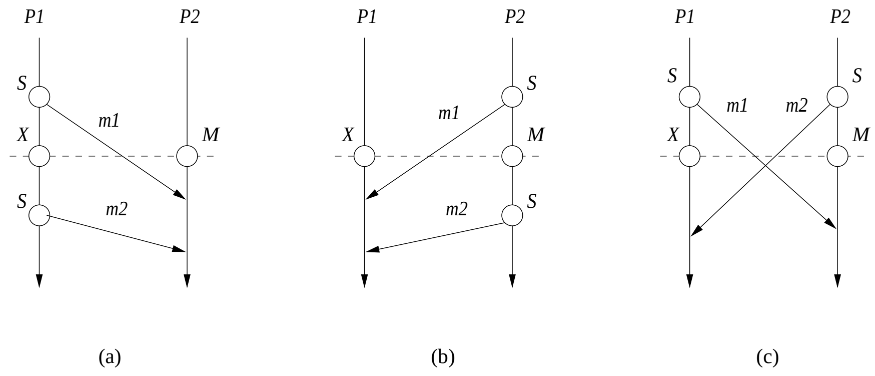


Figure 6.9: Three communication situations.

Situation 1. For situation given in Figure 6.9(a), we show $m1$ is received by the migrating process before $m2$. We consider the situation in two cases. First, we suppose the two processes are connected. From the figure, $m1$ is on its way to $P2$ before or during the migration, while $m2$ is sent after the migration complete. Based on the migrate() algorithm given in Figure 6.7, every message in the communication channel will be received into the received-message-list during process coordination. Since the received-message-list of the migrating process are inserted to the front of the received-message-list of the initialized process and $P1$ has to establish a new connection before sending $m2$, $m1$ always be in the message list before $m2$.

Second, if the connection does not exist before sending $m1$, the send event has to wait for a `conn_nack` in response to its connection request, either from the migrating process or from the virtual machine daemon (after the migration process terminate). Then, according to the send algorithm, the send event will establish a new connection with the initialized process. Since the same connection is shared by both $m1$ and $m2$ and the FIFO behavior is assumed, $m1$ is always received before $m2$.

Situation 2. Similar consideration goes to the situation in Figure 6.9(b). We show that $m1$ is received by $P1$ before $m2$. $m1$ and $m2$ are sent before and after the migration, respectively. In case the two processes are connected, the migration event sends a disconnection signal and `peer_migrating` control message to $P1$, causing $P1$ to receive every message in the connection to its message list. Simultaneously, the migrating process receives all messages sent to the migrating process to the receive message list. It also makes sure that such receive operations on $P1$ finishes before $P2$. As a result, unless $m1$ is a wanted message of any receive event on $P1$ before or during the migration, it would be stored in the message list of $P1$ when the migration completes. Since the connection is closed during the migration, the send event of $m2$ has to initiate a connection again before sending the message. Thus, $P1$ always receive $m2$ after $m1$. In case the prior connection does not exist, the send event of $m1$ (on $P2$) has to wait until at least a send or receive event on $P1$ intercepts its connection request and establishes a connection. Since the migration occur after the send, the connection is always established before a migration, and thus the previous reasoning holds.

Situation 3. In Figure 6.9(c), we test whether a migration can create additional deadlock, other than what could occur in normal operations. The send event on $P1$ can occur before or during a migration; while the send event on $P2$ has to occur before the process migrate. As shown in Figure 6.9(c), the process $P2$ would execute a migration event only after the send event for $m2$ has finished. Therefore, the migration implies the existence of communication channel and a successful execution of a send event in buffered mode for

$m2$. During the migration, the disconnection signal and `peer_migrating` control message will cause $P1$ to receive all messages sent from $P2$ to its receive message list and then close the connection. Concurrently, the migrating process keeps receiving messages from $P1$ until the end-of-message arrives. Since the connection is bi-directional, the receiving operations on both communication ends can be done simultaneously. The migration algorithm does not generate a circular wait. As the result, the migration does not cause deadlock in this situation.

6.4.2 Deadlock among Multiple Processes

In this section, we investigate the possibility of deadlock when a migration occurs during the communication of more than two processes in the environment. Deadlock is a result of circular wait, which could occur in the situation given in Figure 6.10. In case $P3$ has a connection to $P2$, the migration event has to coordinate $P2$ to make a disconnection. Since $P2$ is running a send event, the disconnection would be performed after the send has finished. This, in turn, would cause the migration event to wait. Remind that, in our design, the send algorithm blocks when `(i)` it waits for the response of the connection request, and `(ii)` the transmitted message is too big and causes the control flow mechanism of the underlying communication protocol to operate. If send events on $P2$ and $P1$ are blocked, a circular wait on the three processes would occur. One can scale this problem to the case of multiple processes.

To prevent deadlock, our migration algorithm is designed to receive every message being sent to it during process migration. Therefore, a sender process does not have to block and deadlock is prevented. Reminded that in Figure 6.7, the migration algorithm receives all messages from its connected peers during process coordination. Thus, if $P1$ already has a

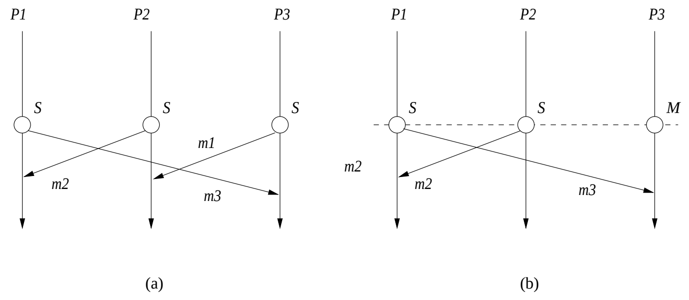


Figure 6.10: A communication situation which may cause deadlock.

connection with $P3$, the migrating process, $m3$ would be received to the receive message list by the migration algorithm. As a result, the circular wait in Figure 6.10 is broken.

On the other hand, suppose that $P1$ and $P3$ are not connected. Since the send event on $P1$ is performed during a migration, the connection request would be redirected to the initialized process. The statement at line 1 of the initialize() algorithm, therefore, would take care of this case by granting a connection and receive $m3$ to the message list. Note that, from the initialize() algorithm in Figure 6.8, the initialize() algorithm does not pass line 3 unless the the process coordination in the migrate() algorithm finishes. Therefore, the Receive statement at line 3 of initialize() would keep on receiving all the messages from new connections which might cause a circular wait to occur. Hence, deadlock is prevented by our protocols.

Chapter 7
Experimental Results

To verify the correctness and efficiency of the process migration software system, we conducted the prototype runtime system and a number of experiments for various purposes.

1. Section 7.1 presents the feasibility study on the potential usages of heterogeneous process migration to improve performance of parallel processing. We run a parallel matrix multiplication program on an early prototype migration software system.
2. In Section 7.2, we have implemented the data collection and restoration runtime library to support memory state transfer. We tested our implementation on a number of programs with different data structures and execution behavior. Experimental results advocate the correctness and efficiency of the MSR model.
3. In Section 7.3, we improved the early prototype and create a virtual machine environment. Based on the well-defined algorithms presented in the previous chapter, we implement data communication and process migration protocol.
4. Finally, in Section 7.4, we have tested our protocols on the parallel kernel MG benchmark on homogeneous and heterogeneous testbeds. Results show the correctness and efficiency of our protocol designs.

7.1 Feasibility Studies

A parallel matrix multiplication program with Master-Slave communication topology developed in [3] was chosen to test an early version of our migration software system, namely the *Migration point-based PVM* (MpPVM) [8, 7]. MpPVM is the first version of our runtime system. It is an extended version of PVM which does not include mechanisms to transfer complex data structures. It is used mainly to investigate potential of process migration in alleviating load-imbalance problem in parallel computation.

In the Matrix Multiplication experiment, both the PVM and MpPVM versions of this program are implemented. These two versions are different in that the MpPVM one contains a number of migration points at various locations in the program. Both experiments are performed on the same LAN environment consisting of four SUN Sparc IPC and fourteen DEC 5000/120 workstations.

7.1.1 Process Migration in a Non-dedicated Environment

The purpose of our first experiment is to verify the heterogeneous process migration of MpPVM and to examine the performance degradation of the PVM version when parallel tasks have to compete with local jobs for computing cycles. We assume the new process at a destination of a process migration is pre-initialized. Since pre-initialization involves a certain hueristic method of the scheduler to foresee the workstations with high availability in the idle workstation pool, an efficient resource allocation policy in the environment is assumed.

In this experiment we run four parallel processes on two SUN and two DEC workstations. One of the two DEC workstations was the test site that had continued requests from the machine owner (or other local users). When the workload on the test site increased, MpPVM migrated its process from the test site to one of the unused SUN workstations on the network. In the competing situation, we simulate the increasing workload of local computation by gradually adding light-weighted processes, i.e. programs with few floating–point operations and small data size, to share more time slices from the CPU.

From Figure 7.1 we can see that MpPVM achieved a superior performance over PVM especially when the owner's jobs request more CPU time. In 7.1(a), a 3x400 and a 400x3 matrix are multiplied for ten times. With competing with local jobs, both parallel processing versions have an increased execution time with the requests from the owner. However, the

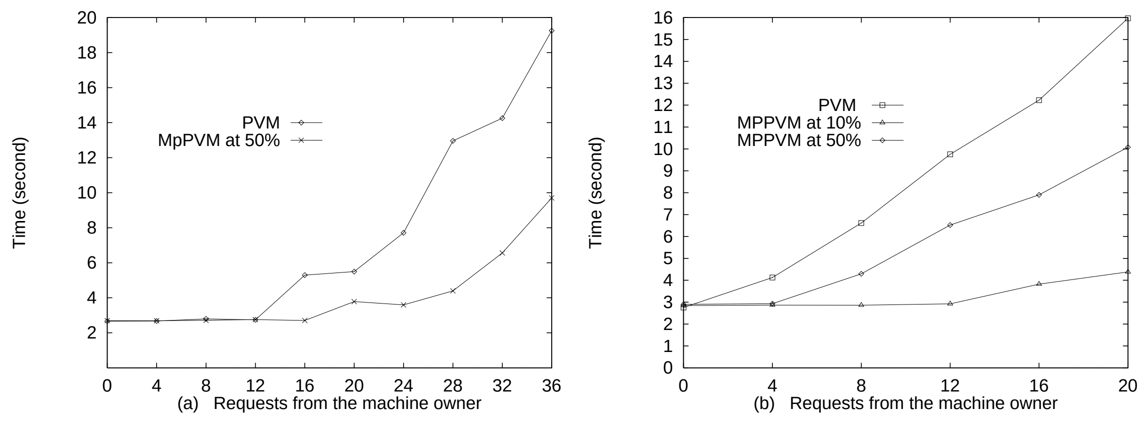


Figure 7.1: Performance comparison between PVM and MpPVM.

execution time increase of MpPVM is about a half of that of PVM when the machine owner actively uses the test site. This is because in this implementation only one migration-point is used and is located at the half-way point of the total required computation. In 7.1(b), we still use the same configuration as 7.1(a) except that we increase the order of the matrices to 3x1600 and 1600x3 respectively. As a result, the parallel processing performance becomes more sensitive to the owner requests than in 7.1(a). In this test we have developed two MpPVM programs which will migrate from the overloaded machine after 10 percent and 50 percent of their executions, respectively. The performance of migration at 10 percent is better than that at 50 percent since the process spends less time competing with the owner computations. In general, the more migration points in a parallel program, the less the duration of competing with local computation will be and, therefore, the better performance is achieved.

7.1.2 Performance of Multiple Process Migration

In the second set of experiments, the goal is to investigate the effects of process migrations when both the problem and ensemble size scale up. Since we want to detect the degradation caused from process migrations only, the tests are conducted in a dedicated environment.

Note that the tests were conducted at midnight on Friday's when no one else was on the system.

We scale the problem size of the MpPVM matrix multiplication program with the number of processors on 4, 6, 8, 10, and 12 workstations by following the memory-bounded scale-up principle [35], respectively. We should note that for the matrix multiplication $A \times B$, the order of matrix A is 3x1600, 3x2400, 3x3200, 3x4000, and 3x4800 for ensemble size 4, 6, 8, 10, 12 respectively. The order of matrix B is symmetric to the order of matrix A. At each ensemble size, at most 3 and 5 slave processes are migrated when the program runs on 4 and 6 workstations respectively; whereas, 6 slave processes are migrated when the program uses a greater number of workstations. New processes of process migration are pre-initialized, and overlapping of process migrations are prevented in this experiment. The experimental results are depicted in Figure 7.2.

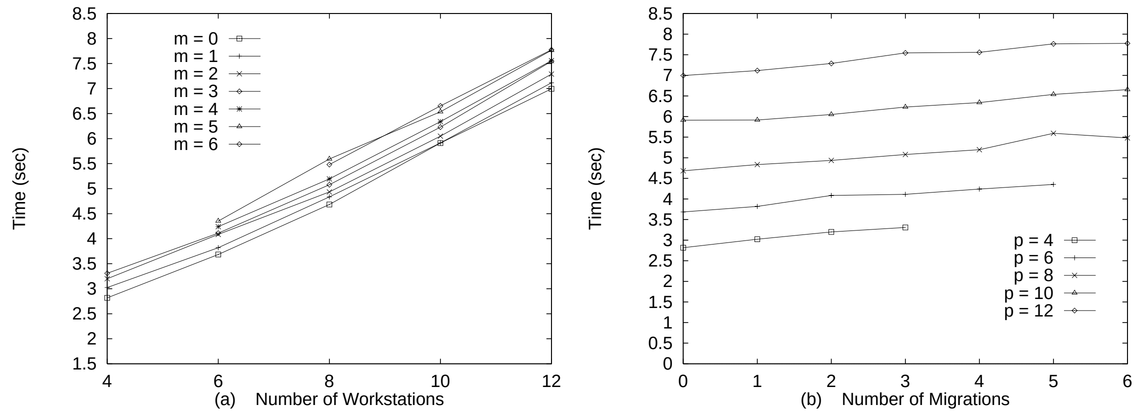


Figure 7.2: Performance comparison when the problem size scales up.

In Figure 7.2(a), each curve represents a relationship between execution time and the scaled problem size, indicated by the number of workstations, at a specific number of process migrations (m). At $m = 0$, there is no process migration. Likewise, the curves at $m = 1$ to $m = 6$ give the results with one to six process migrations, respectively. We can see that these curves increase approximately in parallel. For example, the curve of $m = 6$ (migrate six times) grows in parallel with the curve of $m = 0$ (no migration). Since the difference of execution time of $m = 6$ and $m = 0$ is the migration cost, we can conclude that the migration costs caused by six process migrations at the scaled problem size on 8, 10, and 12 workstations are approximately the same. Therefore, the migration cost is a constant and does not increase with the ensemble size in our experiments. The execution time increase is mainly due to the increase of computation and communication, not due to the migration cost, which is especially true when the ensemble size is large.

Figure 7.2(b) further confirms that the migration cost does not increase with problem and ensemble size. In this figure, different curves depict the relation between the execution time and the number of process migrations at a specific ensemble size. We can see that the increase of execution time with the number of process migrations is consistently small compared with the computing and communication time. For example, at 12 workstations ($p = 12$), with six migrations, the total migration delay is less than 10% of the total execution time. In addition, the increasing rates of the execution time at every ensemble size (p) are approximately the same.

7.2 Data Collection and Restoration

Software for heterogeneous data transfer can be classified into four layers as illustrated in Figure 7.3. 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 [11] 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.

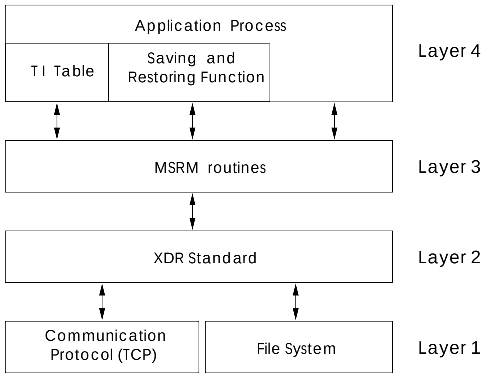


Figure 7.3: Software structure for data collection and restoration

7.2.1 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 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 MSR graph of the tree-like data structure is shown in Figure 7.4(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.

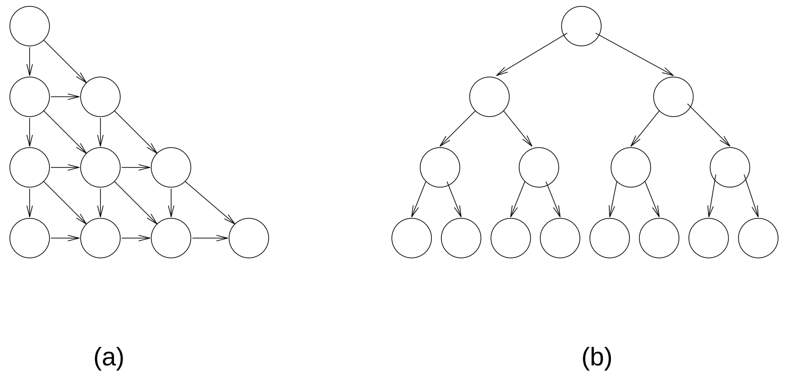


Figure 7.4: MSR graph for the test_pointer (a) and the bitonic sort (b) program

The linpack benchmark from netlib repository at ORNL [29] is a computational 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 run-time, 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 [2, 4] was tested. In this program, a binary tree, illustrated in Figure 7.4(b), is used to store randomly generated integer numbers. The program will manipulate the 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 experiments. One is a tree with 1,024 nodes; the other is a tree with 4,096 nodes. Process migration is conducted in function *bimerge()* with a sequence of function recursive calls, $main() \rightarrow bisort() \rightarrow bisort() \rightarrow bisort() \rightarrow bisort() \rightarrow bimerge() \rightarrow bimerge() \rightarrow bimerge() \rightarrow bimerge()$.

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. It is truly heterogeneous because both systems use different endianness. Both machines 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 7.1, we test all programs on two different data sizes, which create different size 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 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 test_pointer program, despite multiple references to MSR's significant nodes, all memory blocks and pointers are collected and

Table 7.1: 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

restored without duplication. Other data structures such as the pointer to an array of 10 pointers to integers also contain correct values under C dereferencing 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 capabilities of our process migration 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.

7.2.2 Data Collection and Restoration Complexity

Table 7.2 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

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

By Table 7.2 the migration time of the linpack and bitonic programs are 2.096 and 0.526 second, respectively.

Table 7.2: Timing results (in seconds) of migration.

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

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` $+$ `Encoding and Copying`,

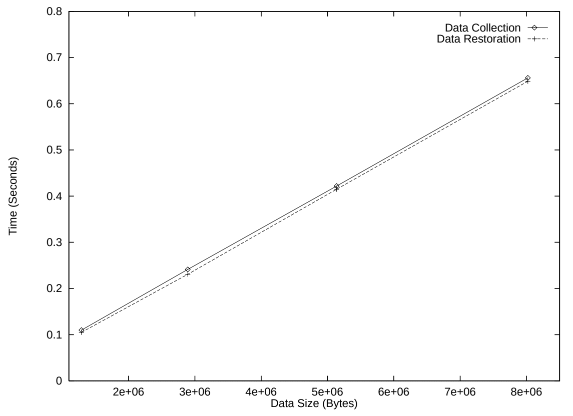
where the `MSRLT Search` time is the time for searching the MSRLT data structure. Suppose that there are n fully-connected MSR nodes in the program memory space and each node has D_i bytes where $1 \leq i \leq n$, then the `MSRLT Search` time depends on n and has the upper bound complexity of $O(nlogn)$, 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` $+$ `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 $MSRLTupdate$ time takes $O(n)$ time complexity, and the `Decoding and Copying` time takes $O(\sum_{i=1}^{n} D_i)$.

Figure 7.5(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 7.5(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



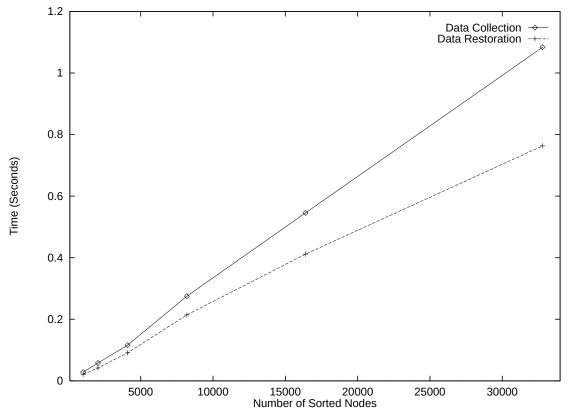


Figure 7.5: Timing for (a) the linpack program, (b) the bitonic sort programs.

time than that of the `MSRLT update` time ($O(n)$) for data restoration, when the number of data to be sorted scales up.

7.2.3 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.

Table 7.3 reports the execution overhead when migration points are placed on different levels of a nested function call. In the linpack benchmark with 1000×1000 problem size, the execution overhead of 0.07% is detected when migration 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 1.6% 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 (over 64%) 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 only invoked once. Moreover, since the amount of work for each call to the $DAXPY()$ is small, the inserted migration macro will cause significant proportion of overhead. The overhead occurred is reasonable and mostly can be avoided. In a practical situation, there is no need to insert a migration point inside of a small kernel.

Table 7.3: Overhead of programs using different migration point placements.

Programs	original	1-level	2-level	3-level
Linpack 1000x1000	764.64	765.21 (0.07%)	777.61 (1.6%)	1256.49 (64.32%)
bitonic 8192	0.82	0.94 (14%)	1.02 (24%)	1.54 (87%)

In case of the bitonic sort, a similar phenomena of that of the linpack benchmark is noticed. As we try to make the inner function of the call sequence migratable, 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 that 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 migration-point placement mechanisms need to be developed. Some works have been given in developing heuristic algorithms [8, 34]. More investigation into this matter is needed.

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

7.3 Complete Prototype Implementation

To test the proposed reliable data communication and process migration protocols, we have implemented software prototypes and performed experiments on a communication intensive, parallel kernel MG benchmark program. The prototypes consist of a software library for the protocol implementation and its supportive runtime system. Figure 7.6(a) shows the protocol stack and programming libraries used by migration-supported application programs. We modify the PVM communication library to accommodate our protocol. PVM has two communication modes. One is direct communication implemented on top of TCP/IP, the other is indirect communication where messages are routed via PVM daemons. We extend the direct communication for our message passing protocol, while we only use the indirect mode for sending control messages. The direct communication establishes TCP connection "on demand" when a pair of `pvm_send` and `pvm_recv` or a pair of `pvm_send` are invoked by the processes on both ends of communication. As shown in Figure 7.6(a), the TCP lies in the lowest layer of the protocol stack and has an extended PVM communication library implemented on top. In the second layer, we modify the `pvm_send` routine to consult the scheduler when it tries to establish a communication channel with a process that has been migrated. We also add a number of *connectivity service routines* providing utilities for higher-level protocols to request, grant, destroy, and make status reports of every communication channel a process has. We found that the extension only causes small changes to PVM source as most of our protocol designs are implemented in next layer of the protocol stack.

We implement our data communication and process migration protocols in the third layer. Although the `send` algorithm has most of its operation performed in the extension to the `pvm_send`, operations to access the PT are implemented in this layer. The receive algorithm maintains the receive-message-list here. Most importantly, the `migrate` algorithm is implemented in two parts consisting of codes to be operated on the migrating and the initialized processes. These codes cooperate the two processes as well as the scheduler throughout a process migration. The `migrate` algorithm uses the extended connectivity service routines to disconnect existing communication channel. Another important programming library in the third layer contains utilities to collect and restore data of a process. For modularity, we implement the data collection and restoration functions separately from that of the communication and migration protocols. This library involves the collection, transfer, and restoration of the execution and memory states of a process. Instead of using the data communication routines, we implement the state transfer directly on the TCP protocol to avoid communication overheads. The details can be found in [9]. Finally, in the forth layer, we have application programs annotated with interface routines from the lower-level libraries. We assume the programs containing migration-supported language features as defined in [32]. Figure 7.6(b) shows the prototypes runtime system. We use the PVM virtual machine to handle process creation and termination and to pass control messages and signals between machines. A simple scheduler is implemented to oversee process migration. In current implementation, the scheduler does not support any advanced allocation policy but basic bookkeeping for process migration records.

7.4 Collaborated Communication

As a case study, we show here the application of the prototype implementations on the parallel kernel MG benchmark program [39]. The benchmark is written in C and originally runs under PVM environment. The kernel MG program is an SPMD-style program executing four iterations of the V-cycle multigrid algorithm to obtain an approximate solution to a discrete Poisson problem with periodic boundary conditions on a $128 \times 128 \times 128$ grid.

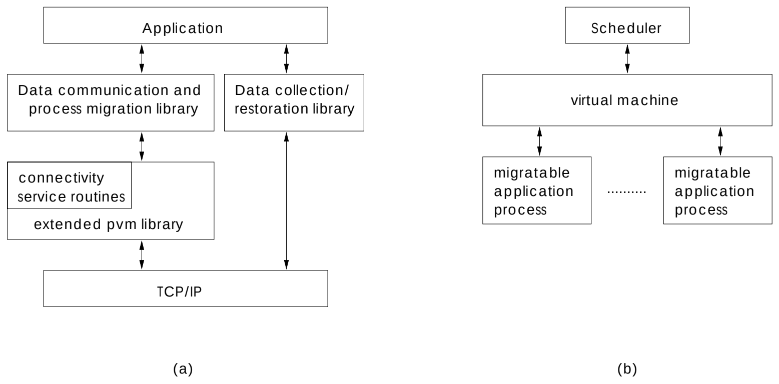

Application
Data communication and process migration library
Data collection/ restoration library
connectivity service routines
extended pvm library
TCP/IP
(a)
Scheduler
virtual machine
migratable application process
migratable application process
(b)

Figure 7.6: (a) Programming libraries and (b) runtime system.

The multigrid algorithm involves arithmetic on vectors u, v, and a work vector r. Each vector is represented by a three dimensional array data structures. In the multigrid recursion, the algorithm operates on the vectors starting at a $128 \times 128 \times 128$ volume to $2 \times 2 \times 2$, shrinking by 2 on each recursion. Then, as the recursion returns, the volume of vectors in operation are expended backward to $128 \times 128 \times 128$. The benchmark program executes the multigrid algorithm for four times.

The kernel MG program applies block partitioning to the vectors for each process. A vector is assigned to an array of size $16 \times 128 \times 128$ when 8 processes are used. Figure 7.7 illustrates the data partitioning and communication pattern on a 4 process example. Since each process has to access data belonging to its neighbors, the data must be distributed to the computation which need them. Such distribution occurs periodically during execution. Every MG process transmits data to its left and right neighbors. Therefore, the communication is a ring topology. Details on how the multigrid algorithm works with such partitioning can be found in [39]. Data communication of the MG program is nontrivial. The application exercises extensive interprocess communication; over 48 Mbytes of data on the total of 1472 message transmissions.

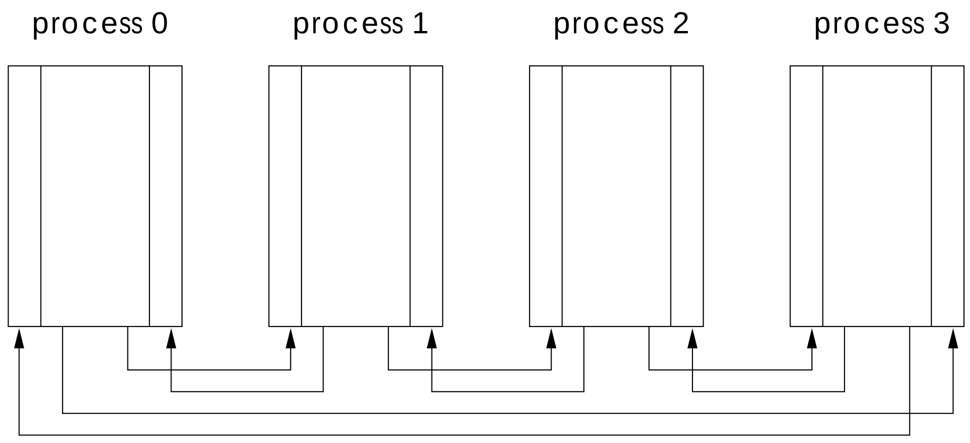


Figure 7.7: Data partitioning and communication pattern of the kernel MG program.

We have annotated the program with process migration operations and linked the annotated program to our protocols. We configure the kernel MG program to generate 8 processes ranking from process `0` to `7`. We insert a migration point in function `kernelMG` which would cause process `0` to migrate when a function call sequence `main` $\rightarrow$ `kernelMG` is made and two iterations of the multigrid solver inside the `kernelMG` function are performed. For reliable data communication, we change the PVM send and receive routines to the send and receive routines of our communication library. With this configuration, the migration is supposed to be performed on process `0` while other processes keep executing and has no prior knowledge of the migrating incident. Note that no barrier is used to synchronize the processes during a migration.

Three experiments are conducted to study process migration of the kernel MG program. First, we verify the correctness of the computation and communication logics and analyze the communication pattern during a process migration. Second, we measure the performance overheads which occur to the annotated code with and without a process migration. Finally, we run an experment to verify that the proposed process migration mechanisms migrates the kernel MG program correctly in a heterogeneous environment.

7.4.1 Correctness of Distributed Execution

To verify the correctness of distributed execution, a number of testings are performed. First, we compare the final outputs of the annotated application with and without a process migration against the output of the original benchmark. The three experiments are run on the same testbed, a cluster of 8 Sun Ultra 5 workstations connected via 100Mbit/s Ethernet network. Each machine hosts a kernel MG process. We consider the final value of the u, v, and r vectors to be the final output of the applications. The comparison shows that outputs from all experiments are identical. Second, we focus on the correctness of the migrating process. Since the process `0` is migrated at the beginning of the third multigrid iteration, we compare the intermediate results of the u, v, and r vectors at the point right after the migration to those generated from the original benchmark and found that they are identical as well.

In addition to the correctness verification, we also has conducted performance study. Figure 7.8 shows a XPVM generated migration diagram of the kernel MG program running on a cluster of 10 Sun Ultra 5 workstations. We set up two machines to run the scheduler and the initialized process. Process `0` spawns seven other processes on different machines as shown in Figure 7.8. Note that a line between two timeline indicates a message passing which starts at the point where `pvm_send` is called and ends when the matching `pvm_recv` returns. Since our communication routines are implemented on top of PVM, these lines also show what are going on inside our prototype implementation. Also, since we implement the execution and memory state transfer directly on TCP, their network transmissions are not displayed in this diagram. In the figure, the execution is separated into different stages. First, all `pvmmg` processes establish connections, distribute data, and perform the first two iterations. Then, the migration is performed by relocating process `0` to the initialized process. After the migration, the kernel MG resumes the computation and finishes.

At each iteration, each MG process periodically exchanges messages with its near neighbors. Within an iteration time frame, the message size as well as the number of participant processes are different as shown in Figure 7.9. From the figure, every process starts the iteration by exchanging the message of size 34848 bytes, then 9248, 2592 and so on, until the size becomes 128. As the message size shrinks, the number of participant processes (in the ring topology) also reduces from 8 to 4 for message size 288, and then to 2 for message size 128. After that, the multigrid computation expands the message size as well as the number of participant processes in a reversed order. The last message size for every iteration is 135200 bytes. Note that we only show the first part of the iteration since the migration occurs in this part.

We have observed a number of interesting facts through the space-time diagram. First, since the migrating process has connection to all other processes, it has to send disconnection signals and `peer_migrating` messages to them. When the migration starts, we find that there is no message sent to the migrating process from any of the connected peers. Therefore, the migrating process does not receive any messages into the receive-message-list when it performs message coordination with connected peers. In other words, the "*comm state*" as shown in Figure 6.3 is empty. After the coordination, every existing connection is closed. This operation is shown in area `A` in Figure 7.10. Note that we have performed ten experiments under the same testing configuration and found that the timing results appeared to be very similar. There is no forwarding message in all tests. The communication pattern during the migration also does not exhibit any variation. Second, while process `0` migrates, other processes proceed with their data exchanges normally. As long as a process

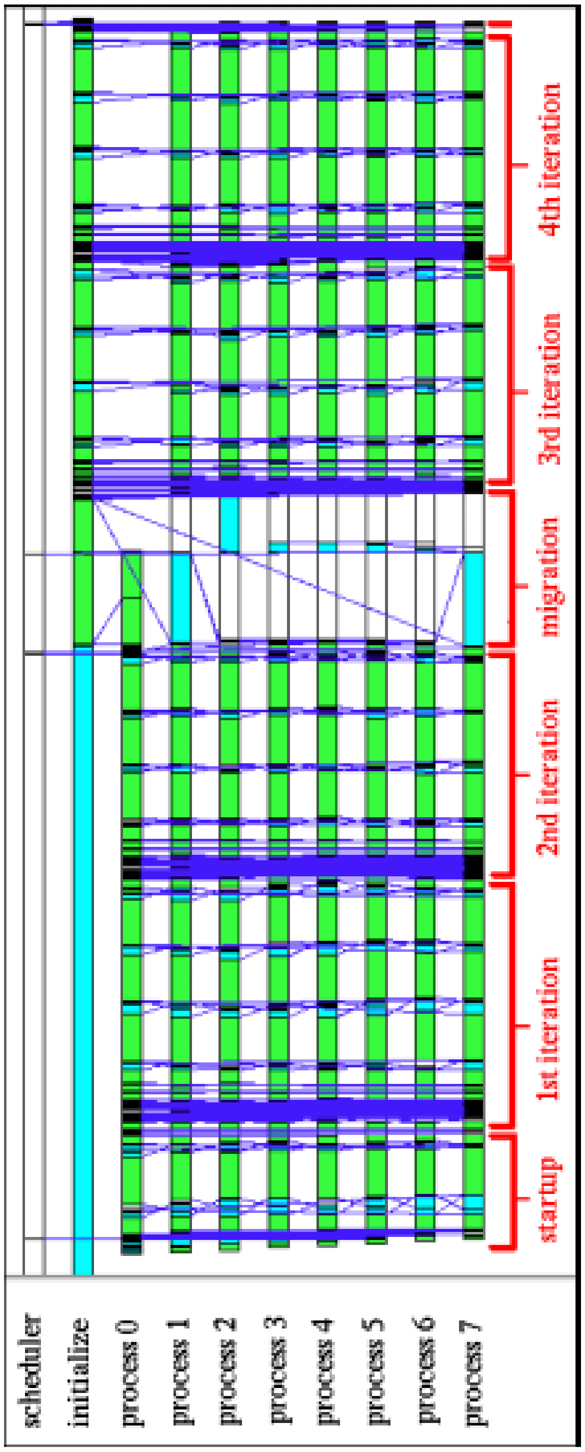


Figure 7.8: A space-time diagram with a process migration.

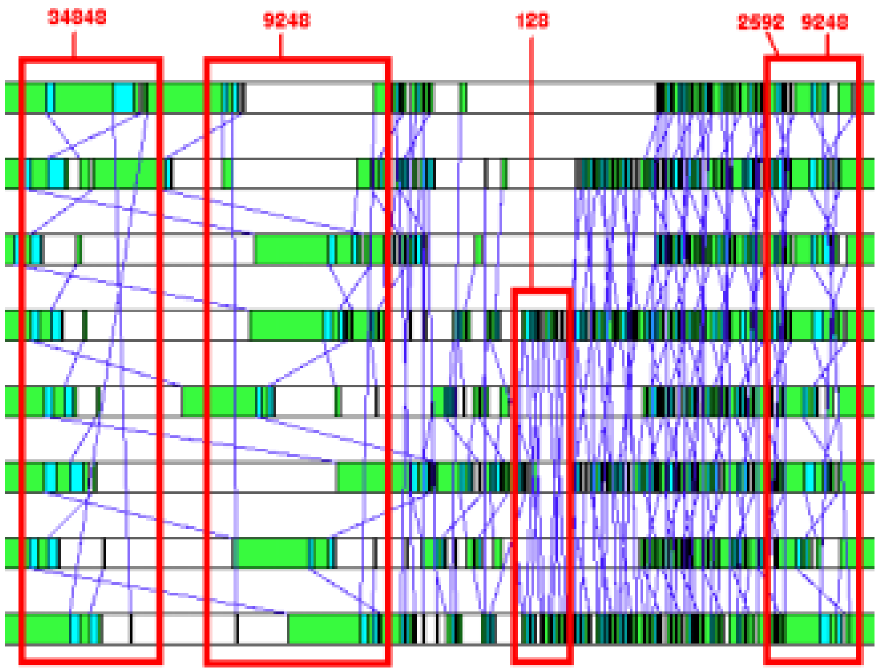


Figure 7.9: The space-time diagram of a multigrid iteration.

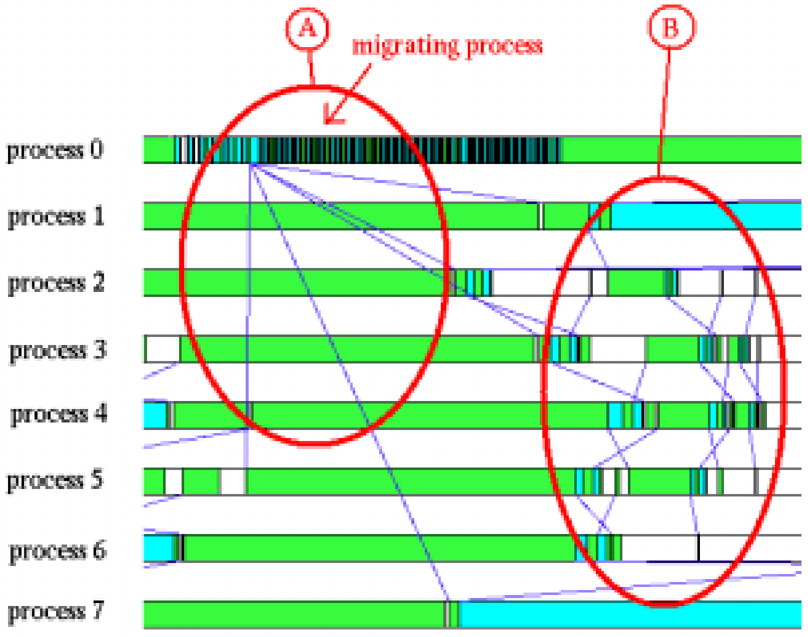


Figure 7.10: A diagram show the beginning of process migration.

does not have to wait for messages, its execution continues. Area B in Figure 7.10 shows an evidence of such execution. In normal operation, the kernel MG process would exchange messages of size 34848 following by 9248 and 2592, etc with its near neighbors. In the area B, some non-migrating processes proceed with the exchanges up to the message size 2592. Then, they have to wait for certain communication to finish before proceeding further until only process 4 can transmit messages of size 800 to its neighbors (area C in Figure 7.11). Beyond this point, due to intensive near-neighbor communication, all non-migrating process becomes idle and waits for process 0 to finish the migration and start sending data.

Finally, following the multigrid algorithm, two messages of size 34848 bytes are sent from process 1 and 7 to process 0 at the start of the third iteration. Since the process 0 is migrating and the communication channels between 0 and 1 and between 0 and 7 are already closed, both senders have to consult the scheduler to acquire location of the initialized process for establishing new connections. Such communication are shown by the two lines captured by label D in Figure 7.11. By a closer analysis of trace data, we find that the communication channels are established before the execution and memory state restoration of the migrating process, allowing the senders (processes 1 and 7) to send their data to the initialized process in parallel to the execution and memory state restoration. Since the send data are copied to low level OS buffers, the sender process can proceed with their next execution so that the computation can continue in the area C. The sent data are received after the restoration finishes, resulting XPVM to display two long lines cut across the migration time frame as shown in area D in Figure 7.11. After that, the migrating process starts resuming its execution, sends two messages of size 34848 bytes back to its neighboring peers, and continues the multigrid computation. These observations confirm that the case study represents general communication situations and validates the proposed communication protocols.

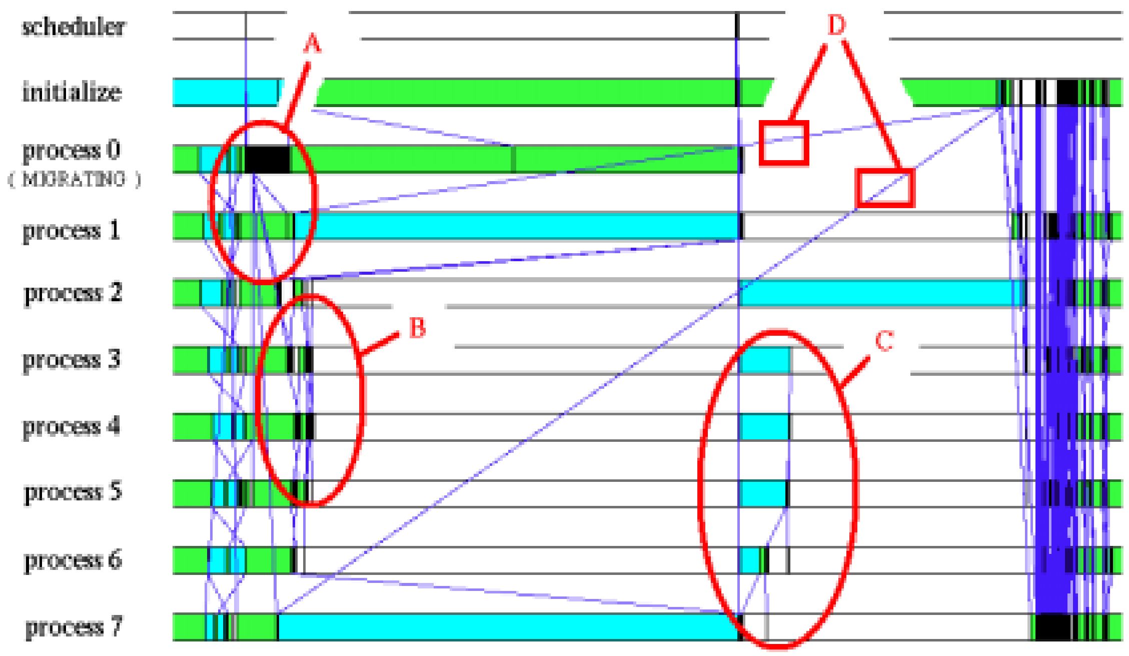


Figure 7.11: The space-time diagram of a process migration.

7.4.2 Overheads

In the second experiment, our objective is to exam the overhead of our communication and migration protocols and the cost of migration. Table 7.4 shows the measured turnaround time of the parallel MG benchmark. All timing reported are averages of ten measurements. `original` means the wallclock time of the original code running on PVM. `modified` is the annotated code running without a migration. Finally, `migration` is the measured time when a process migration is conducted.

Comparing the communication time of our modified program to that of the original code, we see that the overhead is very small. Although over 48 Mbytes of data on the total of 1472 messages are transmitted during execution, the total overhead of the modified code is only about one percent. We believe such small overhead is due to the thin layer protocol design on top of PVM.

By comparing the execution time of the migration to that of the original code, we find that a migration incurs about 16 percents higher turnaround time. Although processes can continue execution while the process `0` migrates, due to the communication characteristic of the kernel MG program, they eventually all have to wait for messages from the process `0` after its migration. The waiting contributes to the migration cost.

Table 7.4: Timing results (in seconds) of the kernel MG program.

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

Table 7.5 gives detailed migration cost. All timing results are the average of ten runs. "Coordinate" indicates overhead of process coordination. "Collect" is the costs of collecting execution and memory states. "Tx" is the cost of transmitting the collected data. "Restore" is the overhead of process state restoration on the initialized process, and, "Migrate" is the average migration overhead. The total migration overhead exhibits a small variation to the summation of the costs of every operation due to miscellaneous implementation costs. From Table 7.5, the migration average time is 2.2922 seconds. The migration transmits over 7.5 Mbytes of execution and memory state data. Both state information are in machine-independent format as defined in [9]. The total migration overhead can be divided into 0.1166 seconds for communication coordination with connected peer processes, 0.73 seconds for collecting the execution and memory state of the migrating process, 0.7662 seconds to

transmit the state to a new machine, and 0.6794 seconds for restoring them before resuming execution.

Table 7.5: Migration performance (in seconds) in a homogeneous environment.

Operations	Time
Coordinate	0.116
Collect	0.730
Tx	0.766
Restore	0.679
Migrate	2.292

7.4.3 Heterogeneity

An experiment is conducted to verify the correctness in a heterogeneous environment. The testbed consists of computers and networks with different powers and speeds. In this experiment, we run the kernel MG program on a heterogeneous testbed where, out of 8 processes, 7 are spawned on Sun Ultra 5 workstations running Solaris 2.6 and 1 is spawned on a DEC 5000/120 workstation running Ultrix. The DEC 5000/120 workstation is significantly slower than the Ultra 5s. Network connections among these machines are also different. All the Ultra 5 machines are connected via a 100Mbit/s Ethernet, while the DEC 5000/120 is connected to the Ultra 5 cluster via a 10Mbit/s Ethernet. We should note that the experiment is performed during the weekend where the utilization of machines and the network traffic are low. We configure the program such that after two iterations of the V-cycle multigrid algorithm, the process on the DEC 5000/120 machine migrates to an idle Ultra 5.

The experimental outputs with and without the migration are identical. The outputs are also consistent to those generated from the homogeneous testbed reported earlier.

Table 7.6 shows the performance of different operations during a process migration. The result is the average of 10 runs. During the migration, over 7.5 Mbytes of execution and memory state are transmitted. The migrating process spends 0.125 seconds to coordinate its connected peer processes, 5.209 seconds to collect the execution and memory state, 8.591 seconds to transfer state information across machines via a 10Mbit/s Ethernet network, and 0.696 seconds to restore the state information on an Ultra 5 machine. The unparallel performance of data collection and restoration is obviously the result of different powers of the two machines.

Table 7.6: Performance (timing in seconds) in a heterogeneous environment.

Operations	Time
Coordinate	0.125
Collect	5.209
Tx	8.591
Restore	0.696
Migrate	14.621

Under the heterogeneous environment, the process migration protocol shows an interesting communication coordination behavior in which two messages are captured and forwarded during the migration. According to the application configuration, the migrating process has to coordinate with 7 other processes to capture all incoming messages to the migrating process before the communication connections can be closed. Because of the significantly slower speed of the DEC 5000/120 machine, the two neighbor processes on the faster Ultra 5 machines already sent messages to the migrating process before the migration started. Thus, the communication coordination of the migration protocol resulted two messages received into the receive-message-list of the migrating process. Therefore, the first part of the communication state contains two messages. The migrating algorithm forwarded these messages to the initialized process and inserted them to the front of the receive-message-list of the initialized process. The communication and migration protocol work well despite the hardware's disparity.

Chapter 8
Conclusions and Future Work

In this dissertation, we have presented the design and implementation of compile-time and runtime systems to support process checkpointing in heterogeneous distributed environments. This dissertation focuses on applying the checkpointing mechanism toward high-performance by means of process migration.

8.1 System Design

There are three major design concepts behind the checkpointing mechanism. First, the program analysis and source code annotation are applied to a source code and transform it into a "migratable" format. We define migration points as locations in the source code where process migration is allowed. The program analysis consists of the migration point analysis, the scheme to choose migration point locations, and the data analysis, the mechanism to find a minimum set of variables whose values have to be transferred at a migration point. For source code annotation, we apply a novel mechanism, namely the data transfer mechanism, which governs the collection and restoration of process state during process migration.

Second, a fundamental technique for heterogeneous process migration, low-level mechanisms for data collection and restoration, is presented. A graph model, namely the Memory Space Representation (MSR) graph, is proposed to identify needed 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 in the MSR graph, respectively. 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. Then, the MSR Lookup Table (MSRLT) data structure is contributed 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. In addition, the representation of pointers in machine independent format is described.

Finally, we have shown the design and implementation of a set of communication and process migration protocols for distributed computation in a heterogeneous environment. The protocols are built on top of a connection-oriented communication model where a connection is established prior to message passing. We have designed and implemented algorithms for send, receive, and process migration operations. These algorithms work collectively to prevent loss of messages and preserve message ordering semantic. In the send algorithm, the sender-initiate technique is implemented so that the sender requests the receiver for a connection. Two vital functionalities are added to the send algorithm to support migration environment. They are the abilities to reconstruct communication channels and to search for the location of a migrated process. In the receive algorithm, we have introduced the receive message list as a user-level buffer that keeps messages arrived before their intended receive operations are executed. The algorithm is capable of assisting a migration of a connected peer process by receiving all messages from the communication channel and disconnecting it. We have also implemented a number of algorithms for process migration. The process initialization algorithm loads processes on potential destination machines to wait for state transfer. The migration request algorithm commands a migrating process to migrate to an initialized computer.

The process migration algorithm is designed to work jointly with the send and receive algorithms to preserve the correctness. It performs the followings:

1. Prohibit additional communication connection during the migration.

2. Coordinate every connected peer to guarantee no message loss and to disconnect existing communication channels.
3. Transfer communication state, following by execution and memory states of the migrating process.
4. Restore the process state on the new computer.

8.2 Software Implementation

We have implemented a number of software components for the runtime system to support heterogeneous process migration. They are 1) the checkpointing library to capture the memory state of the migrating process, 2) the virtual machine daemon to provide basic resource access and process management on top of networks of workstations, 3) the message passing library containing data communication and process migration protocols, and 4) the scheduler to bookkeeping resource utilization in distributed environments.

Three sets of experiments are performed to verify the correctness and practicability of our system design. First, we apply process migration to a parallel matrix multiplication program to show potential performance improvement. Second, we have tested our data collection and restoration mechanisms on three C programs with different data structures and execution behaviors to verify correctness and efficiency of our design. Finally, we have implemented the prototype data communication and process migration protocols by extending the PVM system. We have presented a case study of process migration on the parallel MG benchmark running on homogeneous and heterogeneous testbeds. Experimental results show that our protocols do preserve distributed computation logics and correctly capture and restore the communication state of a process for process migration. They also show the applicability of our design in a heterogeneous environment. The prototype implementation reports small computation and migration overhead and demonstrates the real potential of the protocols.

8.3 Future Work

The need of heterogeneous process migration for future distributed computation is vital [19]. Work is still left to be done in many areas. In the near future, more case studies should be performed on a number of parallel applications with different communication characteristics. We plan to apply our migration system to other parallel benchmarks such as the kernel IS, CG, and FT. Then, we plan to develop a compilation system to support semi-automatic process migration. We believe that the development of such tools will advocate new applications of dynamic programming to distributed network computing.

Bibliography

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

[2] John Reppy Anne Rogers, Martin C. Carlisle and Laurie Hendren. Supporting dynamic data structures on distributed memory machines. *ACM Transactions on Programming Languages and Systems*, 17:233–263, March 1995.

[3] Adam Beguelin and et al. Dome: Distributed Object Migration Environment. Technical Report CMU-CS-94-153, Carnegie Mellon University, School of Computer Science, May 1994.

[4] G. Bilardi and A. Nicolau. Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines. *SIAM Journal of Computing*, 18:216–228, 1989.

[5] 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, August 1995.

[6] Jeremy Casas, Dan Clark, Ravi Konuru, Steve Otto, Robert Prouty, and Jonathan Walpole. Mpvm: A migratable transparent version of PVM. *Computing Systems*, 8(2):171–216, 1995.

[7] K. Chanchio and X.-H. Sun. Efficient process migration for parallel processing on non–dedicated network of workstations. Technical Report 96-74, NASA Langley Research Center, ICASE, 1996.

[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*, August 1996.

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

[10] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed system. *ACM Transactions on Computer Systems*, pages 63 – 75, 1987.

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

[12] E. N. Elnozahy, D. B. Johnson, and Y. M. Wang. A survey of rollback-recovery protocols in message passing. Technical Report CMU-CS-96-181, Carnegie Mellon University, 1996.

[13] Thomas Fahringer and Bernhard Scholz. Symbolic Evaluation for Parallelizing Compilers. In *Proc. of the 11th ACM International Conference on Supercomputing*, Vienna, Austria, July 1997.

[14] Adam J. Ferrari, Stephen J. Chapin, and Andrew S. Grimshaw. 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.

[15] Alice E. Fischer and Frances S. Grodzinsky. *The Anatomy of Programming Languages.* Prentice-Hall, 1993.

[16] Charles N. Fischer and Jr Richard J. LeBlanc. *Crafting A Compiler.* Benjamin/Cummings, 1988.

[17] Message Passing Standard Forum. Mpi: A message-passing interface standard, version 1.1. Available by anonymous ftp from ftp.mcs.anl.gov, June 1995.

[18] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal Supercomputer Applications*, 11(2):115 – 128, 1997.

[19] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann, 1998.

[20] Al Geist and et al. *PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[21] Andrew S. Grimshaw, Wm. A. Wulf, and the Legion team. The Legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1):39–45, 1997.

[22] M. Harchol-Balter and A. Downey. Exploiting Process Lifetime Distribution for Dynamic Load Balancing. *ACM Transactions on Computer Systems*, 15, 1997.

[23] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communication of the ACM*, 21(7):558–565, 1978.

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

[25] Micheal J. Litzkow and et al. Condor–a hunter of idle workstations. In *Proceeding of the 8th IEEE International Conference on Distributed Computing Systems*, pages 104–111, June 1988.

[26] Micheal J. Litzkow, Miron Livny, and Matt W. Mutka. Condor–a hunter of idle workstations. In *Proceeding of the 8th IEEE International Conference on Distributed Computing Systems*, pages 104–111, June 1988.

[27] D. S. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process Migration. Technical report, TOG Research Institute, December 1996.

[28] D. S. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process Migration. Technical report, Hewlett-Packard, December 1998.

[29] ORNL. Available at http:://www.netlib.org.

[30] S. Pakin, V. Karamcheti, and A. A. Chien. Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors. *IEEE Concurrency*, 5, 1997.

[31] J. S. Plank. An overview of checkpointing in uniprocessor and distributed systems, focusing on implementation and performance. Technical Report UT-CS-97-372, University of Tennessee, 1997.

[32] P. Smith and N. Hutchinson. Heterogeneous process migration : The TUI system. Technical Report 96-04, University of British Columbia, Department of Computer Science, February 1996.

[33] G. Stellner. Consistent Checkpoints of PVM applications. Proceeding of the First European PVM Users Group Meeting, 1994.

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

[35] Xian-He Sun and L. Ni. Scalable Problems and Memory-Bound Speedup. *The Journal of Parallel and Distributed Computing*, 19:27–37, 1993.

[36] 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.

[37] 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, November 1994.

[38] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the International Symposium on Computer Architecture*, 1992.

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

[40] Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1996.

Vita

Kasidit Chanchio was born in Cheingmai, Thailand, in 1969. He spent his childhood enjoying nice sea breeze and nature in Sattahip, Chonburi, for about a decade before moving to face reality of life in Bangkok. In 1986, he attended the department of Computer Science at Thammasat University, Bangkok, Thailand, and received his Bachelor of Science degree in 1990. At Thammasat, he has learned not only technical skills, but also the true meanings of human right, freedom, and equity.

Kasidit arrived at Baton Rouge in Fall 1993 to attend the department of Computer Science at Louisiana State University. Three years later, he has earned his Master of Science degree in system science. Since then, he has become very interested in a research in heterogeneous process migration and decided to pursue a doctoral degree in Computer Science in Fall 1996.

His research interests include process migration, parallelizing compiler, parallel processing, mobile agents, distributed operating systems, and distributed computing. He is also an LSU football fan. Geaux Tiger!