HPCM: A Pre-compiler Aided Middleware for the Mobility of Legacy Code*

Cong Du, Xian-He Sun
Department of Computer Science
Illinois Institute of Technology
{ducong, sun}@iit.edu

Kasidit Chanchio
Computer Science and Mathematics Division
Oak Ridge National Laboratory
chanchiok@ornl.gov

Abstract

Mobility is a fundamental functionality of the next generation internet computing. How to support mobility for legacy codes, however, is still an issue of research. The key to solve this outstanding issue is the support of heterogeneous process migration. During the last few years, we have successfully developed mechanisms to support heterogeneous process migration of legacy codes written in C, C++, and Fortran. We present in this paper the design of the High Performance Computing Mobility (HPCM) middleware, the development and implementation of its key components, pre-compiler and its static libraries. Due to the similarity between process migration and checkpointing, the pre-compiler not only makes automatic process migration of legacy codes feasible, but also supports dynamic heterogeneous checkpointing. We perform a set of tests and compare experimental results with Porch, a well-known portable heterogeneous checkpointing system. The experimental results show that our methods are feasible, efficient and very promising.

1. Introduction

Process migration is the act of transferring an active process from one computer to another. The process retains its execution sequence, memory state and communication state during migration. The process is interrupted on the source machine and then resumes its execution at the break point with the same memory state and communication contexts on the destination machine. The break point in the execution sequence where migration occurs is called "migration point". Process migration has many benefits to distributed computing environment, including dynamic load balancing, fault tolerance, data access locality, mobile and pervasive computing. The emergence of grid-computing environment has made process migration even more important and challenging [10]. The grid provides mechanisms to share heterogeneous resources in a wide area network environment. In a grid environment, the cooperating systems are loosely coupled and the resources are widely distributed and highly dynamic. With the help of process migration, the running process can be relocated to approach computation, data, and service resources dynamically. Process migration system helps improve mobility, performance, efficiency and utilization of shared resources in a grid environment. Process migration can be classified into two categories: homogenous and heterogeneous. The heterogeneity can be at the hardware layer, system software layer, communication layer, and runtime environment layer. The grid is a large scale distributed computing environment that features heterogeneity, posing great difficulties and challenges to process migration. For example, the source and destination of a migration may have different computer architectures and instruction formats. For execution state transfer, an application has different binary codes on the source and destination machines. The destination machine cannot directly use the address of migration point stored in the instruction counter of the source machine. Similarly, for memory state transfer, the variable values at the source machine are meaningless to the destination because of different data representations. During a migration, the execution and memory states have to be transferred in a machine independent format.

The High Performance Computing Mobility (HPCM) middleware supports user-level heterogeneous process migration. Several critical mechanisms have been proposed in our previous work [3, 5], including the execution, memory, and communication state transfer mechanisms. Based on these mechanisms, we build the HPCM middleware, an automatic process migration system in a heterogeneous distributed environment. The pre-compiler is an important component that aims to transform a code written in C, or other stack based languages, into a migration capable code. We choose C because of its high performance and popularity. We also build up libraries that can be statically linked with the pre-

* This research was supported in part by national science foundation under NSF grant EIA-0130673, ANI-0123930, ACI-0130458 and by Army Research Office under ARO grant DAAD19-01-1-0432.

compiled applications and provide necessary functionalities. In this paper, we introduce the design of the HPCM middleware and present its implementation and experimental results. In next section, we give an overview of related works on process migration and checkpointing technologies. In section 3, we describe the architecture and components of the HPCM middleware. In Section 4, we introduce the structure and workflow of the pre-compiler system and address several technical issues. The experimental tests are presented and discussed in section 5. In Section 6, the conclusion and future work are presented.

2. Related Research

Intensive research has been done in the area of process migration due to its importance. Some of the early works, such as MOSIX [1], V [2] and Sprite [7], try to combine the migration functionality to cluster operating systems. By migrating a process across workstations, these systems balance workload and provide single system image. These systems rely on specific operating systems and provide services by the kernel extension. The usage of these technologies is limited to a specific system with a specific cluster operating system. To provide transparent access, they use forwarding technology, which keeps user contexts on the igniting host. The igniting host forwards system events such as IPC, file accesses and signals. As a result, the process is not migrated completely. These systems miss some benefits of process migration, including fault tolerance, data access locality, and mobility in grid or pervasive computing. They can only achieve load balancing to a limited cluster environment.

Several other systems have been implemented at user-level such as Condor [16], Dynamite [13] and CoCheck [12], or kernel level such as Linux Zap [17] to support homogeneous process migration. These systems use run-time checkpointing to preserve the memory image of a running process. They assume that the migration is between homogenous machines, where the execution and memory states can be transmitted without being translated or understood. For heterogeneous environments, a pre-compiler is necessary to transfer the type, structure, variable and other required information into a machine independent format. An advantage of these systems is that they are built on top of the operating system layer and can be used on commercial workstations. The second advantage is that through checkpointing mechanisms, they can provide fault tolerance without other instrumentations. There are some outstanding limitations for these systems too. The checkpointed memory images can only be restored on machines with the same architecture. That is, the migration can only be performed on homogeneous platforms.

Most checkpointing systems are designed for homogeneous computing. By capturing the memory image from physical memory or virtual memory, the memory collection and restoration problems can be naturally solved. Some research efforts have been made on developing heterogeneous checkpointing systems. Existing compiler instrumented portable heterogeneous checkpointing systems include Porch [19], PREACHES [15] and [14]. Similar to our system, they support heterogeneity by compiler or pre-compiler. Both checkpointing and process migration systems provide the save and restoration of the process states. However, checkpointing rolls back to a previously checkpoint for fault recovery; migration continues computing till the next migration point to migrate. Checkpointing requires periodic saving of run-time information to the disk, assuming reliable storage, and may have a domino effect in a distributed environment [18, 8]. Migration provides better mobility and efficiency, but requires pre-warning for fault tolerance.

Few early research efforts [20, 9] in heterogeneous process migration have been presented in some early works. They address and discuss several important problems and build their prototype systems. These systems are not widely accepted by the scientific and engineering applications because of their inherent limitations such as range, performance and the integrity.

3. HPCM System Overview

The HPCM middleware is designed to support mobility of legacy codes. It consists of several primary components. They are a pre-compiler, libraries, a console/scheduler, and a run-time environment. The goal is to build an automatic heterogeneous process migration system, which provides a transparent mobility middleware layer. Users can modify the pre-compiler output through the pre-compiler interface and coordinate process operations through the console/scheduler. The libraries include execution and memory state facilities, and the communication library. HPCM can be incorporated into a distributed computing environment such as PVM, MPI, Condor and Grid. Figure 1 shows the structure of the HPCM middleware.

The input of HPCM is the source code of an application. A poll-point is the point where a migration can occur. The pre-compiler chooses poll-points via poll-point analysis with the assistance of the user, if the user chooses to do so. The pre-compiler annotates the source code accordingly and outputs equivalent migration capable code, namely the *annotated* code. The console/scheduler can distribute the annotated code to the destination machine at any time. In general, the annotated code is pre-loaded into a potential destination machine before a migration. The annotated code is then compiled

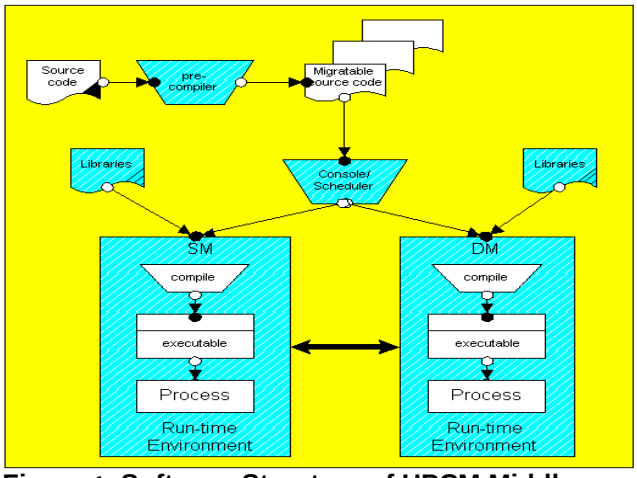

Figure 1. **Software Structure of HPCM Middleware**

and linked with libraries to generate the binary code at the destination machine, which is called pre-initialization. When the migrating process at the source machine requests a migration, the console/scheduler chooses a pre-initialized host from the pool, sends its choice to the migrating process, and initializes a process called initialized process at the destination machine. While receiving the migration command, the migrating process transfers the execution, memory, and communication state to the initialized process. By applying our mechanisms, the process states can be transferred in a pipelined fashion. That is, the data collection, transmission, and restoration can be performed concurrently. Concurrency can save significant time in a networked environment, especially when there are large amount of state data to be transmitted. After transmission, the migrating process is terminated and the initialized process resumes its execution.

3.1. Pre-compiler

We have successfully designed a set of mechanisms to support the process migration in the heterogeneous environments [5]. These mechanisms include the execution, memory and communication state transfer mechanisms. These novel mechanisms have been tested on several practical applications, which confirm their feasibility and effectiveness. To implement these mechanisms, macros and other facilities need to be inserted into the source code. For this purpose, we build a pre-compiler, which transforms the source code to its equivalent migration capable annotated code.

A migration point is a poll-point where a migration occurs. Poll-point analysis is used to determine a set of suitable locations where a migration can be performed safely and efficiently. The pre-compiler performs poll-point analysis to automatically choose suitable poll-points according to the user's requirements. Astute users can make their own decisions based on the pre-compiler's choice. At each poll-point, the process will examine if a migration command has been issued and take action accordingly. Proper choice of poll-points may significantly affect the feasibility and performance of process migration. Firstly, there might be some features in C language that are not safe to migrate at certain locations. These features may lead to errors after migration. Secondly, it is not efficient to migrate at some locations. It may need more memory state data to be transferred or it may need more communication to be redirected to the destination machine. Avoiding those locations as poll-points can save much time during migration. Thirdly, the migrating process should not be executing for a long time before it reaches the nearest poll-point. If there are fewer poll-points in an application, it will take a longer time to reach the nearest poll-point. On the other hand, if there are more poll-points, the cost for checking the poll-points and source code annotation is higher. Poll-point analysis determines an appropriate set of poll-points. It needs to estimate the execution time between poll-points, the amount of data that needs to be transferred, and the overhead of migration. We need to minimize the execution time, migration response time, communication overhead and migration overhead.

3.2. Console/Scheduler

A console is built to monitor and coordinate the running processes. The console accepts user commands to migrate a running process and controls the state transformation. It handles the authentication and authorization problems. It initializes a process, which compiles, distributes the code and starts the run-time environment. The console serves as a commander to running processes and as a user interface to the system administrator. Sometimes for load balancing and fault tolerance, we also need a scheduler to collect current status of performance and resources availability on the source machine and other pooling hosts. When the performance of current hosting machine cannot satisfy the requirements of the users, it will find another host that has more suitable resources. It will also determine when to migrate and where to migrate dynamically. In our system design, we combine the console and the scheduler into one component called console/scheduler. With the console/scheduler, we can choose a suitable destination machine automatically or manually. Users can define conditions as to when the process migration should be invoked or directly issue a command through the console/scheduler interface to migrate at a desired time to a desired machine.

3.3. Libraries and Run-time System

We build a set of libraries and a run-time process working together to provide the functions and run-time

supports needed by process migration. The libraries include a basic library, a data transmission library, and a communication migration library. We can easily extend the HPCM middleware to different platforms by providing more libraries. The basic library contains a set of functions, which are called by the macros inserted by the pre-compiler. These functions help to conduct memory, execution and communication state transfers. The data transmission library contains functions to handle memory and execution transfers over the network. It can be implemented on top of various communication software environments. Currently we have implemented the libraries on top of MPI, PVM and TCP/IP. That is, the memory and execution state can be transmitted across machines using MPI, PVM, or TCP/IP communication facilities. The communication migration library contains the functions needed to handle message passing or network data communication during process migration. We have currently implemented the communication migration library under PVM [11]. However, It also can be extended to support other distributed environments. The run-time system consists of a number of collaborative daemon processes each running on a participant computer within a process migration environment. The console/scheduler starts a daemon on a participant host on demand. An early version of the functions is used [3] to confirm the feasibility of the mechanisms for execution, memory and communication transfer. Recently, we have

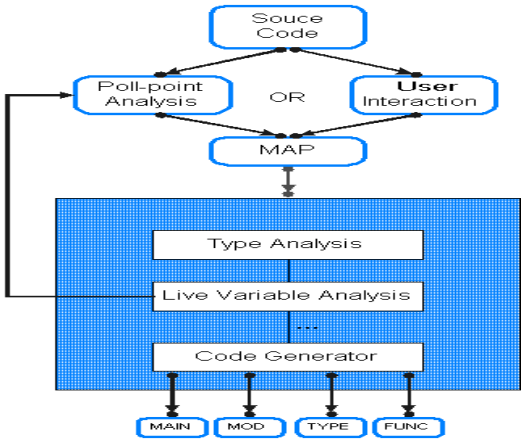

Figure 2. **Pre-Compiler Conceptual Model**

build up a set of libraries based on them and significantly improve their prototypes by providing more functionalities for process migration operations, by making the run-time system more robust and interoperable with other software, and by providing better performance. The run-time system can be bound to different underlying platforms as well.

4. The Pre-compiler and Its Functionality

The pre-compiler is a C-to-C translator, which converts the C source code into its equivalent migration capable C code, and generates related utility files. Figure 2 shows the conceptual model of the pre-compiler. The pre-compiler does the poll-point analysis to select a set of poll-points. It gathers type, structure, variable and memory block information from the source code and performs type analysis, live variable analysis and memory block analysis. Live variable analysis of the pre-compiler also generates the performance prediction and statistical data to evaluate the selection. Users can also choose poll-points and evaluate the poll-points selected by the poll-point analysis manually through the graphic user interface. The GUI is implemented using Java. This process may repeat several times until an appropriate set of poll-points is selected. It annotates the source code and inserts the migration macros and function calls to C source code to form MOD (MODified code). The Code Generator also generates a main function file that takes care of the global data and data type management of the application (*MAIN*), a function definition file (*FUNC*), a type definition file (*TYPE*) containing type information table, a component layout table, as well as saving and restoring functions.

For applications written in C, there are some language features and programming practices that make the process non-migratable. Some of them depend highly on the current execution state and some others depend highly on the underlying computational platform. We called this condition as migration-unsafe. We have to transform these features into their equivalent codes that are safe during migration. We have to use some mechanisms to deal with these features, rewrite this part of program in a different but safe way or use some other mechanisms to gain safe migration. The migration-unsafe features include: usage of pointer, type casting, usage of void pointer, memory allocation and release, *union*, communication, local file access, usage of "sizeof()" operator, function pointer. We need to handle these migration-unsafe problems through the pre-compiler and process migration mechanisms.

Our pre-compiler is implemented based on the framework of Porch (Portable checkpoint compiler), a well-known pre-compiler for a checkpointing system developed by the MIT laboratory of computer science [19]. We adopt the underlying layer including lexical analysis, semantic analysis, framework of data flow analysis and some part of the code generation.

Figure 3 shows the workflow of our pre-compiler. The pre-compiler gets the preprocessed code with *"gcc –E"*, and then performs lexical and semantic analysis to the preprocessed code and builds up the AST (Abstract Syntax Tree). After performing optional poll-point analysis and user interaction one or more times, it detects the migration-unsafe features of the application and transforms them to an equivalent code that is migration-safe, gathers the function information, and performs type, live variable, and memory block analysis. Finally, the pre-compiler generates annotated codes.

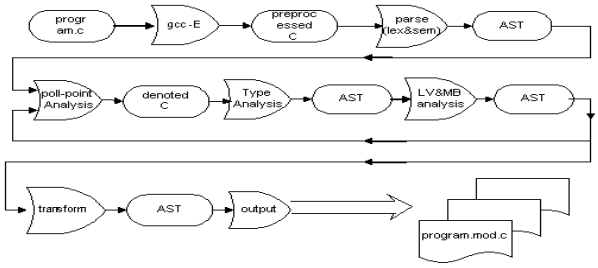

Figure 3. Workflow of the Pre-compiler

4.1. Source Code Annotation

There are six kinds of macros involved in the process of migration. The pre-compiler needs to insert them into the source code. They are *head_macro*, *end_macro*, *jump_macro*, *mig_macro*, *entry_macro*, and *stk_macro*. The *head_macro* is inserted at the beginning of the function body. It put the migration point to the stack of calling sequence. For the migrating process, it registers memory spaces and memory blocks into a global data structure, called MSRLT table; for the main function of the initialized process, it makes connection to the migrating process and initializes the global state of the process. The *end_macro* removes the migration point in the stack of calling sequence. The *jump_macro* follows the *head_macro*. It extracts the calling sequence information and jumps to corresponding location of codes. The *mig_macro* is placed at the poll point. For the migrating process, it collects and transmits global variables; for the initialized process, it receives and restores the values of global variables. The *entry_macro* and *stk_macro* are placed separately before and after the function call in the migration point calling sequence. The *entry_macro* collects and restores the local live variables of the current function before entering the migrating point. The *stk_macro* collects and restores the local live variables of the current function after existing the migrating points.

4.2. Pointers and Dynamic Memory Management

The pointer type in C language poses a great difficulty in process migration. The memory address on the source machine is meaningless to the destination machine. In most cases, we consider the memory piece occupied by a variable or sometimes a group of variables as a memory block. The dynamic allocated memory pieces are also considered as memory blocks. To circumvent the above problem, a logical memory space model MSR (Memory Space Representation) is defined [5].

There could be many memory blocks in the memory space of a big application. In practice, we do not need to register all memory blocks into MSRLT table. The memory blocks registered in the MSRLT shall satisfy one of the following four conditions: dynamic allocated memory blocks; pointers; variables with the types of *struct* and *array*; variables whose addresses have been used as right-value. For dynamic memory management, we replace the system calls of dynamic memory management with our own functions. The MSR function not only requests a piece of memory as required but also registers this memory block to MSRLT table.

4.3. Live variable analysis

A practical computing application may have large number of variables. Some of those variables will never been used after process migration. Transferring those variables to the destination machine will add unnecessary cost. To improve the performance of process migration, we try to find out which variable is useful in future execution and which is not. This process is called live variable analysis [4]. Live variable analysis defines a set of variables whose values are modified before a poll-point and are needed after the poll-point. We perform live variable analysis to global variables and local variables separately to determine the variables that need to be transmitted.

4.4. Struct and Union

Union is one of the three composite types in C. It is widely used in all kinds of applications, especially, in system header files. It will dramatically limit the usage of our method without solving this problem. P. Smith and N. Hutchinson address this problem [20], but the problem is not solved in their system. The difference is caused by the definition of union, which is that every component in a union should occupy the same memory address. In our way, for case C, we do the followings: 1. add a component *int _elem_tag* to the *union*; 2. add a component *union _U* to the *union* and move all the components in the original *union* to *_U*; 3. convert the access (read, write) of the *union* to the access of the component "_U"; 4. record current type in the union to *_elem_tag* after an assignment instruction to a union; 5. check the *_elem_tag* to determine current type of the union while saving/restoring the value of a *union*,, then save/restore the value accordingly.

5. Experimental Testing

We have implemented and tested our system on the following three platforms: a Sun Blade workstation 100 with 1 UltraSparc-IIe 500MHz CPU, 256K L2 cache, 128MB, running SunOS 5.8 operating system (called *w* in this section), a Sun Enterprise 450 server with 4 UltraSparc II 480Hz, 8M cache, 4GB, running SunOS 5.8 operating system (called *s*), and a Dell Precision

Workstation 410MT with 2 Pentium III 500MHz, 512K L2 cache, 768MB, running Redhat Linux 8.0 (called *l*). Their floating-point speed is 32698kflops for the server, 27067kflops for the workstation and 54934kflops for the Linux PC. We have tested the linpack sequential program translated to C by Bonnie Toy that solves a dense system of linear equations with Gaussian elimination [6], and the bitonic program by Joe Hummel, which builds a random binary tree and then sort it. The linpack program includes use of pointer, array and other primary types. The bitonic includes more complex self-defined data structure such as structs and linking pointers. It also includes lots of dynamic memory management operations and recursive function calls, which may incur large memory state data. We configured the matrix size of the linpack as 100, 200, 500, and 1000. The tree size of the bitonic benchmark varies from 1024 to 16384. The communication between the server and the workstation is a 100Mbps internal Ethernet with exclusive use. The server and linux belong to different subnets of our 100Mbps campus network and the communication between them is sometimes interfered non-determinately by other users. Each test with communication is performed ten times at separate time periods to avoid the network interference.

First we test the overhead for the migration capable linpackc application for normal execution. We perform the testing on the server, workstation and linux. The problems scale from 100 to 1000. We find that the overhead of the migration capable code is very low. In most cases, the performance of the linpackc and the migration capable linpackc is almost the same. There is no significant overhead in most testing results shown in Figure 4. The maximum overhead found among these tests is 0.7%, which comes from the 1000x1000 matrix running on the server. These performance results show that generally the overhead is very low when there is no migration occurred. Figure 4 compares the execution time of the linpackc and migration capable linpackc.

Table 1. Homogeneous Process Migration

Seconds	100	200	500	1000
original (*w*)	0.800	5.720	96.475	773.420
original (*s*)	0.793	5.338	75.628	604.676
non-migration (*w*)	0.782	5.699	96.478	772.650
non-migration (*s*)	0.780	5.343	75.353	608.996
migration (*w*=>*s*)	0.813	5.464	77.643	622.620
migration (*w*=>*w*)	0.828	5.750	96.947	774.026
communication data	82240	323440	2007040	8013040
migration overhead (*w*)	3.5%	0.5%	0.5%	0.08%

We test the homogeneous process migration from the workstation to the server and the heterogeneous process migration performance from the server to the Linux PC. Table 1 shows the performance of homogeneous process migration from the workstation to the server and the migration between two workstations. The overheads of the homogeneous migration between the workstations with the same architecture and speed range from 0.08% to 3.5%. For very small application scale, the migration may

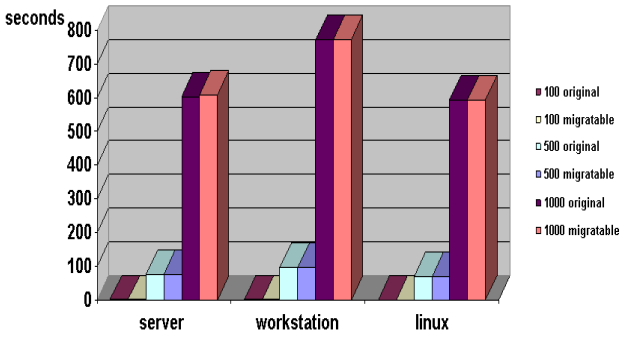

Figure 4. Overhead of Process Migration System

cause higher overhead. For bigger scales, the overheads are from 0.08% to 0.5%. Figure 5 compares the performance of original, non-migration and homogeneous migration.

Table 2. Heterogeneous Migration from Server to Linux

Seconds	500	1000
1. non-migration	75.353	608.996
2. collection	4.708	19.092
3. restoration	4.790	19.334
4. without pipeline 1+2+3	84.851	648.326
5. migration (s=>l)	74.182	610.496

The heterogeneous testing is on the problem scale 1000x1000 and 500x500 matrix. The floating point computing speed of the Linux PC is higher than the server. Table 2 shows the comparisons of performance for the pre-compiled linpack application without migration, collection time, restoration time, migration without pipelining, and migration with pipelining. The collection and restoration time refer to the time of memory state collection and restoration time respectively. Because we pipelined the collection, restoration, and data transmission, the summation of no-migration execution, collection, and restoration time is higher than the actual migration time. Table 2 verifies that our design is efficient and pipelining the collection and restoring of state is beneficial in actual migrations. By overlapping the collection, restoration, and transmission, we save 6%-14% of total execution time or almost 50% of the data collection/restoration time. We also tested process migration for the benefits of load balancing on the server and linux. When the server's one-minute load average is 5.23, the migrated linpackc can save 24.60% time compared with the non-migrated control. When the server's one-minute load average reaches 8.41, the performance gain is over 100%. In this case, migrating a process to a faster machine compensates for the overhead incurred by migration. Because it is difficult to find

heterogeneous platforms with the same computational environments including integer speed, floating point speed, and memory access speed, the exact overhead for heterogeneous migration is not presented here.

Table 3. Heterogeneous Migration of Bitonic

Tree Size	Data size (bytes)			Execution Time (seconds)			Migration Time (seconds)		
level	1	4	8	1	4	8	1	4	8
1024	49416	49932	50620	0.564	0.558	0.570	0.018	0.018	0.018
2048	98568	99084	99772	1.715	1.768	1.786	0.036	0.036	0.047
4096	196872	197388	198076	4.779	4.674	4.742	0.088	0.108	0.108
8192	393480	393996	394684	11.020	11.134	10.994	0.214	0.248	0.253
16384	786696	787212	787900	24.815	24.857	24.724	0.462	0.556	0.557

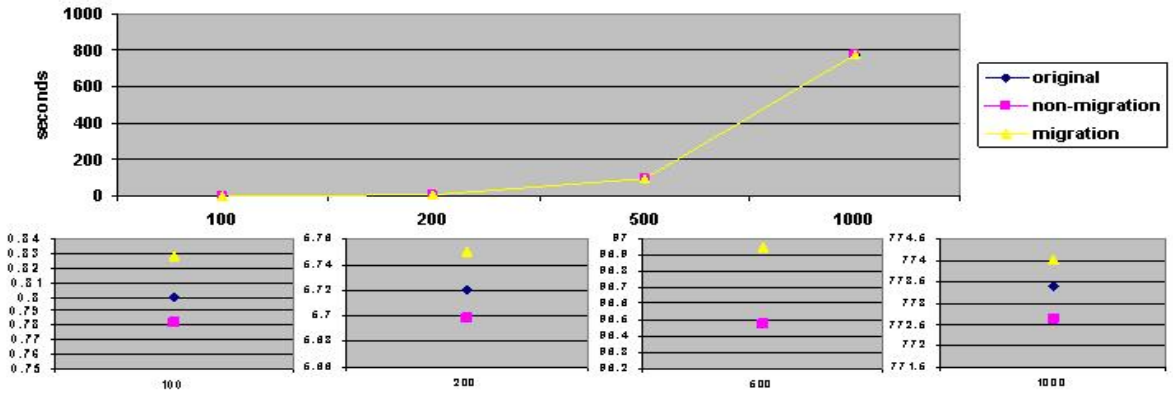

Figure 5. Migration Overhead of Homogeneous Migration (workstation)

We also test the migration performance of the bitonic benchmark from the server to the linux. The bitonic benchmark randomly generates a defined size of tree, then perform bitonic sort on the tree. In our tests, the process repeats for 32 times. There are a large number of pointers referring the memory blocks that need to be transferred because the bitonic benchmark has many memory management operations. The actual migration point is inside a recursive function call, biSort(). The migration command is issued when the execution sequence reaches its depth of 1st, 4th, and 8th level in biSort(). The scale of the application increases with the *Tree Size* from 1024 to 16384. The execution time of the application, the data size of the memory state and the migration time increase accordingly. We compare the experimental data for 1st, 4th, and 8th level for data size, total execution time, and migration time in Table 3. With the increase of the *Tree Size* and the data size, the execution time and migration time increase accordingly. With the increase of the migration point level, the data size increases slowly; the migration time also increases slowly. There is no significant increase for total execution time. Though the application is recursive, the amount of local variable does not incur a dramatic performance overhead for both the execution time and the migration time.

Figure 6 is a comparison of the migration and checkpointing/restoring performance for the HPCM system and Porch, a well-known portable heterogeneous checkpointing system [19]. Porch provides fault tolerance by checkpointing the process state to local reliable disks. The recovery can be performed both on local or remote system. The recovery machine can be both homogenous and heterogeneous. But Porch has the limitations caused by its data conversion mechanisms. They use a data structure called structure metric to provide a specification of the data layout at runtime to accomplish data representation conversion. This means that Porch has to know the data representation format at pre-compile time. The pre-compiled application is static to the given format of data. In the other words, Porch has to know the type of the data format at the source machine before compiling the application on the destination machine. The pre-compilation has to be performed for each source or destination machine and for each data format. Porch does not support the type of *union*. In the contrary, for the HPCM middleware, we do not need to know the architecture of source or destination machine before migration. We can dynamically configure the system at run time. Both the systems are using the pre-compiler technology to solve the problems caused by heterogeneity. In normal execution, Porch generates more overheads compared with HPCM pre-compiler. Figure 6 shows that the performance of HPCM process migration is better than Porch checkpointing. The mechanisms used by HPCM can be used for dynamic heterogeneous checkpointing as well.

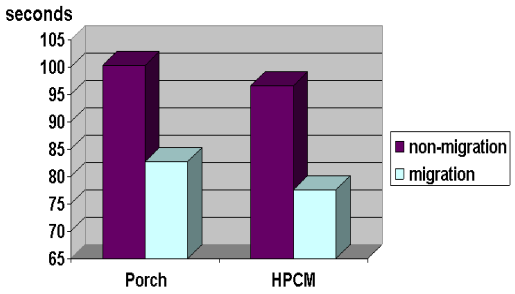

Figure 6. Performance of HPCM and Porch

6. Conclusion and Future Work

This paper presents our recent progress in supporting mobility of legacy codes through heterogeneous process migration. First we introduce the background of process migration and compare the heterogeneous process migration with checkpointing systems. We describe the design of the HPCM middleware and its primary components. Then we introduce the conceptual model, the implementation and the workflow of the pre-compiler. The performance results show that the HPCM middleware is efficient for both the migration and non-migration conditions, and has its real potential in checkpointing as well as in mobility. Heterogeneous process migration of legacy codes is an outstanding research issue. The success of the design and implementation of the pre-compiler is a significant step towards a practical solution for the outstanding issue.

There are still some features in our design that are left for future work. The performance of the HPCM system is highly depending on the amount of memory states to be transferred. This problem is worsened when the application is recursive in major. Local variables need to be transferred for each level of recursive call. We need to further improve the system and tune it for better performance. Currently, the algorithm of poll-points selection is coarse-grained and simple. It needs user interaction for better performance. We still need to refine the poll-point analysis module to select the poll-point wisely and improve the console/scheduler component of the current prototype implementation.

References

[1] A. Barak and R. Wheeler, "Mosix: An Integrated Multiprocessor UNIX", in Proceedings of Winter 1989 USENIX Conf., pp. 101-112, San Diego, CA, Feb. 1989.

[2] D. Cherton, "The V Distributed System", *Communications of the ACM*, 31(3): 314-333, March 1988.

[3] K. Chanchio and X. H. Sun, "Communication State Transfer for Mobility of Concurrent Heterogeneous Computing", in Proceedings of *the International Conference on Parallel Processing* (ICPP 2001, Best Paper Award), September 2001.

[4] M. J. Wolfe, "High Performance Compilers for Parallel Computing", *Addison-Wesley*, 1995.

[5] K. Chanchio and X. H. Sun, "Data collection and restoration for heterogeneous process migration", *Sofware—Practice and experience*, 32:1-27, April 2002.

[6] J. Dongarra, "The Linpack Benchmark: An Explanation", in Proceedings of *the 1st International Conference on Supercomputing*, E. N. Houstis, T. S. Papatheodorou, and C. D. Polychronopoulos, eds., pp. 456-474, Athens, Greece, June 8-12, 1987, Springer-Verlag 1988.

[7] Frederick Douglis, "Transparent Process Migration in the Sprite Operating System", PhD thesis, University of California, Berkeley, Sep. 1990.

[8] E. Elnozahy, L. Alvisi, Y. Wang and D. B. Johnson, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems", *ACM Computing Surveys*, 34:3, September 2002, pp. 375-408.

[9] M. M. Theimer and B. Hayes, "Heterogeneous Process Migration by recompilation", in Proceedings of *the 11th IEEE International Conference on Distributed Computing Systems*, June 1991.

[10] I. Foster, C. Kesselman, J. Nick and S. Tuecke, "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration", June 2002.

[11] A. Geist et al, "PVM: Parallel Virtual Machine: A users' guide and tutorial for networked parallel computing", MIT press, 1994.

[12] Georg Stellner, "CoCheck: Checkpointing and Process Migration for MPI", in Proceedings of *the10th International Parallel Processing Symposium*, April 1996.

[13] K. A. Iskra, F. van der Linden, Z. W. Hendrikse, B.J.Overeinder, G.D. van Albada, and P.M.A. Sloot, "The implementation of Dynamite - an environment for migrating PVM tasks", *Operating Systems Review*, vol. 34, July 2000.

[14] Feras Karablieh and Rida A. Bazzi, "Heterogeneous Checkpointing for Multithreaded Applications", in Proceedings of *21st IEEE Symposium on Reliable Distributed Systems*, Oct. 2002.

[15] K. F. Ssu and W. K. Fuchs, "Portable Recovery and Checkpointing in Heterogenous Systems", in Proceedings of *IEEE Fault-Tolerant Computing Symposium*, pp.38-47, June 1998.

[16] M. Lizkow, M. Livny, and T. Tannenbaum. "Checkpoint and Migration of UNIX Processes in the Condor Distributed Environment", April 1997.

[17] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh, "The Design and Implementation of Zap: A System for Migrating Computing Environment", in Proceedings of *the 5th Operating System Design and Implementation*. Dec 2002.

[18] B. Randell, "System Structures for Software Fault Tolerance", *IEEE Transactions on Software Engineering*, Volume SE--1, Number 3, pp. 221-232, June 1975.

[19] B. Ramkumar, V. Strumpen, "Portable Checkpointing for Heterogeneous Architectures", in Proceedings of *the 27th International Symposium on Fault-Tolerant Computing - Digest of Papers*, Seattle, WA, June 1997.

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