Hindawi
Mathematical Problems in Engineering
Volume 2017, Article ID 6547638, 10 pages
https://doi.org/10.1155/2017/6547638

Research Article

A Migration Method of MPI Program Combining Local Library Replacement and Instruction Translation

Nan Li, Jianmin Pang, and Zheng Shan

State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, Henan 450002, China

Correspondence should be addressed to Jianmin Pang; jianmin_pang@hotmail.com

Received 18 December 2016; Revised 10 April 2017; Accepted 20 April 2017; Published 4 June 2017

Academic Editor: Alberto Borboni

Binary translation acts as a main method used to solve software compatibility among different instruction set architectures (ISAs), yet the main objects that the binary translator deals with are serial programs but not parallel programs. We propose a hybrid method combining local library replacement and instruction translation based on a formal model built to describe the equivalent when migrating MPI programs between different clusters. The shared codes in a MPI program (MPI library function call) are treated by executing local libraries, and the other parts are done by dynamic binary translation. Also, during the course of dealing with local library functions, we propose a method of program flow redirection by designing two algorithms along with hierarchically encapsulating local libraries. A framework called MPI-QEMU is designed to implement migrating MPI program of 64 bits from X86-64/Linux platform to the domestic SW platform which is verified by experiment.

1. Introduction

Dating back to the 1960s, the binary translation technology [1–3] was generated and developed as a tool for equivalent migration of programs. Binary translation is defined as an equivalent migration process of the instruction sequence from one machine to another. Binary translation can be divided into three types according to the way of translation [4]: interpretive execution, static translation, and dynamic translation [5]. Interpretive execution applies the one-by-one instruction simulation; static translation applies the mode of "executing after translating"; dynamic translation applies the mode of "executing while translating." With the production of multicore processors, this enabled the combination of parallel processing and binary translation to further improve the efficiency of program migration [6]. For example, PQEMU [7] uses the multithreading mechanisms for translation; HQEMU [8] uses multiple cores to run the binary translator and dynamic optimizer, respectively, in a multicore platform, so as to implement the coordination of the translator and LLVM-based optimizer. As the new ISA computer system increasingly attaches great importance to high performance applications, binary translation is applied to the processing of a parallel program and the CUDA program [9]. The work [10] proposes a dynamic binary translation framework, Ocelot, based on MCUDA [11], which implements the dynamic translation of the CUDA program to a multicore CPU platform. The work [12] points out the related research lacking in implicit synchronization processing in the process of translating the CUDA program to a multicore platform and puts forward the implicit synchronous detection method based on dependency analysis. The author proposes a method of dividing and conquering in [13] and uses static translation to implement migrating CUDA binary program to the domestic SW platform [14].

MPI (Message Passing Interface) [15] program has been occupying a large share in the parallel program market, the successful migration of which has an important demonstration and guidance significance. However, little related research was carried out both at home and abroad, and there is almost no research result that can be used as a reference. As a result, the migration of the MPI program faces some difficulties and challenges. The first is how to properly deploy the MPI program binary translator. The processing object of the traditional binary translator is a serial program, whose runtime environment is a standalone machine, so such translator will be deployed under the environment of standalone machine, while the MPI program is a parallel program, whose runtime environment is a cluster; the premise of program running is to deploy a copy of the MPI program on each node of the cluster, which requires a binary translator deployed on each node. Therefore, as a whole, there are multiple MPI program binary translators working at the same time and each translator is independent to translate the MPI program on the same node. The second challenge is how to ensure synchronism and consistency among the MPI processes at the time of dynamic translation. The translator is only in charge of loading and dynamic translation of the MPI program on the node, and it does not need to interact with the MPI program or with the translator on other nodes. Therefore, the translator itself does not have to be a MPI program. Although the translator encapsulated into the MPI process can also be used as a solution, there is no need to do so considering the additional encapsulation and communication cost. Indeed, it is the MPI process that performs real message interaction which is translated by the translator on different nodes. Interaction among MPI processes is implemented by calling the MPI library function. If the instruction translation is applied to process the MPI library function, the dynamic generation of the translation codes makes it very difficult to maintain the communication among the MPI processes synchronous and consistent in time and space. In addition, when there are a large number of MPI library functions contained by the MPI program to be translated, this may cause inefficient translation and there will be repeated translation of the same library functions. Thus, the processing of the MPI library function is the key to the migration of the MPI program and is also the focus of this paper.

In view of this, this paper carries out a study on the passive MPI program migration. Firstly, it puts forward a formal representation of the binary translation of the MPI program, based on which a MPI program migration model MPI-QEMU is designed, so as to propose a migration method of MPI program combining code-sharing local library replacement and instruction translation. It can combine the dynamic instant translation and processes message interaction protocol, to intercept and redirect the message interaction function call during the course of dynamic translation. And, also, it shields the interaction details among the MPI processes and implements the transparency processing of the MPI library.

Aiming at solving the dynamic binary translation of the MPI program, the main contributions made by this paper include the following:

(1) This paper proposes a formal representation method of the MPI program binary translation, which provides theoretical support for the MPI program equivalent migration.

(2) A migration method of MPI program combining the local library replacement and the instruction translation is designed to solve the migration of the MPI program.

(3) This paper proposes a method of program flow redirection during the course of dynamic translation, to implement the local replacement of library function call.

(4) A project called MPI-QEMU is designed which implements migrating MPI binary application from X86 platform to the domestic SW platform, which expands the scope of the software supported by a domestic supercomputer to a certain extent.

Section 2 of this paper gives the formal representation of the MPI program binary translation; Section 3 introduces a migration method of MPI program combining local library replacement and instruction translation; Section 4 expounds the implementation process of local library replacement; Section 5 shows the experimental data. In this paper, the installation platform of the binary translation system is called the local platform; the platform to be simulated is called source platform; the program to be translated is called source program and it is a binary executable file; the translated program is called the local program; in addition, the domestic SW (ShenWei) processor mentioned in this paper is a Chinese processor with independent research and development and has been used in a Chinese supercomputer system. On November 14, 2016, the Chinese Sunway TaihuLight supercomputer using the SW processor won the championship in new ranking published by the site https://Top500.org in Salt Lake City, USA; the proposed approach in this paper requires Linux as the OS and is not OS-independent.

2. The Formal Representation of the MPI Program Migration

MPI is a parallel process message interaction interface widely used in distributed memory architecture. In fact, it is a standard specification of a message passing function library, to define such interface library in the form of independent language. It supports a variety of programming languages like C, C++, and Fortran, with different versions of implementation on multiple platforms; the predefined message operation function is used to complete the sending and receiving of information and implement the parallel processing of tasks.

Indeed, the migration of the MPI program is to make a MPI program run on different architecture clusters keeping the semantic consistency. In order to better understand the migration of the MPI program and find a solution, this section uses the formalism method. First of all, the problem of MPI program running on a single cluster is represented by formalization, to introduce quintuple array representing the cluster and the instruction sequence interpretation function representing the program running on a cluster; secondly, according to the computable equivalence theory of Turing machine, the mapping of the program among the clusters is represented by formalization, that is, the equivalence of a program running on different clusters. Through the mapping process from the source platform instructions to the local platform instructions, a binary translator prototype for general program migration among the clusters is represented. Then, the composition of the MPI program is represented by formalization, based on which the binary translator is improved. Finally, the method of MPI program migration is derived.

2.1. The Program Running on a Cluster. The MPI program mainly runs in a cluster environment, which usually consists of multiple computing nodes; the abstract representation of the clusters needs to consider the machine state of multiple nodes. In order to share information between the processes of different computing nodes, the data must be sent, received, or broadcasted through the network; the interaction between the nodes is the fundamental characteristic to distinguish between the cluster and a group of independent computers. The cluster C composed of K nodes can be represented by the following quintuple array:

$$C = \left(S^k, I^k, \theta^k, A, \alpha\right). \quad (1)$$

In the formula,

$S^k = S_1 \times S_2 \times \cdots \times S_k$ represents the state of the cluster, a *Kd* Cartesian product composed of the state of K nodes;

$I^k = I_1 \times I_2 \times \cdots \times I_k$ represents the instruction set of the cluster, a *Kd* Cartesian product composed of the instruction set of K nodes;

$\theta^k = \theta_1 \times \theta_2 \times \cdots \times \theta_k : S^k \times I^k \rightarrow S^k$ represents the state transition of the cluster, an interpretation function of the cluster to the instruction set in a state;

A represents the cluster interactions, including all instructions that can change the state of the cluster; the interactions between the nodes of the cluster are implemented by the elements of the set and the message is sent or received through the elements of the set;

$\alpha : S^k \times A \rightarrow S^k$ represents the interpretation function of the cluster to the message interaction in a state.

Since all $a \in A$ operations of the cluster can be simulated by the I^k sequences, the representation of a cluster can be simplified. Let $C = (\mathbf{S}, \mathbf{I}, \mathbf{\Theta})$ represent the cluster composed of m computing nodes, in which

$\mathbf{S} = s_1 \times s_2 \times \cdots \times s_m$ represents the set of cluster states;

$\mathbf{I} = i_1 \times i_2 \times \cdots \times i_m$ represents the set of the instruction vectors of the cluster;

$\mathbf{\Theta} = \theta_1 \times \theta_2 \times \cdots \times \theta_m : \mathbf{I} \times \mathbf{S} \longrightarrow \mathbf{S}$ represents the interpretation function of the cluster to execute the instruction vector.

When the cluster executes the program, a continual instruction sequence that needs to be executed can be represented as follows:

$$\begin{gathered}\mathbf{\Theta}\left(\mathbf{I}_{n+k}, \ldots, \mathbf{\Theta}\left(\mathbf{I}_{n+1}, \mathbf{\Theta}\left(\mathbf{I}_n, s_n\right)\right)\right) = s_{n+k}, \\ \mathbf{I}_n, \mathbf{I}_{n+1}, \ldots, \mathbf{I}_{n+k} \in \mathbf{I}.\end{gathered} \quad (2)$$

Let $t = \langle \mathbf{I}_n, \mathbf{I}_{n+1}, \ldots, \mathbf{I}_{n+k} \rangle$ represents an orderly sequence of the instruction vectors, $t \in \mathbf{I}^*$; the implementation process of t can be represented by the following function:

$$\mathbf{\Theta}^* : \mathbf{I}^* \times \mathbf{S} \longrightarrow \mathbf{S}. \quad (3)$$

So, the execution process of the instruction vector with the length of k can be represented as

$$\mathbf{\Theta}^*\left(\mathbf{t}, \mathbf{s}_n\right) = \mathbf{s}_{n+k}, \quad (4)$$

where the function $\mathbf{\Theta}^*$ represents the running of the program in the cluster.

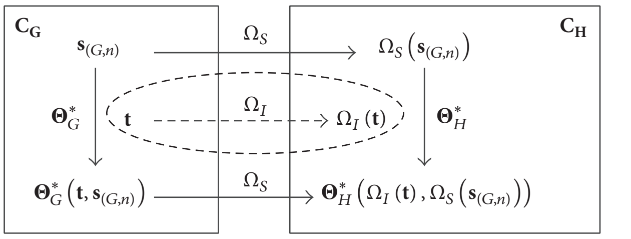
$\mathbf{C_G}$
$\mathbf{C_H}$
$\mathbf{s}_{(G,n)}$
Ω_S
$\Omega_S(\mathbf{s}_{(G,n)})$
$\mathbf{\Theta}_G^*$
$\mathbf{t}$
Ω_I
$\Omega_I(\mathbf{t})$
$\mathbf{\Theta}_H^*$
$\mathbf{\Theta}_G^*(\mathbf{t}, \mathbf{s}_{(G,n)})$
Ω_S
$\mathbf{\Theta}_H^*(\Omega_I(\mathbf{t}), \Omega_S(\mathbf{s}_{(G,n)}))$

FIGURE 1: Formal representation of migrating general programs between clusters.

2.2. The Mapping of the Program among the Clusters. It can be obtained by the computable equivalence of Turing machine that any programs running on a universal Turing machine can be computed by another universal Turing machine. To apply the theory to the cluster environment, we can conclude that a program running on the cluster of the source platform can be simulated by the local cluster, and the simulation process is implemented by executing the source platform instructions on a local platform. The instructions execution can lead to the change of state of the cluster. To this end, the state mapping function and instruction sequence mapping function are, respectively, defined as follows:

$\Omega_s : \mathbf{S}_G \rightarrow \mathbf{S}_H$: mapping the cluster state of source platform to that of the local platform.

$\Omega_I : \mathbf{I}_G^* \rightarrow \mathbf{I}_H^*$: mapping the instruction sequence of the program in the source platform cluster to that of the local platform.

Let C_G represent the cluster of source platform; let C_H represent the local cluster; let $S_{(G,n)}$ represent the initial state of C_G; and let $\mathbf{t}$ represent the instruction sequence of the program on C_G. Figure 1 shows the process of a program of source platform cluster mapped to a local cluster. C_G uses $\mathbf{\Theta}_G{}^*$ to explain and execute $\mathbf{t}$, causing the change in C_G state; Ω_s maps $S_{(G,n)}$ to derive the initial state of C_H, $\Omega_s(S_{(G,n)})$; Ω_I maps $\mathbf{t}$ to derive the instruction sequence of C_H, $\Omega_I(\mathbf{t})$; C_H uses $\mathbf{\Theta}_H{}^*$ to explain and execute $\Omega_I(\mathbf{t})$, causing the change in C_H state. As shown by the dotted box of Figure 1, the process of transformation from $\mathbf{t}$ to $\Omega_I(\mathbf{t})$ is actually an instruction translation process, which outlines a binary translator prototype for general program migration among the clusters.

2.3. The Composition of the MPI Program. The foregoing section introduces the migration of general program among the clusters. However, the special nature of the MPI program determines that the instruction translation cannot be simply regarded as the migration method of the MPI program.

MPI program running on the cluster contains a lot of interaction operations. Each interaction operation corresponds to a MPI function of the platform; these interaction operations have a limited number but are called frequently. Since the MPI is an open-source cross-platform agreement, these interaction operations are shared.

Let W_G represent the program of the source program C_G; let W_H represent the program of the local program C_H;

let W_{share} represent the cross-platform part of the program. W_{share} may vary with different platforms. So, we define it depending on the platform.

Definition 1. $\mathbf{W}_{\text{share}}(\mathbf{C}_G, \mathbf{C}_H) = \{w$: the cross-platform part matching the source code of W_G in the source platform $C_G\}$.

Correspondingly, $\mathbf{W}_{\text{share}}(\mathbf{C}_H, \mathbf{C}_G)$ is the cross-platform part in the platform C_H.

Let Ω_{rep} be the injective function from the sequence $\mathbf{W}_{\text{share}}(\mathbf{C}_G, \mathbf{C}_H)$ to the corresponding sequence $\mathbf{W}_{\text{share}}(\mathbf{C}_H, \mathbf{C}_G)$:

$$\begin{gathered}\Omega_{\text{rep}} : \mathbf{W}_{\text{share}}(\mathbf{C}_G, \mathbf{C}_H) \longrightarrow \mathbf{W}_{\text{share}}(\mathbf{C}_H, \mathbf{C}_G) \\ \forall w \in \mathbf{W}_{\text{share}}(\mathbf{C}_G, \mathbf{C}_H),\ \Omega_{\text{rep}}(w) \in \mathbf{W}_{\text{share}}(\mathbf{C}_H, \mathbf{C}_G).\end{gathered} \tag{5}$$

Since the sequence w and the sequence $\Omega_{\text{rep}}(w)$ are compiled from the same codes shared, they have the same program semantics and function.

Therefore, in the migration of the MPI program from the source platform C_G to a local platform C_H, the translation replacement function $\Omega_{\text{rep}}()$ instead of instruction translation can be used for the W_{share} part.

Definition 2. $\mathbf{W}_{\text{private}}(\mathbf{C}_G, \mathbf{C}_H) = \{w$: the private part matching the source code of $\mathbf{W}_G$ in the source platform $\mathbf{C}_G$, not shared with the local platform $\mathbf{C}_H\}$.

According to Definitions 1 and 2,

$$\begin{gathered}\mathbf{W}_G = \mathbf{W}_{\text{share}} \cup \mathbf{W}_{\text{private}}, \\ \phi = \mathbf{W}_{\text{share}} \cap \mathbf{W}_{\text{private}}.\end{gathered} \tag{6}$$

$\mathbf{W}_G$ represents the program of source platform and consists of $\mathbf{W}_{\text{share}}$ and $\mathbf{W}_{\text{private}}$, which can be processed by different strategies, so as to derive the migration method of the MPI program.

2.4. The MPI Program Migration. The program logic of MPI executable file in the source platform is unique and difficult to analyze, but the behavior of calling the shared library function is predictable. When the MPI program is migrated from the source platform C_G into the local platform C_H, different strategies are applied for the $\mathbf{W}_{\text{share}}$ part and the $\mathbf{W}_{\text{private}}$ part, which together constitute the MPI program. The translation replacement function $\Omega_{\text{rep}}()$ is used for the $\mathbf{W}_{\text{share}}$ part, while the instruction translation function $\Omega_I()$ is used for the $\mathbf{W}_{\text{private}}$ part, for processing. The binary translation function is defined as follows:

$$\Omega_{\text{trans}}(w) = \begin{cases}\Omega_I(w), & w \in \mathbf{W}_{\text{private}} \\ I_{\text{prologe}}, \Omega_{\text{rep}}(w), I_{\text{epiloge}}, & w \in \mathbf{W}_{\text{share}}.\end{cases} \tag{7}$$

When simulating to execute the program of the source platform in the local platform C_H, for the $\mathbf{W}_{\text{private}}$ part, the Ω_I function is directly used to translate the instructions of the source platform to the local platform; for the $\mathbf{W}_{\text{share}}$ part, the local library provided by the local platform is used for processing, that is, to implement it through the function Ω_{rep}. In the local platform, some additional processing work shall be done to ensure the consistency of data during the course of platform switching. I_{prologe} represents the operation of switching the local platform C_H to the source platform; I_{epiloge} represents the operation of switching back to C_H after the simulated execution of the corresponding code. The two instruction sequences are often used to store the register state to memory, pass the parameters to the mapped instructions, and preread the register values and memory information; usually, there are only a few instructions like these, which are frequently called in dynamic binary translation. The above model makes it possible to migrate the MPI program.

3. A Migration Method of MPI Program Combining Local Library Replacement and Instruction Translation

3.1. The Migration Model of the MPI Program. As mentioned in the above section, the migration of the MPI program can be classified for processing according to the code types of the constituent part, based on which the migration model shown in Figure 2 is established. The model ignores the underlying implementation details and shows a method combining the local library replacement and instruction translation from the macro perspective. The MPI program in the source platform to be translated can be deemed to be composed of several parts divided by the MPI function; during the course of migrating the MPI program, for the MPI function call, it belongs to the part of cross-platform, so that it can be processed by Ω_{rep} function (i.e., the local library replacement method); for the non-MPI function call, it can be processed by Ω_I function (the traditional dynamic translation method).

3.2. The Framework and Implementation Process of Migrating MPI Program. Based on this model, this paper constructs a framework for migrating MPI program called MPI-QEMU on the basis of the open-source dynamic binary translator called QEMU. MPI-QEMU can call local MPI library functions through the method of software instrumentation, and the system framework design is as shown in Figure 3.

The implementation of MPI-QEMU is as follows: firstly, the MPI executable file to be migrated is loaded into the local memory through the loading module, during which information extraction is carried out against the MPI function calls, so as to store the related message of MPI function call in the executable file into the special structure, for example, function name, function call address, and function address. Then, taking basic block as a unit, the source binary file is disassembled to generate the corresponding intermediate code; in case of the MPI function calls, the program control flow is redirected according to the function call address identified in the loading process, to generate and execute the instruction sequence that calls the local MPI library function. Otherwise, the intermediate instruction is translated to local instructions for execution in accordance with the rules of translation; during the course of execution, the basic block translated is stored in the code cache for reuse.

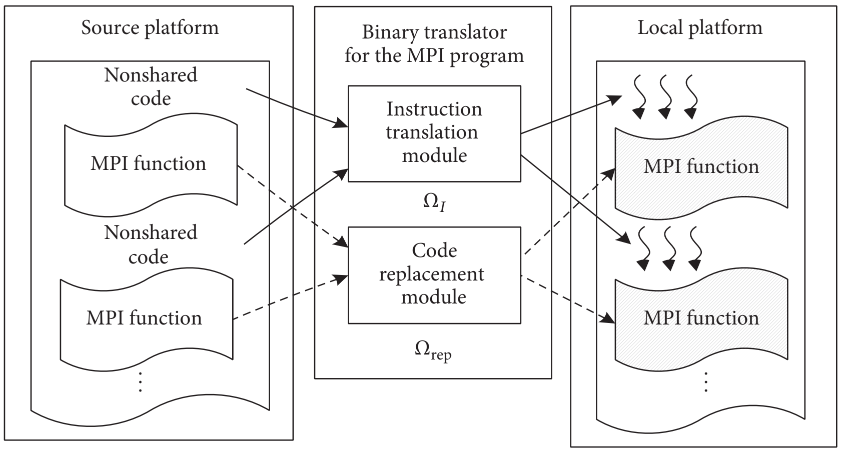

FIGURE 2: The migration model of the MPI program.

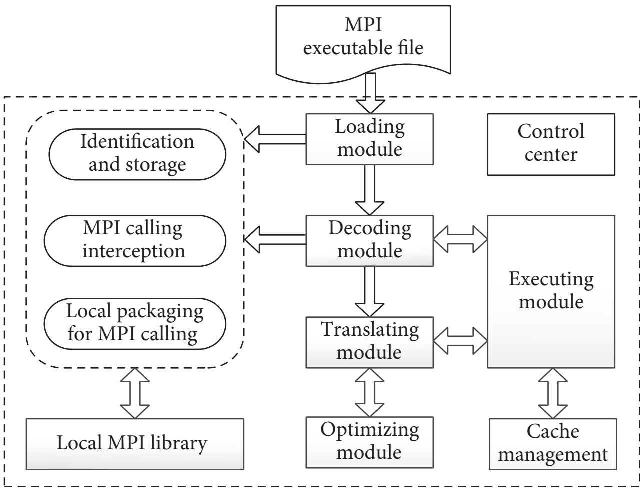

FIGURE 3: The framework of MPI-QEMU.

```
struct func_info{
char * func_name;
unit64_t func_call_addr;
unit64_t func_addr;
struct func_info * next;
}:
```

ALGORITHM 1: Information storage structure on library function calls.

On the local platform, the MPI executable file is executed in a simulated environment realized by the dynamic binary translator without considering the implementation details of the underlying library; when MPI-QEMU processes the calling of the MPI library function, it is to use the local MPI library function to simulate executing instead of dynamically translating them without the consideration of message sending and receiving. Accordingly, under this framework, the dynamic binary translation is the combination of the executable file and the local library; the executable file indirectly calls the local codes to implement the message interaction.

4. The Implementation of Local Library Replacement

The processing of MPI library function is the key to implement the MPI program migration. This paper puts forward a method of program flow redirection, to identify the MPI function call in the process of disassembling instruction and transform the program into local execution. It mainly includes three stages: the first is the library function identification and storage at the program loading stage, to identify the library function call through parsing program and cache the name of library function, call address, and other key information. The second is the opcode interception at the translation stage, comparing the opcode information obtained by the disassembling instruction with the cache information; if the instruction type is the call for MPI library function, the program will go to place of the local MPI function encapsulation; otherwise, the instruction translation will be done. The third is the calling of local MPI library function for execution at the execution stage.

4.1. Library Function Identification and Storage. At the program loading stage, all library functions of the program are identified and cached; the cache information includes the name of the library function, the library function call address, and the MPI function address. The traditional implementation is to use the open-source tools of the Linux platform such as Readelf and Objdump to carry out auxiliary parsing, which needs the manpower's participation. This paper designs and implements an automatic parsing extraction algorithm of library function. Through reverse parsing of MPI executable program, the corresponding information is stored in the data structure called func_info, with its format shown in Algorithm 1. In Algorithm 1, the func_name represents the name of the library function; func_call_addr represents the

```
//function: identify and store dynamic linked library
function when loading executable file
//input: executable file handle, .sym section head,
  .rel.plt section pointer, .plt section pointer
//output: library function message queue (func_info)
init func_info
set null func_info
Foreach elf_rel_plt in .rel.plt
    create a rela_plt_func_info object
    func_addr = extract the offset address of elf_rel_plt
    func_name = extract the .sym name
Endfor
Foreach elf_plt in .plt
    add_var = 32-bit data after moving the elf_plt address
16 bits to right + elf_plt address + 6
if (func_addr = = add_var)
func_call_addr = elf_plt address + index ∗ (unit width)
Endfor
```

Algorithm 2: Automatic extraction algorithm of library function information.

```
//function: intercept opcode to redirect program flow
//input: the code segment of executable file (.text), list of
library function, current environment (env)
//output: instructions calling local library function
Foreach insn in .text
  switch insn
   case: call insn
     if (the address of calling instruction is in the func_info)
   generate instructions of moving local argument
   generate instructions of calling local library function
   generate ret instruction of storing library function
          break;
      case
        ...
      default
        ...
      break
Endfor
```

Algorithm 3: Opcode interception algorithm.

function call address; func_addr represents the function address; next represents the next structure variable.

The extraction and storage of the library function call information are completed through two loops at the program loading stage. An empty func_info queue is initialized prior to the start of the loop; each time, a member of the .rel.plt section is read in the first loop and a new object is added to the func_info queue (representing a dynamic link function); the offset field value of the extracted members is taken as the function address of such object; then, the offset of the member in the .rel.plt is reused to calculate the corresponding position in .sym section and the name field value is extracted as the function name. The second loop is to modify the function call address of each object in the func_info queue; before each modification, consistency validation must be carried out for the function address. When the function address of the object to be processed is consistent with the function address calculated by the .plt field, the address of the current members, index, and its width are used to compute the function call address. The algorithm design is shown in Algorithm 2.

4.2. Opcode Interception. At the translation stage, an opcode interception algorithm is designed to implement the redirection of the program flow. The disassembly of each instruction will judge whether it is MPI function call. When the MPI function call is determined, the program jumps to the local MPI function encapsulation for execution, without instruction translation; the library function call information which has been cached into func_info structure is used for processing; otherwise, the traditional instruction translation will be carried out. Algorithm 3 shows the opcode interception algorithm.

Table 1: Experimental environment.

	Ms	Mt
CPU	Intel(R) Core(TM) 2 Quad CPU Q9500 @ 2.83 GHz	SW410 processor
OS	Fedora 2.6.27.5-117.fc10.i686	NeoKylin 3.8.0
Compiler	gcc-4.3.2	gcc-4.5.3
Frequency of the processor	2.5 GHz	1.6 GHz
Number of processor cores	4	4
Cache sizes of the processor	L2 cache 4 M	L2 cache 4 M
Amount of memory	4 G	4 G
Amount of storage	500 G	500 G
MPICH	3.1.2	3.1.2

Table 2: Test cases and the result of correctness test.

Test cases	Explanation	Correctness
IMB-MPI1	MPI-1 functions	100%
IMB-EXT	One-way interaction capabilities in MPI-2	100%
IMB-NBC	Interaction overlapping nonblocking MPI-3 functions	100%
IMB-RMA	Remote memory access	100%
NPB-IS	Two-dimensional large integer order base on the bucket sort	100%
NPB-CG	The best feature of large sparse symmetric positive definite matrix approximation	100%
NPB-EP	Test float computing	100%
NPB-FT	Discrete Fourier transform 3D problem	100%

When the translation is carried out taking the basic block as a unit, each instruction is disassembled to get the opcode and identify the instruction type. If it is the library function call (e.g., the opcode of call instruction in the x86 platform is E8), the function name will be checked if it exists in the cache array and contains the mpi_ as a prefix; if yes, parameter passing will be carried out to generate local MPI function call instruction. During the course of call instruction interception, the call instruction operand and the address of its next instruction are used to compute the call address of current library function. According to the keyword, search in the cache array; if it is found successfully, it shall be the MPI function call instruction.

4.3. Local Library Function Encapsulation and Call. The library function call is implemented by executing the local library encapsulated in advance, which means that the MPI function shall be encapsulated before the execution. This paper applies a method of hierarchical encapsulation of library function as follows: implementation language → the MPI library function name → parameter type, number of parameters, and return value type; such division can gradually narrow the scope of search when matching library function, so as to improve the hit rate. In the process of implementation, the parameters passing and return values processing shall be carried out with the help of context. Due to the limited number of functions contained in the MPI library, there is a small amount of work to do, which can be implemented by referring to the MPI library of a platform.

Due to the complexity and diversity of library functions, a series of issues, such as the parameter acquisition, return value capture, format character string analysis, and data element acquisition of the structure, need to be processed in the library function encapsulation. The library function encapsulation is a challenging job. When the MPI program to be translated contains a large number of MPI library function calls, the local library replacement method can significantly improve the efficiency.

5. Experimental Test

5.1. Experimental Environment. This paper aims to implement migrating the MPI executable program of the X86/Linux platform to the domestic processor platform, using the experimental environment shown in Table 1. Ms represents the source platform, with the application of X86-64 architecture CPU; Mt represents the local platform, with the application of domestic SW processor (similar to Sunway TaihuLight processor chip); Ms and Mt are equipped with open-source MPI compilation environment MPICH-3. Mt is equipped with MPI-QEMU, using a single multithreaded environment to simulate MPI multiprocess execution. The Intel MPI test set (IMB) [16] and NASA parallel program test set (NPB) [17] are taken as the test cases.

5.2. Correctness Test. This paper uses IMB and NPB test set to test the correctness of MPI-QEMU, with the test items and results shown in Table 2. The result shows that using

Table 3: Test cases of IMB-MPI1.

Operation	Number of processes	Ratio
PingPong	2	0.981
Sendrecv	4	0.942
Exchange	4	0.992
Allreduce	4	0.958
Reduce	4	0.945
Allgather	4	0.935
Allgatherv	4	0.931
Gather	4	0.969
Gatherv	4	0.957
Scatter	4	0.987
Scatterv	4	0.977
Alltoall	4	0.928
Alltoallv	4	0.961
Bcast	4	0.956

Table 4: Test cases of IMB-EXT.

Operation	Number of processes	Ratio
Window	4	0.951
Unidir_Get	2	0.968
Unidir_Put	2	0.961
Bidir_Get	2	0.973
Bidir_Put	2	0.961
Accumulate	2	0.974
Accumulate	4	0.847

Table 5: Test cases of IMB-NBC.

Operation	Number of processes	Ratio
Ibcast	4	1.124
Iallgather	4	0.985
Iallgatherv	4	0.948
Igather	4	0.917
Igatherv	4	0.959
Iscatter	4	0.981
Iscatterv	4	0.948
Ialltoall	4	0.946
Ialltoallv	4	0.968
Ireduce	4	0.985

Table 6: Test cases of IMB-RMA.

Operation	Number of processes	Ratio
Unidir_put	2	0.977
Unidir_get	2	0.925
Bidir_put	2	0.963
Bidir_get	2	0.981
One_put_all	4	0.945
One_get_all	4	0.979
All_put_all	4	0.964
All_get_all	4	0.617
Put_all_local	4	0.989
Exchange_put	4	0.943
Exchange_get	4	0.957
Unidir_put	2	0.957
Unidir_get	2	0.949
Bidir_put	2	0.961

MPI-QEMU can translate and execute the MPI program of the X86-64 platform in the domestic SW processor platform.

5.3. Performance Test. This paper, respectively, uses IMB and NPB to test the performance of MPI-QEMU. Since IMB test set almost covers all the MPI interactive functions, the performance test of the system by IMB also verifies the completeness of the system. In the experimental process, $T_{\text{mpi-qemu}}$ (the execution time of the MPI-QEMU) and T_{native} (running time under the source platform) are recorded and the performance of MPI-QEMU is tested by defining the speed ratio:

$$\text{Ratio} = \frac{T_{\text{native}}}{T_{\text{mpi-qemu}}} = \frac{1/T_{\text{mpi-qemu}}}{1/T_{\text{native}}} = \frac{B_{\text{mpi-qemu}}}{B_{\text{native}}}. \quad (8)$$

In the formula, $B_{\text{mpi-qemu}}$ and B_{native}, respectively, represent the execution speed in the MPI-QEMU and source platform. If the ratio is greater than 1, this represents the notion that the MPI-QEMU is faster; otherwise, the source platform is faster.

(1) IMB Test. IMB focuses on testing the performance of every MPI interactive function. This paper adopts the four items of IMB to test, that is, IMB-MPI1, IMB-EXT, IMB-NBC, and IMB-RMA. The test results are shown in Tables 3–6, including three columns in every table, that is, the MPI library function name, number of processes, and the ratio from left to right. In the 46 MPI functions contained in the four test items, only the ratio of 2 functions did not reach 0.9; the ratios of the others are above 0.9 and close to 1, and there is even 1 function with a ratio greater than 1. This indicates that the library function encapsulation algorithm can make full use of the local MPI environment to accelerate each message interaction; the MPI-QEMU has almost the same performance with the execution on source platform.

(2) NPB Test. NPB test set applies the classic algorithms and modules commonly used, to reflect the system's ability to solve practical engineering problems. This paper uses the NPB of MPI version to test the performance of MPI-QEMU in a multiprocess pattern. Figures 4–7, respectively, show the test results of NPB-IS, NPB-CG, NPB-EP, and NPB-FT. The horizontal axis represents the number of processes and the longitudinal axis represents the running time of the program, with the unit of seconds. The experimental results show that as the number of processes increased, the running time greatly reduced; and the running time is inversely proportional to the number of processes, almost

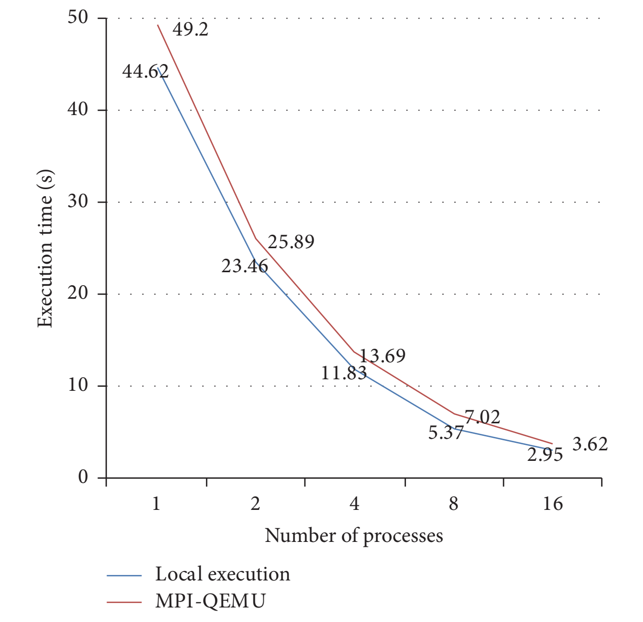

Figure 4: Test result of NPB-IS.

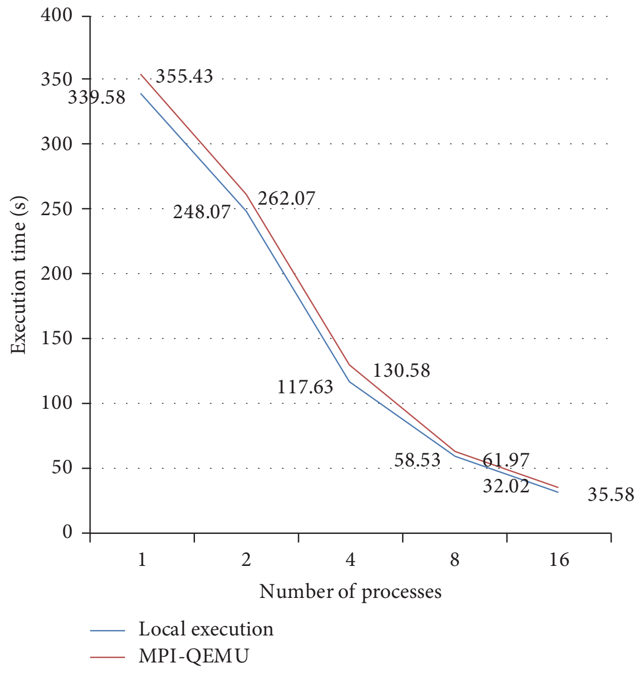

Figure 5: Test result of NPB-CG.

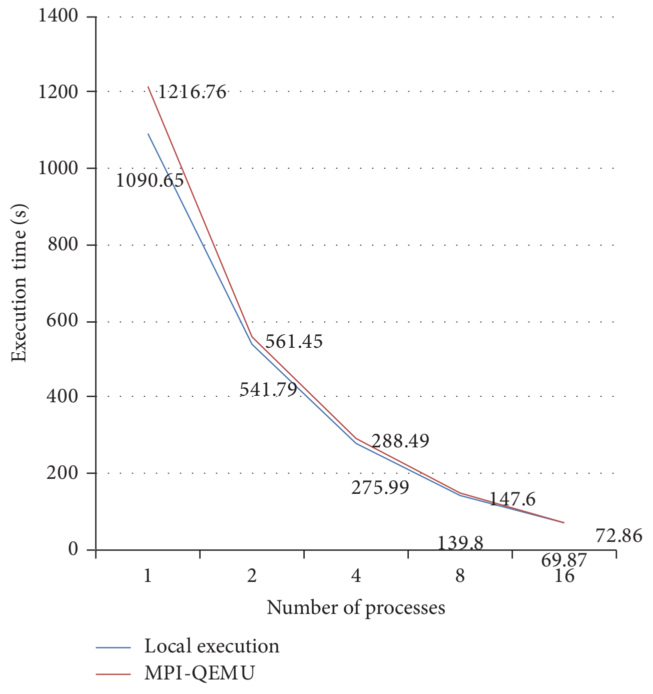

Figure 6: Test result of NPB-EP.

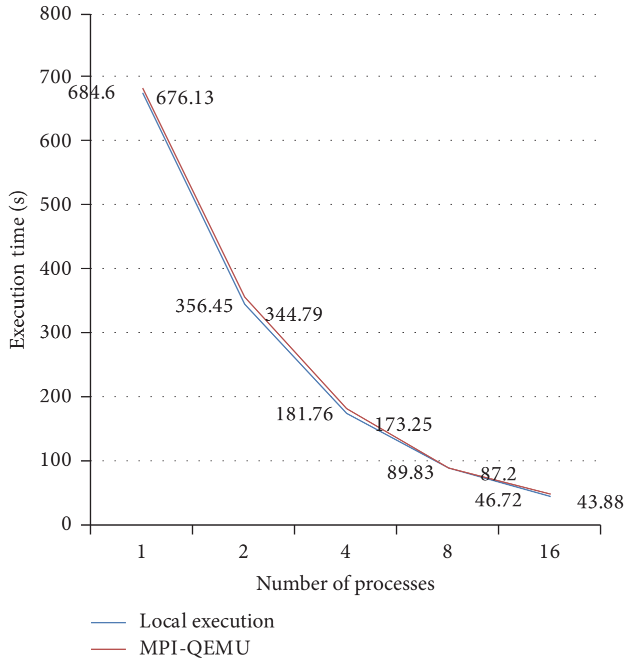

Figure 7: Test result of NPB-FT.

reaching the theoretical process acceleration ratio; through the calculation, the ratios of the four test cases are 85.1%, 92.9%, 94.5%, and 96.4%, with the average acceleration ratio of 92.2%.

6. Conclusion

This paper carried out the beneficial attempt of migrating the MPI program. A migration method of MPI program combining binary translation and local library function call has obtained periodic success. Starting from the equivalence representation between the programs of the clusters and through the analysis of the constitution principle of MPI executable file, this paper put forward different processing means for different parts and built the model for migration of the MPI program: for the cross-platform part of the executable file, the local library functions call is applied to implement migration through the program flow redirection at three stages; for the non-cross-platform part, the binary translation based on instructions translation is applied to implement migration. The experiment verified the correctness and effectiveness of the proposed method. For the next step, we will begin to explore the migration of large-scale commercial MPI binary programs and new ways for the improvement of migration efficiency.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

The authors offer their heartfelt thanks to the funding unit (supported by the National Natural Science Foundation of China under Grant no. 61472447) and the reviewing experts. Also, they would like to pay tribute to the predecessors who provided the basis and research platform for the research of this paper.

References

[1] E. R. Altman, D. Kaeli, and Y. Sheffer, "Welcome to the opportunities of binary translation," *Computer*, vol. 33, no. 3, pp. 40–45, 2000.

[2] M. Gschwind, E. R. Altman, S. Sathaye, P. Ledak, and D. Appenzeller, "Dynamic and transparent binary translation," *Computer*, vol. 33, no. 3, pp. 54–59, 2000.

[3] C. Songsong, L. Qi, and W. Jian, "Optimization of binary translator based on GODSON CPU," *Computer Engineering*, vol. 35, no. 7, pp. 280–282, 2009.

[4] C. Cifuentes and V. Malhotra, "Binary translation: static, dynamic, retargetable?" in *Proceedings of the Conference on Software Maintenance*, pp. 340–349, IEEE, Piscataway, NJ, USA, November 1996.

[5] J. Li, X. Ma, and C. Zhu, "Dynamic binary translation and optimization," *Computer Research and Development*, vol. 44, no. 1, pp. 161–168, 2007.

[6] O. Almer, I. Böhm, T. E. Von Koch et al., "Scalable multi-core simulation using parallel dynamic binary translation," in *Proceeding of the 11th International Conference on Embedded Computer Systems*, pp. 190–199, IEEE, Piscataway, NJ, USA, 2011.

[7] J.-H. Ding, P.-C. Chang, W.-C. Hsu, and Y.-C. Chung, "PQEMU: A parallel system emulator based on QEMU," in *Proceedings of the 17th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2011*, pp. 276–283, IEEE, Piscataway, NJ, USA, 2011.

[8] D.-Y. Hong, C.-C. Hsu, P.-C. Yew et al., "HQEMU: A multi-threaded and retargetable dynamic binary translator on multicores," in *Proceedings of the 10th International Symposium on Code Generation and Optimization, CGO 2012*, pp. 104–113, ACM, New York, NY, USA, April 2012.

[9] CUDA Toolkit Documentation v6.0. 2014 https://developer.nvidia.com/cuda-toolkit-archive.

[10] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark, "Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems," in *Proceeding of the 19th International Conference on Parallel Architectures and Compilation Techniques, PACT 2010*, pp. 353–364, New York, NY, USA, September 2010.

[11] J. A. Stratton, S. S. Stone, and W. W. Hwu, "MCUDA: An effective implementation of CUDA kernels for multi-core CPUs," in *Proceeding of the 21st International Workshop on Languages and Compilers for Parallel Computing*, Springer Verlag, Berlin, Germany, 2008.

[12] F. Yue, J.-M. Pang, and R.-C. Zhao, "Detecting and treatment algorithm of implicit synchronization based on dependence analysis in SPMD program," *Ruan Jian Xue Bao/Journal of Software*, vol. 24, no. 8, pp. 1775–1785, 2013.

[13] N. Li, J. Pang, and Z. Shan, "Migration of CUDA program based on a Divide-and-Conquer method," in *Proceedings of the 17th IEEE International Conference on Computational Science and Engineering*, pp. 1685–1691, IEEE, Piscataway, NJ, USA, December 2014.

[14] SW410, 2014, http://www.shenweimicro.com/.

[15] MPI Forum, 2016, http://mpi-forum.org/.

[16] Nas parallel benchmarks, 2016, http://www.nas.nasa.gov/publications/npb.html.

[17] Intel® MPI Benchmarks 4.1., 2016, https://software.intel.com/en-us/intel-mpi-library/.