Process/Thread Migration and Checkpointing in Heterogeneous Distributed Systems*

Hai Jiang and Vipin Chaudhary
Institute for Scientific Computing
Wayne State University
Detroit, MI 48202
{hai, vipin}@wayne.edu

Abstract

Process/thread migration and checkpointing are indispensable for resource sharing, cycle stealing, and other modes of interaction. To provide a flexible, transparent, and portable solution in heterogeneous environments, we have developed a multi-grained migration/checkpointing package, MigThread, which can migrate/checkpoint multiple threads to different machines or file systems simultaneously, and also perform single coarse-grained process migration/checkpointing. For scalability and portability, computation states are extracted out of their original places and abstracted to the language level. With the user-level stack/heap management, MigThread does not rely on any thread libraries and operating systems. For heterogeneity, a novel data conversion scheme is proposed to analyze data types automatically and convert data only on the receiver side. For safety, MigThread detects and overcomes "unsafe" factors to qualify virtually all C programs for migration/checkpointing. Some performance measurements are given to illustrate its effectiveness.

1. Introduction

From cluster computing to internet computing and now Grid computing, current computation technologies have focused more on collaboration, data sharing, cycle stealing, and other modes of interaction among dynamic and geographically distributed organizations [1]. Multi-grained computation migration and checkpointing become indispensable for load balancing, load sharing, fault tolerance and data locality improvement. The major obstacle preventing them from achieving widespread use is the complexity of adding them transparently to systems originally designed to run stand-alone [2]. Heterogeneity further complicates this situation. Effective solutions are required.

To hide the different levels of heterogeneity, we have developed an application-level process/thread migration and checkpointing package, *MigThread*, which consists of a preprocessor and a run-time support module [3, 4]. At compile time, the preprocessor transforms user's source code and extracts the computation state out of its original place to abstract it up at the language level so that the run-time support module can construct it precisely. Since the physical states are transformed into a logical form, there is no restriction on thread types and operating systems. *MigThread* provides a full-fledged solution, including user-level stack and heap management as well as pointer updating scheme.

A process is an operating system abstraction representing an instance of a running computer program [2], whereas a thread is an execution path within a process. *MigThread* supports both coarse-grained processes and fine-grained threads. Each process may contain multiple threads. These threads can be treated as a whole for a process; or they can be handled individually. Some threads might be migrating to different destinations whereas others can be saving their states into file systems for checkpointing.

For heterogeneity, computation states are abstracted and represented in terms of data. Then *MigThread* is equipped with a novel "plug-and-play" style data conversion scheme, called coarse-grain tagged "Receiver Makes Right" (CGT-RMR), which can detect data types, generate tags, and convert data only on the receiver side. Aggregate type data are handled as a whole instead of being flattened down by programmers manually. It is efficient in handling large data chunks which are common in migration/checkpointing.

Migration/checkpointing-safety concerns ensuring the correctness: computation states should be constructed precisely on source nodes, and restored correctly on destination nodes [4]. Type-unsafe programming languages like C challenge this, since during execution each memory block

*This research was supported in part by NSF IGERT grant 9987598, NSF MRI grant 9977815, and NSF ITR grant 0081696.

can hold data of any type, which could be totally different from the type declared in the program. Although type-safe languages can avoid such type uncertainty, setting it as a requirement for migration and checkpointing will be too conservative. The unsafe factors are identified as harmful pointer casting, pointers in unions, third-party library calls, and incompatible data conversion. With help from a friendly user interface and CGT-RMR, *MigThread* manages to detect and recover from unsafe cases. Therefore, almost all programs are qualified for migration and checkpointing.

Beyond our previous homogeneous thread migration scheme [3], the system is strengthened with new functionalities, including process migration, checkpointing, heterogeneity, and safety protection. Then, through configurations, *MigThread* can accomplish various tasks.

The remainder of this paper is organized as follows: Section 2 introduces the background of migration/checkpointing technologies. Section 3 describes the design and implementation of *MigThread* in detail. Complexity analysis and some experimental results from benchmark programs are shown in Section 4. Section 5 gives an overview of related work. Finally, our conclusions and continuing work are presented in Section 6.

2. Migration and Checkpointing Technologies

Process/thread migration concerns saving the current computation state, transferring it to remote machines, and resuming the execution at the statement following the migration point. Checkpointing concerns saving the computation state to file systems and resuming the execution by restoring the computation state from saved files. Therefore, process/thread migration is a real-time "memory-network-memory" state-transfer scenario while checkpointing follows an off-line "memory-file-memory" route to save/retrieve computation state. Although the state-transfer medium differs, migration and checkpointing share the same strategy in state handling.

2.1. Process Migration and Checkpointing

A process indicates a coarse-grained computation unit. Process migration/checkpointing is utilized to move a sequential job or a whole parallel computation. According to the levels at which they are implemented, they can be classified into three categories: kernel-level, user-level, and application-level [2].

Kernel-level schemes are implemented in operating systems which can access process states efficiently and support preemptive migration at virtually any point [2]. Thus, this approach provides good transparency and flexibility. But it adds a great deal of complexity to the kernels, and only works within the same platforms. User-level schemes are implemented as libraries in user space and linked to user applications at compile time [6]. This approach relies on certain non-portable UNIX system calls to fetch computation state, and thus user-level threads cannot be identified and scheduled. Application-level schemes are implemented as a part of the application. Since the computation state is moved into applications, this approach possesses a great potential to support heterogeneous computations. However, it typically sacrifices transparency and reusability.

2.2. Thread Migration

Threads are lightweight processes and thread migration enables fine-grained computation adjustment in parallel computing. As multi-threading becomes a popular programming practice, thread migration is increasingly important in fine-tuning Grid computing in dynamic and non-dedicated environments. The core of thread migration is to construct and transfer thread states. Since applications can only contact user-level threads directly, thread migration packages adopt user/application-level strategies and they are classified based on how they handle pointers.

The first approach requires language and compiler support to maintain adequate type information and identify pointers [7]. Portability is the major drawback. The second approach scans stacks at run-time to detect and translate pointers [8]. However, it is possible that some pointers cannot be detected and the resumed execution might go wrong. The third approach necessitates the partitioning of the address space and reservation of unique virtual address for the stack of each thread so that the internal pointers remain the same values. A common solution is to preallocate memory space for threads on all machines and restrict each thread to migrate only to its corresponding address on other machines[9]. This homogeneous "iso-address" solution requires large address space and scalability is poor.

2.3. Heterogeneity

For heterogeneous computations, both programs and data should be portable. Naturally, the intuitive option is virtual machines [18], which usually suffer with performance drawback. Most practical migration systems choose to install corresponding execution code on all different machines and only manage to achieve portability for data.

Most data conversion schemes, such as XDR (External Data Representation) [10], adopt a canonical intermediate form strategy which provides an external representation for each data type. This approach requires the sender to convert data to canonical form and the receiver to convert data from canonical form, even on the same platforms. Obviously, this incurs tremendous overhead in homogeneous environments. XDR only encodes the data instead of data types,

which have to be determined by application protocols. For each data type, there are corresponding routines for encoding/decoding. And each platform is only equipped with a set of routines for conversion between local and intermediate formats. Zhou and Geist [11] proposed an asymmetric data conversion technique, called "receiver makes it right" (RMR), where data conversion is performed only on the receiver side. Thus, the receiver should be able to convert and accept data from all other machines. If there are n kinds of different machines, the number of conversion routine groups will be $(n^2 - n)/2$.

Current symmetric/asymmetric conversion strategies require flattening complex data structures and associating packing/unpacking routines with each basic data type because the padding patterns in aggregate types are a consequence of the processor, operating system and compiler, and cannot be determined until run-time. This type flattening process incurs tremendous coding burden for programmers.

2.4. Migration/Checkpointing Safety

Migration/checkpointing-safety concerns ensuring correctness, e.g., computation states should be constructed and restored correctly [12, 4]. Type-unsafe programming language such as C challenge this since during execution each memory block can hold data of any type, which could be totally different from the type declared in the program. Then, states of such programs can be hard to construct and migration/checkpointing might go wrong. Type-safe languages can avoid certain type uncertainty and help achieve correct states. However, "type safety" is not the same as "migration/checkpointing safety" [12]. "Type safety" concerns type clarity and program correctness whereas the latter one focuses on constructing correct states. They cannot guarantee each other. "Type safety" does eliminate some migration/checkpointing-unsafe factors in languages, meanwhile some type-unsafe programs can still be qualified for migration/checkpointing. Without formal definitions of migration/checkpointing-safety, most existing migration/checkpointing systems declare to only work with "safe" programming [12, 14] and leave the safety issues to programmers. Mechanism is on demand to identify unsafe factors in order to free programmers.

3. Design and Implementation of MigThread

MigThread provides multi-grained migration and checkpointing functionalities for sequential and parallel computations in heterogeneous or Grid computing systems. Both coarse-grained processes and fine-grained threads are supported and both migration and checkpointing are available, as shown in Figure 1. For a certain process, its threads can simultaneously checkpoint to file systems or migrate

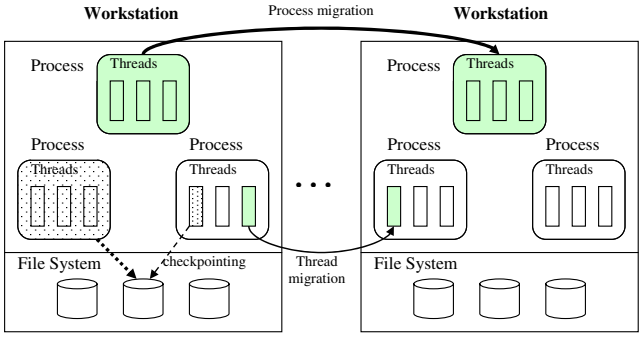

Figure 1. Process/thread and migration/checkpoing options in *MigThread*.

to different destinations. This brings sufficient flexibility into parallel computing. For process migration and checkpointing, all internal threads as well as their shared global data are processed together. Therefore, process migration/checkpointing could be the summation of the ones for all the internal threads.

Typically states consist of process data segments, stacks, heaps and register contents. In *MigThread*, the computation state is moved out from its original location (libraries or kernels) and abstracted up to the language level. Thus, the physical state is transformed into a logical form to achieve platform-independence and reduction in restrictions. Both the portability and the scalability of stacks are improved.

MigThread consists of two parts: a preprocessor and a run-time support module. The preprocessor is designed to transform user's source code into a format from which the run-time support module can construct the computation state precisely and efficiently. Its power can improve the transparency drawback in application-level schemes. The run-time support module constructs, transfers, and restores computation state dynamically as well as provides other run-time safety checks.

3.1. State Construction

Handling computation state, the core of migration/checkpointing, is done by both the preprocessor and the run-time support module. At compile time, all information related to stack variables, function parameters, program counters, and dynamically allocated memory regions, is collected into certain pre-defined data structures [3].

For globally shared variables, non-pointer variables are collected in *GThV* whereas pointers are gathered in *GThP*. For each user defined function, its local variables are put into *MThV*/*MThP* instead of *GThV*/*GThP* pair. Since the address spaces of a thread could be different on source and destination machines, and stacks and heaps need to be recreated on destination machines, values of pointers refer-

```
foo()
{
  int      a;
  double   b;
  int     *c;
  double **d;
         .
         .
         .
}
```

Figure 2. The original function.

```
MTh_foo()
{
  struct MThV_t {
      void   *MThP;
      int     stepno;

      int        a;
      double     b;
  } MThV;

  struct MThP_t {
      int       *c;
      double   **d;
  } MThP;

  MThV.MThP = (void *)&MThP;
         .
         .
         .
}
```

Figure 3. The transformed function.

encing stacks or heaps might become invalid after migration/checkpointing. It is the preprocessor's responsibility to identify and mark pointers at the language level so that they can easily be traced and updated later. Collecting all pointers in *GThP* and *MThP* eases the pointer updating process.

Figures 2 and 3 illustrate an example for this data collection process. A function **foo()** is defined with four local variables as in Figure 2. *MigThread*'s preprocessor transforms the function and generates a corresponding **MTh_foo()** shown in Figure 3. Within *MThV*, field *MThV.MThP* is the only pointer, pointing to the second structure, *MThP*, which may or may not exist. Similarly, *GThV.GThP* is used to trace *GThP* for global pointers. In stacks, each function's activation frame contains *MThV* and *MThP* to record the current function's computation status. The overall stack status can be obtained by collecting all of these *MThV* and *MThP* data structures spread in activation frames.

The program counter (PC) is the memory address of the current execution point within a program. It indicates the starting point after migration/checkpointing. When the PC is moved up to the language level, it should be represented in a portable form. We represent the PC as a series of **integer** values declared as *MThV.stepno* in each affected function, as shown in Figure 3. Since all possible adaptation points have been detected at compile-time, different **integer** values of *MThV.stepno* correspond to different adaptation points. In the transformed code, a **switch** statement is inserted to dispatch execution to each labelled point according to the value of *MThV.stepno*, and executed after the function initialization. The **switch** and **goto** statements help control jump to

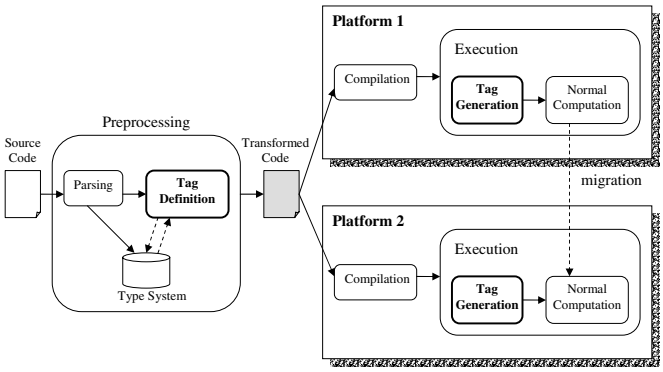

Figure 4. Tag definition and generation.

resumption points quickly.

In C, the parameters of a function carry information and status, and consequently the state. Therefore, this information needs to be restored on the destination nodes. Fields with the same types and names are defined in *MThV* or *MThP* depending on whether they are non-pointer variables or pointers. During initialization, the values of these fields are set by function parameters. Later on, all references to function parameters will be substituted by the ones in these fields. This strategy benefits from the "pass-by-copy" function call method in C.

MigThread also supports user-level memory management for heaps. In application programs, when **malloc()** and **free()** are invoked to allocate and deallocate memory space, statements **MTh_mem_reg()** and **MTh_mem_unreg()** are appended to trace memory blocks correspondingly. Eventually, all computation state related contents, including stacks and heaps, are moved out to the user space and handled by *MigThread* directly. This builds the foundation for correct state retrieval and fulfills the pre-condition for portability.

3.2. The Data Conversion Scheme

Computation states can be abstracted to language level and transformed into pure data. For different platforms, states constructed on one platform need to be interpreted by another. Thus, data conversion is unavoidable.

In *MigThread*, we proposed a data conversion scheme, called Coarse-Grain Tagged "receiver makes it right" (CGT-RMR) [5], to tackle data alignment and padding physically, convert data structures as a whole, and eventually generate a lighter workload compared to existing standards. It accepts ASCII character sets, handles byte ordering, and adopts IEEE 754 floating-point standard because of its dominance in the market.

The programmers do not need to worry about data formats since the preprocessor parses the source code, sets up type systems, transforms source code, and communicates with the run-time support module through inserted primi-

```
MTh_foo()
{
                        .
                        .
                        .
  char MThV_heter[60];
  char MThP_heter[41];

  int MTh_so2 = sizeof(double);
  int MTh_so1 = sizeof(int);
  int MTh_so4 = sizeof(struct MThP_t);
  int MTh_so3 = sizeof(struct MThV_t);
  int MTh_so0 = sizeof(void *);

  MThV.MThP = (void *)&MThP;

  sprintf(MThV_heter,
"(%d,-1)(%d,0)(%d,1)(%d,0)(%d,1)(%d,0)(%d,1)(%d,0)", MTh_so0,
(long)&MThV.stepno-(long)&MThV.MThP-MTh_so0, MTh_so1,
(long)&MThV.a-(long)&MThV.stepno-MTh_so1, MTh_so1,(long)&MThV.b-
(long)&MThV.a- MTh_so1, MTh_so2,(long)&MThV+MTh_so3-(long)&MThV.b-
MTh_so2);

  sprintf(MThP_heter, "(%d,-1)(%d,0)(%d,-1)(%d,0)",
MTh_so0,(long)&MThP.d-(long)&MThP.c-MTh_so0,
MTh_so0,(long)&MThP+MTh_so4-(long)&MThP.d-MTh_so0);
                        .
                        .
                        .
}
```

Figure 5. Tag definition at compile time.

```
char MThV_heter[60]="(4,-1)(0,0)(4,1)(0,0)(4,1)(0,0)(8,0)(0,0)";
char MThP_heter[41]="(4-1)(0,0)(4,-1)(0,0)";
```

Figure 6. Tag calculation at run-time.

tives. It can also analyze data types, flatten down aggregate types recursively, detect padding patterns, and define tags as in Figure 4. But the actual tag contents can be set only at run-time and they may not be the same on different platforms. Since all of the tedious tag definition work has been performed by the preprocessor, the programming style becomes extremely simple. Also, with global control, low-level issues such as data conversion status can be conveyed to upper-level scheduling modules. Therefore, easy coding style and performance gains come from the preprocessor.

In *MigThread*, tags are used to describe data types and paddings so that data conversion routines can handle aggregate types as well as common scalar types. As we discussed above, global variables and function local variables are collected into their corresponding structure type variables *GThV*/*GThP* and *MThV*/*MThP*. Tags are defined and generated for these structures as well as dynamically allocated memory blocks in the heap.

For the example in Figures 2 and 3, its transformed code will be equipped with tags *MThV_heter* and *MThP_heter* for *MThV* and *MThP*, respectively, as in Figure 5. At compile time, it is still too early to determine tag contents. The preprocessor defines rules to calculate structure members' sizes and variant padding patterns, and inserts **sprintf()** to glue partial results together. The actual tag generation has to take place at run-time when the **sprintf()** statement is executed. On a Linux machine, the simple example's tags can be two character strings as shown in Figure 6.

A tag is a sequence of *(m,n)* tuples, and can be expressed as follows (where m and n are positive numbers):

- (m, n) : scalar types. The item "m" is simply the size of the data type, and "n" indicates the number of such scalar types.
- $((m', n')...(m'', n''), n)$: aggregate types. The "m" in the tuple (m, n) can be substituted with another tag (or tuple sequence) repeatedly. Thus, a tag can be expanded recursively for those enclosed aggregate type fields until all fields are converted to scalar types. The second item "n" still indicates the number of the top-level aggregate types.
- $(m, -n)$: pointers. The "m" is the size of pointer type on the current platform. The "-" sign indicates the pointer type, and the "n" still means the number of pointers.
- $(m, 0)$: padding slots. The "m" specifies the number of bytes this padding slot can occupy. The $(0, 0)$ is a frequently occuring case and indicates no padding.

At compile time, only one statement is issued for each data type, whether it is a scalar or aggregate type. The flattening procedure is accomplished by the preprocessor during tag definition instead of the encoding/decoding process at run-time. Hence, programmers are freed from this burden.

All memory segments for *GThV*/*GThP* and *MThV*/*MThP* are represented in a "tag-block" format. Each stack becomes a sequence of these structures and their tags. Memory blocks in heaps are also associated with tags for the actual layout in memory space. Therefore, the computation state physically consists of a group of memory segments associated with their own tags in a "tag-segment" pair format.

3.3. State Restoration

Only the receivers or processes resuming from checkpointing files need to convert the computation state, i.e., data, as required. Since activation frames in stacks are re-run and heaps are recreated, a new set of segments in "tag-block" format is available on the new platform. *MigThread* first compares architecture tags by **strcmp()**. If they are identical and the blocks have the same sizes, the platforms remain unchanged and the old segment contents are simply copied over by **memcpy()** to the new architectures. This enables prompt processing between homogeneous platforms while symmetric conversion approaches still suffer data conversion overhead on both ends.

If platforms have been changed, conversion routines are applied on all memory segments. For each segment, a "walk-through" process is conducted against its corresponding old segment from the previous platform, as shown in Figure 7. In these segments, according to their tags, memory blocks are viewed to consist of scalar type data and padding slots alternately. The high-level conversion unit is data slots rather than bytes in order to achieve portability. The "walk-through" process contains two index pointers pointing to a

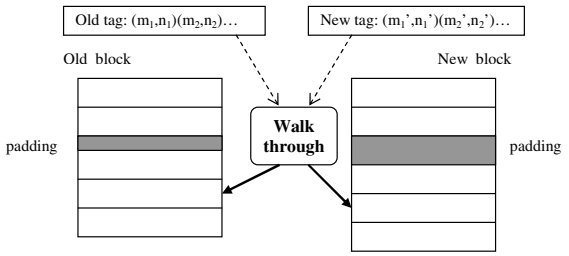

Figure 7. Walk through segments.

pair of matching scalar data slots in both blocks. The contents of the old data slots are converted and copied to the new data slots if byte ordering changes, and then the index pointers are moved down to the next slots. In the mean time, padding slots are skipped over, although most of them are defined as $(0, 0)$ to indicate that they do not physically exist. In *MigThread*, data items are expressed in "scalar type data - padding slots" pattern to support heterogeneity.

3.4. Safety Issues

In order to remove the "safe programming" requirement on programmers, *MigThread* detects and handles some "unsafe" features, including pointer casting, pointers in unions, library calls, and incompatible data conversion. States of more programs will be precisely constructed to make them eligible for migration/checkpointing.

Pointer casting does not mean the cast between different pointer types, but the cast to/from integral types, such as integer, long, or double. The problem is that pointers might hide in integral type variables. The central issue is to detect those integral variables containing pointer values (or memory addresses) so that they could be updated during state restoration. Casting could be direct or indirect. There are four ways to hide pointers in integral type variables (shown in Figure 8):

1. Cast pointers directly or indirectly. In Figure 8, case (1) only shows the direct cast. If num is assigned to another integral type variable, indirect cast happens and it also can cause problems.
2. Memory addresses are cast into integral type variables directly.
3. Functions' returning values are cast in.
4. Integral variables are referenced indirectly by pointers or pointer arithmetic and their values are changed by all the above three cases.

To avoid dangerous pointer casting, *MigThread* investigates pointer operations at compile time. The preprocessor creates a pointer-group by collecting pointers, functions with pointer type return values, and integral variables that have already been cast in pointer values. When the left-hand

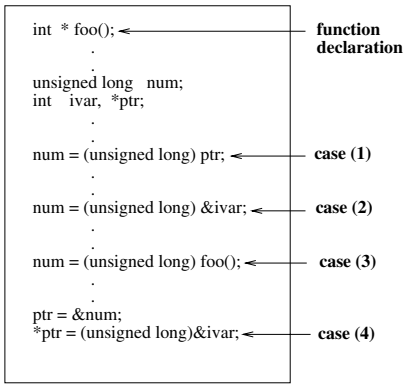

Figure 8. Hiding pointers in integral variables.

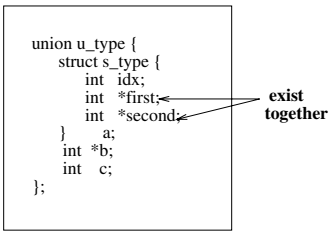

Figure 9. Pointers in Union.

side of an assignment is an integral type variable, the preprocessor checks the right-hand side to see if pointer casting can happen. If members of pointer-group exist without changing their types, the left-hand side variable should also be put into pointer-group for future detection and reported to the runtime support module for possible pointer update. All other cases may be ignored.

The preprocessor is insufficient for the indirect access and pointer arithmetic as in case (4) in Figure 8. A primitive **MTh_check_ptr(mem1, mem2)** is inserted to check if $mem1$ is actually an integral variable's address (not on pointer trees) and $mem2$ is an address (pointer type). If so, $mem1$ will be registered as a pointer which could also be unregistered later. Here $mem1$ is the left-hand side of assignment and $mem2$ is one member of right-hand side components. If there are multiple components on the right-hand side, this primitive will be called multiple times. Frequently using pointer arithmetic on the left-hand side can definitely cause heavy burden on tracing and sacrifice performance. This is a rare case since normally pointer arithmetic is applied more on the right-hand side. Thus, computation is not affected dramatically. During the migration/checkpointing, registered pointers will be updated no matter if their original types are pointer ones or not. The preprocessor and run-time support module work together to find out memory addresses hidden in integral variables and update them for safety [4].

Union is another construct where pointers can evade updating. In the example of Figure 9, using member a means

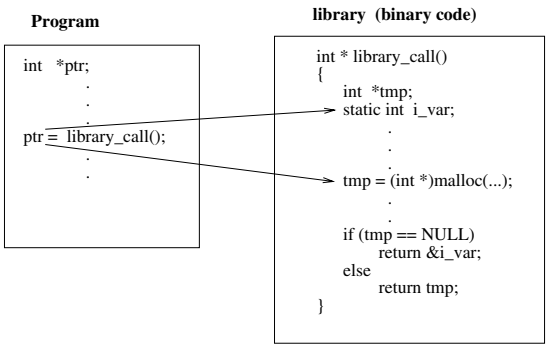

Figure 10. Pointers dangling in libraries.

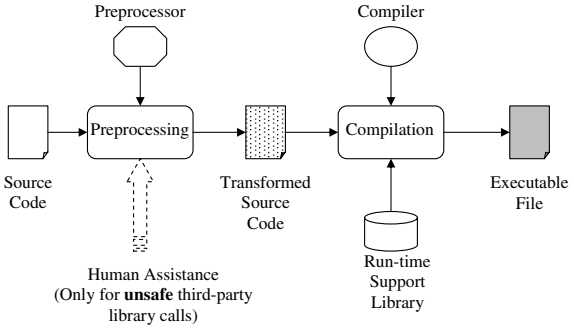

Figure 11. Compile/run-time support.

two pointers are meaningful; member b indicates one; and member c requires no update. Migration schemes have to identify dynamic situations on the fly. When a **union** variable is declared, the compiler automatically allocates enough storage to hold the largest member of the **union**. In the program, once the preprocessor detects that a certain member of the **union** variable is in use, it inserts primitive **MTh_union_upd()** to inform the runtime support module which member and its corresponding pointer fields are in activation. The records for previous members' pointer subfields become invalid because of the ownership changing in the **union** variable. We use linked list to maintain these inner pointers and get them updated later.

Library calls bring difficulties to all schemes since it is hard to detect what happens inside the library code. Without source code, it is even harder for application-level schemes. In the example of Figure 10, the pointer ptr might be pointing to an address of static local variable i_var for which compilers create permanent storage or a dynamically allocated memory block. Both of them are invisible to migration/checkpointing schemes. Pointers pointing to these unregistered locations are also called "dangling pointers", as those pointing to de-allocated memory blocks. This phenomena indicates that schemes are unable to catch all memory allocations because of the "blackbox" effect. The current version of *MigThread* provides a user interface to specify the syntax of certain library calls so that the preprocessor can know how to insert proper primitives for memory management and trace pointers.

The final unsafe factor is the incompatible data conversion. Between incompatible platforms, if data items are converted from higher precision formats to lower precision formats, precision loss may occur. But if the high end portions contain all-zero content, it is safe to throw them away since data values still remain unchanged. *MigThread* intends to convert data unless precision loss occurs. This aggressive data conversion enables more programs for migration and checkpointing without aborting them too conservatively. Detecting incompatible data formats and conveying this low-level information up to the scheduling module can help abort data restoration promptly.

3.5. Compile/Run-time Support

The compile-time support is implemented in our preprocessor that is based on LEX. First, the user compiles his programs using a conventional C compiler to ensure syntax correctness. Then, if process/thread migration and checkpointing functionalities are required, the preprocessor is used to transform the source code. Finally, the transformed code is compiled and linked with *MigThread*'s run-time support library to generate the eventual executable files, as shown in Figure 11. Most of the time, programmers are not required for assistance unless the preprocessor encounters unsolvable third-party library calls which are troubles to all migration/checkpointing systems. Manual support is a necessity for this case.

The preprocessor conducts the following tasks:

- **Information Collection** : Collect related information for future state construction.
  - Fetch stack data, such as globally shared variables, local variables, function parameters, and program counters, etc.
  - Fetch heap data, such as dynamically allocated memory segments.
- **Tag Definition** : Create tags for data structures and memory blocks.
- **Renaming** : Rename functions and variables.
- **Position Labelling** : Detect and label adaptation positions.
- **Control Dispatching** : Insert **switch** statements to orchestrate execution flows.
- **Safety Protection** : Detect and overcome unsafe cases; seek human assistance/instruction for third-party library calls; and leave other unresolved cases to the runtime support module.

Table 1. Complexity comparison in data collecting

System	Collect Variables	Collect Pointers	Collect Memory Blocks	Save Variables	Save Pointers	Allocate Memory Blocks
Porch	$O(N_{var})$	$O(N_{ptr})$	$O(N_{mem})$	$O(N_{var})$	$O(N_{ptr})$	$O(N_{mem} * logN_{mem})$
SNOW	$O(N_{var})$	$O(N_{ptr})$	$O(N_{mem} * logN_{mem})$	$O(N_{var})$	$O(N_{ptr})$	$O(N_{mem})$
MigThread	1	1	$O(N_{mem})$	0	0	$O(N_{mem} * logN_{mem})$

Table 2. Complexity comparison in data restoration

System	Restore Variables	Restore Pointers	Update Pointers	Re-allocate Memory Blocks	Delete Memory Blocks
Porch	$O(N_{var})$	$O(N_{ptr})$	$O(N_{ptr} * logN_{mem})$	$O(N_{mem})$	$O(N_{mem} * logN_{mem})$
SNOW	$O(N_{var})$	$O(N_{ptr})$	$O(N_{ptr} * N_{mem})$	$O(N_{mem})$	$O(N_{mem}^2)$
MigThread	1	1	$O(N_{ptr} * logN_{mem})$	$O(N_{mem})$	$O(N_{mem} * logN_{mem})$

The run-time support module is activated through primitives inserted by the preprocessor at compile time. It is required to link this run-time support library with user's applications in the final compilation. During the execution, its task list includes:

- **Stack Maintenance** : Keep a user-level stack of activation frames for each thread.
- **Tag Generation** : Fill out tag contents which are platform-dependent.
- **Heap Maintenance** : Keep a user-level memory management subsystem for dynamically allocated memory.
- **Migration and Checkpointing** : Construct, transfer, and restore computation state.
- **Data Conversion** : Translate computation states for destination platforms.
- **Safety Protection** : Detect and recover the unsafe cases unresolved at compile time.
- **Pointer Updating** : Identify and update pointers after migration and checkpointing.

4. Complexity Analysis and Experiments

Besides *MigThread*, there are several other application level migration/checkpointing systems, including Porch [13] and SNOW [14] which collect and restore variables one-by-one explicitly at each adaptation point in time $O(N)$. This makes it hard for users to insert adaptation points by themselves. *MigThread* only registers variables once in time $O(1)$ and at adaptation points the programs only check for condition variables. Therefore, *MigThread* is much faster in dealing with computation states.

For memory blocks, Porch and *MigThread* have similar complexity because they both maintain memory information in red-black trees. SNOW uses a memory space representation graph, which is quick to create a memory node, but extremely slow for other operations because searching for a particular node in a randomly generated graph is time-consuming. Also, SNOW traces all pointers and slows down much for pointer-intensive programs. *MigThread* virtually only cares about results ("ignore process") and is therefore less dependent on the types of programs. We summarize the complexity of these three systems, and list the results in Table 1 and 2. The N_{var}, N_{ptr} and N_{mem} represent numbers of variables, pointers and dynamically allocated memory blocks. It is obvious that *MigThread* is extremely fast in stack data collection and shows better overall performance in heap data operations.

One of our experimental platforms is a SUN Enterprise E3500 with 330Mhz UltraSparc processors and 1Gbytes of RAM, running Solaris 5.7. The other platform is a PC with a 550Mhz Intel Pentium III processor and 128Mbytes of RAM, running Linux. CGT-RMR is applied for data conversion between these two different machines. With predefined adaptation points, FFT, continuous and non-continuous versions of LU from the SPLASH-2 suite, and matrix multiplication applications are used for evaluation. The detailed overheads are listed in Table 3. Under program names, the numbers in parentheses indicate the function activation times. In our design, tags for each function are only generated once. This optimized tag generation strategy reduce overhead dramatically since some functions might be activated many times. For example, in LU-c, functions are called 2.8M times, which can cause noticeable slowdown. Numbers outside of parentheses indicate the problem sizes for these applications.

To test the heterogeneity, we perform migration and checkpointing across same and different platforms. For migration, computation states are constructed on the fly and transferred to the memory space on remote machines. Meanwhile, for checkpointing, computation states are first saved into file systems and then read by another process on

Table 3. Migration and Checkpointing Overheads in real applications (Microseconds)

Program (Func. Act.)	Platform Pair	State Size (B)	Save Files	Read Files	Send Socket	Convert Stack	Convert Heap	Update Pointers
FFT (2215) 1024	Solaris-Solaris	78016	96760	24412	26622	598	1033	364
	Linux-Solaris	78024	48260	24492	29047	1581	57218	459
	Solaris-Linux	78016	96760	13026	16948	923	28938	443
	Linux-Linux	78024	48260	13063	17527	387	700	399
LU-c (2868309) 512x512	Solaris-Solaris	2113139	2507354	4954588	4939845	589	27534	5670612
	Linux-Solaris	2113170	1345015	4954421	5230449	1492	3158140	6039699
	Solaris-Linux	2113139	2507354	7011277	7045671	863	2247536	8619415
	Linux-Linux	2113170	1345015	7058531	7131833	385	19158	8103707
LU-n (8867) 128x128	Solaris-Solaris	135284	165840	51729	53212	528	2359	306
	Linux-Solaris	135313	85053	51501	62003	1376	103735	322
	Solaris-Linux	135284	165840	40264	44901	837	52505	359
	Linux-Linux	135313	85053	40108	56695	357	1489	377
MatMult (6) 128x128	Solaris-Solaris	397259	501073	166539	164324	136	2561	484149
	Linux-Solaris	397283	252926	120229	220627	385	306324	639281
	Solaris-Linux	397259	501073	166101	129457	862	604161	482380
	Linux-Linux	397283	252926	120671	130107	100	3462	640072

another similar or different platform. In this experiment, computation state sizes can vary from 78K to 2M bytes. Even for the same applications, sizes of their computation states might be different on same platforms.

The saving, re-reading, and transferring costs are listed in Table 3. Checkpointing splits the state transfer process into two parts and therefore causes more overheads. But in the meantime, it provides more flexibility. These I/O events can cause dominant overheads, no matter whether they are file I/O operations or TCP/IP communications. Normally, data conversion costs are relatively small, especially in homogeneous environments. This indicates the advantage of our data conversion scheme CGT-RMR [5]. Many other conversion standards perform conversion twice even on the same platforms, and therefore, incur more overheads.

In most cases, migrating and restarting processes after checkpointing produce similar overheads because the only difference between them is the medium where they send and fetch the computation states. Total costs (including the time to checkpoint and restart) in homogeneous environments are smaller than heterogeneous ones. Again, this is the consequence of our data conversion scheme CGT-RMR. Migration/checkpointing schemes under symmetric data conversion standards are not able to achieve such results [5].

5. Related Research

There have been a number of notable attempts at designing process migration and checkpointing schemes. An extension of the V migration mechanism is proposed in [15]. It required both compiler and kernel support. Data has to be stored at the same address in all migrated versions of the process to avoid pointer updating and variant padding patterns in aggregate types. Another approach is proposed by Theimer and Hayes in [16]. Their idea was to construct an intermediate source code representation of a running process at the migration point, migrate the new source code, and recompile it on the destination platform. This requires a low-level, non-portable debugger interface to examine all available process state information on each platform. An extra compilation might incur more delays. The Tui system [12] is an application-level process migration package which utilize a compiler support and a debugger interface to examine and restore process state. It applies an intermediate data format, just as in XDR. Relying on the ACK (Amsterdam Compiler Kit) hurts its portability. Process Introspection (PI) [17] is a general approach for checkpointing and applies the "Receiver Makes Right" (RMR) strategy. Data types are maintained in tables and conversion routines are deployed for all supported platforms. Programmers have to flatten down aggregate data types by themselves. SNOW [14] is another heterogeneous process migration system which tries to migrate live data instead of the stack and heap data. However, with pointer casting and pointer arithmetic, it is virtually impossible to capture the precise process states. PVM installation is a requirement and based on this, communication states are supported.

Virtual machines are the intuitive solution to provide abstract platforms in heterogeneous environments. Some mobile agent systems such as the Java-based IBM Aglet [18]

use such an approach to migrate computation. However, it suffers from slow execution due to interpretation overheads and cannot handle legacy systems.

There are few thread migration systems. Emerald[19] is a new language and compiler designed to support fine-grained object mobility. Compiler-produced templates are used to describe data structures and translate pointers. Arachne[7] supports thread migration by adding three keywords to the C++ language and using a preprocessor to generate pure C++ code. No pointer and heap are supported here. Ariadne[8] achieves thread context-switch by calling C-library setjmp() and longjmp(). On destination nodes, stacks are scanned for pointer detection which can fail and lead to wrong results. Many thread migration systems, such as Millipede[9], adopt "iso-address" strategy. This approach's strict restrictions on resources affect their scalability and make them inapproprate for Grid computing.

6. Conclusions and Future Work

With *MigThread*, flexible combinations of process/thread and migration/checkpointing enable applications, especially parallel computations, to adjust themselves based on dynamic situations in heterogeneous non-dedicated computing environments. Both application performance and system resource utilization are improved.

New data conversion scheme CGT-RMR does not perform actual conversion in homogeneous environments and only converts once in heterogeneous environments. It analyzes data layout, assigns tags to data segments, flattens complex data structures, and converts data only on the receiver side. It reduces coding complexity dramatically and improves overall performance. Unlike other systems, *MigThread* defines, detects and overcomes most unsafe cases automatically. More programs become qualified for migration and checkpointing.

Future work is to add more new functionalities such as "I/O operations forwarding". These I/O operations include communication, file, and other input/output device accesses. Shadow process [6] concept will be applied to forward communication messages or send back I/O commands back to improve transparency and achieve a virtually unchanged working platform. With this feature, virtually all programs will be qualified for migration/checkpointing.

References

[1] I. Foster, C. Kesselman, J. Nick, and S. Tuecke, "Grid Services for Distributed System Integration", Computer, 35(6), pp. 37-46, 2002.

[2] D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler and S. Zhou, "Process Migration", ACM Computing Surveys, 32(8), pp. 241-299, 2000.

[3] H. Jiang and V. Chaudhary, "Compile/Run-time Support for Thread Migration", Proc. of 16th Int'l Parallel and Distributed Processing Symposium, pp. 58-66, 2002.

[4] H. Jiang and V. Chaudhary, "On Improving Thread Migration: Safety and Performance", Proc. of Int'l Conf. on High Performance Computing, pp. 474-484, 2002.

[5] H. Jiang and V. Chaudhary, "Data Conversion for Process/Thread Migration and Checkpointing", Proc. of Int'l Conf. on Parallel Processing, 2003.

[6] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny, "Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System", Technical Report 1346, University of Wisconsin-Madison, April 1997.

[7] B. Dimitrov and V. Rego, "Arachne: A Portable Threads System Supporting Migrant Threads on Heterogeneous Network Farms", IEEE Transactions on Parallel and Distributed Systems, 9(5), pp. 459-469, May 1998.

[8] E. Mascarenhas and V. Rego, "Ariadne: Architecture of a Portable Threads system supporting Mobile Processes", Technical Report CSD-TR 95-017, CS, Purdue Univ., 1995.

[9] A. Itzkovitz, A. Schuster, and L. Wolfovich, "Thread Migration and its Applications in Distributed Shared Memory Systems", J. of Systems and Software, 42(1), pp. 71-87, 1998.

[10] R. Srinivasan, XDR: External Data Representation Stndard, RFC 1832, Aug. 1995.

[11] H. Zhou and A. Geist, ""Receiver Makes Right" Data Conversion in PVM", Proc. of the 14th Int'l Conf. on Computers and Communications, pp. 458-464, 1995.

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

[13] V. Strumpen, "Compiler Technology for Portable Checkpoints", submitted for publication (http://theory.lcs.mit.edu/ strumpen/porch.ps.gz), 1998.

[14] K. Chanchio and X.H. Sun, "Data Collection and Restoration for Heterogeneous Process Migration", Proc. of Int'l Parallel and Distributed Processing Symposium, pp. 51-51, 2001.

[15] C. Shub, "Native Code Process-Originated Migration in a Heterogeneous Environment", Proc. of the 1990 Computer Science Conference, pp. 266-270, 1990.

[16] M. Theimer and B. Hayes, "Heterogeneous Process Migration by Recompilation", Proc. of the 11th Int'l Conf. on Distributed Computing Systems, pp. 18-25, 1991.

[17] A. Ferrari, S. Chapin, and A. Grimshaw, "Process introspection: A checkpoint mechanism for high performance heterogenesous distributed systems", Technical Report CS-96-15, Computer Science Dept., University of Virginia, 1996.

[18] D. Lange and M. Oshima, Programming Mobile Agents in Java - with the Java Aglet API, Addison-Wesley Longman: New York, 1998.

[19] E. Jul, H. Levy, N. Hutchinson, and A. Blad, "Fine-Grained Mobility in the Emerald System", ACM Transactions on Computer Systems, 6(1), pp. 109-133, 1998.