Data Conversion for Process/Thread Migration and Checkpointing*

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

Abstract

Process/thread migration and checkpointing schemes support load balancing, load sharing and fault tolerance to improve application performance and system resource usage on workstation clusters. To enable these schemes to work in heterogeneous environments, we have developed an application-level migration and checkpointing package, MigThread, to abstract computation states at the language level for portability. To save and restore such states across different platforms, this paper proposes a novel "Receiver Makes Right" (RMR) data conversion method, called Coarse-Grain Tagged RMR (CGT-RMR), for efficient data marshalling and unmarshalling. Unlike common data representation standards, CGT-RMR does not require programmers to analyze data types, flatten aggregate types, and encode/decode scalar types explicitly within programs. With help from MigThread's type system, CGT-RMR assigns a tag to each data type and converts non-scalar types as a whole. This speeds up the data conversion process and eases the programming task dramatically, especially for the large data trunks common to migration and checkpointing. Armed with this "Plug-and-Play" style data conversion scheme, MigThread has been ported to work in heterogeneous environments. Some microbenchmarks and performance measurements within the SPLASH-2 suite are given to illustrate the efficiency of the data conversion process.

1. Introduction

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. Although the state-transfer medium differs, migration and checkpointing share the same strategy in state handling. To improve application performance and system resource utilization, they support load balancing, load sharing, data locality optimization, and fault tolerance.

The major obstacle preventing migration and checkpointing from achieving widespread use is the complexity of adding transparent migration and checkpointing to systems originally designed to run stand-alone [1]. Heterogeneity further complicates this situation. But migration and checkpointing are indispensable to the Grid [2] and other loosely coupled heterogeneous environments. Thus, effective solutions are on demand.

To hide the different levels of heterogeneity, we have developed an application level process/thread migration and checkpointing package, *MigThread*, which abstracts the computation state up to the language level [3, 4]. For applications written in the C language, states are constructed in the user space instead of being extracted from the original kernels or libraries for better portability across different platforms. A preprocessor transforms source code at compile time while a run-time support module dynamically collects the state for migration and checkpointing.

The computation state is represented in terms of data. To support heterogeneity, *MigThread* is equipped with a novel "plug-and-play" style data conversion scheme called coarse-grain tagged "Receiver Makes Right" (CGT-RMR). It is an asymmetric data conversion method to perform data conversion only on the receiver side. Since common data representation standards are separate from user applications, programmers have to analyze data types, flatten down aggregate data types, such as structures, and encode/decode scalar types explicitly in programs. With help from *MigThread*'s type system, CGT-RMR can detect data types, generate application-level tags for each of them, and ease the burden of data conversion work previously left to the programmer. Aggregate type data are handled as a whole instead of being flattened down recursively in programs. Therefore,

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

Proceedings of the 2003 International Conference on Parallel Processing (ICPP'03)
0190-3918/03 $ 17.00 © 2003 IEEE

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

Figure 1. 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 2. The transformed function.

compared to common standards, CGT-RMR is more convenient in handling large data chunks. In migration and checkpointing, computation states spread out in terms of memory blocks, and CGT-RMR outperforms normal standards by a large margin. Also, no large routine groups or tables are constructed as in common RMR [8].

Architecture tags are generated on the fly so that new computer platforms can be adopted automatically. CGT-RMR takes an aggressive data conversion approach between incompatible platforms. The low-level data conversion failure events can be conveyed to the upper-level *MigThread* scheduling module to ensure the correctness of real-time migration and checkpointing. Empowered with CGT-RMR, *MigThread* can handle migration and checkpointing across heterogeneous platforms.

The remainder of this paper is organized into seven sections. Section 2 provides an overview of migration and checkpointing. Section 3 discusses data conversion issues and some existing schemes. In Section 4, we provide the detail of designing and implementing CGT-RMR in *MigThread*. Section 5 presents some microbenchmarks and experimental results from real benchmark programs. In Section 6, we discuss related work. Section 7 discusses our conclusions and future work.

2. Migration and Checkpointing

Migration and checkpointing concerns constructing, transferring, and retrieving computation states. Despite the complexity of adding transparent support, migration and checkpointing continue to attract attention due to the potential for computation mobility.

2.1. *MigThread*

MigThread is an application-level multi-grained migration and checkpointing package [3], which supports both coarse-grained processes and fine-grained threads. *MigThread* consists of two parts: a preprocessor and a run-time support module.

The preprocessor is designed to transform a user's source code into a format from which the run-time support module can construct the computation state efficiently. Its power can improve the transparency drawback in application-level schemes. The run-time support module constructs, transfers, and restores computation states dynamically [4].

Originally, the state data consists of the process data segment, stack, heap 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. Therefore, the physical state is transformed into a logical form to achieve platform-independence. All related information with regard to stack variables, function parameters, program counters, and dynamically allocated memory regions, is collected into pre-defined data structures [3].

Figures 1 and 2 illustrate a simple example for such a process, with all functions and global variables transformed accordingly. A simple function **foo()** is defined with four local variables as in Figure 1. *MigThread*'s preprocessor transforms the function and generates a corresponding **MTh_foo()** shown in Figure 2. All non-pointer variables are collected in a structure *MThV* while pointers are moved to another structure *MThP*. Within *MThV*, field *MThV.MThP* is the only pointer, pointing to the second structure, *MThP*, which may or may not exist. Field *MThV.stepno* is a logical construction of the program counter to indicate the program progress and where to restart. In process/thread 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.

Since address spaces could be different on source and destination machines, values of pointers referencing stacks or heaps might become invalid after migration. 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. *MigThread* also supports user-level memory management for heaps. Eventually, all state related contents, including stacks and heaps, are moved out to the user space and handled by *MigThread* directly.

2.2. Migration and Checkpointing Safety

Migration and checkpointing safety concerns ensuring the correctness of resumed computation [4, 5]. In other words, computation states should be constructed precisely, and restored correctly on similar or different machines. The major identified unsafe factors come from unsafe type systems (such as the one in C) and third-party libraries [4]. But for heterogeneous schemes, if data formats on different machines are incompatible, migration and resuming execution from checkpoint files might lead to errors. This requires that upper level migration/checkpointing schemes be aware of the situation in lower level data conversion routines.

MigThread supports aggressive data conversion and aborts state restoration only when "precision loss" events occur. Thus, the third unsafe factor for heterogeneous schemes, incompatible data conversion, can be identified and handled properly.

3. Data Conversion

Computation states can be transformed into pure data. If different platforms use different data formats and computation states constructed on one platform need to be interpreted by another, the data conversion process becomes unavoidable.

3.1. Data Conversion Issues

In heterogeneous environments, common data conversion issues are identified as follows:

- **Byte Ordering** : Either big endian or little endian.
- **Character Sets** : Either ASCII or EBCDIC representation.
- **Floating Point Standards** : IEEE 754, IEEE 854, CRAY, DEC or IBM standard.
- **Data Alignment and Padding** : Data is naturally aligned when the starting address is on a "natural boundary." This means that the starting memory address is a multiple of the data's size. Structure alignment can result in unused space, called *padding*. Padding between members of a structure is called *internal padding*. Padding between the last member and the end of the space occupied by the structure is called *tail padding*. Although natural boundary can be the default setting for alignment, data alignment is actually determined by processors, operating systems, and compilers. To avoid such indeterministic alignment and padding, many standards flatten native aggregate data types and re-represent them in their own default formats.
- **Loss of Precision** : When high precision data are converted to their low precision counter-parts, loss of precision may occur.

3.2. Data Conversion Schemes

Data representations can be either tagged or untagged. A tag is any additional information associated with data that helps a decoder unmarshal the data.

Canonical intermediate form is one of the major data conversion strategies which provides an external representation for each data type. Many standards adopt this approach, such as XDR (External Data Representation) [6] from Sun Microsystems, ISO ASN.1 (Abstract Syntax Notation One) [7], CCSDS SFDU (Standard Formatted Data Units), ANDF (Architecture Neutral Data Format), IBM APPC GDS (General Data Stream), ISO RDA (Remote Data Access), and others [8]. Such common data formats are recognized and accepted by all different platforms to achieve data sharing. Even if both the sender and receiver are on the same machine, they still need to perform this symmetric conversion on both ends. XDR adopts the untagged data representation approach. Data types have to be determined by application protocols and associated with a pair of encoding/decoding routines.

Zhou and Geist [8] took another approach, called "receiver makes it right", which performs data conversion only on the receiver side. If there are n machines, each of a different type, the number of conversion routine groups will be $(n^2 - n)/2$. In theory, the RMR scheme will lead to bloated code as n increases. Another disadvantage is that RMR is not available for newly invented platforms.

4. Coarse-Grain Tagged RMR in *MigThread*

The proposed data conversion scheme is a "Receiver Makes Right" (RMR) variant which only performs the conversion once. This tagged version can tackle data alignment and padding physically, convert data structures as a whole, and eventually generate a lighter workload compared to existing standards.

An architecture tag is inserted at the beginning. Since the byte ordering within the network is big-endian, simply comparing data representation on the platform against its format in networks can detect the endianness of the platform. Currently *MigThread* only accepts ASCII character sets and is not applicable on some IBM mainframes. Also, IEEE 754 is the adopted floating-point standard because of its dominance in the market, and *MigThread* can be extended to other floating-point formats.

4.1. Tagging and Padding Detection

For data conversion schemes, the tagged approaches associate each data item with its type attribute so that receivers can decode each item precisely. With this, fewer conversion routines are required. But tags create an extra workload and slow down the whole process. However, untagged approaches maintain large sets of conversion routines and encoding/decoding orders have to be handled explicitly in application programs. Performance improvement comes from the extra coding burden.

In existing data format standards, both tagged and untagged approaches handle basic (scalar) type data on a one-by-one basis. Aggregate types need to be flattened down to a set of scalar types for data conversion. The main reason is to avoid the padding issue in aggregate types. Since the padding pattern is a consequence of the processor, operating system, and compiler, the padding situation only becomes deterministic at run-time. It is impossible to determine a padding pattern in programs and impractical for programmers to convey padding information from programs to conversion routines at run-time. This is because programs can only communicate with conversion routines in one direction and programming models have to be simple. Most existing standards choose to avoid padding issues by only handling scalar types directly.

MigThread is a combination of compile-time and run-time supports. Its programmers do not need to worry about data formats. The preprocessor parses the source code, sets up type systems, and transforms source code to communicate with the run-time support module through inserted primitives. With the type system, the preprocessor can analyze data types, flatten down aggregate types recursively, detect padding patterns, and define tags. 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 the global control, low-level issues such as the 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 their padding situations so that data conversion routines can handle aggregate types as well as common scalar types. As we discussed in Section 2, global variables and function local variables in *MigThread* are collected into their corresponding structure type variables *MThV* and *MThP* which are registered as the basic units. Tags are defined and generated for these structures as well as dynamically allocated memory blocks in the heap.

For the simple example in Section 2 (Figures 1 and 2), tag definitions of *MThV_heter* and *MThP_heter* for *MThV* and *MThP* are shown in Figure 3. It is still too early to determine the content of the tags within programs. 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 4.

```
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 *);

  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 3. 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 4. Tag calculation at run-time.

A tag is a sequence of *(m,n)* tuples, and can be expressed in one of the following cases (where m and n are positive numbers):

- (m, n) : scalar types. The item "m" is simply the size of the data type. The "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 popular case and indicates no padding.

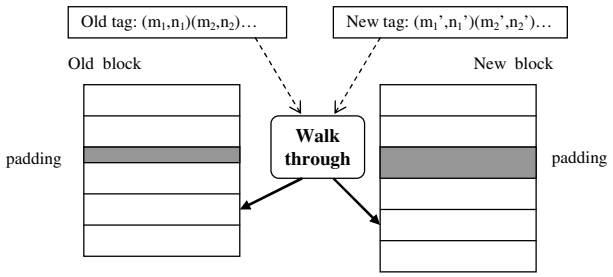

Figure 5. Walk through "tag-block" format segments.

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

4.2. Data Restoration

Each function contains one or two structures and corresponding tags depending on whether *MThP* exists. In *MigThread*, all memory segments for these structures are represented in a "tag-block" format. The process/thread stack becomes a sequence of *MThV*, *MThP* and their tags. Memory blocks in heaps are also associated with such tags to express 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. To support heterogeneity, *MigThread* executes data conversion routines against these coarse-grained memory segments instead of the individual data object. Performance gains are guaranteed.

The receivers or reading processes of 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 blocks have the same sizes, this means the platform remains 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 5. In these segments, according to their tags, memory blocks are viewed to consist of scalar type data and padding slots alternatively. 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 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 index pointers 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.

4.3. Data Resizing

Between incompatible platforms, if data items are converted from higher precision formats to lower precision formats, precision loss may occur. Normally higher precision format data are longer so that the high end portion cannot be stored in lower precision formats. But if the high end portions contain all-zero content, it is safe to throw them away since data values still remain unchanged. *MigThread* takes this aggressive strategy and intends to convert data until precision loss occurs. More programs are qualified for migration and checkpointing. Detecting incompatible data formats and conveying this low-level information up to the scheduling module can help abort data restoration promptly for safety.

4.4. Plug-and-play

We declare CGT-RMR as a "plug-and-play" style scheme, and it does not maintain tables or routine groups for all possible platforms. Since almost all forthcoming platforms are following the IEEE floating-point standard, no special requirement is imposed for porting code to a new platform. However, adopting old architectures such as IBM mainframe, CRAY and DEC requires some special conversion routines for floating-point numbers.

5. Microbenchmarks and Experiments

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. The CGT-RMR scheme is applied for data conversion in migration and checkpointing between these two different machines.

PVM (Parallel Virtual Machine) uses the XDR standard for heterogeneous computing. Thus, some process migration schemes, such as SNOW [12], apply XDR indirectly by calling PVM primitives. Even the original RMR implementation was based on XDR's untagged strategy [8]. Since

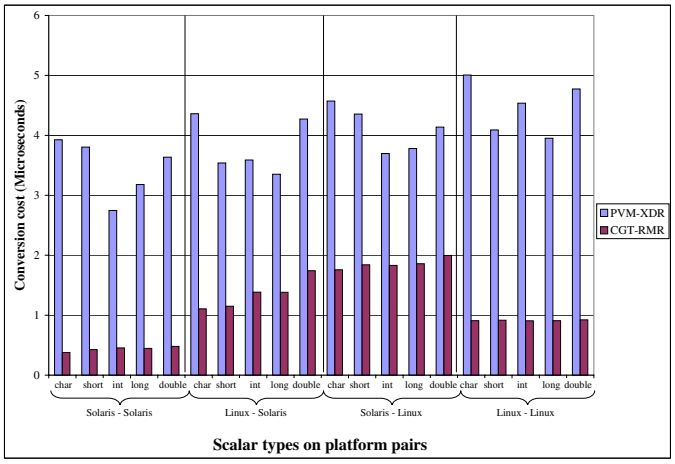

Figure 6. Conversion costs for scalar types.

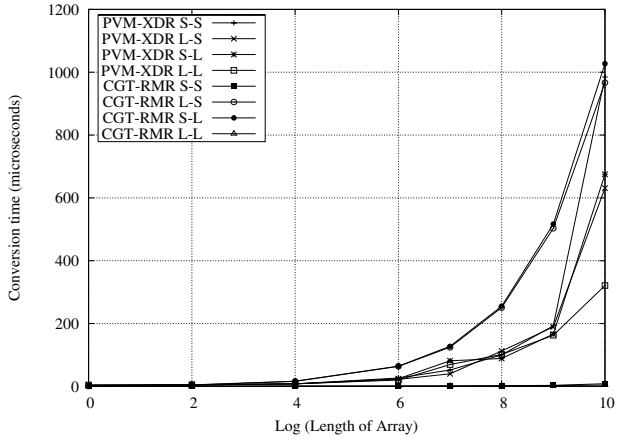

Figure 7. Conversion costs of integer arrays.

most existing systems adopt XDR or similar data conversion strategies [11, 5, 12], we compare our CGT-RMR with XDR implementation in PVM to predict the performance of *MigThread* and other similar systems.

The costs of converting scalar data types are shown in Figure 6. Data are encoded on one platform and decoded on another platform. For scalar types, such as char, short, int, long and double, the PVM's XDR implementation (PVM-XDR) is slower than CGT-RMR, which is even faster in homogeneous environments since no conversion actually occurs. Also XDR forces the programmer to encode and decode data even on the same platforms. Figure 6 indicates that CGT-RMR can handle basic data units more efficiently than PVM-XDR.

To test the scalability, we apply the two schemes on integer and structure arrays. Figure 7 shows an integer array's behavior. In homogeneous environments, i.e., both encoding and decoding operations are performed on either the Solaris or Linux machine, CGT-RMR demonstrates virtually no cost and excellent scalability. In heterogeneous

```
struct {
   char     a;
   short    b;
   int      c;
   long     d;
   double   e;
} s[n];
```

Figure 8. The structure array.

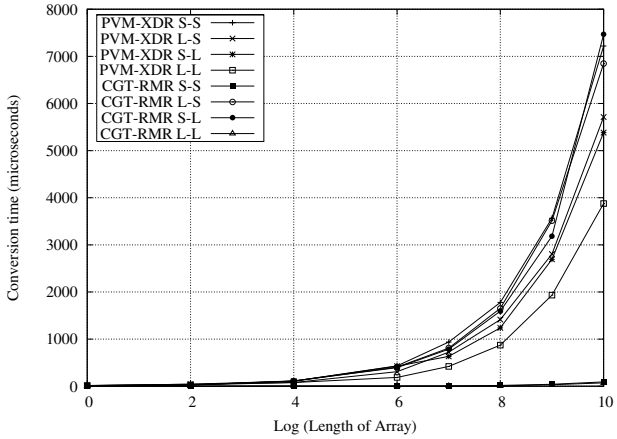

Figure 9. Conversion costs of structure arrays.

environments, i.e., encoding data on Solaris or Linux and decoding data on a different machine, CGT-RMR incurs a little more overhead, shown by the top two curves. The four curves in the middle are from PVM-XDR which does not vary much by different platform combinations, nor can it take advantage of homogeneous environments. This indicates that PVM-XDR has a little better scalability on scalar type arrays in heterogeneous environments.

The conversion overheads of structure arrays are simulated in Figure 9. Figure 8 lists the sample structure array which contains 5 common scalar type fields with different data alignment requirements. Again, in a homogeneous environment, CGT-RMR causes virtually no overhead. Its heterogeneous cases, the top two curves start merging with the 4 PVM-XDR curves. Because of the padding issue in structures, programmers have to encode/decode each field explicitly which diminishes XDR's advantage in scalar arrays and incurs tremendous programming complexity. In the simple case shown in Figure 8, assuming that n is the number of scalar types, there will be $10n$ encoding and decoding statements hand-coded by programmers. In CGT-RMR, only one primitive is required on each side, and the preprocessor can handle all other tag generation details automatically. Therefore, CGT-RMR eases the coding complexity dramatically in complex cases such as migration and checkpointing schemes where large computation states are common.

To evaluate the CGT-RMR strategy in real applications,

Table 1. 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

we apply it on FFT, continuous and non-continuous versions of LU from the SPLASH-2 suite, and matrix multiplication applications. We predefine the adaptation points for migration or checkpointing. The detailed overheads are listed in Table 1. With certain input sizes, applications are paused on one platform to construct computation states whose sizes can vary from 78K to 2M bytes in these sample programs. Then the computation states are transferred to another platform for migration, or saved into file systems and read out by another process on another platform for checkpointing.

CGT-RMR plays a role in data conversion in stacks, data conversion in heaps, and pointer updating in both areas. In FFT and LU-n, large numbers of memory blocks are dynamically allocated in heaps. In homogeneous environments, stacks and heaps are recreated without data conversion. But in heterogeneous environments, converting data in large heaps dominates the CPU time. On the other hand, LU-c and MatMult are deployed as pointer-intensive applications. When computation states are restored, pointer updating is an unavoidable task. In homogeneous environments with no data conversion issue, the CPU simply devotes itself to pointer updating. Even in heterogeneous cases, pointer updating is still a major issue although their large heaps also incur noticeable overheads. The time spent on stacks is negligible. It is clear that overhead distribution is similar for both homogeneous and heterogeneous environments in XDR or similar standards. CGT-RMR runs much faster in homogeneous environments and is similar in performance of XDR in heterogeneous environments.

From the microbenchmarks, we can see that CGT-RMR takes less time in converting scalar type data and provides distinct advantages in programming complexity. XDR only shows limited advances in scalar array processing with tremendous coding effort from programmers. The experiments on real applications detail the overhead distribution and indicate that CGT-RMR helps provide a practical migration and checkpointing solution with minimal user involvement and satisfactory performance.

6. Related Work

There have been a number of notable attempts at designing process migration and checkpointing schemes, however, few implementations have been reported in literature with regard to the fine-grain thread migration and checkpointing. An extension of the V migration mechanism is proposed in [9]. It requires both compiler and kernel support for migration. 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. Obviously this constraint is inefficient or even impossible to meet across different platforms.

Another approach is proposed by Theimer and Hayes in [10]. 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. An extra compilation might incur more delays. Efficiency and portability are the drawbacks.

The Tui system [5] is an application-level process migration package which utilizes compiler support and a de-

bugger interface to examine and restore process states. It applies an intermediate data format to achieve portability across various platforms. Just as in XDR, even if migration occurs on the same platform, data conversion routines are still performed twice.

Process Introspection (PI) [11] uses program annotation techniques as in *MigThread*. PI 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. Aggregate data types are still flattened down to scalar types (e.g., int, char, long, etc) to avoid dealing with data alignment and padding. *MigThread* does this automatically.

SNOW [12] is another heterogeneous process migration system which tries to migrate live data instead of the stack and heap data. SNOW adopts XDR to encode and decode data whereas XDR is slower than the RMR used in PI [8]. PVM installation is a requirement.

Virtual machines are the intuitive solution to provide abstract platforms in heterogeneous environments. Some mobile agent systems such as the Java-based IBM Aglet [13] use such an approach to migrate computation. However, it suffers from slow execution due to interpretation overheads.

7. Conclusions and Future Work

We have discussed a data conversion scheme, (CGT-RMR), which enables *MigThread* to not flatten down aggregate types into scalar (primitive) data types within data conversion routines. In fact, type flattening takes place in the tag definition process which is conducted by *MigThread*'s preprocessor. The contents of tags are determined at run-time to eliminate possible alignment affecting factors from CPUs, operating systems, and compilers. Since tags help resolve data alignment and padding, CGT-RMR provides significant coding convenience to programmers and contributes a feasible data conversion solution in heterogeneous migration and checkpointing.

Performing data conversion only on the receiver side, CGT-RMR exhibits tremendous efficiency in homogeneous environments and performance similar to XDR in heterogeneous environments. Without tables or special data conversion routines, CGT-RMR adopts "plug-and-play" design and can be applied on new computer platforms directly.

Our future work is to build a new data conversion API so that programmers can use CGT-RMR directly rather than through *MigThread*. To accomplish a universal data conversion scheme, CGT-RMR requires *MigThread*'s preprocessor as its back-end because the preprocessor's type system is still indispensable. Working as a stand-alone standard such as XDR, CGT-RMR will benefit other migration and checkpointing schemes, and any applications running in heterogeneous environments.

References

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

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

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

[4] H. Jiang and V. Chaudhary, "On Improving Thread Migration: Safety and Performance", *Proc. of the International Conference on High Performance Computing*(HiPC), pp. 474-484, 2002.

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

[6] R. Srinivasan, "XDR: External Data Representation Stndard", RFC 1832, http://www.faqs.org/rfcs/rfc1832.html, Aug. 1995.

[7] C. Meryers and G. Chastek, "The use of ASN.1 and XDR for data representation in real-time distributed systems", Technical Report CMU/SEI-93-TR-10, Carnegie-Mellon University, 1993.

[8] H. Zhou and A. Geist, ""Receiver Makes Right" Data Conversion in PVM", *In Proceedings of the 14th International Conference on Computers and Communications*, pp. 458– 464, 1995.

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

[10] M. Theimer and B. Hayes, "Heterogeneous Process Migration by Recompilation", *In Proceedings of the 11th International Conference on Distributed Computing Systems*, Arlington, TX, pp. 18-25, 1991.

[11] A. Ferrari, S. Chapin, and A. Grimshaw, "Process Introspection: A Checkpoint Mechanism for High Performance Heterogenesous Distributed Systems", Technical Report CS-96-15, University of Virginia, Department of Computer Science, October, 1996.

[12] K. Chanchio and X. Sun, "Data collection and restoration for heterogeneous process migration", *Software - Practice and Experience*, 32(9), pp. 845-871, 2002.

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

[14] "PVM: Parallel Virtual Machine", http://www.csm.ornl.gov/pvm/pvm_home.html.