Compile/Run-time Support for Thread Migration*

Hai Jiang
Institute for Scientific Computing
Wayne State University
Detroit, MI 48202
haj@cs.wayne.edu

Vipin Chaudhary
Institute for Scientific Computing
Wayne State University
and Cradle Technologies, Inc.
vipin@wayne.edu

Abstract

This paper describes a generic mechanism to migrate threads in heterogeneous distributed environments. To maintain high portability and flexibility, thread migration is implemented at language level. At compile-time, a preprocessor scans the C and C++ programs to build thread state, detects possible thread migration points, and transforms the source code accordingly. Run-time support helps migrate threads physically. Since the physical thread state is transformed into a logical form, and pointers and dynamically allocated memory in heap are supported, the proposed solution places no restriction on thread types and migration-enabled systems. We implemented this approach in Strings: a multithreaded software distributed shared memory system. Some microbenchmarks and performance measurements on SPLASH-2 suite are reported.

1 Introduction

A Network of Workstations (NOW) provides an opportunity to support high-performance parallel applications within an everyday computing infrastructure. Studies have indicated that a large fraction of workstations could be unused for a large fraction of time[10]. Batch-processing systems that utilize idle workstations for running sequential jobs have been in production use for many years. However, the utility of harvesting idle workstations for parallel computation is less clear. When a workstation running a parallel job is reclaimed by its primary user, the remaining processes of the same job have to stop. To make progress, a parallel job requires that a group of processor be continuously available for a sufficiently long period of time. If the state of a large number of processors rapidly oscillates between *available* and *busy*, a parallel computation will be able to make little progress even if each processor is available for a large fraction of time. Also, parallel programs are often not perfectly parallel, they are able to run only on certain configurations - for example, configurations with 2^n processors. Addition or deletion of a single workstation may have no effect, a small effect or a very significant effect on the performance depending on the application requirements and the number of available machines.

In order for the parallel computation to proceed, one should allow the computation to be reconfigured, especially in terms of the degree of parallelism, or the number of processors required. Reconfiguration may need one or more of data and loop repartitioning, data and/or process migration, and updating data location information. To exploit idle cycles on remote workstations, thread/process migration is the necessity. Compared to processes, threads need shorter time for creation, context switch, and synchronization so that threads are more efficient. Multi-threading helps achieve better load distribution between nodes in the system by splitting the application into smaller chunks of work [2]. More efficient threads allow programs to be finer-grained, which benefits both structure and performance [11]. Consequently, thread migration is the first step to exploit idle cycles and balance load in NOWs. Recently, thread migration is becoming increasingly important as distributed shared memory (DSM) systems and meta-computing become efficient alternatives to traditional supercomputing.

The semantic of thread migration is to stop the thread computation on the source node, migrate the thread state to the destination node, and resume the execution at the statement following the migration point on the destination node. In homogeneous environments, similar hardware and software configurations ease thread migration. But in heterogeneous systems, such as meta-computing environments, a generic thread migration scheme is necessary.

In this paper, we describe the design and implementation of a thread migration and make the following contributions:

- Design thread migration scheme for any type of thread on heterogeneous systems.
- Translate physical thread state into a logical form so

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

1530-2075/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: SUNY Buffalo. Downloaded on October 23, 2008 at 10:57 from IEEE Xplore. Restrictions apply.

that it could be restored easily on different machines.

- Design and implement a preprocessor to work with regular C/C++ compilers. The preprocessor is used only if the users want to add the thread migration feature into their applications.
- Detect migration points automatically and insert essential statements to dynamically contact the run-time support subsystem, which communicates with the global scheduler to migrate threads across nodes in a network of workstations.
- Support dynamically allocated memory in heap.
- Update all pointers correctly on destination nodes.
- Place no restriction on thread stacks and local heaps.

We implement thread migration on our multi-threaded software DSM system: *Strings*[9].

The remainder of this paper is organized as follows: In Section 2 we discuss related work. Section 3 gives the overview of *Strings* and the thread migration strategy. Section 4 describes the thread migration design and implementation details. In Section 5, we analyze the migration overhead. Section 6 presents some microbenchmarks to evaluate the performance of thread migration. In Section 7, we show some experiment results from real benchmark programs. We wrap up with conclusions and continuing work in Section 8.

2 Related Research

A lot of research has been conducted in the general area of thread migration. The core of thread migration is about how to transfer thread state and necessary data in local heap to the destination. However, since the addresses on the destination machine may be different from the original ones, internal self-referential pointers may no longer be valid. There are three approaches in the research literature to deal with this problem.

The first approach is to use language and compiler support to maintain enough type information and identify pointers[7, 6]. This approach implies that thread migration will be bound to languages instead of operating systems or platforms. Emerald[3] and Arachne[6] are examples of this approach.

Emerald[3] is a new language and compiler designed to support fine-grained object mobility. Compiler-produced templates are used to describe the format of data structures and help translate pointers. Arachne[6] supports thread migration between heterogeneous platforms with dynamic stack size management. It adds three keywords to the C++ language and a preprocessor is used to generate pure C++ code. No pointers are supported here.

The second approach requires scanning the stacks at run-time to detect and translate the possible pointers dynamically[1]. Since some pointers in stack cannot possibly be detected (as pointed out by [2]), the resumed execution can be incorrect. The representative implementation of this is Ariadne[1] which is a user-space threads library. It achieves thread migration over thread context-switch by calling C-library setjmp() and longjmp() primitives. On the destination node, the migrated stack is scanned to identify and translate possible pointers. It is possible that some misidentified pointers incur wrong execution. Moreover, pointers referencing data in heap will not be updated because these data are not migrated.

The third approach is most popular and 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 to its corresponding ones on other machines. This solution requires large address space and is not scalable. The total number of threads and the stack size is limited by the address space of single node. Even though the 64-bit address architectures will ease this restriction[2, 7, 12] in the near future, it is not a desired long term solution. Another drawback of this “iso-address” approach is that thread migration is restricted to homogeneous systems.

Amber[4] supports data and thread migration and the location of objects is managed explicitly by applications. UPVM system[5] utilizes User Level Processes (ULPs) with characteristics from both threads and processes. The migrated ULP state includes context, stack, private data, and heap. The private heap could worsen the severe memory limitations. PM2[13] is a DSM system whose protocols may mix page replication, page migration, and thread migration. Millipede[2] is a system which migrates lightweight processes on Windows-NT. The thread migration in the Nomad system[14] is designed to be very lightweight, and in principle involves only the migration of the top page of the thread’s stack and the current register values to the new node[14]. Missing stack pages are fetched later. This could reduce the startup time. This is the only approach whose migration time is not closely related to the stack size of the thread. In the approach by Cronk et al.[8], each thread has its own heap, and both the heap and stack are migrated. This places a limitation on the amount of heap space for a thread.

3 Thread Migration in *Strings*

Currently we implement a generic thread migration scheme on *Strings*[9], a multi-threaded DSM system.

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: SUNY Buffalo. Downloaded on October 23, 2008 at 10:57 from IEEE Xplore. Restrictions apply.

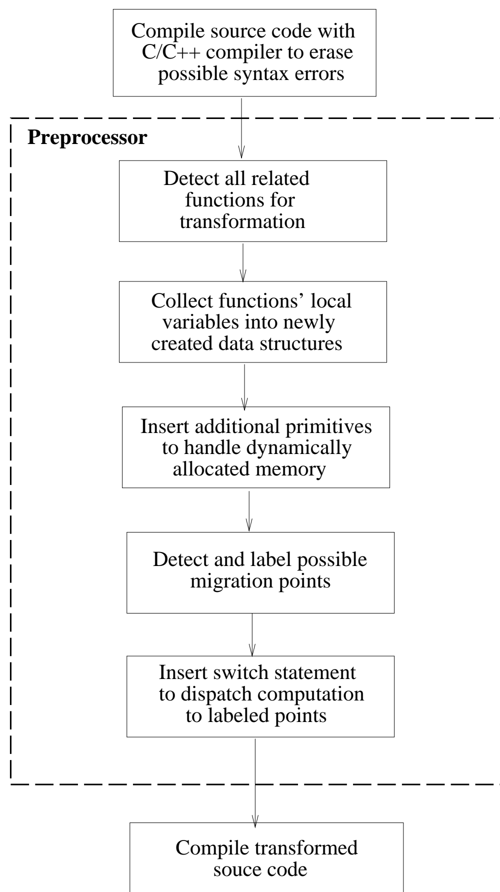

Figure 1. Flow chart for compile-time support.

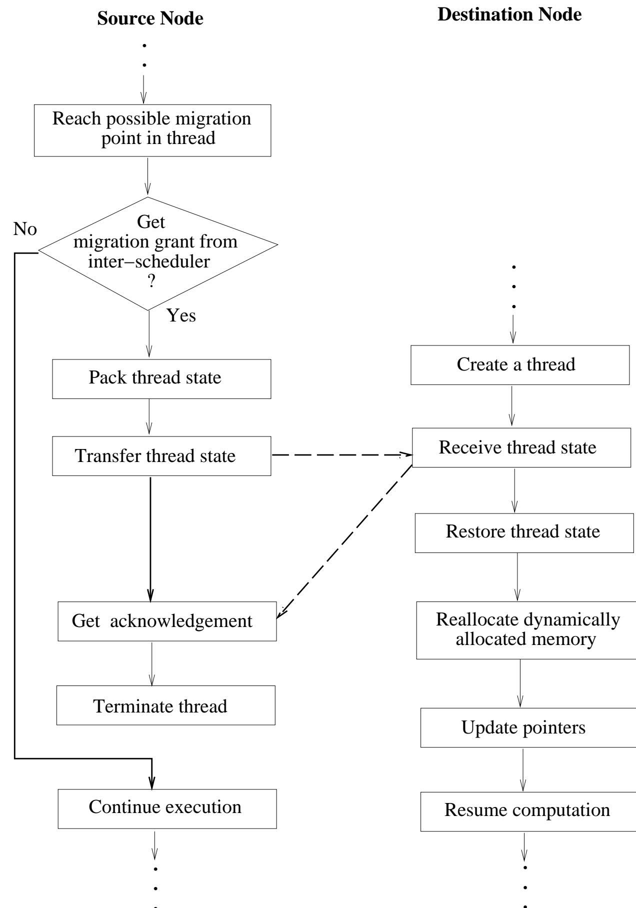

Figure 2. Flow chart for run-time support.

3.1 Strings

Strings is built using POSIX threads, which can be multiplexed on kernel lightweight processes. The kernel can schedule these lightweight processes across multiple processors on symmetrical multiprocessors (SMPs) for better performance. *Strings* is designed to exploit data parallelism at the application level and task parallelism at the run-time level.

3.2 Thread migration

To reduce the restriction on thread migration, our approach uses language level support to transform the physical thread state into a logical form. Thus, it could be used in heterogeneous environments for any type of thread libraries or subsystems. Both the portability and the scalability of stack are improved. There is no need to preallocate addresses of stacks on each machine which is too costly and almost impossible in meta-computing environments. The limitation on virtual memory address space is also removed. Since thread data in heap is migrated with stack and restored remotely, dynamically allocated memory is supported by this approach. Pointers referencing stack or heap are represented in a generic manner so that they could be updated precisely on destination nodes. Compile-time support is built in a preprocessor compatible with traditional C/C++ compilers. There are no new languages and compilers as in Arachne[6] and Emerald[3]. The preprocessor is run only when users want to add thread migration functionality into their applications, not an essential step as in Arachne. An overview of the tasks involved during the compile-time and run-time process of our thread migration strategy is given in Figures 1 and 2, respectively.

3.3 Migration points

The overheads associated with destroying a thread, transferring thread state, creating a thread and initiating remote execution are relatively high. Therefore, a thread should have sufficient amount of computation and two consequent migrations should keep distant spans in order to minimize

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: SUNY Buffalo. Downloaded on October 23, 2008 at 10:57 from IEEE Xplore. Restrictions apply.

```
void  test(int arg1, int *arg2)
{
   int   i;
   int  *list;

   list = arg2;
   for (i=0; i<arg1; i++)
       printf("%d", list[i]);

   STR_barrier();
   printf("\n Done.\n");
   return;
}
```

Figure 3. Original function definition.

the impact from the cost of thread migration.

If the memory model of the distributed system is the traditional sequential consistency, then the system appears like a multiprogrammed uniprocessor and threads can be migrated randomly with a guarantee of correctness of resumed execution. However, software DSMs, for better performance, adopt relaxed memory models such as release consistency model to reduce both the number of messages and the amount of data transfers between processors. In such models, some virtually shared data between two synchronization points or barriers could be in inconsistent states. If migrated threads access such data in the destination node, the resumed computation could be incorrect, especially when they are used as input data. To ensure correctness, thread migration can only be allowed at synchronization points or barriers. The preprocessor scans the application code, detects synchronization points automatically, and inserts suitable thread migration primitives. However, users can insert migration primitives at any points they prefer. But they need to ensure the correctness after migration. Note that *Strings* uses release consistency model.

```
void  STR_test(int arg1, int *arg2)
{
   struct sr_var_t {
       void   *sr_ptr;
       int     stepno;
       int     arg1;
       int     i;
   } sr_var;

   struct sr_ptr_t {
       int     *arg2;
       int     *list;
   } sr_ptr;

   sr_var.sr_ptr = (void *)&sr_ptr;
   sr_var.stepno = 0;

   STR_mig_init((void *)&sr_var,
       sizeof(struct sr_var_t),
       (void *)&sr_ptr,
       sizeof(struct sr_ptr_t));

   if (sr_var.stepno == 0) {
      sr_var.arg1 = arg1;
      sr_ptr.arg2 = arg2;
   }
   else {
      arg1 = sr_var.arg1;
      arg2 = sr_ptr.arg2;
   }

   switch(sr_var.stepno) {
      case 0:
          break;
      case 1:
          goto STR_step_1;
          break;
      default:
          break;
   }


   sr_ptr.list = sr_ptr.arg2;
   for (sr_var.i=0; sr_var.i<sr_var.arg1;
           sr_var.i++)
     printf("%d", sr_ptr.list[sr_var.i]);

   STR_barrier();

   sr_var.stepno = 1;
   STR_checkpoint();
   STR_step_1:

   printf("\n Done.\n");
   return;
}
```

Figure 4. Transformed function definition.

4 Design and Implementation of Thread Migration

The proposed strategy is implemented as a combination of compile/run-time modules.

4.1 Compile-time support

To support thread migration in heterogeneous environments, the physical thread state needs to be transformed into a logical one for portability. In our system, compile-time support helps achieve this at the language level instead of burying it in run-time library or kernel. Currently it supports C/C++ and can be extended to other languages easily.

A thread state consists of a program counter (PC), a set of registers, and a stack of procedure records containing variables local to each procedure. We abstract the thread state up to the language level by representing the program counter and the local variables of procedures in two data structures at compile-time. The stack and dynamic memory management is done at run-time. No information from low level devices, such as registers, needs to be retrieved. Thus, the control of thread is moved up to the application level.

4.1.1 Data variables

Threads can access three types of variables:

- Global variables shared by all threads on any machine in the system.
- Variables shared by all threads in the current process on a local machine.

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: SUNY Buffalo. Downloaded on October 23, 2008 at 10:57 from IEEE Xplore. Restrictions apply.

- Variables local to procedures.

The global shared variables are placed in the global shared memory areas in *Strings*. Page/object migration is the essential tool for data consistency. Different memory models follow different protocols to achieve this goal, thereby constraining thread migration only to synchronization or barrier points. The access to variables shared among local threads in the current process is forbidden during thread migration due to the difficulty in identifying the scope of data sharing. The local variables, which represent the state of the threads, are transferred during thread migration.

During migration, the values of all local variables for related functions should be packed, transferred to destination nodes, and restored remotely. To speed up this process, the preprocessor collects all local variable declarations, puts them into a local structure sr_var at the beginning of the functions, and migrates sr_var (instead of searching and transferring local variables individually at run-time). The preprocessor scans all the functions, locates all references to the original local variables and replaces them with the references to the corresponding fields in sr_var. For example, the declaration of local variable i, as shown in Figure 3, will be moved into sr_var, and all references to i in the thread function will be changed to $sr_var.i$, as in Figure 4. This tedious task is done by the preprocessor.

4.1.2 Pointers

Pointers are the main problem in thread migration because the memory address space on destination node could be totally different. Pointers will become invalid unless they are updated accordingly. This updating is handled at run-time and could be very complicated. To simplify this, many systems choose to reserve memory space on all machines to keep the pointers valid. In our approach, pointers are identified at language level and collected by the preprocessor into a structure type variable sr_ptr, as shown in Figure 4.

On the destination node, the memory area for sr_ptr is scanned to translate all the pointers. If some structure type variables in sr_var or pointers of structure variables in sr_ptr contain pointer fields, these need to be referenced by new pointer variables in sr_ptr and initialized right after the sr_ptr declaration. All pointer declarations are moved to sr_ptr or represented by newly created fields in sr_ptr. All the above work is done by the preprocessor.

Overall, there are two structure type variables created by the preprocessor: sr_var and sr_ptr. Since some functions may not contain pointers, sr_ptr could be missing. For detecting the existence of sr_ptr at run-time, $sr_var.sr_ptr$ in sr_var is used to keep the address of sr_ptr. To keep the records of sr_var and sr_ptr, both their address and size are passed into **STR_mig_init()** during function initialization.

4.1.3 Function parameters

In C/C++, the parameters of a function carry information and status, and consequently the state of threads. Therefore, this information needs to be restored on the destination nodes. Fields with the same types and names are defined in sr_var or sr_ptr depending on whether they are 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 to these fields. This strategy benefits from the "pass-by-copy" function calling in C/C++.

4.1.4 Program counter

The program counter (PC) is the memory address of the current execution point within a program. It indicates the start of computation after the migration. 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 $sr_var.stepno$ in each affected function, as shown in Figure 4.

Since all possible migration points have been detected at compile-time, different **integer** values of $sr_var.stepno$ correspond to different migration points. In our approach, **STR_barrier()** is a typical synchronization point. The preprocessor appends **STR_checkpoint()** after the barrier, **STR_barrier()**, to contact the scheduler for any possible thread migration. If the migration request is granted, the run-time support will help execute the migration procedure and the computation should be resumed right after this statement on the destination node. We set the value of $sr_var.stepno$ and label the jump locations as STR_step_n (see Figure 4).

In the preprocessor transformed code, a **switch** statement is inserted to dispatch execution to each labeled point according to the value of $sr_var.stepno$, and executed after the function initialization. Non-zero values of $sr_var.stepno$ indicate that this thread was migrated and has executed on another node (partially). However, a zero value does not indicate that this thread was not migrated since migration could take place before reaching the current function. The **switch** and **goto** statements help control jump to resumption points quickly.

4.1.5 Preprocessor

The compile-time support is implemented in our preprocessor that is based on LEX. We borrow tokens generated by LEX and analyze their relations according to C/C++ syntax and semantics. The high level scheme is shown in Figure 1. First, the user compiles his program using a conventional C/C++ compiler. Then, the preprocessor phase is used if thread migration is needed.

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: SUNY Buffalo. Downloaded on October 23, 2008 at 10:57 from IEEE Xplore. Restrictions apply.

The overall transformation procedure is outlined as follows (refer to Figure 4 too):

1. Rename function names to **STR_xxx()**.
2. Collect non-pointer variables and function parameters into sr_var that contain the fields $sr_var.sr_ptr$ and $svar_var.stepno$.
3. Collect pointer variables and function parameters into sr_ptr.
4. Set $sr_var.sr_ptr$ to reflect the existence of sr_ptr.
5. Set $sr_var.stepno$ to zero.
6. Register the addresses and sizes of sr_var and sr_ptr through **STR_mig_init()** to the run-time support subsystem.
7. Set function parameter fields in sr_var and sr_ptr according to the migration status.
8. Construct a **switch** statement to dispatch execution to the proper location according to the value of $sr_var.stepno$.
9. Scan the original code to replace references to local variables and parameters with the ones to the corresponding fields in sr_var and sr_ptr.
10. Detect synchronization points and barriers, such as **STR_barrier()**, or other suitable migration points, and insert the following statements after them:
    (a) Set the value of $sr_var.stepno$.
    (b) Insert migration primitive, such as **STR_checkpoint()** to contact the scheduler for possible thread migration permission.
    (c) Insert proper label STR_step_n to resume computation on destination machines.
11. Detect other transformed user functions, and do the following around them:
    (a) Set the value of $sr_var.stepno$ before calling these functions.
    (b) Insert proper label STR_step_n ahead.
    (c) Append cleanup primitives **STR_mig_cleanup()** afterwards to pop the top record on stack for the last finished function.
12. Use **STR_mig_reg()** and **STR_mig_unreg()** to register and unregister dynamically allocated memory space in heap and keep records of them in run-time support subsystem (discussed in Section 4.2.2).

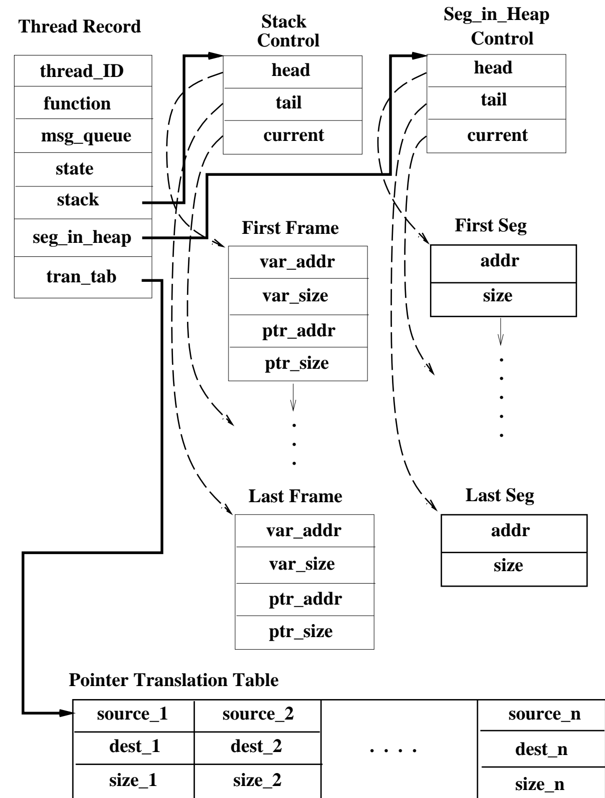

Figure 5. Thread records in Thread Control Area (TCA).

4.2 Run-time support

The run-time support subsystem maintains a thread control area (TCA) which holds a record for each thread containing references to a thread stack, a control block of memory segments in the heap and a pointer translation table, as shown in Figure 5.

4.2.1 Stack

One of the important aspects of a thread state is the stack of activation records containing variables local to each procedure. In our approach, we duplicate activation records in user applications, name them as sr_var and sr_ptr, and register them through **STR_mig_init()**. The stack is a linked list of simplified activation frames in the run-time subsystem, as shown in Figure 5. Each activation frame consists of the addresses and sizes of sr_var and sr_ptr. Pointers in this format of stack can be identified and updated easily. Note that our representation of the stack is different from those used in libraries or kernels. The kernels need to take care of both user functions and system calls. Our user version

Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'02)
1530-2075/02 $17.00 © 2002 **IEEE**
IEEE COMPUTER SOCIETY

Authorized licensed use limited to: SUNY Buffalo. Downloaded on October 23, 2008 at 10:57 from IEEE Xplore. Restrictions apply.

stack is simpler and fits the requirement well. When a user function is called, it contacts the run-time support through **STR_mig_init()** to create and push an activation frame onto the stack. When control returns from a user function, the calling function applies **STR_mig_cleanup** to pop the top frame from the stack.

4.2.2 Memory segments in heap

Threads can have their own local heaps or share the heap in the processes. When threads allocate memory segments for themselves, no matter in which manner the heap exists, this will bring complexity into thread migration. In our approach, a control of memory segments in heap (see Figure 5) maintains a linked list of segment records, traces all dynamically allocated memory in local or shared heap, and provides the information for pointer updating. Each segment record consists of the address and size of the referenced memory block.

In user applications, when **malloc()** and **free()** are invoked to allocate and deallocate memory space, the statements **STR_mig_reg()** and **STR_mig_unreg()** are appended, correspondingly, to create and delete memory segment records in the run-time support subsystem. Essentially, the dynamically allocated memory management is duplicated at application level to support thread migration.

4.2.3 Thread state transfer

To migrate the thread computation correctly, we need to transfer the thread state from the source node to the destination node. In our approach, thread state is transformed at language level and maintained by the run-time support subsystem. Thread state consists of a stack, a linked list of memory segment records, and memory areas referenced by both of them. The run-time support subsystem packs the thread state, contacts the scheduler to identify the destination nodes, and transfers the state by UDP/IP.

Since the aforementioned two data structures and related memory areas are ready when the migration is activated, the communication cost is the major overhead.

4.2.4 State restoration and pointer translation

When a thread state arrives at the destination node, the run-time support subsystem restores it by recreating everything as in Figure 5. A new thread is created to continue the execution of the incoming thread. The thread stack implies the order and depth of the execution of user functions. In fact, the new thread just re-runs those functions in the same order. When functions call **STR_mig_init()** to pass their local variables sr_var and sr_ptr, variable values are reset to the ones from source nodes. The computation restarts from the right place based on the value of $sr_var.stepno$.

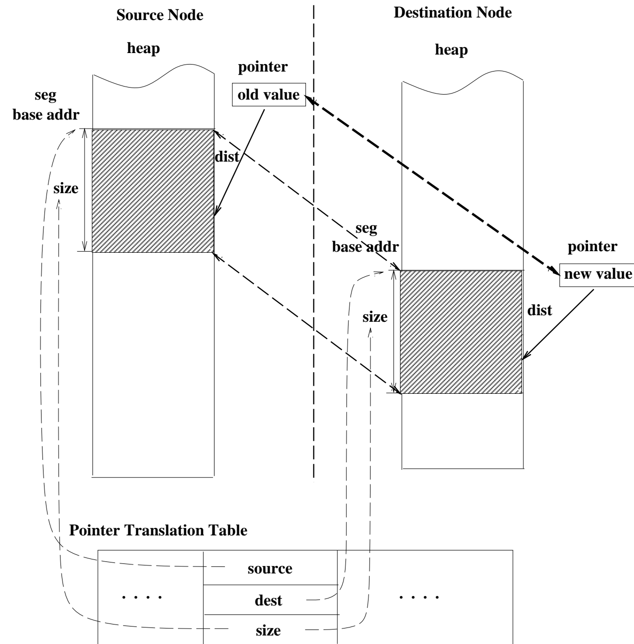

Figure 6. Mapping memory segments.

A new linked list of memory segment records is rebuilt. For each segment record, the value of **size** is the same as the one from source node and is used to re-allocate new memory space on destination nodes. Again, the contents of the newly allocated ones are substituted by the ones from source nodes.

Now we can update the Pointer Translation Table (PTT) in the Thread Control Area. Each element in PTT consists a field, **size**, which holds the size of the memory segment, and the **source** and **dest** fields, which point to the base memory addresses on the source and destination nodes. This depicts a one-to-one mapping between two memory blocks on two machines. Each record in the memory segment control updates one element in PTT. When those related user functions are rerun on the newly created thread, the memory areas for the sr_var and sr_ptr also need to be recorded at two elements in PTT.

As the new thread runs through the stack to restore the thread state, the run-time support subsystem needs to translate the pointers in sr_ptr when **STR_mig_init()** is called. The content of sr_ptr is scanned. The memory segment for pointers is determined by their base addresses and sizes. If a memory address lies in a memory segment on the source nodes, its value should be changed to the one with the same distance from the mapped base addresses on the destination nodes (see Figure 6).

If a pointer is referencing a field in sr_var or sr_ptr, there is one more indirect reference. We should change the value of the pointed field instead of the pointer itself.

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: SUNY Buffalo. Downloaded on October 23, 2008 at 10:57 from IEEE Xplore. Restrictions apply.

This is only used to handle pointers in nested data structures within local variable definition areas sr_var and sr_ptr, which group variables by determining if the top-level type is a pointer or not.

5 Performance Analysis

As a generic scheme, our approach tries to reduce overheads to the minimum. The overhead comes from three areas: compile-time support, run-time support, and migration.

At compile-time, the preprocessor transforms the application code to enable migration feature. This is a one-time cost. The size of the generated executable file is slightly larger than the original one. For each affected function, at least two assignment statements and one switch statement are added (see Figure 4). Each function parameter and migration point will bring in one assignment statement.

The time spent at run-time support plays a bigger role. Each affected function needs one initialization primitive **STR_mig_init()** , and one **STR_checkpoint()** at each possible migration point. The **STR_mig_init()** registers the local variables and allocates memory to maintain records in stack. Similar cost is incurred for dynamic memory management only if **STR_mig_reg()** or **STR_mig_unreg()** is invoked.

Since **STR_checkpoint()** is only invoked at migrated functions to contact the scheduler for possible migration, its cost is determined by the scheduler's decision making speed and communication overhead. The migration costs occurred at **STR_checkpoint()** are classified as follows:

- *Scheduling Cost* t_S : Plays the main role in **STR_checkpoint()** by contacting the scheduler; determining the destination node; creating a new thread; and setting up communication.
- *Transfer Cost* t_T : Transfers a thread state from the source node to the destination node and builds raw state on destination one.
- *Restoration Cost* t_R : Restores the thread state and translates pointers on destination nodes.

Given the cost of each category, we can derive the overall thread migration cost, as follows:

$$C_{thread} = t_{static} + t_{init} + t_S + t_T + t_R \quad (1)$$

where t_{static} is the time to execute statements inserted at compile-time, and t_{init} is the state initialization time at runtime. Since t_{static} and t_{init} may vary with the thread stack length or amount of activation frames, and t_T and t_R are proportional to the thread stack size, the cost could be rewritten as:

$$C_{thread} = \sum_{i=1}^{n} c_i + c_0 f_{len} + t_S + \delta(s_{size}) + \lambda s_{size} \quad (2)$$

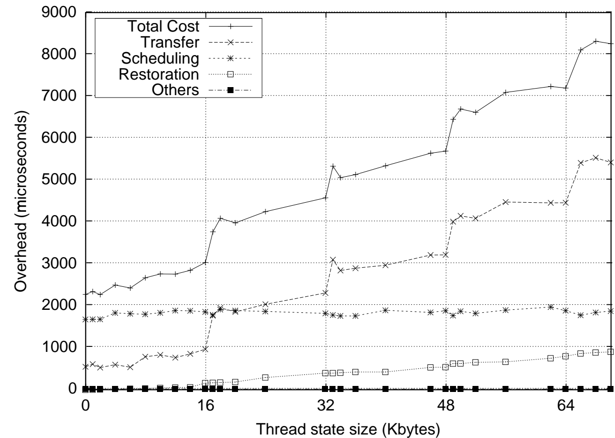

Figure 7. Thread migration costs.

where t_{static} is represented by a summation of statement execution time c_i for all functions, c_0 is the cost of t_{init} in a single function, f_{len} is the amount of activation frames in the stack, s_{size} is the thread state size, λ is a constant, and δ is a function of the stack size. Obviously, besides a constant fundamental cost, stack size is the main factor for thread migration overhead, which must be significantly lower than the average thread lifetime to make migration worthwhile.

6 Microbenchmarks

Our main migration platform is a cluster of SMPs (SUN UltraEnterprise 3000s) connected by fast Ethernet. Each SMP contains four 330Mhz UltraSparc processors.

A thread migration overhead breakdown for a simple example with one thread function and one 40-byte stack is listed in Table 1. It indicates the typical overhead distribution trend in thread migration. More complicated programs just increase the costs proportionally.

Item	Cost (nsec)	Percentage (%)
function activation	276	-
Statements (t_{static})	361	0.015
Initialization (t_{init})	28,869	1.206
Scheduling (t_S)	1,779,428	74.320
Transfer (t_T)	574,794	24.007
Restoration (t_R)	10,825	0.452
Total (C_{thread})	2,394,277	100.000

Table 1. Overhead breakdown in nanoseconds.

The complete thread migration cost is shown in Figure 7 for up to 70 Kbytes thread state size. Transfer cost t_T plays a major role for migrations with big state size. Scheduling cost t_S is a constant and the restoration cost t_R increases linearly. Other costs including t_{static} and t_{init} are generally negligible. This validates our analysis in Section 5.

1530-2075/02 $17.00 © 2002 **IEEE**
IEEE COMPUTER SOCIETY

Authorized licensed use limited to: SUNY Buffalo. Downloaded on October 23, 2008 at 10:57 from IEEE Xplore. Restrictions apply.

7 SPLASH-2 and other real applications

We use several applications from the SPLASH-2 [15] application suite and matrix multiplication to evaluate the thread migration cost. At compile-time, the preprocessor scan and transforms programs. The preprocessor's running time varies from 1.34 to 5.87 seconds. The file size changes resulting from the transformation are listed in Table 2.

Program	Original (lines)	Altered (lines)	Increase (%)
FFT	1059	1408	33
LU-c	1047	1430	37
LU-n	829	1189	43
MatMult	415	562	35
RADIX	1012	1414	40

Table 2. Program sizes changed by preprocessor.

If there are no migrations, our approach does not incur any noticeable overhead despite the compile-time transformations. There are minimal overheads when migrations occur. Table 3 shows the migration costs for the various applications. Most overheads are less than 3% (and 2.5ms) of the computation time even though the application input sizes are very small. The absolute overhead for RADIX is larger than others due to its larger stack size. From these applications, it is clear that our strategy incurs negligible or small overhead on thread migration.

Program	Input Size	Stack size (bytes)	Exec. (ms)	Migration (ms)
FFT	64 Points	160	85	2.5
LU-c	16 x 16	184	77	2.5
LU-n	16 x 16	176	346	2.5
MatMult	16 x 16	100	371	2.5
RADIX	64 keys	32,984	688	5.1

Table 3. Migration Overhead in real applications.

8 Conclusion and future work

The thread migration scheme proposed in this paper is shown to be generic in its scope. It handles pointers accurately, supports dynamic memory management in the heap, and is applicable for heterogeneous environments. To achieve efficient and flexible implementation, we found it practical to transform the physical thread state into a logical one at the language level. Microbenchmarks and real applications indicate that the overhead of our scheme is minimal.

We are currently investigating scheduling strategies for the scheduler that would encompass the choice of data or thread migration and the implementation of our scheme on several different platforms to verify wide scale heterogeneity.

References

[1] 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 .

[2] A. Itzkovitz, A. Schuster, and L. Wolfovich, Thread Migration and its Applications in Distributed Shared Memory Systems, *Journal of Systems and Software*, vol. 42, no. 1, 1998.

[3] E. Jul, H. Levy, N. Hutchinson, and A. Blad, Fine-Grained Mobility in the Emerald System, *ACM Transactions on Computer Systems*, Vol. 6, No. 1, 1998.

[4] J. Chase, F. Amador, E. Lazowska, H. Levy and R. Littlefield, The Amber System: Parallel Programming on a Network of Multiprocessors, *ACM Symposium on Operating System Principles*, Dec. 1989.

[5] J. Casa, R. Konuru, S. Otto, R. Prouty, and J. Walpole, Adaptive Migration systems for PVM, *Supercomputing '94*.

[6] 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), May 1998.

[7] K. Thitikamol and P. Keleher, Thread Migration and Communication Minimization in DSM Systems, *The Proceedings of the IEEE*, March 1999.

[8] D. Cronk, M. Haines, and P. Mehrotra, Thread migration in the presence of pointers, *Proc of the Mini-track on Multithreaded Systems, 30th Hawaii International Conference on System Sciences*, 1997.

[9] S. Roy and V. Chaudhary, Strings: A High-Performance Distributed Shared Memory for Symmetrical Multiprocessor Clusters, *Proc. of IEEE conf. on High Performance Distributed Computing*, 1998.

[10] A. Acharya, G. Edjlali and J. Saltz, The Utility of Exploiting Idle Workstations for Parallel Computation *Procx of the Conference on Measurement and Modeling of Computer Systems*, 1997.

[11] T. Anderson, B. Bershad, E. Lazowska, and H. Levy, Thread Management for Shared-Memory Multiprocessors, *Handbook for Computer Science.*

[12] B. Weissman, B. Gomes, J. Quittek, M. Holtkamp, Efficient Fine-Grain Thread Migration with Active Threads, *Proc. of 12th International Parallel Processing Symposium*, 1998.

[13] G. Antoniu and L. Boug, DSM-PM2: A portable implementation platform for multithreaded DSM consistency protocols, *Proc. 6th Intl Workshop on High-Level Parallel Programming Models and Supportive Environments*, 2001.

[14] S. Milton, Thread Migration in Distributed Memory Multicomputers, *Technical Report TR-CS-98-01*, CS, The Australian National University, February 1998.

[15] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, The SPLASH-2 Programs: Characterization and Methodological Considerations, *Proc. of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[16] A. Scherer, H. Lu, T. Gross and W. Zwaenepoel, Transparent Adaptive Parallelism on NOWs using OpenMP, *Proc. of the Thirteenth Intl Parallel Processing Symposium*, April 1999.

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: SUNY Buffalo. Downloaded on October 23, 2008 at 10:57 from IEEE Xplore. Restrictions apply.