MigThread: Thread Migration in DSM Systems*

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

Distributed Shared Memory (DSM) systems provide a logically shared memory over physically distributed memory to enable parallel computation on Networks of Workstations (NOWs). In this paper, we propose an infrastructure for DSM systems to utilize idle cycles in the network by thread migration.

To maintain high portability and flexibility, a generic thread migration package, MigThread, is implemented at language level. At compile-time, a preprocessor scans C programs to build thread state, detects possible thread adaptation points, and transforms the source code accordingly. At runtime, MigThread moves DSM threads around to utilize idle cycles on remote machines. Since the physical thread state is transformed into a logical form, MigThread is ready to be used in heterogeneous DSM systems. We implemented MigThread in a DSM system Strings. A comparison with other migration schemes and some performance measurements on real applications are reported to show the efficiency.

Key Words: Software Distributed Shared Memory, adaptive parallel computing, thread migration, thread scheduling, compile-time code transformations .

1 Introduction

Recent improvements in commodity processors and networks have provided an opportunity to support high-performance parallel applications within an everyday computing infrastructure. However, applications for such distributed systems are cumbersome to develop due to the need for programmers to handle communication primitives explicitly. Distributed shared memory (DSM) systems are gaining popularity for providing a logically shared memory over physically distributed memory. The programmer is given the illusion of a large global address space encompassing all available memory, eliminating the task of explicitly moving data between processes located on separate machines [5, 2]. DSM systems combine programming advantages of shared memory and the cost advantages of distributed memory.

Studies have indicated that a large fraction of workstations could be unused for a large fraction of time[1]. 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, such as the ones on DSM systems, is less clear. When a workstation is reclaimed by its primary user, the remaining workstations in the same DSM system have to stop. In order for DSM systems to proceed and exploit idle cycles in networks, one should require the computation and DSM systems be reconfigurable, especially in terms of the degree of parallelism, or the number of processors required. Reconfiguration may need data and loop repartitioning, and data and/or computation migration. Since threads are the computation units in multi-threaded DSM systems, we focus on thread migration when computation migration is required. Besides idle cycle utilization, thread migration is also useful for load balancing, fault tolerance, data access locality and system administration [7, 8].

The semantic of thread migration is to stop the thread computation, migrate the thread state to the destination node, and resume the execution at the statement following the migration point there. The thread state consists of process state (inherited from the process containing the current thread), computation state (variables and register contents), and communication state (e.g., open files and message channels). Normally the process state should be shared by threads within the same process. But in DSM systems, it should stay in globally shared areas so that all threads can access it for the same data image. Thus, DSM systems help handle the process state efficiently. Freedman [6] ob-

*This research was supported in part by NSF IGERT grant 9987598, NSF MRI grant 9977815, NSF ITR grant 0081696, US Army Contract DAEA-32-93-D-004, Ford Motor Company Grants 96-136R and 96-628R, and Institute for Scientific Computing.

serves that long-running computations typically use operating system services in the beginning and ending phases of execution, while most of their time is spent in number crunching. We prefer to put more attention on the dominant phase of execution. Therefore, in DSM systems, the thread state refers to the computation state. Communication state requires "migration-aware" communication protocols [15] and is not addressed here.

In this paper, we make the following contributions:

Propose an infrastructure to utilize idle cycles locally and remotely.

Design and implement an efficient and scalable thread migration scheme, *MigThread* that

- Supports fast thread state retrieval.
- Does not trace on pointers and pointer arithmetic to guarantee efficiency.
- Detects adaptation points automatically for DSM systems.

Incorporate adaptivity into a DSM system, *Strings*.

Like other language-level migration schemes [7, 9, 10], *MigThread* enables migration feature for "migration-safe" C programs. For unsafe code, *MigThread* gives warning messages for the possible risks.

The remainder of this paper is organized as follows: Section 2 describes the adaptive DSM systems. Section 3 shows the details of *MigThread* improved from an existing implementation [3]. In Section 4, we analyze the performance of *MigThread*. In Section 5, we show some experimental results on benchmark programs. Section 6 gives an overview of related work. We wrap up with conclusions and continuing work in Section 7.

2 Adaptive DSM systems

Parallel programs running on top of DSM systems know nothing about the physical situation in networks, i.e., they cannot distinguish if they are running on supercomputers or NOWs. DSM systems are in charge of hiding system configurations from programs. Traditionally, DSM systems are static and never change their configuration once the execution starts. To fit the dynamic situation in NOWs, we incorporate adaptivity into DSM systems to reconfigure them dynamically, utilize idle cycles on local or remote machines, and evict when machine owners come back.

2.1 *Strings*

DSM system *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. Therefore, in *Strings*, each thread could be assigned to any processor on the SMP if there is no special request, and all local threads could run in parallel if there are enough processors. *Strings* is designed to exploit data parallelism at the application level and task parallelism at the run-time level [5].

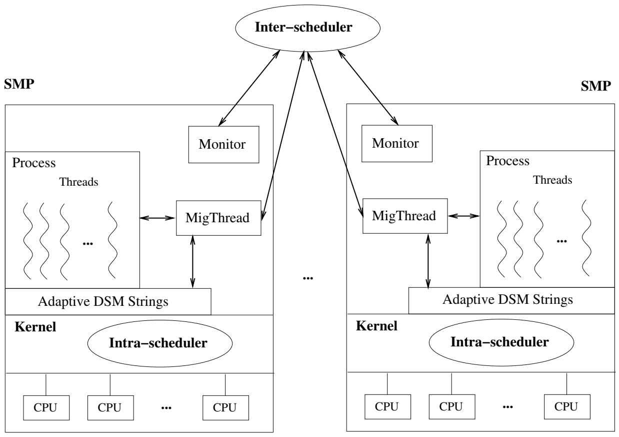


Figure 1. Two-level thread scheduling in *Strings*

2.2 Thread scheduling

To import *MigThread*, *Strings* utilizes a two-level thread scheduling scheme: intra-scheduling and inter-scheduling, as shown in Figure 1. The intra-scheduling comes from the scheduler in the kernel on each SMP. It can assign, context switch, and migrate POSIX threads across multiple processors within the SMP because of kernel-level thread characteristic. When a processor becomes idle, the kernel can assign any available thread to that CPU. Operating systems utilize the idle cycles efficiently on local SMPs, i.e., the kernel supports local thread migration.

Monitor is a daemon process running on each SMP machine in the network. It monitors and reports local machine's workload (including CPU, memory and network card usages) and the "owner-is-returning" event to inter-scheduler periodically. Monitors are separate from DSM systems.

The inter-scheduler runs on a predefined machine which maintains load distribution information, obtains migration policy decision, and orchestrates migration activity across multiple machines. The inter-scheduler can only talk with DSM systems through *MigThread*. Normally the inter-scheduler passes the thread migration request to DSM systems based on load information collected by monitors. When machines leave or join, the inter-scheduler helps DSM systems re-distribute data and threads for reconfiguration. There is a potential for DSM systems to activate thread migration for load balance under more elaborate migration policy.

With this two-level scheduling scheme, each DSM thread can virtually move to any idle processor in the system. Threads can utilize new machines and evict from old ones without programs' notification. Since adaptive DSM systems can use more processors dynamically, eventually the whole system's job throughput is expected to improve.

2.3 DSM migration policy

DSM systems could be more aggressive if they apply complicated migration policy[4]. For example, they can make a choice between page and thread migration. The default option is page migration/replication. But if a thread accesses data on a certain remote machine too frequently and page migration cost is much larger than thread migration, the latter one is a better choice. To identify such situations, DSM systems need to check thread access history, investigate data and thread locality, and contact inter-scheduler for the load information on remote machines. After all, if thread migration seems more efficient, DSM systems will work with *MigThread* and inter-scheduler to start migration. This fine-tuned DSM system works well based on heuristic algorithms in migration policy which is out of the scope of this paper. The point is that DSM systems have sufficient utility to balance their workload well for parallel computing speedup and thread migration is the necessity.

2.4 Adaptation points

The overheads associated with destroying a thread, transferring the thread state, creating a thread and initiating remote execution are not negligible. Hence, there should be sufficient amount of computation between two consequent migration actions to amortize the overhead.

MigThread provides thread migration at certain predefined points. Programs check the condition variables in *MigThread* for possible thread migration requests from the inter-scheduler or DSM systems. Since programs cannot contact the inter-scheduler directly, no communication is involved here. Therefore, the cost of adaptation points is limited if no migration happens. This adaptive strategy enables *MigThread* to insert adaptation points with low cost and improves the sensitivity to the environment.

If the memory model of the DSM system is the traditionally sequential consistency, the system is always in consistent state and threads can be migrated anytime with a guarantee of correctness of resumed execution. It is *MigThread*'s responsibility to identify how far apart each two adaptation points should be. If they are too far away, DSM systems might be too insensitive to the dynamic situation. But if they are too close, it will slow down the computation.

Most DSM systems, for better performance, adopt relaxed memory models such as release consistency model to reduce both the number of messages and the amount of data transferred between processors. Under such speculative models, some virtually shared data could be in inconsistent states when they are between two synchronization points (barriers), i.e., their copies on physically different machines might have different values. If migration takes place between these machines at this time and these data are accessed (especially read) later, the resumed computation could be incorrect. To ensure correctness, thread migration can only be allowed at synchronization points or barriers (adaptation points). Note that *Strings* uses release consistency model.

MigThread scans the source code, detects adaptation points automatically, and inserts suitable thread migration primitives. If two adaptation points are too far away, *MigThread* can insert barriers and migration primitives at certain suitable positions, such as before and after loops. At the same time users can insert such adaptation points as they want which is difficult in some thread migration schemes because of their inefficient thread state construction strategies.

3 Thread Migration system: *MigThread*

MigThread extends our previous implementation [3] and aims at providing thread migration feature with high portability and scalability. As other language-level migration schemes, *MigThread* supports "migration-safe" C programs without unsafe features, such as pointer casting. *MigThread* consists of two parts: preprocessor and runtime support module. At compile time, its preprocessor scan the source code and collect related thread state information into some data structures which will be integrated into the thread state at runtime. With the support of dynamically allocated memory, data in heap are also migrated and restored remotely. Because source and destination nodes might use different address spaces, pointers referencing stack or heap could be invalid after migration. *MigThread* detects and marks pointers at language level so that at runtime it just accesses the predefined data structures to update pointers precisely. Since the physical thread state is transformed into a logical form, *MigThread* has great potential to be used in heterogeneous environments without relying on the types of thread library or operating system. More design and implementation details are in [3].

3.1 *MigThread*'s preprocessor

The preprocessor conducts the source-to-source transformation. This includes marking pointers, defining the thread state, and detecting adaptation points.

3.1.1 The thread state

The thread state consists of a program counter (PC), a set of registers, and a stack of procedure records containing variables local to each procedure. Normally this state is buried in thread libraries or kernels. *MigThread*'s preprocessor abstracts the thread state up to the language level. No information from low level devices, such as registers, need to be retrieved. Therefore, the whole control of thread is moved up to the application level.

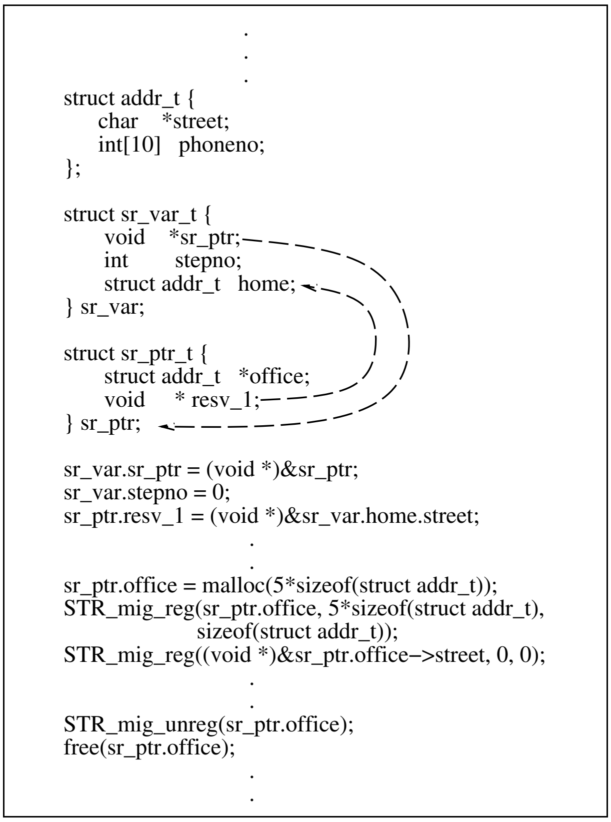
In DSM systems, some variables can be locally or globally shared by threads. Threads belong to different processes which contain locally shared variables for their contained threads. Since all threads should have identical images about shared information, these locally shared variables are disallowed in DSM systems. Therefore, the process state of the thread state is only a group of globally shared variables placed in *Strings*'s global regions.

The thread state contains local variables, parameters, and Program Counters (PC) of functions activated by threads. The preprocessor collects all local variables, and puts them into two data structures, $r\_\ \ r$ and $r\_\ \ r$, depending on if they are pointers or not. For variables with complicated data types, the outer most type will determine which data structure they belong to. Then the preprocessor re-scans the functions, locates all references to the original local variables and replaces them with the references to the corresponding fields in $r\_\ \ r$ and $r\_\ \ r$ which contain most of the thread state.

One of major advances of *MigThread* is its efficient thread state generation. Many application level migration schemes [9, 10] have to add variables one-by-one into the thread state at each adaptation point. This is very time-consuming and places a big obstacle if users want to insert more adaptation points by themselves. *MigThread* only reports the thread state once and the runtime system knows how to construct it. This efficient design benefits *Strings* substantially.

3.1.2 Pointers and pointer arithmetic

Manipulating pointers determines the thread migration approach's efficiency and practicality. Some approaches [14, 16] keep pointers the same by pre-allocating memory spaces on each involved machine which is a hard restriction on systems. Otherwise, pointers will become invalid and be required to be updated accordingly after the migration. *MigThread*'s strategy is to identify pointers at language level and collect them into $r\_\ \ r$. On the destination node, *MigThread* scans the memory area for $r\_\ \ r$ to translate all pointers. If some structure type variables in $r\_\ \ r$ contain pointer fields, they need to be referenced by newly created pointer variables in $r\_\ \ r$ which are initialized right after the $r\_\ \ r$ declaration. For example, in Figure 2, structure type variable $r\_\ \ r.h\ \ m$ contains a pointer field r . Then, an extra pointer field $r\ \ \ \_1$ is declared in $r\_\ \ r$ pointing to the address of $r\_\ \ r.h\ \ m\ .\ \ r$. At the initialization time, the value of $r\_\ \ r.r\ \ \ \_1$ is set, as shown in Figure 2.



```
                    .
                    .
                    .
struct addr_t {
    char    *street;
    int[10]  phoneno;
};

struct sr_var_t {
    void     *sr_ptr;
    int       stepno;
    struct addr_t  home;
} sr_var;

struct sr_ptr_t {
    struct addr_t  *office;
    void     * resv_1;
} sr_ptr;

sr_var.sr_ptr = (void *)&sr_ptr;
sr_var.stepno = 0;
sr_ptr.resv_1 = (void *)&sr_var.home.street;
                    .
                    .
sr_ptr.office = malloc(5*sizeof(struct addr_t));
STR_mig_reg(sr_ptr.office, 5*sizeof(struct addr_t),
                sizeof(struct addr_t));
STR_mig_reg((void *)&sr_ptr.office−>street, 0, 0);
                    .
                    .
STR_mig_unreg(sr_ptr.office);
free(sr_ptr.office);
                    .
                    .
```

Figure 2. Operations related to pointers.

If some pointer variables in $r\_\ \ r$ contain pointer type subfields, such as $r\_\ \ r.$'s field r in Figure 2, the preprocessor will register them by calling **STR_mig_reg()** when they are used by dynamically allocated memory operations and *MigThread* will take care of them at runtime. The typical situation for this is the creation of linked lists.

MigThread does not trace pointers, making it more efficient than approaches[9, 10]. Therefore, pointer arithmetic is not a concern for *MigThread*. No matter how pointers are manipulated, only the current values of variables and pointers hold the correct thread state. This "ignore-strategy" reduces side-effects on performance.

3.1.3 Adaptation points

In Section 2.4, we have discussed the strategy to determine adaptation points in adaptive DSM systems. These positions have to be labeled and matched with certain PC's values. A **switch** statement is inserted to dispatch the execution based on the value of PC. Not only are the adaptation points labeled, but also the positions right before the transformed function call. The reason is that the resumed computation on destination nodes needs to go through the execution path once again. Thus, we need to trace execution at application level. The implementation details are in [3].

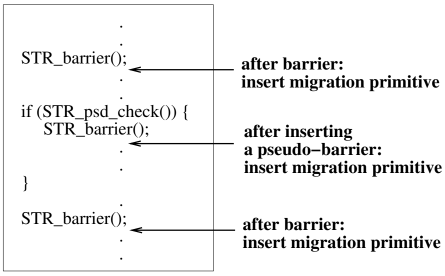


Figure 3. Adding pseudo-barriers for more adaptation points

To work with relaxed memory model of DSM systems, *MigThread* is extended to detect more adaptation points. If two barriers are too far away, the preprocessor inserts pseudo-barriers (see Figure 3) where a primitive **STR_psd_check()** is called to contact the runtime support module. When thread migration is requested, the actual barrier command and migration primitive are executed to migrate threads with consistent data state. This strategy enables more adaptation points in *Strings* without sacrificing the performance much.

3.2 *MigThread* at runtime

At runtime, *MigThread* needs to maintain, transfer and restore thread states. After updating pointers, it can resume the computation on destination nodes.

3.2.1 Thread Control Area

MigThread maintains a thread control area (TCA) which holds a record for each thread which contains references to a stack, a control block for memory segments in the heap, and a pointer translation table.

Once a function calls **STR_mig_init()** to register the addresses and sizes of *r*_ *r* and *r*_ *r*, *MigThread* generates an activation frame and pushes it onto the stack. After each function call, primitive **STR_mig_cleanup()** is used to pop up the top frame from the stack. This application-level stack enables the portability.

In user applications, when **malloc()** and **free()** are invoked to allocate and deallocate memory spaces, the preprocessor inserts **STR_mig_reg()** and **STR_mig_unreg()** accordingly to let *MigThread* create and delete memory segment records at runtime. *MigThread* maintains a linked list of memory 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, size and single data structure's size (unit size) of the referenced memory block, with an extra linked list of offsets for inner pointer subfields, *r*_ , which is used to trace and update the pointer fields in the memory segments. Each element in *r*_ contains an offset from the base address of the current memory segment in the first unit block, as shown in Figure 4 (the shadowed areas are pointer fields). At runtime, *MigThread* uses pointer arithmetic to detect other pointer fields in the whole memory block. Primitive **STR_mig_reg()** is called with 0 to report offsets of pointer subfields (see Figure 2). Again, the dynamically allocated memory management is moved up to application level.

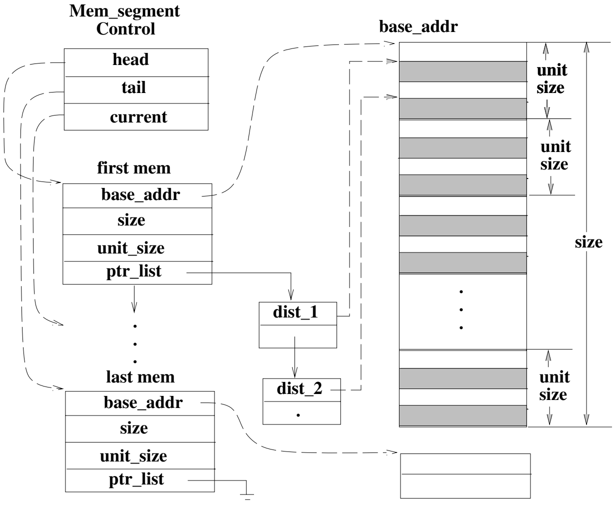


Figure 4. Handling pointers in dynamically allocated memory

3.2.2 Thread state transfer and restoration

The application-level thread state consists of a stack, a linked list of memory segment records, and their associated memory blocks. *MigThread* packs the thread state, transfers it by UDP/IP, and restores it by recreating everything on new nodes. To update pointers, *MigThread* simply scans *r*_ *r* for local pointer variables and the linked list of memory records for dynamic pointer fields. The stack implies the order and depth of the execution of functions. A new thread just re-runs the functions in the same order to resume computation with the same state. The details are in [3].

4 Performance Analysis

We compare the complexity of *MigThread* with similar application level schemes Porch [9] and SNOW [10]. The major difference is the state construction. Porch and SNOW are similar in collecting and restoring data. They need to deal with variables and pointers one-by-one explicitly at each adaptation point. This slows down the whole system dramatically and makes it hard for users to insert adaptation points by themselves if they prefer. On the other hand, *MigThread* only registers variables and pointers once and is much faster to construct the thread state. The complexity comparisons are listed in Table 1 where $N\ {}_{r}$ and $N\ {}_{r}$ represent number of variables and pointers.

Table 1. Complexity comparison in state construction

System	Collect/restore Variables	Collect/restore Pointers
Porch	$(N\ {}_{r})$	$(N\ {}_{r})$
SNOW	$(N\ {}_{r})$	$(N\ {}_{r})$
MigThread	1	1

Pseudo-barriers added for more adaptation points in DSM systems introduce extra overhead. Whereas regular barriers synchronize both computation progress and data copies, pseudo-barriers are light-weight and intended only for synchronizing threads' progress. The data synchronization happens only when migration is required. Pseudo-barriers start affecting computation after the slowest thread arrives. The actual time depends on how the centralized inter-scheduler activates threads. For n threads, the cost of pseudo-barrier $\quad {}_{s} \quad$ is

$$TT \qquad {}_{s} \qquad TT + \frac{(n-1) * \quad TT}{2} \tag{1}$$

where $\ TT$ is the Round-Trip-Time for a thread to contact the inter-scheduler. We borrow *Strings*'s synchronization mechanism to combine local threads' messages into single one for cost reduction. Pseudo-barriers slow down the relaxed consistency model DSM systems for migration options. But the overall system is still faster than DSM systems with sequential consistency model which keep consistent data all the time.

The overhead of *MigThread* comes from two areas: preprocessor and runtime support module. The preprocessor transforms the application code to enable migration feature. For each function, at least two assignment statements and one switch statement are added and each function parameter will bring in one assignment statement. The migration cost of *MigThread* at runtime could be expressed as follows:

$${}_{r} \quad = {}_{s} \quad {}_{c} + \qquad + \qquad + {}_{T} + \tag{2}$$

where ${}_{s} \quad {}_{c}$ is the time to execute assignment statements inserted by preprocessor, $\quad$ is the state initialization time at runtime, $\quad$ is the time to contact inter-schedulers, ${}_{T}$ is the thread state transfer cost, and $\quad$ is the state restoration time. Since ${}_{s} \quad {}_{c}$ and $\quad$ may vary with the thread stack length or amount of activation frames, and ${}_{T}$ and $\quad$ are proportional to the thread stack size, the cost could be rewritten as:

$${}_{r} \quad = \quad \underset{1}{} \quad + {}_{0} \quad + \quad + \ ({}_{s} \quad) + \lambda \ {}_{s} \tag{3}$$

where ${}_{s} \quad {}_{c}$ is represented by a summation of statement execution time $\quad$ for all functions, ${}_{0}$ is the cost of $\quad$ in a single function, $\quad$ is the amount of activation frames in the stack, ${}_{s} \quad$ is the thread state size, λ is a constant, and $\ $ is a function of the stack size. Obviously, besides a constant fundamental cost, stack size is the factor for thread migration overhead [3].

In page-based DSM systems, the page migration cost $\quad$ is a function of page size ${}_{s} \quad$:

$$= \ '({}_{s} \quad) \tag{4}$$

With *MigThread*, DSM systems can compare costs and calculate a threshold as follows:

$${}_{r} \quad = \frac{{}_{1} \quad + {}_{0} \quad + \quad + \ ({}_{s} \quad) + \lambda \ {}_{s}}{'({}_{s} \quad)} \tag{5}$$

Based on $\quad {}_{r} \quad$, DSM systems should be able to determine which migration scheme is more efficient and fine-tune the runtime performance well.

The migration platform is a cluster of SMPs (SUN UltraEnterprise 3000s) connected by fast Ethernet. Each SMP contains four 330Mhz UltraSparc processors. The thread migration costs vary based on thread state size [3]. For most software DSM systems, page migration is the main mechanism to maintain locality. In *Strings*, the page size is 8 Kbytes and the page migration cost is 2.0 ms. Since data are moved by pages, the page migration cost is discrete and could be expressed by a function of page quantity. The cost relationship between page and thread migration is shown in Figure 5. For bigger thread states and smaller shared data, page migration is preferred. Otherwise thread migration should be considered. At runtime, DSM systems calculate and compare thread migration cost by the state size and page migration cost by page quantity to determine efficiency. Since these costs are predictable, DSM systems are able to fine-tune their performance dynamically.

5 *MigThread* on real applications

To evaluate the thread migration cost, we use several applications from the SPLASH-2 application suite, matrix

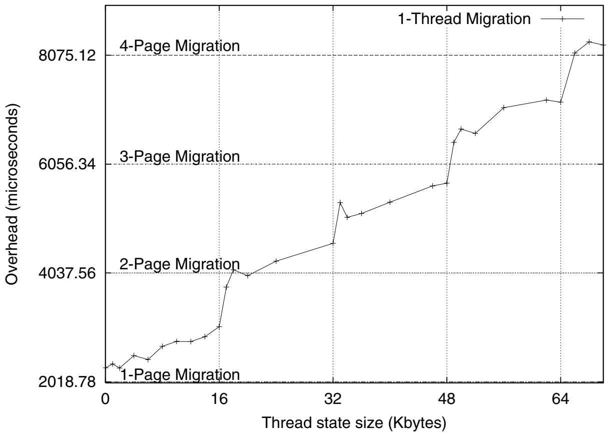


Figure 5. Cost relationship between page and thread migration

multiplication, and Molecular Dynamics (MD) simulation which is used to study friction forces of sliding hydroxylated α-aluminum oxide surfaces [17]. The parallelized programs are run on two SMP machines with one thread on each. The communication layer is UDP/IP. Since the inserted primitives do not cause any noticeable slowdown, we only focus on the migration cost and compare it with pure execution time on two SMP nodes.

MigThread's preprocessor scans and transforms these C programs. It is the preprocessor's responsibility, not program's, to conduct this transformation. Normally programmers do not need to be involved unless they want to add more precise adaptation points. This one-time transform procedure takes only 1-8 seconds for these programs.

The runtime overheads are shown in Table 2 which fits previous analysis and microbenchmarks well. For most applications, their thread states range from 100 to 184 bytes, and their migration time is around 2.4 ms. Even though the thread state of OCEAN-c is increased to 432 bytes, it is still not big enough to change migration time since the dominant factor transfer cost $_T$ and overall cost display stair-like behavior [3]. Only the thread states of RADIX and MD are big enough to make difference. For most programs, we only choose small problems (by input sizes) to indicate that even in these small cases migration overhead is small. Cases with bigger problem sizes make thread migration look more efficient since in *Strings* thread state sizes are invariant to problem sizes (see Table 2). Shared data are in global shared regions which do not migrate with threads. This makes *MigThread* very efficient in *Strings*. Compared to programs' execution time, migration cost is very small (most of them are less than 1% and at most 3%) for benchmark programs and real programs like MD.

For most cases, page migration happens frequently for data accesses. Their overall cost is much higher than single thread migration's. This suggests that thread migration may be an effective tool for DSM systems to balance workload and improve data locality. The only exception one is the DM application whose huge thread state makes the thread migration cost about half the page migration cost. For such applications, thread migration is only suitable for idle cycle utilization situation.

The chosen programs are popular, but all array-based. Fortunately *MigThread* does not slow down particularly for pointer-intensive applications because pointers are not traced at runtime. Definitely more memory blocks incur bigger overhead, which is inevitable.

6 Related Work

The major concern in thread migration is that the address space could be totally different on different machines and internal self-referential pointers may no longer be valid. There are three approaches to handle the pointer issue. The first approach is to use language and compiler support to maintain enough type information, identify and update pointers[4, 11], such as Emerald[12], Arachne[11], and Tui system [7]. But they reply on new languages and compilers. The second approach requires scanning the stacks at runtime to detect and translate the possible pointers dynamically, as in Ariadne[13]. Since some pointers in stack are probably misidentified, the resumed execution can be incorrect. The third approach is most popular and necessitates the partitioning of the address space and reservation of unique virtual addresses for the stack of each thread so that the update of internal pointers becomes unnecessary. This solution requires large address space and is not scalable[16, 4]. Another drawback of this method is that thread migration is restricted to homogeneous systems. Amber[14] and Millipede[16] use this "iso-address" approach.

Application-level migration schemes could be heterogeneous without relying on new languages and compilers. The SNOW [10] is one of them. It handles communication state with "connection-aware" protocols. The Porch system [9] uses the same way as in SNOW to construct process/thread state dynamically. They need to register pointers one-by-one which can cause flexibility and efficiency problems. The thread migration approach in [3] is similar to *MigThread* but has limitations. *MigThread* improves the handling of pointer arithmetic, pointers in heap, and memory management. A significant improvement in *MigThread* is that it cooperates with the inter-scheduler to exchange information with DSM systems, and eventually incorporates adaptivity into DSM systems.

7 Conclusion and future work

MigThread is shown to be generic in its scope. It handles

Table 2. Migration Overhead in real applications

Program	**Input Size**	**State size**	**Transform**	**Execution**	**# of**	**Page**	**Single**	**Migr.**
		(bytes)	(sec)	(ms)	**Page**	**Migr.**	**Thread**	**/ Exec.**
					Faults	(ms)	**Migr.** (ms)	**Rate**(%)
FFT	64 Points	160	5.87	85	12	22.68	2.42	2.85
	1024 Points	160	5.87	112	21	42.63	2.46	2.20
LU-c	16 x 16	184	4.19	77	5	9.65	2.35	3.05
	512 x 512	184	4.19	7,699	368	885.58	2.41	0.03
LU-n	16 x 16	176	4.17	346	2	4.20	2.34	0.68
	128 x 128	176	4.17	596	34	61.21	2.37	0.40
MatMult	16 x 16	100	1.34	371	13	24.23	2.32	0.63
	128 x 128	100	1.34	703	56	113.25	2.47	0.35
OCEAN-c	18 x 18	432	7.98	2,884	89	166.94	2.45	0.08
	258 x 258	432	7.98	14496	1082	2041.31	2.40	0.02
RADIX	64 keys	32,984	2.86	688	25	46.46	5.12	0.74
	1024 keys	32,984	2.86	694	25	45.84	5.14	0.74
MD	5,286 Atoms	7,040,532	2.45	52,243	74	148.60	83.65	0.16

pointers accurately, constructs thread state promptly, brings adaptivity into DSM systems, and helps utilize idle cycles in networks of SMPs. *MigThread* erases many restrictions placed on most thread migration approaches and only requires that migration happen at certain predefined points which fits well with DSM systems. The language level migration strategy makes it ready for heterogeneous DSM systems. Comparison with similar approaches and experiments on real applications indicate that the overhead of *MigThread* is minimal.

We are currently investigating more complex scheduling policies to make a choice of data or thread migration for better data locality and communication minimization, expanding *MigThread* to handle more "migration-unsafe" cases, and implementing it on multiple platforms for heterogeneity.

References

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

[2] E. Speight and J. K. Bennett, Brazos: A Third Generation DSM System, *Proc. of the USENIX Windos NT Workshop*, 1997.

[3] Hai Jiang and Vipin Chaudhary, Compile/Run-time Support for Thread Migration, *Proc. of 16th International Parallel and Distributed Processing Symposium*, Fort Lauderdale, Florida, April 15-19, 2002.

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

[5] S. Roy and V. Chaudhary, Design Issues for a High-Performance Distributed Shared Memory on Symmetrical Multiprocessor Clusters,*Cluster Computing: The Journal of Networks, Software Tools and Applications*, No. 2, 1999.

[6] D. Freedman, Experience Building a Process Migration Subsystem for UNIX, *Proceedings of the Winter USENIX Conference*, page 349-355, Jan. 1991.

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

[8] D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler and S. Zhou, Process Migration, *ACM Computing Surveys*, 2000.

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

[10] K. Chanchio and X.H. Sun, Data Collection and Restoration for Heterogeneous Process Migration, *Proceedings of 21st International Conference on Distributed Computing Systems* (ICDCS), April 2001.

[11] 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.

[12] 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, Feb. 1998.

[13] E. Mascarenhas and Vernon Rego, Ariadne: Architecture of a Portable Threads system supporting Mobile Processes, *Technical Report CSD-TR 95-017*, Dept. of Computer Sciences, Purdue University, 1995 .

[14] 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.

[15] K. Chanchio, and X.H. Sun, Communication State Transfer for the Mobility of Concurrent Heterogeneous Computing, *International Conference on Parallel Processing (ICPP)*, September, 2001.

[16] 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.

[17] V. Chaudhary, W. Hase, H. Jiang, L. Sun, and D. Thaker, Comparing Various Parallelizing Approaches for Tribology, *4th International Workshop on High Performance Scientific and Engineering computing with Applications*, Aug. 2002.