Purdue University
Purdue e-Pubs

Computer Science Technical Reports | Department of Computer Science

1994

Thread Migration on Heterogeneous Systems via Compile-Time Transformations

Janche Sang

Geoffrey W. Peters

Vernon J. Rego
Purdue University, rego@cs.purdue.edu

Report Number:
94-022

Sang, Janche; Peters, Geoffrey W.; and Rego, Vernon J., "Thread Migration on Heterogeneous Systems via Compile-Time Transformations" (1994). *Computer Science Technical Reports.* Paper 1125.
http://docs.lib.purdue.edu/cstech/1125

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

Thread Migration on Heterogeneous Systems via Compile-Time

Janche Sang, Geoffrey W. Peters, and Vernon Rego
Computer Sciences Department
Purdue University
West Lafayette, IN 47907

CSD-TR-94-022
March, 1994

Thread Migration on Heterogeneous Systems via Compile-Time Transformations *

Janche Sang
Geoffrey W. Peters
Vernon Rego
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
Email: {sang,gwp,rego}@cs.purdue.edu
Contact: Prof. Rego
Phone: (317) 494-7835
Fax: (317) 494-0739

Abstract

This paper describes an alternative technique to provide multithreading in an enhanced C language. In contrast to the traditional design of a thread library, which usually utilizes a few lines of assembly code to switch control between threads, the technique we use is based on compile-time program transformations and a run-time library. Since this approach transforms a thread's physical states into logical forms, thread migration in a heterogeneous distributed environment becomes practically feasible. Performance measurements of the current implementation are reported.

Keywords: Thread migration, Heterogeneous computing, Process abstraction, Lightweight process, Preprocessor, Parallel programming

* Research supported in part by PRF, NATO-CRG900108, NSF CCR-9102331, ONR-9310233, and ARO-93G0045.

1 Introduction

Lightweight processes or *threads* have emerged as a representation of computational entities, co-operating with each other within a single address space. In fundamental structure, a lightweight process is no different from a process; each has its own stack, local variables, and program counter, describing the state of its execution. However, as compared to a process, a lightweight process is lighter in terms of the overhead associated with creation, context-switching, inter-process communication, and other routine functions. This is because these primitives can be executed within the same address space. The work required to maintain page tables, register values, and so on, can be greatly reduced. In general, the purpose of a threads system is to provide a cheap concurrent programming environment within a process.

Spreading execution of threads over several processors can exploit parallelism and thus achieve improved performance. However, in distributed-memory parallel systems, two factors may degrade the performance gains of multi-threading. The first factor is load imbalance. During program execution, there may be a dense cluster of threads resident on a single processor while only few threads exist on other processors. Thus, lightly loaded processors have to wait for heavily loaded processors to complete their work. A systematic scattering of threads across processors which allows heavily loaded processors to efficiently balance their load with lightly loaded processors gives the executing system an opportunity to achieve a better overall throughput.

The second factor is non-local data access. In a program execution, threads will typically access remote data and thus require unavoidable inter-processor communication. If cross-processor data access tends to be frequent, then relocating an accessing thread to the site hosting the remote information can reduce inter-processor communication traffic. Therefore, these two factors bring the need to provide threads with a dynamic migration capability.

The semantics of the thread migration function is that a migrant thread will resume its execution at the statement following its point of migration. To achieve this, a thread's state must be transported from the source processor to the destination processor. In a homogeneous environment, the task of translating the migration information can be reduced because both the source and destination machines consist of the same hardware and software configuration.

For example, it is not necessary to translate a thread's execution resumption address because the address represents the same resumption point at both source and destination processors. However, this is normally not true in a heterogeneous environment.

In this paper, we describe design and implementation issues of a heterogeneous thread library with a dynamic migration capability. To migrate a thread between heterogeneous machines, we need sufficient knowledge of the underlying systems to generate machine-independent information about a thread's running state. This information can be transfered to the destination machine to re-generate an equivalent state for the migrant thread. Our approach is based mainly on compile-time program transformations which map a thread's physical states into logical forms.

Our goal in this work is to provide concurrent programming primitives for high performance heterogeneous computing, such as numerical computation, simulations, etc.; therefore, efficiency is our major concern. To show this, we have compared the performance of the new *Ythreads* package with our homogeneous thread library *Xthreads*[8, 10]. The results show that the latency of operations, such as creation, context switching, and migration, etc. in Ythreads are close to the latency in Xthreads.

2 Related work

Multi-threaded systems can be categorized into two levels: kernel-level and user-level. Many contemporary operating systems, such as Mach[13] and SunOS 5.0 [6], support kernel-level threads. The kernel can directly schedule application's threads onto the available physical processors. In a user-level multi-threaded system, all thread operations, such as creation, synchronization, context switching, etc. require no intervention from the operating system kernel. Invocations of threads can be as cheap as simple function calls rather than traps. Therefore, a user-level thread library can be more efficient than a kernel supported multi-threaded system. The major disadvantage of user-level thread packages is that a blocking system call performed by a thread may prevent execution of other runnable threads. Examples of user-level thread packages can be found in [3, 5].

Recent multi-threaded systems supporting dynamic thread migration for distributed memory systems include IVY[4] and Amber[2]. The IVY system was designed at the kernel level

to support shared virtual memory. The stack of a thread is allocated from shared virtual memory. When a thread migrates, the current page of the thread's stack must also move to the destination processor to avoid a page fault. The Amber system integrates several shared-memory multiprocessors connected by an Ethernet local area network. The threads in Amber execute in a shared object space. Thread migration is used for remote invocation of objects. As far as we know, thread migration on these two systems appears to be limited to between homogeneous processors.

Two approaches to heavy-weight process migration in a heterogeneous environment can be found in [11] and [14], respectively. In the former approach, the migrant process has to issue a request to initiate its migration. During migration, it is necessary o search the stack and process control block to translate the return and resumption addresses. In the latter approach, the authors build a machine-independent *migration program* which consists of the current code and state of the migrant process when migration occurs. The migration program is sent, recompiled, and then re-started at the destination site. Both approaches consider migration an infrequent event. Therefore, efficiency is not their major concern and no performance results are provided.

3 The Multithreaded Programming Environment

The proposed multithreaded programming model supports logical *concurrency* within each processor and physical *parallelism* across processors in distributed-memory systems. Logical concurrency is provided by using lightweight processes. Multiple threads of control sharing a single address space are created within a heavyweight process. This heavyweight process in turn is executed on all machines in the system. Physical parallelism is realized through distributing and executing these processes across processors.

Figure 1 depicts the structure of the programming model. A process is a logical unit which hosts threads. It initializes the threads environment at the beginning of its execution. Threads can be created and migrated dynamically across process and/or processor boundaries.

Thread creation is semantically equivalent to an asynchronous function call. That is, the caller and callee can execute in parallel. A thread is a computational entity which defines its control flow through a set of statements. When control reaches the end of a thread body,

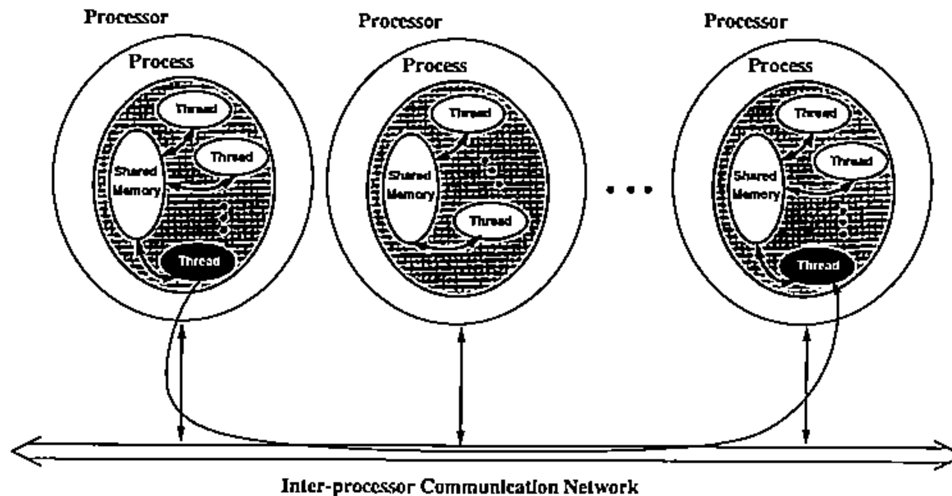

Figure 1: The Multi-threaded Programming Model

execution of the thread terminates naturally. Hence, control will return to the thread scheduler which selects a next thread to run.

The current environment supports a priority-based scheduling policy for threads within a process. The scheduling principle is that the highest priority thread which is not blocked has the right to run. If two threads have the same highest priority, the First-Come First-Served rule is applied. The currently running thread can execute until it is terminated or suspended.

3.1 Xthreads vs. Ythreads

The multi-threaded model was originally realized by the Xthreads library[8, 10] running in a homogeneous distributed-memory multiprocessor environment. The Xthreads library enhances the C programming language with concurrent capability in the form of library functions and sets of predefined data structures. It provides a simple and efficient context-switch primitive through a few lines of assembly code. The thread migration mechanism is built on top of the context-switch function to make a snapshot of the thread state. Since Xthreads is designed and implemented in a homogeneous environment, the thread's state, such as the stack, resumption address, etc., can be copied from the source to the destination without translation.

The design objective of Ythreads is to provide a cheap concurrent programming environment, particularly in thread migration over heterogeneous networks. In contrast to homoge-

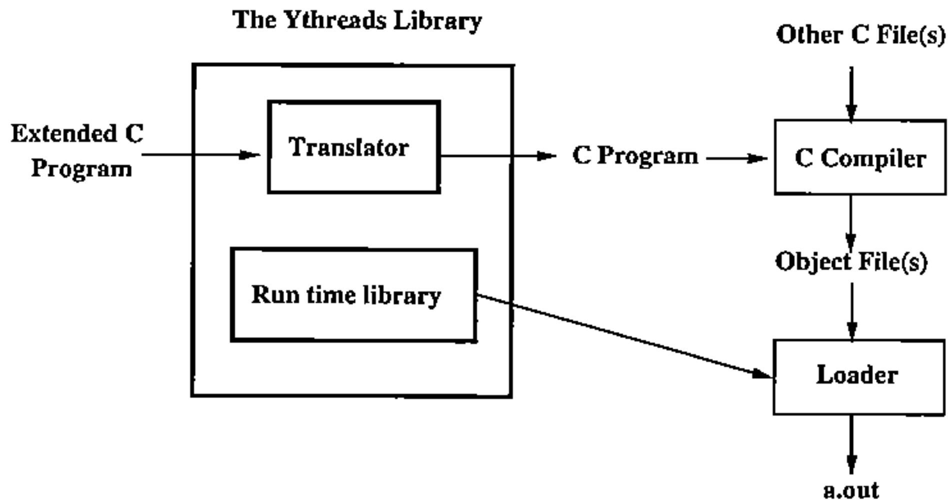

Figure 2: The Ythreads Library

neous thread migration, migration of a heterogeneous thread relies on a machine-independent representation of the thread's state. The typical problem is the representation of the resumption address. In our approach, the resumption address is represented by a pair of integers `<fun_num, resume_pt>`, where the variable `fun_num` is an index of some function in our translator-generated function tables, and the variable `resume_pt` is a logical re-entry point inside the function. Therefore, the translation of physical resumption address from one machine to another can be simplified to integer transformation. Note that the resumption points can be determined at compile time because in our model the resumption will only occur during one of the thread system function invocations (e.g. yield, wait, etc.).

In order to generate the logical state of a thread, we extend the C language with the thread construct. The declaration of a thread has the same syntax as the function declaration, except that the keyword `thread` is used in place of the function type. The translator translates the extended C program into a normal C program (see Figure 2). This translated program is then compiled and linked with the Ythreads run-time support library which provides routines dealing with the thread management issues such as scheduling, synchronization, migration, and so on. The primitive facilities of Ythreads are shown in Table 1. In fact, the run-time routines re-use most of the code in Xthreads.

```
/* creation and destruction */
YTCALL ythread_create(*ypid,attrp,func,narg,arg1,arg2,...);
YTCALL ythread_destroy(ypid);

/* destroy itself */
YTCALL ythread_exit();

/* yield control to a thread pointed to by ypid */
YTCALL ythread_yield(ypid);

/* find out who I am */
YTCALL ythread_self();

/* find out if a thread is alive or not */
YTCALL ythread_ping(ypid);

/* event */
YTCALL ythread_event();
YTCALL ythread_wait(e);
YTCALL ythread_set(e);

/* message passing */
YTCALL ythread_send(ypid,msg);
YTCALL ythread_receive();
```

Table 1: The Primitive Functions in Ythreads

4 Implementation Issues

The implementation involves a translator and run-time system routines. The translator converts the `thread` construct to the C constructs, and converts some Ythreads facilities into calls to run-time system routines and in-line code. A similar conversion technique was briefly described in [7]. However, as far as we know, our work is a first attempt to employ this technique to implement heterogeneous migration.

4.1 The Translator

We implemented the translator using *lex* and *yacc*. The main task of the translator is to convert the thread construct into a function and an associated structure declaration in C. The structure is used to store the information which must be preserved for later resumption of the thread. There are two kinds of information in the structure. One is generated from the translator and the other is directly copied from the source code. For each thread construct, the translator automatically inserts two variables `fun_num` and `resume_pt` into the corresponding structure. The variable `fun_num` will hold the logical identifier of the function when the thread

is created, while the variable resume_pt will store the next logical resumption point of the thread. Note that these two fields are represented by integers, not in the form of machine-dependent addresses. Therefore, the work of address translation in thread migration can be reduced if two machines use the standard integer format.

The structure also contains the user-defined data for the thread. The translator copies the declaration of the formal parameters and the local variables of the thread from the source program into the structure. Therefore, the structure replaces the role of the stack which is used to store the function activation record. An instance of the structure type represents an instance of a thread during execution. To access the translated parameters and local variables in the structure, a pointer _this is inserted into the converted function heading. In fact, the pointer is analogous to the traditional stack pointer which indicates the current activation frame. Whenever a thread resumes, the pointer which indicates the appropriate structure location will be passed to the function. For efficiency, it is better to place the pointer _this in a machine register because it will be heavily used. An example written in the extended C language and its translated code can be seen in Figure 3 and Figure 4, respectively.

The behavior of a thread is defined through a few statements inside the thread body. The preprocessor translates these statements into the body of the function with the following two major conversions. First, if there is a local variable or formal parameter, e.g. i, it will be converted to be _this->i since the location of the variable i is in the structure indicated by the pointer _this.

Second, if it is a Ythreads system primitive which may lead to suspension of the current running thread, the translator will insert at least three statements. That is, a statement which assigns the next re-entry point (an integer value) to the variable resume_pt, followed by a return statement to give control back to the thread scheduler, and followed by a new label for locating the physical re-entry address. That is, multiple exit and re-entry points are preserved logically in the variable resume_pt using the integer format.

When control reaches the end of a thread body, the execution of the thread terminates naturally. Hence, control should return to the thread scheduler which selects the next thread to run. To achieve this, the translator inserts an invocation of the Ythreads routine ythread_destroy() into the converted function to eliminate the currently running thread, effectively a suicide op-

```
thread foo()
{
   int i;

   i = 1000;
   while(--i > 0) {
       printf("This is foo %d\n",i);
       ythread_yield(bar_yid);
   }
}

thread bar(k)
{
   int j;

   j = k;
   while(--j > 0) {
       printf("This is bar %d\n",j);
       ythread_yield(foo_yid);
   }
}

thread ymain()
{
   ythread_create(&foo_yid,NULL,foo,0);
   ythread_create(&bar_yid,NULL,bar,1,1000);
}
```

Figure 3: An example in extended C

```
struct foo_act_rec {
   int fun_num;
   int resume_pt;
   int i;
};

foo(_this)
register struct foo_act_rec * _this;
{
   goto _findstate;
_s00: ;
   _this->i = 1000;
   while(--_this->i > 0) {
      ythread_yield(bar_yid);
      _this->resume_pt = 1;
      return;
_s01: ;
   }
   ythread_destroy(ythread_self());
   return;

_findstate:
   switch(_this->resume_pt) {
      case  0: goto _s00;
      case  1: goto _s01;
   }/* end switch */
}


struct bar_act_rec {
   int fun_num;
   int resume_pt;
   int k;
   int j;
}

bar(_this)
register struct bar_act_rec * _this;
{
   goto _findstate;
_s00: ;
   _this->j = _this->k;
   ...
}
```

```
struct ymain_act_rec {
     int fun_num;
     int resume_pt;
};

ymain(_this)
register struct ymain_act_rec * _this;
{
   goto _find_resume_pt;
_s00: ;
   ythread_create(&foo_yid,NULL,foo,0);
   _this->resume_pt = 1;
   return;
_s01: ;
   ythread_create(&bar_yid,NULL,bar,1,1000);
   _this->resume_pt = 2;
   return;
_s02: ;
   ythread_destroy(ythread_self());
   return;

_findstate:
   switch(_this->resume_pt) {
      case  0: goto _s00;
      case  1: goto _s01;
      case  2: goto _s02;
   }/* end switch */
}


int (* _pfunc[ 3])() = {
   foo,
   bar,
   ymain
}


int  _act_size[ 3] = {
   sizeof(struct foo_act_rec),
   sizeof(struct bar_act_rec),
   sizeof(struct ymain_act_rec)
}
```

Figure 4: The translated code

eration.

After inserting the thread self-termination statement, the translator generates a switch statement to realize the multiple exit and re-entry points of the thread. Each case in the switch statement represents a re-entry point in the translated function through a goto statement and a previously generated label. When the function is invoked, control immediately transfers to the switch statement. Then, based on the value stored in the variable `resume_pt`, execution will either start from the beginning or resume following the statement at which the thread previously suspended.

When all of the thread constructs are converted to functions and structures, the translator builds two tables `_pfunc[]` and `_act_size[]` containing the converted function addresses and the sizes of the structures, respectively. These tables provide the necessary information for run-time support routines, especially the thread creation and migration functions. We will discuss the usage of these tables later.

4.2 The Ythreads Run-Time System

The Ythreads library provides the `main()` function. It initializes system data structures such as the thread table, priority list, etc. and then creates the first thread which will execute the user-supplied main function `ymain()`. The `main()` function then plays the role of the scheduler in the system. The scheduler selects the thread with the highest priority from the ready list and calls the converted function for the thread by passing the location of the structure to the pointer `_this`. The converted function carries out the actions of the thread. A sketch of pseudo-code outlining the main scheduler is shown in Figure 5.

The Ythreads system forces any suspension of the running thread to return control back to the scheduler. Each Ythreads primitive function stores a value representing the suspension reason in the variable `_prev_yat_stm`. Based on the value in the variable, the scheduler will then take appropriate action, such as invoking the yielded thread's function, sending the migration information, etc.

A thread can be created by calling the `ythread_create()` function (see Figure 6). It first allocates an instance of the thread's corresponding structure declaration and initializes the members in the structure. The size of the structure can be obtained from the structure

```
schedule()
{
    while(TRUE) {
        switch(_prev_yat_stm) {
        case CREATE:
            • if the newly created thread has a higher priority,
                invoke the new thread's converted function.
            else
                invoke the current thread's converted function.
            break;
        case YIELD:
            • check the yielding is legal.
            • insert the current running thread into the ready queue.
            • delete the designated thread _yyid from the ready queue.
            • invoke the designated thread's converted function.
            break;
        case MIGRATE:
            • send the migrant thread's state to the remote site.
            • kill the migrant thread.
            • if it is self-migration, select the next thread in the ready
              queue to run.
            break;
        • • •
        }
        check_mig_msg();
    }
}
main()
{
    •initialize thread table, ready queue, etc.
    •create the first user thread running ymain().
    schedule();
}
```

Figure 5: Pseudo code of the main scheduler

```
YTCALL ythread_create(yid,aptr,procaddr,nargs)
{
    • allocate a structure and initialize the members.
    • find a free xentry in the xtab table.
    if(aptr == NULL)
        • initialize the new thread using the default priorities, etc.
    else
        • initialize the new thread using specified values through aptr pointer.
    • insert the new thread to the ready list.
    _prev_yat_stm = CREATE;
}

YTCALL ythread_yield(yid)
{
    _yyid = yid;
    _prev_yat_stm = YIELD;
}
```

Figure 6: The function ythread_create() and ythread_yield()

size table `_act_size[]`. Next, the member `resume_pt` in the structure is initialized to be zero. This setting will let control start from the first labelled statement (i.e. `_s00`) in the converted thread's function. As discussed before, the member `fun_num` is set to the converted thread's function identifier which is the index of the converted function in the function address table `_pfunc[]`. The creation function then finds a free entry in the thread table for the new thread. The entry stores the thread's unique identifier, priority, and the pointer to the allocated structure. Note that creation of a new thread may cause a suspension of the currently running thread if the new thread has a higher priority.

A thread can yield control to another thread if they have equal priorities. The control transfer must go through the scheduler. If we allow a thread to transfer control by directly invoking the converted function for another thread, the system stack may overflow when two threads yield control to each other recursively. The principle of our approach is to let the resumption and suspension of a thread be realized by a pair of call and return operations. Therefore, the stack will return to its original state when the currently running thread suspends itself through a return statement. The scheduler then invokes the function for the designated thread. Figure 7 depicts a run-time state of the system.

To implement the thread migration primitive, we need to send a thread state from one

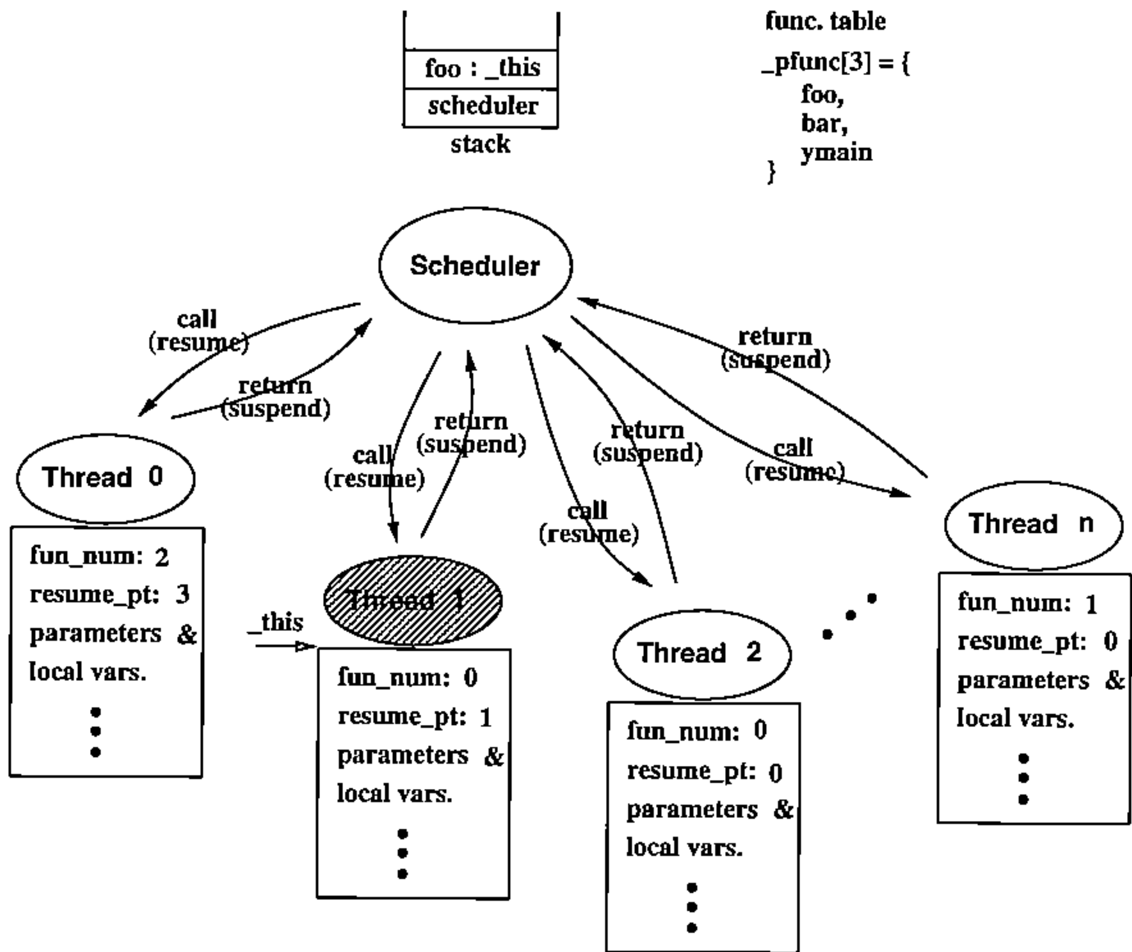

Figure 7: The Ythreads run-time system state on a single processor

machine to another. In our proposed programming model, an instance of a structure, including the logical identifier `fun_num`, the next re-entry point `resume_pt`, and local variables, represents a thread state. Note that we do not need to send code because the translated program has been compiled on all machines. Each processor has its own compiled code and its own `_pfunc[]` and `_act_size[]` tables. The destination site only needs the identifier `fun_num` to find out the corresponding function from the previously generated function table `_pfunc[]`. Furthermore, because the resumption point has been translated to an integer value, there is no need to translate the resumption address at migration time. Hence, migration can be realized in an efficient way.

Synchronization between processes can be achieved through two distinct coordination mechanisms. One mechanism is through *events*. A thread is suspended and put in a waiting queue for an event e if it *waits* for event e while event e has not yet occurred. Event e is said to occur when it is *set* by some other thread. At this point, all threads waiting for event e are reactivated simultaneously and put back on the ready list. The other mechanism for thread synchronization is through message-passing. The non-blocking *send* and blocking *receive* operations are provided.

4.3 Limitations and Current Work

Our current implementation restricts the Ythreads primitives to be used within the `thread` construct. This limitation sacrifices flexibility, but is desirable for efficiency. Without such a restriction, we would have to keep an activation chain which preserves the function calling sequence for a suspended thread. When the thread resumes, we can reach the previous suspension point through the activation chain. Of course, the chain must be transfered with the thread to be migrated. This will undoubtedly increase the context-switch and migration costs. Note that the Xthreads library does not have this limitation. A thread can be suspended in any function invoked by the thread. The function calling sequence which is stored on the stack can be sent across processors without translation for thread migration in a homogeneous environment.

In a heterogeneous environment, data conversion is always the major problem. Our current implementation is on the Sun SPARC IPC and IBM RS6000 workstations. These two machines

use the IEEE standard format to represent integer and floating-point numbers. The C compilers on both machines also generate the same arrangement for the members in a structure. This greatly reduces the migration cost. However, it is not always true that the source and the target machines use the same format, byte order, and data alignment. To solve this problem, the translator needs to generate more information about the structure, such as the number of members and the data type of each member, for the run-time migration routines. Therefore, the source machine can translate the members in the structure one by one using either an XDR protocol or even a string of ASCII numbers. The destination site can decompose the message in the reverse manner. Fortunately, there is a well-known package PVM[12] that provides such functions for many different kinds of machines. We are currently integrating the Ythreads library with PVM.

Pointers conversion also makes thread migration complicated and inefficient. For this reason, our current system prohibits a migrant thread from carrying pointers. To alleviate this constraint, it is necessary to build our own heap allocation mechanism. Each machine allocates a piece of memory and exchanges the starting address with each other at initialization stage. Instead of returning a physical address, the memory allocation routine returns an logical offset to the starting address. In fact, thread migration can be fully transparent if a heterogeneous shared memory environment[15] is supported.

5 Performance Measurements

The Ythreads library has been ported successfully on the nCUBE2 distributed-memory multiprocessor, the Sun SPARC IPC workstation, and the IBM RS6000 machine. In our environment, the nCUBE2 hypercube contains 64 nodes and each node has a 7 MIPS peak performance, while the Sun SPARC IPC and the IBM RS6000 workstations have the vendor-claimed peak performance rate of 15.7 MIPS and 28.5 MIPS, respectively.

5.1 Cost of Operations

We have conducted a few experiments to evaluate the operation costs of thread creation, thread switching, and thread migration. For measuring the thread switching cost, we created two threads which yield to one another. For measuring the creation plus deletion time, we created

Operations	Cre. + Del.	Ctx. Switch	Migration
nCUBE2 Xthreads	87	30	259
nCUBE2 Ythreads	96	36	278
RS6000 Xthreads	15	6	1200
RSS600 Ythreads	25	13	1200
SparcIPC Ythreads	43	29	940

Table 2: Comparison of Operation Latency (in μs) between Xthreads and Ythreads

a thread which will terminate immediately after starting to execute. Note that the times presented here also include the overhead resulting from priority-based scheduling. To evaluate the migration cost, we created a thread which travels back and forth on two processors. Each operation was executed for a large number of times. The cost per operation was obtained by dividing the elapsed time by the number of times the operation was performed. Also included in Table 2, for the purpose of comparison, are the corresponding overheads of the Xthreads operations on the nCUBE2 and the IBM RS6000 machines. We have not ported the Xthreads library on the Sun SPARC IPC workstation yet.

It is not surprising to see that the Ythreads system exhibits slightly larger overheads than its Xthreads counterpart, most likely due to its lack of assembly code for thread context-switching. A thread switch in the Ythreads system also needs to return to the scheduler which requires one more function call/return as compared to the direct switch in Xthreads. On an nCUBE processor, a null function call and return requires approximately 1.5 μs.

The migration overheads presented in Table 2 were performed on two homogeneous processors. That is, they were measured on two of the nCUBE2 processors through its internal interconnection network, and on two IBM RS6000 workstations (or two Sun Sparc IPC machines) through an Ethernet. The communication facilities in Ythreads are implemented on top of UDP/IP, the Internet Universal Datagram Protocol. The cost on two workstations is higher than on the nCUBE2 because passing a message through inter-processor communication networks is usually three to five orders of magnitudes cheaper than on an Ethernet. Note that the latency of the migration operation is highly sensitive to machine workload and network traffic. Therefore, the timings may vary under different workloads.

Local data (in bytes)	16	64	256	512	1024
Migration Cost	5.6	6.0	6.5	7.2	9.5

Table 3: Heterogeneous Thread Migration Latency (in ms) with Different Size of Local Data

5.2 Cost of Heterogeneous Migration

We also performed an experiment to evaluate migration cost between the Sun Sparc IPC and IBM RS6000 workstations. Table 3 shows the results obtained by varying the size of local data (in bytes) defined inside a thread. The more local data used in a thread, the larger the converted structure that is needed, the longer transmission overhead there is, and therefore the higher a migration cost that is incurred. Because the two workstations are located on different local area networks, the migration cost is higher than the cost shown in Table 2. Passing migration information through gateways undoubtedly increases the overhead.

5.3 Simulation Benchmark Measurements

Based on the proposed multithreaded paradigm, a novel *mobile-process* approach has been proposed for parallelizing process-oriented simulation systems (i.e., systems consisting of active lightweight processes and passive objects). This approach entails the migration of a requesting process with its timestamp to the remote site hosting the requested passive object. As a result, the migrant process can make subsequent accesses of the object locally and continue execution transparently, as if on its original host. The advantages of one-time transmission, fixed communication topology and increased data locality make the mobile-process approach more effective than corresponding send-reply based remote-procedure-call paradigm. A detailed description of the mobile-process approach and related experiments can be found in [9].

We have implemented two versions of the parallel simulation systems, PSi_X and PSi_Y, based on the Xthreads and Ythreads libraries, respectively. We used the Cluster Queueing Network (CQN) as a benchmark to evaluate and compare the systems performance. An $M \times N$ cluster queueing network consists of M tandem queues, each containing N FIFO servers. A job which arrives at a queue is served by the N servers sequentially. After completing service at the last server in a queue, the job is routed back to the front of any queue based on a given

No. of procs	1	2	4	8	16	32	64
PSi_X (4x64)	469	357	209	99	46	22	11
PSi_Y (4x64)	522	374	217	104	48	23	11
PSi_X (4x256)	1872	1235	671	315	143	64	30
PSi_Y (4x256)	1925	1285	700	325	147	67	32

Table 4: Times (in seconds) for simulating cluster queueing networks

probability.

In the simulation exercise, each server is modelled by a passive pre-defined object in PSi, and each job is represented by a dynamic process (i.e. a thread) traveling around the network to access the servers. We executed a 4×64 network model and a 4×256 network model, each initialized with a total of 1024 jobs. We distributed the servers equally between the processors. Table 4 shows the performance figures (in seconds) obtained on the nCUBE2 machine. As compared with PSi_X, PSi_Y typically performs less than 5% slower in most cases. However, both systems can achieve good speedups. Note that superlinear speedup occurs because our current system uses a heap [1] to maintain the event calendar, where both inserting an event and deleting a minimum timestamp event have a time complexity of $\log(q)$, where q is the size of the event calendar. When running on p processors in parallel, the simulation calendar on each processor has an average of q/p events, and thus an insertion/deletion cost of $\log(q/p)$, resulting in super-linear cost reduction.

5.4 Code Length Measurements

In addition to slightly larger overheads, our thread transformation also imposes a penalty on code expansion. The more threads declared, the more translated code inserted. Also, the more synchronization between threads, the more re-entry points that will be generated. We measured the sizes of the source code and the translated code for each of the context-switch benchmark program, the M/M/1 program in PSi, and the CQN benchmark program. Each program is evaluated in terms of lines of C statements and the size in bytes of its compiled object file, respectively. Table 5 shows the results.

Program	Ctx. Switch		M/M/1		CQN	
	C	Obj.	C	Obj.	C	Obj.
Source	63	1340	99	2304	172	3944
Translated	124	1836	168	3436	236	5224

Table 5: Comparison of Code Length

6 Conclusion

We have delineated a prototype multi-threaded system that permits rapid development of process-based descriptions of computations, allowing for migration of computations, via threads, across heterogeneous wide-area networks. We have implemented the system successfully on various machines and demonstrated applications in parallel simulation. The experimental results show that the performance of heterogeneous threads is comparable to performance of homogeneous threads.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.

[2] J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors. In *Proceedings of the Symposium on Operating System Principles*, pages 147–158, 1989.

[3] T. W. Doeppner Jr. Threads - a system for the support of concurrent programming. Technical Report CS-87-11, Computer Sciences Department, Brown University, 1987.

[4] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of the International Conference on Parallel Processing*, pages 147–158, 1988.

[5] F. Mueller. A Library Implementation of POSIX Threads under UNIX. In *Proceedings of the Winter USENIX Conference*, 1993.

[6] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. SunOSmultithread Architecture. In *Proceedings of the Winter USENIX Conference*, 1991.

[7] T. W. Pratt. *Programming Languages Design and Implementation.* Prentice-Hall, Englewood Cliffs, NJ 07362, second edition, 1984.

[8] J. Sang, F. Knop, V. Rego, J. K. Lee, and C.-T. King. The Xthreads Library: Design, Implementation, and Applications. In *Proceedings of the COMPSAC*, 1993.

[9] J. Sang, E. Mascarenhas, and V. Rego. Process Mobility in Distributed-memory Simulation Systems. In *Proceedings of the Winter Simulation Conference*, 1993.

[10] J. Sang and V. Rego. Thread Migration on Distributed-memory Multiprocessors. In *Proceedings of High Performance Computing Symposium*, 1994.

[11] C. Shub. Naive code process-originated migration in a heterogeneous environment. In *Proceedings of the ACM 18th Annual Computer Science Conference*, pages 266–270, 1990.

[12] V. S. Sunderam. PVM: a Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, Dec. 1990.

[13] A. Tevanian, R. F. Rashid, D. B. Golub, D. L. Black, E. Cooper, and M. W. Young. Mach Threads and the UNIX Kernel: The battle for control. In *Proceedings of the summer USENIX Conference*, 1987.

[14] M. M. Theimer and B. Hayes. Heterogeneous Process Migration by Recompilation. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 18–25, 1991.

[15] S. Zhou, M. Stumm, K. Li, and D. Wortman. Heterogeneous Distributed Shared Memory. *IEEE Trans. on Parallel and Distributed Systems*, 3:540–554, Sep. 1992.