A Machine Independent Interface for Lightweight Threads

Bodhisattwa Mukherjee (bodhi@cc.gatech.edu)
Greg Eisenhauer (eisen@cc.gatech.edu)
Kaushik Ghosh (kaushik@cc.gatech.edu)

GIT–CC–93/53

Abstract

Recently, lightweight thread libraries have become a common entity to support concurrent programming on shared memory multiprocessors. However, the disparity between primitives offered by operating systems creates a challenge for those who wish to create portable lightweight thread packages. What should be the interface between the machine-independent and machine-dependent parts of the thread library? We have implemented a portable lightweight thread library on top of Unix on a KSR-1 supercomputer, BBN Butterfly multiprocessor, SGI multiprocessor, Sequent multiprocessor and Sun 3/4 family of uniprocessors. This paper first compares the nature and performance of the OS primitives offered by these machines. We then present a procedure-level abstraction that is efficiently implementable on all the architectures and is a sufficient base upon which a user-level thread package can be built.

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332–0280

1 Introduction

The effectiveness of parallel computing depends on the performance of the primitives offered by a system to express parallelism. If the cost of creating and managing parallelism is high, even a coarse-grained parallel program shows poor performance. On the other hand, if the cost of creating and managing parallelism is low, even a fine-grained program can achieve excellent performance.

One way to express parallelism is by using 'Unix-like' processes. Such a process consists of an address space and a single thread of control. Operating system kernels supporting such processes do not distinguish between a thread and its address space. Parallelism expressed using such heavyweight processes must be coarse-grained and is often not suitable for high performance parallel programs. Hence, in many contemporary operating system kernels, address space and threads are decoupled so that a single address space can have multiple execution threads[2, 4]. Threads are an emerging model for expressing concurrency within Unix processes[7]. In multiprocessors, threads are primarily used to simultaneously utilize all the available processors. Threads are even useful on a uniprocessor system for mapping asynchronous behavior into equivalent synchronous behavior[7, 4]. Though such kernel-level threads offer a general programming interface to an application, they are expensive and therefore are not used in fine-grained parallel programs[4, 17].

Unlike kernel-level threads, *user-level* threads (also known as *lightweight* threads) are managed by runtime library routines linked into each application[3, 6]. They are efficient because thread management operations do not need to cross the kernel protection boundary. Furthermore, lightweight threads enable an application program to use a thread management system which is most appropriate to the problem domain. Recently, thread libraries have become a common element of new languages and operating systems for shared memory multiprocessor and uniprocessors to support concurrent programming. Mach Cthreads[6, 16], the University of Washington threads[15, 2], POSIX Pthreads[7], SunOS LWP and threads[12, 13, 20], are a few popular lightweight thread implementations.

We have implemented a portable user-level thread package[1][16] on several multiprocessor platforms such as BBN Butterfly parallel processor, Kendall Square Research parallel processor, Sequent Symmetry multiprocessor, Silicon Graphics multiprocessor and on several uniprocessors such as Sun 3 and Sun 4 families. Though these machines run implementations of UNIX as their native operating systems, each machine offers a different set of parallel programming primitives to applications.

In the next sections, we first mention the salient points about some hardware platforms. Then, we report the performance of some frequently used Unix primitives and parallel programming support offered by the vendors of said hardware. Further, we present a set of operating system primitives that we found sufficient for a portable, efficient implementation of user-level thread package on these hardware platforms.

[1] The interface of the library is an extension of the Mach Cthreads library

2 Machine Architectures

In this section, we state the relevant characteristics of the machines on which we implemented our threads. This serves to develop a perspective on performance measurements of the basic Unix functionality on each of these hardware platforms (such measurements are reported at the end of this section).

The KSR supercomputer is a NUMA (non-uniform memory access) shared memory, cache-only architecture with an interconnection network that consists of hierarchically interconnected rings, each of which can support up to 32 nodes or 34 rings. Each node consists of a 64-bit processor, 32 MBytes of main memory used as a local cache, a higher performance 0.5 Mbytes sub-cache, and a ring interface. CPU clock speed is 20 MHz, with peak performance of 40 MIPS per node (due to two functional units that can operate in parallel). Access to non-local memory results in the corresponding cache line being migrated to the local cache, so that future accesses to that memory element are relatively cheap[9, 18]. At the lowest level, the parallel programming model offered by the KSR's OSF Unix operating system is one of kernel-level threads which offer constructs for thread fork, thread synchronization, shared memory between threads, etc..

The 8-node SGI multiprocessor, on the other hand, is a UMA (uniform memory access) shared memory machine, and consists of four 33 MHZ IP7 and four 25 MHZ IP7 processors. Each node contains a MIPS R2010A/R3010 FPU, a MIPS R2000A/R3000 CPU, 64 Kbytes data cache and 64 Kbytes instruction cache. The parallel programming support provided by SGI's IRIX operating system (a version of Unix) includes creation/management of "share group processes", shared memory arena, lock control operations etc..

The 10-node Sequent Symmetry multiprocessor is also a UMA bus based machine consisting of Intel 80386 microprocessors, each of which contains 64 Kbytes of cache. Symmetry systems can be configured with 8 to 240 Mbytes of memory, and they provide 256 Mbytes of virtual address space per process. Sequent multiprocessors run the DYNIX operating system, a version of Unix 4.2BSD that also supports most utilities, libraries, and system calls provided by Unix System V. The DYNIX parallel programming library includes three sets of routines: a micro-tasking library (to create/manage a set of child processes), a set of routines for general use with data partitioning programs, and a set of routines for memory allocation in data partitioning programs.

The 32-node BBN Butterfly GP1000 multiprocessor is a NUMA machine which consists of a set of processors and memory connected in a wraparound Omega network. Each processor node consists of a 16 MHZ MC68020 processor, a custom Processor Node Controller, and 1 MB of memory, expandable to a 4 MB by a daughter-board arrangement resulting in a computing capacity of 1 MIPS. BBN Butterfly multiprocessors run a version of Unix which supports a few Mach[5, 10] features.

Table 1 compares the cost of a few basic operations of these Unix implementations. In the table, context save and restore operations refer to Unix `setjmp` and `longjmp` operations,

Operation	BBN	SGI25	SEQUENT	SPARC	KSR
Context save (saves signal mask)	810	38	140	46	270
Context save (without signal mask)	28.7	3.90	8.06	2.17	5.34
Context restore (with signal mask)	1200	45	140	39	210
Context restore (without signal mask)	32.87	1.70	8.81	22.33	6.17
Test-and-Set	31.19	0.73	5.73	-	1.32
Fork	6300	50	2100	240	49000
Create Virtual Proc.	59000	25	740	240	71

Table 1: Cost (μsecs) of basic OS operations

test-and-set refers to the most primitive atomic instruction (`atomior` for the BBN Butterfly, `uscsetlock` for SGI, `s_lock` for Sequent Symmetry, `gspnwt` for KSR1, and none for Sparc). A virtual processor is created on the BBN Butterfly using `fork_and_bind`, on SGI using `sproc`, on Sequent Symmetry using `m_fork`, and on the KSR1 using `pthread_create` calls. This table serves to set up a perspective for performance measurements of similar operations on lightweight threads, as listed in table 3.

3 The Proposed Portable Interface

The performance of applications using a thread library depends on the performance of the primitives offered by the library, which in turn depend on the primitives offered by the underlying operating system and the machine architecture. We implemented a portable lightweight thread package on a number of different multiprocessors (ranging from UMA to NUMA) and uniprocessors. In the process of implementation, we have been able to identify a machine-independent interface and a set of OS primitives which can be used to develop efficient user-level thread libraries. The proposed portable OS interface consists of three well defined parts: (1) a hardware library (Fig 1) (2) process and scheduling support and (3) memory management support.

4 Getting to the hardware

The hardware library provides applications with easy access to the machine hardware. This section identifies the hardware primitives that can be directly used by the thread library (without OS interference).

Synchronization. Threads of a fine-grain parallel application synchronize very often. Therefore, a small increase in the cost of synchronization operations may have a considerable impact on application performance. Conversely, a less expensive thread synchronization primitive can

significantly improve performance.

Scheduling and Dispatching cost. A large fine-grain parallel application typically consists of "many threads" communicating via "many messages". The cost of synchronization operations determines the cost of these "many messages" whereas, the scheduling cost determines the cost to schedule those "many threads" on a limited number of available processors. Though a well-designed application-specific scheduler minimizes the number of context switches between threads by carefully avoiding any useless context switch (arising due to synchronization, inter-thread communication etc), it is impossible to completely avoid context switches among threads (especially when the number of threads is more than the number of processors). Thread context switches add to the overhead of parallel applications. Apart from the primary effect of saving and restoring context, there is also the secondary effect of destroying cache footprints [21].

A few popular classes of applications (e.g. interactive, real-time) often use temporal information to make scheduling decisions. Such information is obtained by accessing the hardware provided "timer". For such applications, the cost of generating/handling timer interrupts plays a vital role in application performance.

Dynamic operation costs. Thread based parallel programs spend significant amount of time in dynamic operations such as creating/destroying threads, allocating/freeing memory segments etc. Though some of these operations are necessary, several of them can be done statically (at compilation time) or once at initialization time. For example, it is less expensive to statically share memory between threads than allocating dynamic shared memory. Our future research focuses on minimizing such dynamic operations using user hints. However, that is not in the scope of this paper.

The total cost of a parallel application can be expressed as:

$$C_a = \alpha(C_{sync}) + \beta(C_{csw}) + \delta(C_{dyn}) + C_I \text{ where,}$$

C_a, C_{sync}, C_{csw}, C_{dyn}, C_I are the application cost, synchronization cost, context switch cost, dynamic operation cost, and other (thread-independent) cost respectively and α, β, and δ are arbitrary functions monotonically increasing with the number of threads. We posit that C_I remaining constant, the minimum value of C_{sync}, C_{csw}, and C_{dyn} produce the minimum value of C_a. Furthermore, the minimum possible value of C_{sync} and C_{csw} are bounded below by the hardware speed and primitives. The remainder of this section lists these necessary hardware primitives.

Saving and restoring hardware context. The recent generation of processors define the context buffer as hardware objects and implement context save and restore operations as hardware instructions. The operating system should export these, or similar instructions, to the application layer. Followings are the proposed library routines for such purpose:

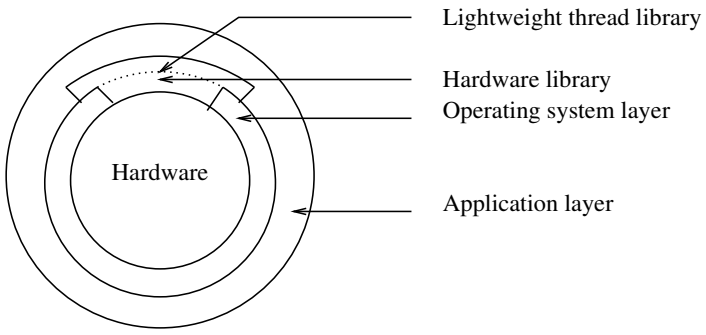

Figure 1: Software layers

```
hw-save-context(buffer hw-buf);
hw-restore-context(buffer hw-buf);
```

Hardware atomic instruction. Though most processors support a simple atomic `test-and-set`[2], and a `bus-lock` instruction at the hardware level, most operating systems do not export them directly to the user level. Instead, they export a set of expensive and sophisticated synchronization primitives to the application layer. Unix processes have distinct address spaces; hence, synchronization has to be achieved through system-provided (shared) semaphores. Such synchronization traps to the kernel, and is therefore expensive. Threads on a processor share an address space, and a set of shared variables can be used to achieve synchronization without trapping to the kernel, or expensive memory-mapping techniques (e.g., `mmap()`, `shmget()` or `usinit()`[3]). Hence, the hardware library should provide these inexpensive hardware-implemented synchronization primitives to the user:

```
hw-test-and-set(lock hw-lock);
```

Timer access. It is important to have support for (timer) interrupts to develop preemptible thread libraries[23]. Most operating systems support an expensive interface to the timer interrupts and interrupt handling capabilities (e.g., the UNIX "signal" mechanism). These primitives are often so expensive (Table 2) that they are practically unusable in a lightweight thread implementation. The following are a set of proposed timer-related routines to be exported by the hardware library to be used in user-level thread implementation.

```
hw-load-timer(timer hw-timer, int time);
hw-catch-interrupt(interrupt-type interrupt, handler intr-hndlr);
hw-check-timer(timer hw-timer);
hw-mask-timer(mask-type hw-mask);
```

[2] For example, the `XCHG` instruction of the Intel x86 family
[3] On SGI multiprocessors.

Operations	BBN	SGI25	SEQUENT	SPARC	SUN360	SUN386
Load Timer	6588.51	91.50	254.78	72.33	509.65	281.66
Read Timer	5891.33	46.09	106.62	31.33	203.49	220.16
Timer Interrupt	1985.58	262.50	442.17	126.49	777.14	936.96
Mask Timer	441.41	41.90	67.14	19.33	135.49	136.83

Table 2: Timer operations in different machines (μseconds)

Miscellaneous. It is often necessary to access, and manipulate a thread stack for easy parameter passing. The newer generation of microprocessors support some primitives for structured access to a stack[4] in their instruction set. Following are the two proposed stack manipulation functions needed in a thread library:

```
hw-push(stack_t stack, hw_record a_record);
hw-pop(stack_t stack);
```

It is often convenient to have access to a fast block copy operation (For example: stack copying for thread-migration). Since, a few machines implement such primitives at the hardware level, the hardware-library should export it to the user as a library routine.

4.1 Managing Processes

Typically, lightweight threads execute in the context of middleweight or heavyweight kernel supported threads. Specifically, lightweight threads share the "thread of control" from kernel-provided active entities. An application level scheduler schedules lightweight threads on top of kernel supported threads (let ϕ be the mapping function), which in turn are scheduled by a kernel scheduler (κ) on the available physical processors (referred to as *two-level scheduling*) [2, 3]. Figure 2 illustrates such a computation model where kernel-level active entities form a cluster of virtual processors for user-level threads. If both ϕ and κ map 1:1, it results in a true parallelism.

The implementation of a lightweight thread library depends on the kernel scheduler (ϕ). Earlier research in scheduling [14, 22, 11] has compared the performance of applications under "processor partitioning" scheme[5] vs. "time sharing" scheme[6], and has observed that processor partitioning considerably improves application performance. We implemented the thread library on top of both of these schemes (Sequent/DYNIX and SGI/IRIX implement time-sharing whereas BBN/Mach implements processor partitioning). Our results support the observation above.

Concurrent execution of application threads results either due to concurrency of threads on a virtual processor (κ) or concurrency of virtual processors on physical processors (ϕ). However,

[4] The `ENTER` and `LEAVE` instructions in Intel x86 instruction set, e.g..

[5] The processors in a machine are partitioned into a few clusters. An application runs on a dedicated cluster and does not share its processors with other applications

[6] The processors of the machine are time-shared among all the applications.

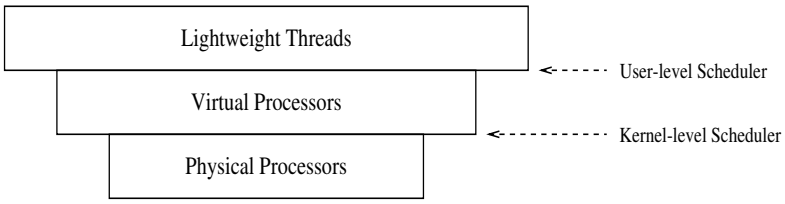

Figure 2: Computation model

true parallelism (parallel execution of threads) results only due to parallel execution of the virtual processors. Therefore, to obtain true application parallelism, the operating system has to provide support to execute the virtual processors simultaneously (also known as "coscheduling" or "gang scheduling"[19]).

Thus, an operating system should provide the following functionality:

```
os-make-cluster(int no_of_processors);
os-create-and-bind_a_process(int processor_number);
```

As the names suggest, `os-make-cluster` creates a cluster of a "user provided" number of processors whereas the function `os-create-and-bind-a-process` creates a process and binds it to a specific processor in the cluster. These two functions are sufficient to satisfy the processor partitioning and coscheduling requirements of lightweight threads.

4.1.1 Process address space

Different operating systems export different process models to the user. For example, the address space of a (UNIX) process consists of three well-defined segments – code segment, data segment, and stack segment. The code segment is the instruction space of the process and contains instructions to be executed by the thread of control, whereas, the data segment contains the (global) data used by the process and the stack segment contains the context of the computation. Although multiple processes may share their code and data segments, each process has its own separate stack segment.

Data Sharing among processes is important in multiprocessors to implement inter-process communication. Such sharing may occur either statically (at compilation time) or dynamically (at execution time). For example, in Sequent/DYNIX, a user may explicitly specify a data item to be shared among a group of processes using the "shared" keyword. Similarly, some operating systems implement process groups that inherently share an address space. A few operating systems[8, 1] also support dynamic address space sharing using sophisticated address mapping techniques.

Depending on the amount of static data sharing, we have classified the process model in

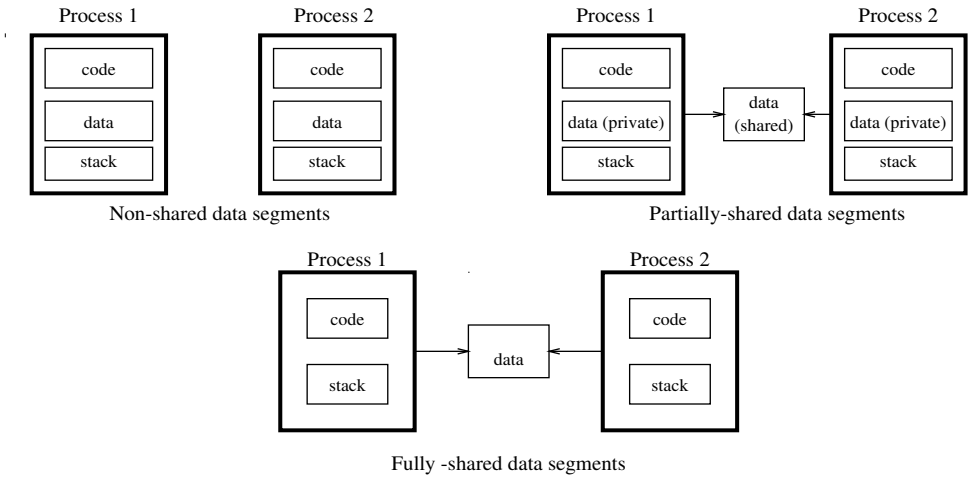

Figure 3: Process models

three categories (Figure 3):

Fully shared data segments. In this model, a group of related processes share one data segment. Any change in the data segment affects all the processes. Stack segments are used for private data.

Partially shared data segments. In this model, a data segment consists of two well defined portions – a private part which contains private data of processes and a shared part which is used to contain shared information. Communication between processes happen via the shared data segments.

Non shared data segments. Processes in such a model do not statically share data among them. Data sharing among such processes is implemented by expensive (dynamic) memory mapping techniques.

We have implemented the lightweight thread library on all of these three models. Since threads implement a shared memory programming paradigm, the fully shared data model of process groups are best suited for a lightweight thread implementation. In such a model, threads in all virtual processors share the entire data, thus implementing simple inter-thread and inter-processor communication. However, it introduces some inefficiency in the implementation of the library itself due to excessive use of synchronization operations. It needs careful programming and synchronization to maintain consistency of node (virtual processor) specific data structure. On the other hand, a non-shared-data process model uses less synchronization operations in the implementation of thread routines. However, information sharing across nodes requires

expensive dynamic memory allocation techniques resulting in inefficient communication and load balancing. For example, if a thread wants to update data in a remote node, it does not have much choice except creating a "messenger thread" in the remote node.

Therefore, we found partially shared data model most suitable for lightweight threads. In this model, node specific data structure are allocated in the private data segment (thus avoiding unnecessary synchronization operations), and global shared data (e.g., global application variables) are allocated in a shared data segment (thus maintaining the shared memory programming paradigm). Though accessing a node-private data is still a problem, such accesses can be avoided by careful programming.

To summarize, non-shared data model is bad because of unnecessary dynamic memory operations required for inter-processor coordination. Fully shared data segments are bad due to unnecessary synchronization operations required to maintain consistency of internal data structure of the library. Partially shared data model provides a middle ground between the above two extremes and avoids most of these problems.

4.2 Managing Memory

Memory management requirements for a thread library are extremely dependent on the underlying architecture. For NUMA multiprocessor architecture, it is important to allocate as much data in local memory as possible. Furthermore, it is often necessary to replicate a segment in different nodes (example: replication of code segment of virtual processors). The following is a list of memory management functions required for NUMA machines:

```
os-allocate-and-bind(int memory-size, int processor-number);
os-allocate-and-replicate(int memory-size, int cluster-id);
```

Since, all memory is equidistant in a UMA architecture, UMA machines do not need the above functions. However, they may benefit from some application control on the cache memory. Our future research focuses on a minimal set of primitives required by lightweight threads for the control of cache memory.

Thread libraries supporting dynamic memory allocation need support for dynamic memory from the operating system. The following are the essential functions for such purpose.

```
os-allocate-and-copy(int memory-size, int processor-number);
os-allocate-and-share(int memory-size, int cluster-id);
```

The function `os-allocate-and-copy()` allocates node-private memory whereas the function `os-allocate-and-share()` allocates memory to be shared by virtual processors. Note that `os-allocate-and-bind` and `os-allocate-and-replicate` are used at the physical processor level where as the functions `os-allocate-and-copy` and os-allocate-and-share are used at the virtual processor level.

Though most of the recent multiprocessor operating systems support efficient paging mechanisms, performance improves considerably if the address space of an application is pre-paged into the local memory of its cluster during initialization. Therefore, the following operating system function is required to improve the performance of the thread library :

```
os-page-in(address-space-t ad-space);
```

4.2.1 Summary

This section summarizes the portable interface:

Hardware library:

```
hw-save-context(buffer hw-buf);
hw-restore-context(buffer hw-buf);
hw-test-and-set(lock hw-lock);
hw-load-timer(timer hw-timer, int time);
hw-catch-interrupt(interrupt-type interrupt, handler intr-hndlr);
hw-check-timer(timer hw-timer);
hw-mask-interrupt(interrupt-type hw-interrupt);
hw-push(stack-t stack, hw-a_record a_record);
hw-pop(stack-t stack);
hw-block-copy(char-t source, char-t dest, int size);
```

Process and Scheduling support:

```
os-make-cluster(int no-of-processors);
os-create-and-bind-a-process(int processor-number);
```

– *Process family with partially shared data.*

Memory Management support:

```
os-allocate-and-bind(int memory-size, int processor-number);
                              (For NUMA architecture)
os-allocate-and-replicate(int memory-size, int cluster-id);
                              (For NUMA architecture)
os-allocate-and-copy(int memory-size, int processor-number);
os-allocate-and-share(int memory-size, int cluster-id);
os-page-in(address-space-t ad-space);
```

5 Implementation

Once the above mentioned operating system support is available, a threads library can be easily and efficiently implemented (a few algorithms are listed in the appendix). We have implemented

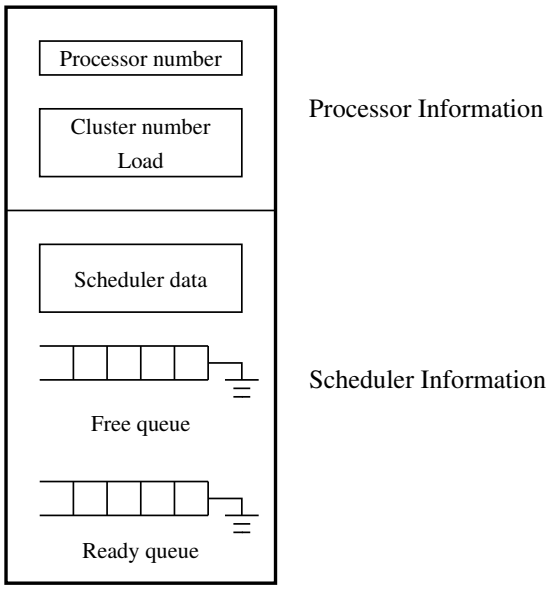

Figure 4: Process-private data

the above interface on different machines using the existing OS and hardware primitives, then implemented the library on top of the interface.

The library is initialized by creating a number of virtual processors and node-specific data structures. The structure of a sample node-private data structure is shown in Figure 4.

When the library is initialized, a number of thread control blocks are pre-allocated, initialized and put in the free thread queue. This strategy of one time creation of the thread pool reduces the time required to create a thread dynamically (C_{dyn}). A thread is represented using a thread control block which contains (among other things) a stack and a buffer to save the hardware context (`hw-buf`). The thread scheduler goes through the run queue and dispatches threads onto the virtual processor. Table 3 lists the costs of the basic threads operations for different machines implemented on top of the proposed interface.

Operation	BBN	SGI25	SEQUENT	SPARC	KSR
null function	3.33	0.44	5.03	0.599	1.78
thread-fork	475.67	55	183.95	20.83	71.5
thread-fork-and-detach	536.94	60	210.03	33.33	378.37
thread-yield	219.96	17.2	93.20	51.66	38.69
pingpong	434.81	35	181.52	116.66	78.09

Table 3: Costs (μseconds) of the basic thread operations

6 Conclusion

Efficiency of lightweight threads depends on the primitives offered by an underlying kernel. In this paper, we have proposed a sufficient set of primitives that a multiprocessor operating system might provide for an efficient implementation of a user-level thread library. Specifically, the contributions of this paper are:

- Proposing a portable OS interface targeted toward efficient implementation of user-level thread packages.
- Experimentation with different OS trade-offs such as "time-sharing" vs. "processor-partitioning", various memory models etc.
- Implementation of the portable library on various multiprocessors including a 32-node KSR super computer, a Sequent symmetry multiprocessor, a BBN Butterfly parallel processor, and a SGI multiprocessor.

In the future, we will investigate the effect of user-level cache control on lightweight threads for Uniform Memory Access multiprocessors.

7 Availability

The portable lightweight thread library is available in the public domain by remote FTP access to ftp.cc.gatech.edu

References

[1] Jr. A. Tevanian. *Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments*. PhD thesis, School of Computer Science, Carnegie-Mellon University, December 1987. Also as Technical Report CMU-CS-88-106.

[2] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *Transactions on Computer Systems, ACM*, 10(1):53–79, February 1992.

[3] Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[4] Brian N. Bershad. High performance cross-address space communication. Technical Report 90-06-02, Dept. of Computer Science and Eng., University of Washington, June 1990. Ph.D. dissertation.

[5] David. L. Black. Scheduling support for concurrency and parallelism in the mach operating system. *IEEE Computer*, May 1990. Also as CMU Technical Report CMU-CS-90-125, April 1990.

[6] E. Cooper and R. Draves. C threads. Technical Report CMU-CS-88-154, Dept. of Computer Science, Carnegie Mellon University, June 1988.

[7] POSIX P1003.4a IEEE Draft. *Threads Extension for Portable Operating Systems.*

[8] R. Fitzgerald and R. Rashid. The integration of virtual memory management and interprocess communication in accent. *ACM Transactions on Computer Systems*, May 1986.

[9] Kaushik Ghosh, Bodhisattwa Mukherjee, and Karsten Schwan. Experimentation with configurable, lightweight threads on a ksr multiprocessor. Technical Report GIT-CC-93/37, College of Computing, Georgia Institute of Technology, 1993.

[10] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an application program. In *Proceedings of the Summer Usenix Technical Conference*, June 1990.

[11] A. Gupta, A. Tucker, and S. Urushibara. The impact of operating systems scheduling policies and synchronization methods of the performance of parallel applications. In *Proc. 1991 ACM SIGMETRICS Conf. on Meas. and Mod. of Comp. Sys.*, May 1991.

[12] Sun Microsystem Inc. *Sun OS 4.0 Reference Manual*, November 1987. Section 3L.

[13] J. Kepecs. Lightweight processes for unix implementation and application. In *Proc. 1985 USENIX Summer Conference*, pages 299–308, 1985.

[14] S. Leutenegger and M. Vernon. The performance of multiprogrammed multiprocessor scheduling policies. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1990.

[15] P. McJones and G. Swart. Evolving the unix system interface to support multithreaded programs. In *Proc. USENIX Winter Conference*, pages 393–404, 1989.

[16] Bodhisattwa Mukherjee. A portable and reconfigurable threads package. In *Proceedings of Sun User Group Technical Conference*, pages 101–112, June 1991. TR# GIT-ICS-91/02.

[17] Bodhisattwa Mukherjee and Karsten Schwan. A survey of multiprocessor operating system kernels. Technical Report GIT-CC-92/05, College of Computing, Georgia Institute of Technology, January 1992.

[18] Bodhisattwa Mukherjee and Karsten Schwan. Experimentation with a reconfigurable micro-kernel. In *Proc. of the USENIX Symposium on Microkernels and Other Kernel Architectures*, pages 45–60, September 1993.

[19] J. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of Distributed Computing Systems Conference*, pages 22–30, 1982.

[20] M. Powell, S. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. Sunos multi-thread architecture. In *Proc. USENIX winter conference*, pages 1–14, 1991.

[21] R. Vaswani and J. Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Proceedings of the thirteenth ACM Symposium on Operating Systems Principles*, pages 26–40, October 1991.

[22] J. Zahorjan and C. McCann. Processor scheduling in shared memory multiprocessors. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 214–225, May 1990.

[23] Hongyi Zhou and Karsten Schwan. Dynamic scheduling for hard real-time systems: Toward real-time threads. In *Proceedings of Joint IEEE Workshop on Real-Time Operating Systems and Software and IFAC Workshop on Real-Time Programming, Atlanta, GA*, pages 13–21. IEEE, May 1991. Also in IEEE Real-Time Systems Newsletter, Vol. 7, No. 4, Fall 1991, pp. 14-22.

A Implementation of a few thread routines

lwt-init(int no-of-processors):

```
      (* initializes the library *)
{
      cluster = os-make-cluster(no-of-processors);
      foreach processor in cluster do
            os-create-and-bind-a-process(processor);
            os-allocate-and-copy(private-data-size, processor);
                  (* to allocate virtual processor private data *)
            Initialize node-private data;
            os-page-in(self-address-space);
      od
      os-allocate-and-share(shared-data-size, cluster);
      Initialize shared data;
      start per-processor thread scheduler in each processor;
}
```

lwt-fork-a-thread(int processor-no, function func, argument arg):

```
      (* creates a thread to execute func(arg) *)
{
      new-thread-control-block = dequeue(scheduler.freeq);
      Initialize new-thread-control-block.hw-buf;
      Initialize other fields of new-thread-control-block;
      a-record = make-activation-record(func, arg);
      hw-push(new-thread-control-block.stack, a-record);
```

```
    enqueue(scheduler.freeq, new-thread-control-block.runq);
}
```

lwt-sched-dispatch(tcb new-tcb, tcb current-tcb):

```
    (* switches the virtual processor from the current thread
    to the new thread *)
{
    hw-save-context(current-tcb.hw-buf);
    hw-restore-context(new-tcb.hw-buf);
}
```

lwt-spin-lock(lock slock):

```
{
    while (hw-test-and-set(slock.hw-lock) == LOCKED);
}
```

lwt-block-lock(lock b-lock):

```
{
    if (hw-test-and-set(b-lock.hw-lock) == LOCKED) do
        enqueue(b-lock.queue, current-tcb);
    od
}
```