Strands: An Efficient and Extensible Thread Management Architecture

Emin Gün Sirer Przemysław Pardyak Brian N. Bershad

May 15, 1996

1 Introduction

Applications can significantly benefit from specializing thread packages, schedulers and synchronization primitives to their needs. In prior systems, specialization has been accomplished through a partitioning of service across the user-kernel boundary. The kernel provides some basic control flow services while user code implements the specialized interface. This approach, though, has been shown to suffer from poor performance or poor integration [Anderson et al. 92].

In this paper, we describe a new architecture for thread and scheduling subsystems that provides correct, extensible and efficient thread management for applications. The *strand* architecture enables applications to place their specialized thread management code in the kernel address space. This allows the operating system to perform upcalls without crossing costly hardware boundaries. As well, it enables applications to contact other system services with low overhead.

The system safety issues that arise when placing application code in the kernel are handled in two ways. First, application handlers are written in a typesafe language, Modula-3, to ensure memory-safety. A user thread package executing in the kernel cannot corrupt the kernel's memory or call inappropriate kernel procedures. Second, the strand interface is structured to prohibit the failure of any application handler from affecting threads not directly managed by that handler. The end result is that the strand architecture allows application code to be tightly integrated with system services, thereby enabling correct, safe and efficient implementations of specialized thread managers.

We have implemented the strands architecture in the context of *SPIN*, which is an extensible operating system being developed at the University of Washington. Using strands, we have implemented several threads packages, including CThreads[Cooper & Draves 88], Mach kernel threads[Accetta et al. 86] and UNIX processes[Ritchie & Thompson 74], as well as low overhead synchronization primitives[Bershad et al. 92], application-specific schedulers[Zahorjan & McCann

[0]This research was sponsored by the Advanced Research Projects Agency, the National Science Foundation (Grants no. CDA-9123308 and CCR-9200832) and by an equipment grant from Digital Equipment Corporation. Bershad was partially supported by a National Science Foundation Presidential Faculty Fellowship. Sirer was supported by an IBM Graduate Student Fellowship.

90, Marsh et al. 91], debugging tools[Redell 88] and sampling profilers. In a related publication [Bershad et al. 95], we briefly described the performance of some of the threads packages. In this paper, we focus on the strand architecture, and provide more detailed information about performance.

1.1 Prior Work

Application-specific thread management has been a commonly recurring theme in many systems projects. The lesson from prior work is that flexible and fast thread management is difficult to provide.

Kernel threads[Dijkstra 66, Brinch Hansen 70], where the operating system directly supports a thread management interface, suffer from lack of extensibility. Since the thread implementation resides in the kernel, it cannot be specialized or extended by applications. Further, a kernel thread implementation must be sufficiently general purpose in order to support the needs of a diverse range of applications. The lack of extensibility in kernel-threads combined with their general purpose nature has forced applications to seek more flexible and higher performance alternatives.

User-level threads[Bershad et al. 88, Cormack 88] have been suggested as a means of addressing the lack of extensibility in traditional, static thread systems. User-level threads multiplex a number of application threads on top of a single kernel thread. This arrangement, though, suffers from a lack of integration with the operating system. Namely, the kernel is unaware of the amount of application-level concurrency, and may thus make unfavorable decisions such as suspending the kernel thread currently underlying a particular user thread. As a result, user-level thread systems may exhibit incorrect behavior in the face of I/O or asynchronous events such as page faults.

Finally, user-upcall based schemes such as scheduler activations[Anderson et al. 92] and shared threads[Marsh et al. 91] have been proposed as a way of combining the desirable aspects of user and kernel thread systems. In approaches based on user-upcalls, the kernel manages the threads cooperatively with user-level handlers by making upcalls to user space in response to kernel scheduling events. However, this approach suffers from the partitioning of functionality across hardware protection barriers. The services that an application-specific thread handler might need, e.g. memory managers, pagers, device drivers and IPC services, require subsequent system calls back into the kernel. Further, the cost of the upcall itself may be prohibitive. The Firefly implementation of upcalls for scheduler activations, for example, required 2.4 milliseconds per kernel upcall alone, on a roughly 1 MIP machine. An alternative implementation based on the Mach microkernel required up to .915 milliseconds per upcall on a Sequent [Davis et al. 93]. Hence, despite various optimizations aimed at batching requests and avoiding communication by sharing state, user-upcall based have suffered from poor performance due to the frequent crossing of hardware protection boundaries.

2 Strands

To address the shortcomings of kernel-threads (poor extensibility), user-threads (poor integration) and scheduler activations (poor performance), we have designed and implemented a new thread management architecture. Application-specific threads, called *strands*, are the central abstraction of our architecture. A strand is an opaque handle to a kernel resource that represents a thread of control in user-space. Essentially, it is a kernel object that names a user-level thread and provides application supplied implementations for its methods.

The kernel cooperates with application-specific thread management code (*strand packages*) to manage strands. A strand's state and semantics are determined by the application to which the

strand belongs. In particular, the application decides how much state the strand should carry (e.g. CPU registers, FPU state, device status, system call emulation modes, etc.) and how that state ought to be updated in response to system events. The kernel cooperates with strand packages by sending notifications whenever the strand's execution state changes.

The interface by which the kernel communicates with strand packages consists of a set of upcalls. These upcalls invoke user-supplied code, but differ from user-upcall based schemes in that they do not cross hardware boundaries to do so. The kernel invokes application code wherever it would otherwise have to make a policy or implementation decision. Hence, strand packages can have complete control over both the scheduling policies that govern their strands, and the exact responses of their strands to system events. For instance, it is possible to implement fast synchronization primitives based on restartable atomic sequences [Bershad et al. 92] by writing a strand handler that will readjust the user program counter when a thread is preempted within a critical section.

Strand packages execute in the kernel's address space, which enables low-overhead communication between the operating system and application-specific thread managers. Once the user thread management code is placed in the same memory context as the kernel, it can interact with kernel services without crossing hardware protection boundaries. Strand upcalls differ in this respect from scheduler activations, in that they do not require a hardware protection boundary crossing. As a result of being located in the kernel, user code can contact system services and other application-specific code via regular procedure calls and share data by passing references.

Application supplied upcall handlers that execute within the kernel are restricted in their memory accesses. A typesafe language, Modula-3[Nelson 91], restricts applications from issuing illegal reads, writes and jumps. A safe linker prohibits applications from accessing interfaces for which access has not been explicitly granted [Sirer et al. 96]. Combined, a typesafe language and a safe linker provide the same level of isolation for application code in a shared address space as is normally provided by separate hardware based address spaces. Users can of course run unsafe code in separate address spaces, but this code must cross hardware protection domain to access kernel services.

Though unprivileged applications may implement arbitrary semantics for user-level threads, they are not allowed to change the state of kernel threads. Such an ability would allow applications to circumvent system safety guarantees, for instance by modifying kernel register values. The operating system itself uses a thread package implemented in terms of strands for in-kernel threads that is not extensible.

Figure 1 illustrates the overall strands architecture. Multiplexing of the CPU between competing applications is performed by the global scheduler. The kernel threads package provides a protected execution context for threads executing in the kernel. Application specific schedulers and thread packages reside above the global scheduler and communicate via the strand interface.

In the next section, we describe the interface between the operating system and user supplied thread packages, and give examples of its use. The rest of the paper describes our strand implementation in the context of the *SPIN* operating system. Section 4 provides implementation details and Section 5 demonstrates the performance advantages of strands. Section 6 concludes with an overview of our experience.

3 Strand Upcalls

The kernel communicates system events to strand packages through the strand upcall interface. The strand interface forms the bridge between application supplied thread implementations and the rest

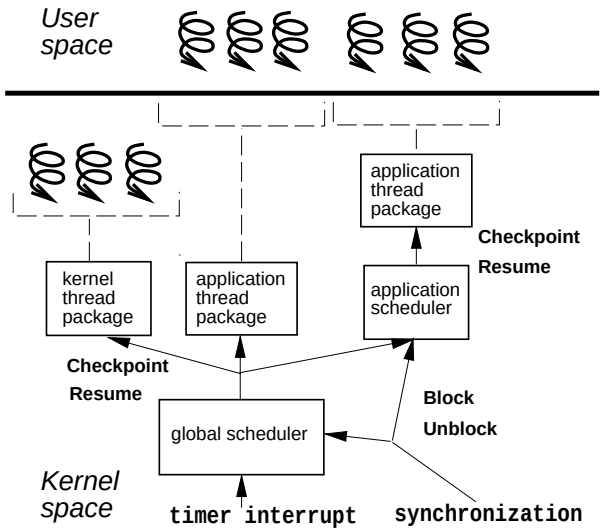

Figure 1: *This figure shows the requisite parts of a strands implementation, and demonstrates the use of the strand interface. Trusted kernel threads are for use within the kernel. Applications implement threads by providing thread packages that handle strand events. Additionally, application-specific scheduling policies may be implemented by installing a specialized scheduler that handles scheduling events. Scheduling and context related events events are propagated through the scheduling hierarchy to thread packages.*

of the operating system services which are oblivious to the particular thread implementation. The interface is minimal, provides machine independence, and enables incremental extensions to thread behavior. It consists of a set of upcalls for tracking the scheduling status of threads and another set for managing their state. Decoupling the state management from scheduling and making it explicit allows clients to extend thread functionality at fine granularity. Applications may provide pieces of a threads package or pieces of a scheduler, or they may implement the entire upcall interface, depending on their needs.

Scheduling:	**Context:**
`PROCEDURE GetCurrent() : Strand.T;`	`PROCEDURE Checkpoint(s: Strand.T);`
Return a handle for the currently executing strand.	*Checkpoint the current state of the strand.*
`PROCEDURE Block(s: Strand.T);`	`PROCEDURE Resume(s: Strand.T);`
The strand is no longer eligible to run.	*Resume strand from last checkpointed state.*
`PROCEDURE Unblock(s: Strand.T);`	
The strand has become eligible to run.	

Table 1: *The strand upcall interface.*

The scheduling related upcalls are used by the system to query application-specific schedulers, and to indicate whether or not an application thread is eligible for execution. The **GetCurrent** upcall is used to acquire a handle to the currently running user thread. This handle identifies an application thread and may be used as an argument to subsequent calls in the strand interface. The **Block** upcall indicates that the named thread is not going to make any progress and should be stopped. For example, the virtual memory subsystem may, in response to a page fault, acquire a handle to the current strand and perform the **Block** upcall to indicate that it is unable to make

progress. A typical scheduler would handle the upcall by stopping this thread and scheduling a new one using a custom scheduling policy. The **Unblock** upcall is the opposite of **Block**. The VM subsystem may perform this upcall when a page is fetched from disk, and a typical scheduler would place the suspended thread back on its run queue. Hence, the strand interface enables subsystems that are oblivious of the thread and scheduler implementations, such as virtual memory, device drivers, input/output, etc., to communicate with application specific schedulers.

The **Checkpoint** and **Resume** upcalls comprise the subset of thread upcalls that are related to execution state. They are used to communicate between schedulers, which decide *when* threads should be scheduled, and thread packages, which determine *how* they should be executed. The **Checkpoint** and **Resume** upcalls are used by schedulers to indicate to thread packages that a strand has been selected to run, and that its state should be loaded onto the machine. For example, a typical thread package would handle the **Resume** upcall by setting the user registers to their appropriate values. An application-specific scheduler that handles the **Block** upcall could in turn raise the **Checkpoint** upcall in order to signal to a thread package that the state of the current computation should be saved.

The *SPIN* operating system provides default handlers for all of the upcalls in the strand interface. The system scheduler handles scheduling related upcalls, while a bootstrap user-thread package handles the state related upcalls. Applications that do not care about application-specific thread behavior do not need to provide handlers for the strand upcalls. Further, applications can extend their behavior on the granularity of single upcalls, and do not need to write whole thread packages or schedulers in order to add incremental functionality.

3.1 Examples

Figure 2 demonstrates how a generic threads package might use the strand interface to implement lightweight user threads. The **Resume** handler is invoked whenever the thread is scheduled to run. It sets the current translation mappings to the thread's address space, and transfers control to the user-level thread. The **Checkpoint** handler saves the thread's state and notifies the address space manager that the user's address space is no longer active. Although both **Checkpoint** and **Resume** are executed with the scheduler lock held, a rogue application cannot keep the system from making forward progress by failing to relinquish control. The EPHEMERAL keyword indicates that the handler may be safely terminated by the caller if it fails to relinquish control within an allotted time frame. Section 4.4 further describes ephemeral handlers and the restrictions placed on them by the language.

A real-time scheduler can be constructed just as readily by installing handlers on strand upcalls. The application-specific real-time scheduler handles **Block** and **Unblock** events for its threads, and performs **Checkpoint** and **Resume** upcalls which are caught by a real-time threads package. Since the real-time scheduler intercepts **Block** and **Unblock** events, it has complete knowledge about the schedulability of its threads, and can make application-specific policy decisions on which thread to run first. For the user-level schedulers to be able to make real-time guarantees, the global scheduler itself has to be real-time. Our current implementation of the global scheduler provides fixed-priorities fair round-robin scheduling within a given priority level.

4 Implementation

We have implemented the strand interface in the context of the *SPIN* kernel. *SPIN* is an extensible operating system which allows untrusted applications to extend system functionality by download-

```
MODULE BasicThreads;
IMPORT Space;
IMPORT CPUState;

TYPE T = Strand.T OBJECT cpustate: CPUState.T; space: Space.T; END;

EPHEMERAL PROCEDURE ResumeHandler(s: Strand.T) =
  VAR th : T := NARROW(s, T);
  BEGIN
    (* switch to user address space *)
    Space.Activate(th.space);
    (* context switch to thread *)
    CPUState.SetUserRegisters(th.cpustate);
  END ResumeHandler;

EPHEMERAL PROCEDURE CheckpointHandler(s: Strand.T) =
  VAR th : T := NARROW(s, T);
  BEGIN
    (* protect the user address space *)
    Space.Deactivate(th.space);
    (* save thread context *)
    CPUState.GetUserRegisters(th.cpustate);
  END CheckpointHandler;

BEGIN
END BasicThreads.
```

Figure 2: *A generic thread package, which illustrates a typical client of the strands interface.*

ing code into the kernel address space. It relies on typesafety and safe dynamic linking in order to retain system wide safety guarantees. *SPIN* runs standalone on Alpha workstations. Some clients of the system include a Unix server, a web server, and an NFS server.

In order to deliver high performance, our strand implementation takes advantage of the extension, integration and protection services offered by *SPIN*. First, we reduce upcall overhead by placing application-specific thread packages in the same address space as the kernel. We further optimize strand upcall dispatch through run-time code generation[Pardyak & Bershad 96]. In order to reduce the latency of user-kernel boundary crossings, we delay scheduling actions until they are necessary. Finally, we expose machine dependent architectural events to applications, and allow applications to handle low-level trap events. Combined, these techniques yield competitive conventional interfaces, such as CThreads, built on strands, as well as deliver high performance to application-specific thread packages.

Although strands were designed and implemented in the context of *SPIN*, we believe that the general architecture could be implemented in more conventional systems such as UNIX provided they support kernel extensions in a safe fashion. Fundamentally, our strands architecture requires that the kernel export two facilities: dynamic linking and in-kernel firewalls. Dynamic linking is necessary to install and remove new strands packages into and from the kernel. An in-kernel firewall makes it possible to run user code within the kernel without compromising system integrity. Typesafe languages are not the only mechanism for implementing firewalls. Alternative strategies include interpreted languages [Gosling & McGilton , Ousterhout 94] and software based fault

isolation techniques [Wahbe et al. 93].

4.1 Low-Overhead Communication

We colocate application-specific thread packages within the kernel to achieve high performance for strand upcalls. Colocation enables the kernel to perform application upcalls without crossing architectural protection boundaries.

The high cost of crossing hardware protection boundaries has been a barrier for thread management schemes based on user-level upcalls. The overhead of crossing protection boundaries has forced previous systems to implement complicated schemes to reduce the number of cross-domain upcalls. For instance, scheduler activations amortizes the cost of an upcall over multiple events by batching them, which trades off latency and simplicity of design for throughput. Psyche and Symunix, on the other hand, share thread state between the kernel and user via mapped memory regions in order to reduce the number of protection boundary crossings. In strands, however, downloading the application thread packages into the kernel enables operating system services to contact user-supplied code through regular procedure calls.

An in-kernel dynamic linker provided by *SPIN* performs the dynamic linking. The linker accepts object code from applications, ensures that it is typesafe, and that it only references kernel interfaces for which it possesses capabilities. Once linked, applications can name kernel services directly and without any overhead. The typesafety of code is ensured by only permitting code written in the safe subset of Modula-3[Nelson 91], which prohibits pointer arithmetic, validates all array accesses and forbids arbitrary pointer casting. The linker validates all accesses to interfaces against a capability list presented by the application, to ensure that no application can access a kernel interface unless authorized to do so. Hence, *SPIN* provides compile and link time facilities to draw firewalls around code in a single privileged address space[Sirer et al. 96].

4.2 Efficient Upcall Dispatch

Application-specific handlers that implement the strand interface need to be bound to the strand upcalls in order to be invoked. Our implementation relies on the *SPIN* event dispatcher to establish this binding. The dispatcher is a manager for control transfer, whose function is to invoke the right set of handlers for a given event. Events in *SPIN* are typed Modula-3 procedures which carry arguments and a return value. When a strand handler registers interest in a kernel upcall, the dispatcher transparently interposes itself between the upcall sites and the handler. Upon an event raise, it evaluates boolean guard expressions to determine which handlers should be executed, and transfers control to the set of handlers that need to run.

The dispatcher utilizes run-time code generation in order to maximize invocation performance. The guard predicates are possibly unrolled and inlined into a run-time generated dispatch stub, which is transparently placed between event handlers and raisers. The stub is responsible for evaluating the guards and invoking the handlers. The *SPIN* dispatcher allows arbitrary interposition of code between event raisers and handlers, and provides a uniform method of extending system functionality within *SPIN*. The guard expressions provide flexibility in specifying arbitrary predicates for when a handler should execute. Although it is possible to achieve the binding functionality offered by the dispatcher through method lookups on objects, our implementation relies on *SPIN* event machinery to take advantage of unlimited interposition.

4.3 User-kernel crossings

Although users normally have full access to the execution state and semantics of a user thread, their access needs to be revoked whenever the user thread crosses the kernel boundary. As discussed in section 2, allowing users to manipulate the state of threads in the kernel could result in system-wide safety problems. The naive way of revoking users' access rights to a user thread is to perform a full context-switch from a user strand to a kernel strand upon kernel entry. However, a full context switch is a relatively costly operation that will impact the performance of traps and system calls, especially in the common case where the thread completes system call processing without performing any scheduling operations.

We optimize the kernel entry path by performing a *deferred context switch*. A user thread is logically paired with a kernel-level thread when it is activated. Upon a subsequent entry into the kernel in response to a system call, fault, or interrupt, the system merely transfers control flow to a kernel stack and establishes a binding between the kernel thread and the user thread that entered the kernel. It defers all scheduling related operations until a scheduling event occurs. For instance, if a user thread enters the kernel and is subsequently blocked, the **Block** upcall for the user strand is delayed until the kernel thread performs a scheduler operation, such as yielding the processor.

This optimization allows us to eliminate redundant scheduler operations in the common case of no scheduling state change during a system call, even though a logical context switch from a user strand to a kernel strand has occurred. *SPIN* is thus able to achieve system call and trap handling performance that is comparable to DEC OSF/1 and Mach.

4.4 Safety and Failure Isolation

In order to guarantee system safety, an implementation of the strand interface needs to provide protection against rogue clients and failure isolation against erroneous handlers.

Our implementation limits thread manipulation to properly authorized handlers via *imposed guards*. An imposed guard is a predicate that ensures that a user handler will not handle events to which the user does not have access capabilities. For instance, a rogue client might try to subvert other users' threads by installing a handler for all strand events, e.g. by specifying a guard that always returns true. However, the installation is not allowed to complete unless the client can present a strand capability to the system. The system then imposes a guard which ensures that the strand argument to the imposed event matches the capability presented by the client.

Further, the strand implementation needs to protect the system from the failures of application-specific handlers. The strand interface is designed such that any invocation of a handler affects at most one strand. This limits the adverse effects of the failure of any handler to the user threads managed by that handler. Failures in handlers are turned into language exceptions by the runtime, which can be caught and appropriately handled [Hsieh et al. 96]. Hence, a program error in an application-specific thread package causes the failure of the corresponding user strand, but does not compromise system integrity.

It is important to note that a failure of a strand handler to relinquish the processor does not cause system failure. Our scheduler and base kernel are preemptive, and a user handler that goes into an infinite loop is dealt with by preemption in the same manner as infinite loops at user level. Sometimes, however, user-supplied handlers need to be invoked with a lock held, and their failure to relinquish a lock may keep other unrelated strands from making progress. For instance, the system scheduler invokes the context related upcalls, **Checkpoint** and **Resume**, with the global scheduler lock held. While this allows atomicity guarantees to be made about the system state changes made by these handlers, it poses a problem should a handler block or go into an

infinite loop. The straightforward solution is to guarantee some grace period to user handlers, as measured by successive clock ticks, and then to terminate the offending handler. However, sudden forceful termination may leave the system in an inconsistent state. For example, a handler that is performing allocation may leave the heap in an inconsistent state if forcefully killed. A rogue client may knowingly invoke an unsuspecting service until it is terminated in order to destroy the internal consistency of this service. We have adopted a language-level solution which allows handlers who agree to a revocation protocol to handle such restricted events, while being effectively prevented from damaging the state of clients that are sensitive to termination. We have added a new subtype of procedures to Modula-3, called EPHEMERAL, which designates that the procedure may be terminated at any time. An ephemeral procedure can only invoke other ephemeral procedures, which are similarly resilient against forceful termination. The strand context upcalls **Checkpoint** and **Resume**, for instance, are declared to be ephemeral, and the system scheduler will forcefully terminate any handlers that do not relinquish the scheduler lock within a small number of clock ticks.

5 Results

In this section, we show that the strands architecture allows applications to construct application-specific thread packages that deliver high-performance. More specifically, we first demonstrate via a set of microbenchmarks that the strand interface has low overhead. This enables conventional thread systems to be constructed out of strands without any performance degradation. In fact, our strands based implementation is a factor of two to ten faster than its equivalent in DEC OSF, even though it incurs a cost for providing extensibility. We then show that the strand architecture enables application-specific thread packages to implement functionality not possible with user threads, and to perform better than alternative approaches based on kernel threads.

We compare the strands performance in the *SPIN* release 1.22 operating system against DEC OSF/1.0 V3.2, a monolithic operating system. The measurements are performed on an AlphaStation 250 4/266Mhz workstation rated at 4.18 SPECint95. The operating systems were run in single-user mode during the course of the measurements.

5.1 Microbenchmarks

The purpose of the microbenchmarks is to highlight specific benefits of the strand system structure. First, we compare the relative performance of protected control transfer between different domains under monolithic systems versus the in-kernel control transfer provided by *SPIN*. In Table 2, *System Call* reflects the time for a user-level program to invoke a system service by crossing the user-kernel boundary and returning. *Protected Procedure Call* reflects a protected service invocation between two kernel extensions in *SPIN*. The low overhead for software based protected domain transfer demonstrates that there is a large incentive to shift application-specific thread management code from user-space to inside the kernel. Further, the table also shows that *SPIN* provides comparable system call performance despite performing a logical context switch on the system call path.

We now compare the performance of generic thread operations under a conventional monolithic implementation model versus a user extension based on the strand interface. We examine Pthreads under DEC OSF, which offers a mature implementation of the POSIX threads interface. Each pthread is is bound to a separate kernel supported thread, which is required for correct concurrent behavior with respect to I/O and page faults. We also measure an in-kernel implementation of CThreads using strands. For the purposes of this experiment, the CThread and Pthread interfaces

	DEC OSF	SPIN
Protected Procedure Call	N/A	0.10
System Call	1.7	2.9

Table 2: *Communication latencies in microseconds. This table shows that there is a large incentive to shift thread management code from user-space to inside the kernel.*

are identical. As Table 3 demonstrates, the overhead of the strand based implementation is substantially lower, which shows that the overhead of extensibility does not interfere with high-performance thread implementations.

Operation	DEC OSF Pthreads	SPIN Strands
Fork-Join	1440	135
Ping-Pong	180	95

Table 3: *Thread management overhead in microseconds. This table demonstrates that it is possible to construct conventional thread systems based on strands that are competitive with hand-tuned, non-extensible implementations.*

5.2 Applications

In this section, we compare the performance of several applications and corresponding application-specific thread packages under different thread models. We compare kernel threads, user threads, and strands. Our results show that the strand architecture can deliver high-performance thread subsystems, due to two factors: (1) specialization, where no more code than is strictly necessary for the application's task is executed, and (2) protection, where low overhead software protection schemes obviate costly hardware protection boundary crossings.

For kernel threads, we report measurements made on DEC OSF PThreads. For user threads, we report numbers from DEC OSF MiniThreads, which is a purely user-level thread package that provides a similar interface to that of Pthreads. Finally, we examine the SPIN CThread extension.

	Kernel threads	User threads	Strands
Application	DEC OSF Pthreads	DEC OSF MiniThreads	SPIN CThreads
MemRegions	N/A	2950	1540
CAS2Synch	309	309	64

Table 4: *Application execution time in seconds. This table demonstrates that the strands architecture offers high-performance for applications which need to interface closely with kernel services.*

The first benchmark, MemRegions, is a local DSM system simulator, where each thread has separate memory regions that are protected to track updates. Upon every context switch, the memory mapping is updated to contain the pages in the thread's working set (i.e. pages it has locked) and to protect the pages that the thread does not have access to. Since kernel thread implementations are both immutable and do not expose preemptions and other scheduling events to the user, implementing this behavior with kernel threads is not possible. A strands based

implementation is a factor of two faster than a user-level thread implementation, due largely to the low communication overhead between the thread package and the VM system.

CAS2Synch is a multithreaded application that synchronizes through double compare and swap (CAS2) operations. When used to synchronize with threads in other address spaces via shared memory, the kernel and user thread based implementations both require the same sequence of heavyweight synchronization mechanisms provided by the architecture. In this instance, the kernel and user thread implementations both need to use load-linked/store-conditional primitives exported by the hardware. However, the strands interface allows its clients to take advantage of their application-specific knowledge by exposing all scheduling related events. For instance, on a single processor, it is possible to implement CAS2 by executing application-specific code on each thread preemption that checks the pc and either resets it to the beginning of the CAS region if no writes have been performed, or rolls it forward from within the context save routine [Bershad 93]. This optimistic software implementation of CAS is roughly five times faster than the equivalent implemented with hardware primitives.

Acknowledgements

We would like to thank all members of the *SPIN* group, in particular Marc Fiuczynski and Stefan Savage, for their assistance during the course of this work.

6 Conclusion

In this paper, we have described an architecture for building extensible, efficient and robust thread implementations. For conventional thread interfaces, the strand architecture is able to provide high performance competitive with hand-tuned, monolithic implementations. In addition, the strand interface allows applications to build specialized thread implementations that are either hard to construct or inefficient under alternative concurrency models.

References

[Accetta et al. 86] Accetta, M. J., Baron, R. V., Bolosky, W., Golub, D. B., Rashid, R. F., Tevanian, Jr., A., and Young, M. W. Mach: A New Kernel Foundation for Unix Development. In *Proceedings of the 1986 Summer USENIX Conference*, pages 93–113, July 1986.

[Anderson et al. 92] Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[Bershad 93] Bershad, B. N. Practical Considerations for Non-Blocking Concurrent Objects. In *Proceedings of the Thirteenth International Conference on Distributed Computing Systems*, pages 264–274, Pittsburgh, PA, May 1993.

[Bershad et al. 88] Bershad, B., Lazowska, E., and Levy, H. Presto: A system for object-oriented parallel programming. *Software: Practice and Experience*, 18(8), August 1988.

[Bershad et al. 92] Bershad, B. N., Redell, D. D., and Ellis, J. R. Fast Mutual Exclusion for Uniprocessors. In *Proceedings of the Fifth International Conference on Architectural*

Support for Programming Languages and Operating Systems (ASPLOS-V), pages 223–233, Boston, MA, October 1992.

[Bershad et al. 95] Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M., Becker, D., Eggers, S., and Chambers, C. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, December 1995.

[Brinch Hansen 70] Brinch Hansen, P. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4):238–250, April 1970.

[Cooper & Draves 88] Cooper, E. C. and Draves, R. P. C Threads. Technical Report CMU-CS-88-154, Carnegie Mellon University, June 1988.

[Cormack 88] Cormack, G. V. A Microkernel for Concurrency in C. *Software—Practice and Experience*, 18(5):485–491, May 1988.

[Davis et al. 93] Davis, P.-B., McNamee, D., Vaswani, R., and Lazowska, E. Adding Scheduler Activations to Mach 3.0. In *Proceedings of the Third USENIX Mach Symposium*, pages 119–136, Santa Fe, NM, April 1993.

[Dijkstra 66] Dijkstra, E. W. The Structure of the THE Multiprogramming System. *Communications of the ACM*, 9(3):341–346, March 1966.

[Gosling & McGilton] Gosling, J. and McGilton, H. The Java Language Environment: A White Paper. http://java.sun.com.

[Hsieh et al. 96] Hsieh, W. C., Fiuczynski, M. E., Garrett, C., Savage, S., Becker, D., and Bershad, B. N. Language Support for Extensible Systems. In *First Annual Workshop on Compiler Support for System Software*, January 1996.

[Marsh et al. 91] Marsh, B., Scott, M., LeBlanc, T., and Markatos, E. First-Class User-Level Threads. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 110–121, Pacific Grove, CA, October 1991.

[Nelson 91] Nelson, G., editor. *Systems Programming with Modula-3.* Prentice Hall, 1991.

[Ousterhout 94] Ousterhout, J. K. *Tcl and the TK Toolkit.* Addison-Wesley Publishing Company, 1994.

[Pardyak & Bershad 96] Pardyak, P. and Bershad, B. Dynamic Binding for Extensible Systems. Submitted to 1996 OSDI, May 1996.

[Redell 88] Redell, D. Experience with Topaz Teledebugging. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, October 1988.

[Ritchie & Thompson 74] Ritchie, D. M. and Thompson, K. The UNIX Time-Sharing System. *Communications of the ACM*, 17(6):365–375, July 1974.

[Sirer et al. 96] Sirer, E. G., Fiuczynski, M., Pardyak, P., and Bershad, B. N. Safe Dynamic Linking in an Extensible Operating System. In *First Annual Workshop on Compiler Support for System Software*, January 1996.

[Wahbe et al. 93] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient Software-Based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, NC, December 1993.

[Zahorjan & McCann 90] Zahorjan, J. and McCann, C. Processor Scheduling in Shared Memory Multiprocessors. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 214–225, May 1990.