# User-Level Interprocess Communication for Shared Memory Multiprocessors

BRIAN N. BERSHAD
Carnegie Mellon University

THOMAS E. ANDERSON, EDWARD D. LAZOWSKA, and HENRY M. LEVY
University of Washington

Interprocess communication (IPC), in particular IPC oriented towards *local communication* (between address spaces on the same machine), has become central to the design of contemporary operating systems. IPC has traditionally been the responsibility of the kernel, but kernel-based IPC has two inherent problems First, its performance is architecturally limited by the cost of invoking the kernel and reallocating a processor from one address space to another. Second, applications that need inexpensive threads and must provide their own thread management encounter functional and performance problems stemming from the interaction between kernel-level communication and user-level thread management.

On a shared memory multiprocessor, these problems can be solved by moving the communication facilities out of the kernel and supporting them at the user level within each address space. Communication performance is improved since kernel invocation and processor reallocation can be avoided when communicating between address spaces on the same machine.

These observations motivated User-Level Remote Procedure Call (URPC). URPC combines a fast cross-address space communication protocol using shared memory with lightweight threads managed at the user level. This structure allows the kernel to be bypassed during cross-address space communication. The programmer sees threads and RPC through a conventional interface, though with unconventional performance.

*Index Terms*—thread, multiprocessor, operating system, parallel programming, performance, communication

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*modules, packages*; D.4.1 [**Operating Systems**]: Process Management—*concurrency, multiprocessing / multiprogramming*; D.4.4 [**Operating Systems**]: Communications Management; D.4.6 [**Operating Systems**]: Security and Protection—*access controls, information flow controls*; D.4.7 [**Operating Systems**]: Organization and Design; D.4.8 [**Operating Systems**]: Performance—*measurements*

This material is based on work supported by the National Science Foundation (grants CCR-8619663, CCR-8700106, CCR-8703049, and CCR-897666), the Washington Technology Center, and Digital Equipment Corporation (the Systems Research Center and the External Research Program). Bershad was supported by an AT&T Ph.D. Scholarship, and Anderson by an IBM Graduate Fellowship.

Authors' addresses: B. N. Bershad, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213; T. E. Anderson, E. D. Lazowska, and H. M. Levy, Department of Computer Science and Engineering, FR-35, University of Washington, Seattle, WA 98195.

General Terms: Design, Performance, Measurement

Additional Key Words and Phrases: Modularity, remote procedure call, small-kernel operating system

## 1. INTRODUCTION

Efficient interprocess communication is central to the design of contemporary operating systems [16, 23]. An efficient communication facility encourages system decomposition across address space boundaries. Decomposed systems have several advantages over more monolithic ones, including failure isolation (address space boundaries prevent a fault in one module from "leaking" into another), extensibility (new modules can be added to the system without having to modify existing ones), and modularity (interfaces are enforced by mechanism rather than by convention).

Although address spaces can be a useful structuring device, the extent to which they can be used depends on the performance of the communication primitives. If cross-address space communication is slow, the structuring benefits that come from decomposition are difficult to justify to end users, whose primary concern is system performance, and who treat the entire operating system as a "black box" [18] regardless of its internal structure. Consequently, designers are forced to coalesce weakly related subsystems into the same address space, trading away failure isolation, extensibility, and modularity for performance.

Interprocess communication has traditionally been the responsibility of the operating system kernel. However, kernel-based communication has two problems:

—*Architectural performance barriers.* The performance of kernel-based synchronous communication is architecturally limited by the cost of invoking the kernel and reallocating a processor from one address space to another. In our earlier work on Lightweight Remote Procedure Call (LRPC) [10], we show that it is possible to reduce the overhead of a kernel-mediated cross-address space call to nearly the limit possible on a conventional processor architecture: the time to perform a cross-address LRPC is only slightly greater than that required to twice invoke the kernel and have it reallocate a processor from one address space to another. The efficiency of LRPC comes from taking a "common case" approach to communication, thereby avoiding unnecessary synchronization, kernel-level thread management, and data copying for calls between address spaces on the same machine. The majority of LRPC's overhead (70 percent) can be attributed directly to the fact that the kernel mediates every cross-address space call.

—*Interaction between kernel-based communication and high-performance user-level threads.* The performance of a parallel application running on a multiprocessor can strongly depend on the efficiency of thread management operations. Medium and fine-grained parallel applications must use a thread management system implemented at the user level to obtain satisfactory performance [6, 36]. Communication and thread management have strong interdependencies, though, and the cost of partitioning them across protection boundaries (kernel-level communication and user-level thread management) is high in terms of performance and system complexity.

On a shared memory multiprocessor, these problems have a solution: eliminate the kernel from the path of cross-address space communication. Because address spaces can share memory directly, shared memory can be used as the data transfer channel. Because a shared memory multiprocessor has more than one processor, processor reallocation can often be avoided by taking advantage of a processor already active in the target address space without involving the kernel.

User-level management of cross-address space communication results in improved performance over kernel-mediated approaches because

—Messages are sent between address spaces directly, without invoking the kernel.

—Unnecessary processor reallocation between address spaces is eliminated, reducing call overhead, and helping to preserve valuable cache and TLB contexts across calls.

—When processor reallocation does prove to be necessary, its overhead can be amortized over several independent calls.

—The inherent parallelism in the sending and receiving of a message can be exploited, thereby improving call performance.

User-Level Remote Procedure Call (URPC), the system described in this paper, provides safe and efficient communication between address spaces on the same machine without kernel mediation. URPC isolates from one another the three components of interprocess communication: processor reallocation, thread management, and data transfer. Control transfer between address spaces, which is the communication abstraction presented to the programmer, is implemented through a combination of thread management and processor reallocation. Only processor reallocation requires kernel involvement; thread management and data transfer do not. Thread management and interprocess communication are done by application-level libraries, rather than by the kernel.

URPC's approach stands in contrast to conventional "small kernel" systems in which the kernel is responsible for address spaces, thread management, and interprocess communication. Moving communication and thread management to the user level leaves the kernel responsible only for the mechanisms that allocate processors to address spaces. For reasons of performance and flexibility, this is an appropriate division of responsibility for operating systems of shared memory multiprocessors. (URPC can also be appropriate for uniprocessors running multithreaded applications.)

The latency of a simple cross-address space procedure call is 93 $\mu$secs using a URPC implementation running on the Firefly, DEC SRC's multiprocessor workstation [37]. Operating as a pipeline, two processors can complete one call every 53 μsecs. On the same multiprocessor hardware, RPC facilities that involve the kernel are markedly slower, without providing adequate support for user-level thread management. LRPC, a high performance-kernel-based implementation, takes 157 μsecs. The Fork operation for the user-level threads that accompany URPC takes 43 μsecs, while the Fork operation for the kernel-level threads that accompany LRPC take over a millisecond. To put these figures in perspective, a same-address space procedure call takes 7 μsecs on the Firefly, and a protected kernel invocation (trap) takes 20 μsecs.

We describe the mechanics of URPC in more detail in the next section. In Section 3 we discuss the design rationale behind URPC. We discuss performance in Section 4. In Section 5 we survey related work. Finally, in Section 6 we present our conclusions.

## 2. A USER-LEVEL REMOTE PROCEDURE CALL FACILITY

In many contemporary uniprocessor and multiprocessor operating systems, applications communicate with operating system services by sending messages across narrow channels or *ports* that support only a small number of operations (create, send, receive, destroy). Messages permit communication between programs on the same machine, separated by address space boundaries, or between programs on different machines, separated by physical boundaries.

Although messages are a powerful communication primitive, they represent a control and data-structuring device foreign to traditional Algol-like languages that support synchronous procedure call, data typing, and shared memory for communication within an address space. Communication between address spaces is with untyped, asynchronous messages. Programmers of message-passing systems who must use one of the many popular Algol-like languages as a systems-building platform must think, program, and structure according to two quite different programming paradigms. Consequently, almost every mature message-based operating system also includes support for Remote Procedure Call (RPC) [11], enabling messages to serve as the underlying transport mechanism beneath a procedure call interface.

Nelson defines RPC as the synchronous language-level transfer of control between programs in disjoint address spaces whose primary communication medium is a narrow channel [30]. The definition of RPC is silent about the operation of that narrow channel and how the processor scheduling (reallocation) mechanisms interact with data transfer. URPC exploits this silence in two ways:

—Messages are passed between address spaces through logical channels kept in memory that is pair-wise shared between the client and server. The channels are created and mapped once for every client/server pairing, and are used for all subsequent communication between the two address spaces so that several interfaces can be multiplexed over the same channel. The integrity of data passed through the shared memory channel is ensured by a combination of the pair-wise mapping (message authentication is implied statically) and the URPC runtime system in each address space (message correctness is verified dynamically).

—Thread management is implemented entirely at the user level and is integrated with the user-level machinery that manages the message channels. A user-level thread sends a message to another address space by directly enqueuing the message on the outgoing link of the appropriate channel. No kernel calls are necessary to send a call or reply message.

Although a cross-address space procedure call is synchronous from the perspective of the programmer, it is asynchronous at and beneath the level of thread management. When a thread in a client invokes a procedure in a server, it blocks waiting for the reply signifying the procedure's return; while blocked, its processor can run another ready thread in the same address space. In our earlier system, LRPC, the blocked thread and the ready thread were really the same; the thread just crosses an address space boundary. In contrast, URPC *always* tries to schedule another thread from the same address space on the client thread's processor. This is a scheduling operation that can be handled entirely by the user-level thread management system. When the reply arrives, the blocked client thread can be rescheduled on any of the processors allocated to the client's address space, again without kernel intervention. Similarly, execution of the call on the server side can be done by a processor already executing in the context of the server's address space, and need not occur synchronously with the call.

By preferentially scheduling threads within the same address space, URPC takes advantage of the fact that there is significantly less overhead involved in switching a processor to another thread in the same address space (we will call this *context switching*) than in reallocating it to a thread in another address space (we will call this *processor reallocation*). Processor reallocation requires changing the mapping registers that define the virtual address space context in which a processor is executing. On conventional processor architectures, these mapping registers are protected and can only be accessed in privileged kernel mode.

Several components contribute to the high overhead of processor reallocation. There are scheduling costs to decide the address space to which a processor should be reallocated; immediate costs to update the virtual memory mapping registers and to transfer the processor between address spaces; and long-term costs due to the diminished performance of the cache and translation lookaside buffer (TLB) that occurs when locality shifts from one address space to another [3]. Although there is a long-term cost associated with context switching within the same address space, that cost is less than when processors are frequently reallocated between address spaces [20].

To demonstrate the relative overhead involved in switching contexts between two threads in the same address space versus reallocating processors between address spaces, we introduce the concept of a *minimal latency same-address space context switch*. On the C-VAX, the minimal latency same-address space context switch requires saving and then restoring the machine's general-purpose registers, and takes about 15 $\mu$secs. In contrast, reallocating the processor from one address space to another on the C-VAX takes about 55 $\mu$secs *without* including the long-term cost.

Because of the cost difference between context switching and processor reallocation, URPC strives to avoid processor reallocation, instead context switching whenever possible.

Processor reallocation is sometimes necessary with URPC, though. If a client invokes a procedure in a server that has an insufficient number of processors to handle the call (e.g., the server's processors are busy doing other work), the client's processors may idle for some time awaiting the reply. This is a load-balancing problem. URPC considers an address space with pending incoming messages on its channel to be "underpowered." An idle processor in the client address space can balance the load by reallocating itself to a server that has pending (unprocessed) messages from that client. Processor reallocation requires that the kernel be invoked to change the processor's virtual memory context to that of the underpowered address space. That done, the kernel upcalls into a server routine that handles the client's outstanding requests. After these have been processed, the processor is returned to the client address space via the kernel.

The responsibility for detecting incoming messages and scheduling threads to handle them belongs to special, low-priority threads that are part of a URPC runtime library linked into each address space. Processors scan for incoming messages only when they would otherwise be idle.

Figure 1 illustrates a sample execution timeline for URPC with time proceeding downward. One client, an editor, and two servers, a window manager (WinMgr) and a file cache manager (FCMgr), are shown. Each is in a separate address space. Two threads in the editor, $T_1$ and $T_2$, invoke procedures in the window manager and the file cache manager. Initially, the editor has one processor allocated to it, the window manager has one, and the file cache manager has none. $T_1$ first makes a cross-address space call into the window manager by sending and then attempting to receive a message. The window manager receives the message and begins processing. In the meantime, the thread scheduler in the editor has context switched from $T_1$ (which is blocked waiting to receive a message) to another thread $T_2$. Thread $T_2$ initiates a procedure call to the file cache manager by sending a message and waiting to receive a reply. $T_1$ can be unblocked because the window manager has sent a reply message back to the editor. Thread $T_1$ then calls into the file cache manager and blocks. The file cache manager now has two pending messages from the editor but no processors with which to handle them. Threads $T_1$ and $T_2$ are blocked in the editor, which has no other threads waiting to run. The editor's thread management system detects the outgoing pending messages and donates a processor to the file cache manager, which can then receive, process, and reply to the editor's two incoming calls before returning the processor back to the editor. At this point, the two incoming reply messages from the file cache manager can be handled. $T_1$ and $T_2$ each terminate when they receive their replies.

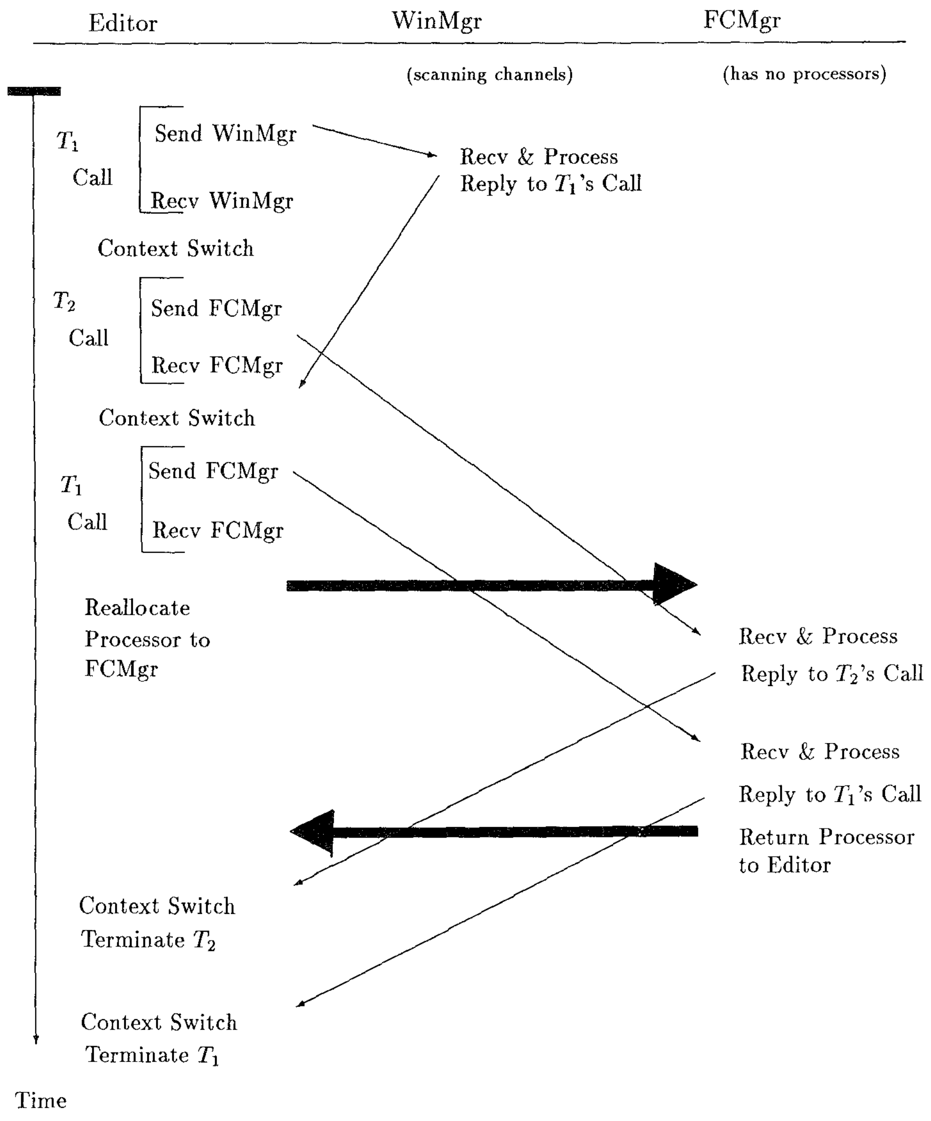

Fig. 1. URPC timeline.

URPC's treatment of pending messages is analogous to the treatment of runnable threads waiting on a ready queue. Each represents an execution context in need of attention from a physical processor; otherwise idle processors poll the queues looking for work in the form of these execution contexts. There are two main differences between the operation of message channels and thread ready queues. First, URPC enables on address space to spawn work in another address space by enqueuing a message (an execution context consisting of some control information and arguments or results). Second,

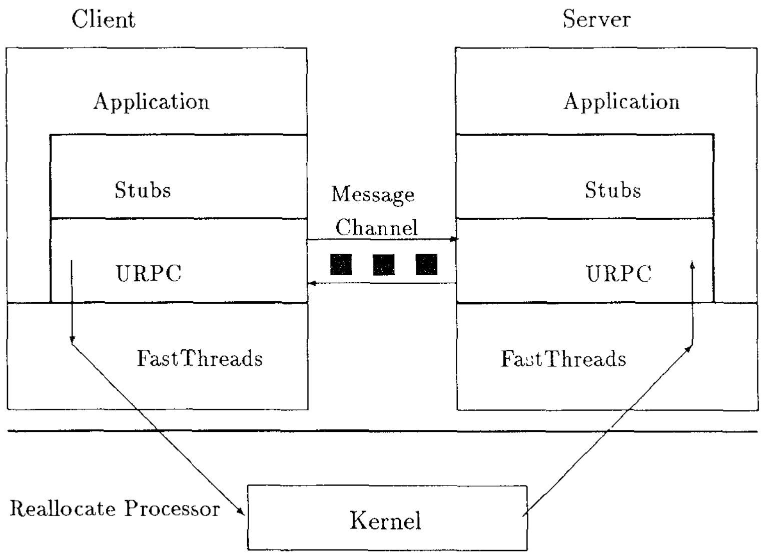

Fig. 2 The software components of URPC.

URPC enables processors in one address space to execute in the context of another through an explicit reallocation if the workload between them is imbalanced.

### 2.1 The User View

It is important to emphasize that all of the URPC mechanisms described in this section exist "under the covers" of quite ordinary looking Modula2 + [34] interfaces. The RPC paradigm provides the freedom to implement the control and data transfer mechanisms in ways that are best matched by the underlying hardware.

URPC consists of two software packages used by stubs and application code. One package, called *FastThreads* [5], provides lightweight threads that are managed at the user level and scheduled on top of middleweight kernel threads. The second package, called *URPC*, implements the channel management and message primitives described in this section. The *URPC* package lies directly beneath the stubs and closely interacts with *FastThreads*, as shown in Figure 2.

## 3. RATIONALE FOR THE URPC DESIGN

In this section we discuss the design rationale behind URPC. In brief, this rationale is based on the observation that there are several independent components to a cross-address space call and each can be implemented separately. The main components are the following:

— *Thread management*: blocking the caller's thread, running a thread through the procedure in the server's address space, and resuming the caller's thread on return,

—*Data transfer*: moving arguments between the client and server address spaces, and

—*Processor reallocation*: ensuring that there is a physical processor to handle the client's call in the server and the server's reply in the client.

In conventional message systems, these three components are combined beneath a kernel interface, leaving the kernel responsible for each. However, thread management and data transfer do not require kernel assistance, only processor reallocation does. In the three subsections that follow we describe how the components of a cross-address space call can be isolated from one another, and the benefits that arise from such a separation.

### 3.1 Processor Reallocation

URPC attempts to reduce the frequency with which processor reallocation occurs through the use of an *optimistic reallocation policy.* At call time, URPC optimistically assumes the following:

—The client has other work to do, in the form of runnable threads or incoming messages, and a potential delay in the processing of a call will not have serious effect on the performance of other threads in the client's address space.

—The server has, or will soon have, a processor with which it can service a message.

In terms of performance, URPC's optimistic assumptions can pay off in several ways. The first assumption makes it possible to do an inexpensive context switch between user-level threads during the blocking phase of a cross-address space call. The second assumption enables a URPC to execute in parallel with threads in the client's address space while avoiding a processor reallocation. An implication of both assumptions is that it is possible to amortize the cost of a single processor reallocation across several outstanding calls between the same address spaces. For example, if two threads in the same client make calls into the same server in succession, then a single processor reallocation can handle both.

A user-level approach to communication and thread management is appropriate for shared memory multiprocessors where applications rely on aggressive multithreading to exploit parallelism while at the same time compensating for multiprogramming effects and memory latencies [2], where a few key operating system services are the target of the majority of all application calls [8], and where operating system functions are affixed to specific processing nodes for the sake of locality [31].

In contrast to URPC, contemporary uniprocessor kernel structures are not well suited for use on shared memory multiprocessors:

—Kernel-level communication and thread management facilities rely on pessimistic processor reallocation policies, and are unable to exploit concurrency within an application to reduce the overhead of communication. *Handoff scheduling* [13] underscores this pessimism: a single kernel operation blocks the client and reallocates its processor directly to the server.

Although handoff scheduling does improve performance, the improvement is limited by the cost of kernel invocation and processor reallocation.

—In a traditional operating system kernel designed for a uniprocessor, but running on a multiprocessor, kernel resources are logically centralized, but distributed over the many processors in the system. For a medium- to large-scale shared memory multiprocessor such as the Butterfly [7], Alewife [4], or DASH [25], URPC's user-level orientation to operating system design localizes system resources to those processors where the resources are in use, relaxing the performance bottleneck that comes from relying on centralized kernel data structures. This bottleneck is due to the contention for logical resources (locks) and physical resources (memory and interconnect bandwidth) that results when a few data structures, such as thread run queues and message channels, are shared by a large number of processors. By distributing the communication and thread management functions to the address spaces (and hence processors) that use them directly, contention is reduced.

URPC can also be appropriate on uniprocessors running multithreaded applications where inexpensive threads can be used to express the logical and physical concurrency within a problem. Low overhead threads and communication make it possible to overlap even small amounts of external computation. Further, multithreaded applications that are able to benefit from delayed reallocation can do so without having to develop their own communication protocols [19]. Although reallocation will eventually be necessary on a uniprocessor, it can be delayed by scheduling within an address space for as long as possible.

3.1.1 *The Optimistic Assumptions Won't Always Hold.* In cases where the optimistic assumptions do not hold, it is necessary to invoke the kernel to force a processor reallocation from one address space to another. Examples of where it is inappropriate to rely on URPC's optimistic processor reallocation policy are single-threaded applications, real-time applications (where call latency must be bounded), high-latency I/O operations (where it is best to initiate the I/O operation early since it will take a long time to complete), and priority invocations (where the thread executing the cross-address space call is of high priority). To handle these situations, URPC allows the client's address space to force a processor reallocation to the server's, even though there might still be runnable threads in the client's.

3.1.2 *The Kernel Handles Processor Reallocation.* The kernel implements the mechanism that support processor reallocation. When an idle processor discovers an underpowered address space, it invokes a procedure called *Processor. Donate*, passing in the identity of the address space to which the processor (on which the invoking thread is running) should be reallocated. *Processor.Donate* transfers control down through the kernel and then up to a prespecified address in the receiving space. The identity of the donating address space is made known to the receiver by the kernel.

3.1.3 *Voluntary Return of Processors Cannot Be Guaranteed.* A service interface defines a contract between a client and a server. In the case of URPC, as with traditional RPC systems, implicit in the contract is that the server obey the policies that determine when a processor is to be returned back to the client. The URPC communication library implements the following policy in the server: upon receipt of a processor from a client address space, return the processor when all outstanding messages from the client have generated replies, or when the server determines that the client has become "underpowered" (there are outstanding messages back to the client, and one of the server's processors is idle).

Although URPC's runtime libraries implement a specific protocol, there is no way to enforce that protocol. Just as with kernel-based systems, once the kernel transfers control of the processor to an application, there is no guarantee that the application will voluntarily return the processor by returning from the procedure that the client invoked. The server could, for example, use the processor to handle requests from other clients, even though this was not what the client had intended.

It is necessary to ensure that applications receive a fair share of the available processing power. URPC's direct reallocation deals only with the problem of load balancing between applications that are communicating with one another. Independent of communication, a multiprogrammed system requires policies and mechanisms that balance the load between noncommunicating (or noncooperative) address spaces. Applications, for example, must not be able to starve one another out for processors, and servers must not be able to delay clients indefinitely by not returning processors. Preemptive policies, which forcibly reallocate processors from one address space to another, are therefore necessary to ensure that applications make progress.

A processor reallocation policy should be work conserving, in that no high-priority thread waits for a processor while a lower priority thread runs, and, by implication, that no processor idles when there is work for it to do anywhere in the system, even if the work is in another address space. The specifics of how to enforce this constraint in a system with user-level threads are beyond the scope of this paper, but are discussed by Anderson et al. in [6].

The direct processor reallocation done in URPC can be thought of as an optimization of a work-conserving policy. A processor idling in the address space of a URPC client can determine which address spaces are not responding to that client's calls, and therefore which address spaces are, from the standpoint of the client, the most eligible to receive a processor. There is no reason for the client to first voluntarily relinquish the processor to a global processor allocator, only to have the allocator then determine to which address space the processor should be reallocated. This is a decision that can be easily made by the client itself.

## 3.2 Data Transfer Using Shared Memory

The requirements of a communication system can be relaxed and its efficiency improved when it is layered beneath a procedure call interface rather than exposed to programmers directly. In particular, arguments that are part of a cross-address space procedure call can be passed using shared memory while still guaranteeing safe interaction between mutually suspicious subsystems.

Shared memory message channels do not increase the "abusability factor" of client-server interactions. As with traditional RPC, clients and servers can still overload one another, deny service, provide bogus results, and violate communication protocols (e.g., fail to release channel locks, or corrupt channel data structures). And, as with traditional RPC, it is up to higher level protocols to ensure that lower level abuses filter up to the application layer in a well-defined manner (e.g., by raising a *call-failed* exception or by closing down the channel).

In URPC, the safety of communication based on shared memory is the responsibility of the stubs. The arguments of a URPC are passed in message buffers that are allocated and pair-wise mapped during the binding phase that precedes the first cross-address space call between a client and server. On receipt of a message, the stubs unmarshal the message's data into procedure parameters doing whatever copying and checking is necessary to ensure the application's safety.

Traditionally, safe communication is implemented by having the kernel copy data from one address space to another. Such an implementation is necessary when application programmers deal directly with raw data in the form of messages. But, when standard runtime facilities and stubs are used, as is the case with URPC, having the kernel copy data between address spaces is neither necessary nor sufficient to guarantee safety.

Copying in the kernel is not necessary because programming languages are implemented to pass parameters between procedures on the stack, the heap, or in registers. When data is passed between address spaces, none of these storage areas can, in general, be used directly by both the client and the server. Therefore, the stubs must copy the data into and out of the memory used to move arguments between address spaces. Safety is not increased by first doing an extra kernel-level copy of the data.

Copying in the kernel is not sufficient for ensuring safety when using type-safe languages such as Modula2+ or Ada [1] since each actual parameter must be checked by the stubs for conformity with the type of its corresponding formal. Without such checking, for example, a client could crash the server by passing in an illegal (e.g., out of range) value for a parameter.

These points motivate the use of pair-wise shared memory for cross-address space communication. Pair-wise shared memory can be used to transfer data between address spaces more efficiently, but just as safely, as messages that are copied by the kernel between address spaces.

3.2.1 *Controlling Channel Access.* Data flows between URPC packages in different address spaces over a bidirectional shared memory queue with test-and-set locks on either end. To prevent processors from waiting indefinitely on message channels, the locks are nonspinning; i.e., the lock protocol is simply *if the lock is free, acquire it, or else go on to something else–never spin-wait.* The rationale here is that the receiver of a message should never have to delay while the sender holds the lock. If the sender stands to benefit from the receipt of a message (as on call), then it is up to the sender to ensure that locks are not held indefinitely; if the receiver stands to benefit (as on return), then the sender could just as easily prevent the benefits from being realized by not sending the message in the first place.

## 3.3 Cross-Address Space Procedure Call and Thread Management

The calling semantics of cross-address space procedure call, like those of normal procedure call, are synchronous with respect to the thread that is performing the call. Consequently, there is a strong interaction between the software system that manages threads and the one that manages cross-address space communication. Each underlying communication function (*send* and *receive*) involves a corresponding thread management synchronization function (*start* and *stop*). On call, the client *sends* a message to the server, which *starts* a server thread. The client thread *stops*, waiting on a *receive* from the server. When the server finishes, it *sends* a response back to the client, which causes the client's waiting thread to be *started*. Finally, the server's thread *stops* again on a *receive*, waiting for the next message.

In this subsection we explore the relationship between cross-address space communication and thread management. In brief, we argue the following:

—High performance thread management facilities are necessary for fine-grained parallel programs.

—While thread management facilities can be provided either in the kernel or at the user level, *high-performance* thread management facilities can only be provided at the user level.

—The close interaction between communication and thread management can be exploited to achieve extremely good performance for both, when both are implemented together at the user level.

3.3.1 *Concurrent Programming and Thread Management.* Multiple threads within an address space simplify the management of concurrent activities. Concurrency can be entirely internal to the application, as with parallel programs for multiprocessors, or it can be external, as with an application that needs to overlap some amount of its own computation with activity on its behalf in another address space. In either case, the programmer needs access to a thread management system that makes it possible to create, schedule, and destroy threads of control.

Support for thread management can be described in terms of a cost continuum on which there are three major points of reference: heavyweight, middleweight, and lightweight, for which thread management overheads are on the order of thousands, hundreds, and tens of $\mu$secs, respectively.

Kernels supporting heavyweight threads [26, 32, 35] make no distinction between a thread, the dynamic component of a program, and its address space, the static component. Threads and address spaces are created, scheduled, and destroyed as single entities. Because an address space has a large amount of baggage, such as open file descriptors, page tables, and accounting state that must be manipulated during thread management operations, operations on heavyweight threads are costly, taking tens of milliseconds.

Many contemporary operating system kernels provide support for middleweight, or *kernel-level* threads [12, 36]. In contrast to heavyweight threads, address spaces and middleweight threads are decoupled, so that a single address space can have many middleweight threads. As with heavyweight threads, though, the kernel is responsible for implementing the thread management functions, and directly schedules an application's threads on the available physical processors. With middleweight threads, the kernel defines a general programming interface for use by all concurrent applications.

Unfortunately, the cost of this generality affects the performance of fine-grained parallel applications. For example, simple features such as time slicing, or saving and restoring floating point registers when switching between threads contexts can be sources of performance degradation for all applications, even those for which the features are unnecessary. A kernel-level thread management system must also protect itself from malfunctioning or malfeasant programs. Expensive (relative to procedure call) protected kernel traps are required to invoke any thread management operation, and the kernel must treat each invocation suspiciously, checking arguments and access privileges to ensure legitimacy.

In addition to the direct overheads that contribute to the high cost of kernel-level thread management, there is an indirect but more pervasive overhead stemming from the effect of thread management *policy* on program performance. There are a large number of parallel programming models, and within these, a wide variety of scheduling disciplines that are most appropriate (for performance) to a given model [25, 33]. Performance, though, is strongly influenced by the choice of interfaces, data structures, and algorithms used to implement threads, so a single model represented by one style of kernel-level thread is unlikely to have an implementation that is efficient for all parallel programs.

In response to the costs of kernel-level threads, programmers have turned to lightweight threads that execute in the context of middleweight or heavyweight threads provided by the kernel, but are managed at the user level by a library linked in with each application [9, 38]. Lightweight thread management implies *two-level scheduling*, since the application library schedules lightweight threads on top of weightier threads, which are themselves being scheduled by the kernel.

Two-level schedulers attack both the direct and indirect costs of kernel threads. Directly, it is possible to implement efficient user-level thread management functions because they are accessible through a simple procedure call rather than a trap, need not be bulletproofed against application errors, and can be customized to provide only the level of support needed by a given application. The cost of thread management operations in a user-level scheduler can be orders of magnitude less than for kernel-level threads.

3.3.2 *The Problem with Having Functions at Two Levels.* Threads block for two reasons: (1) to synchronize their activities within an address space (e.g., while controlling access to critical sections), and (2) to wait for external events in other address spaces (e.g., while reading keystrokes from a window manager). If threads are implemented at the user level (necessary for inexpensive and flexible concurrency), but communication is implemented in the kernel, then the performance benefits of inexpensive scheduling and context switching are lost when synchronizing between activities in separate address spaces. In effect, each synchronizing operation must be implemented and executed at two levels: (1) once in the application, so that the user-level scheduler accurately reflects the scheduling state of the application, and then again (2) in the kernel, so that applications awaiting kernel-mediated communication activity are properly notified.

In contrast, when communication and thread management are moved out of the kernel and implemented together at user level, the synchronization needed by the communication system can control threads with the same efficient user-level synchronization and scheduling primitives that are used by applications.

### 3.4 Summary of Rationale

This section has described the design rationale behind URPC. The main advantages of a user-level approach to communication are that calls can be made without invoking the kernel and without unnecessarily reallocating processors between address spaces. Further, when kernel invocation and processor reallocation do turn out to be necessary, their cost can be amortized over multiple calls. Finally, user-level communication can be cleanly integrated with user-level thread management facilities, necessary for fine-grained parallel programs, so that programs can inexpensively synchronize between, as well as within, address spaces.

## 4. THE PERFORMANCE OF URPC

This section describes the performance of URPC on the Firefly, an experimental multiprocessor workstation built by DEC's Systems Research Center. A Firefly can have as many as six C-VAX microprocessors, plus one MicroVAX-II processor dedicated to I/O. The Firefly used to collect the measurements described in this paper had four C-VAX processors and 32 megabytes of memory.

### 4.1 User-Level Thread Management Performance

In this section we compare the performance of thread management primitives when implemented at the user level and in the kernel. The kernel-level threads are those provided by Taos, the Firefly's native operating system.

Table I shows the cost of several thread management operations implemented at the user level and at the kernel level. PROCEDURE CALL invokes the null procedure and provides a baseline value for the machine's performance. FORK creates, starts, and terminates a thread. FORK;JOIN is like FORK, except that the thread which creates the new thread blocks until the forked thread completes. YIELD forces a thread to make a full trip through the scheduling machinery; when a thread yields the processor, it is blocked (state saved), made ready (enqueued on the ready queue), scheduled

Table I. Comparative Performance of Thread Management Operations

| Test | URPC FastThreads (μsecs) | Taos Threads (μsecs) |
|---|---|---|
| PROCEDURE CALL | 7 | 7 |
| FORK | 43 | 1192 |
| FORK;JOIN | 102 | 1574 |
| YIELD | 37 | 57 |
| ACQUIRE,RELEASE | 27 | 27 |
| PINGPONG | 53 | 271 |

Table II. Component Breakdown of a URPC

| Component | Client (μsecs) | Server (μsecs) |
|---|---|---|
| *send* | 18 | 13 |
| *poll* | 6 | 6 |
| *receive* | 10 | 9 |
| *dispatch* | 20 | 25 |
| **Total** | 54 | 53 |

again (dequeued), unblocked (state restored), and continued. ACQUIRE; RELEASE acquires and then releases a blocking (nonspinning) lock for which there is no contention. PING-PONG has two threads "pingpong" back and forth on a condition variable, blocking and unblocking one another by means of paired signal and wait operations. Each pingpong cycle blocks, schedules, and unblocks two threads in succession.

Each test was run a large number of times as part of a loop, and the measured elapsed time was divided by the loop limit to determine the values in the table. The *FastThreads* tests were constrained to run on a single processor. The tests using the kernel-level threads had no such constraint; Taos caches recently executed kernel-level threads on idle processors to reduce wakeup latency. Because the tests were run on an otherwise idle machine, the caching optimization worked well and we saw little variance in the measurements. The table demonstrates the performance advantage of implementing thread management at the user-level, where the operations have an overhead on the order of a few procedure calls, rather than a few hundred, as when threads are provided by the kernel.

## 4.2 URPC Component Breakdown

In the case when an address space has enough processing power to handle all incoming messages without requiring a processor reallocation, the work done during a URPC can be broken down into four components: *send* (enqueuing a message on an outgoing channel); *poll* (detecting the channel on which a message has arrived); *receive* (dequeueing the message); and *dispatch* (dispatching the appropriate thread to handle an incoming message). Table II shows the time taken by each component.

The total processing times are almost the same for the client and the server. Nearly half of that time goes towards thread management (*dispatch*), demonstrating the influence that thread management overhead has on communication performance.

The send and receive times do not reflect the cost of data copying and marshalling that is incurred when parameters are passed between address spaces. On the Firefly, each word of data passed in the shared memory buffers adds about one additional $\mu$sec of latency. Because the amount of data passed in most cross-address space procedure calls is small [8, 16, 17, 24], average call performance is most influenced by the components shown in Table II, and not the cost of data transfer.

## 4.3 Call Latency and Throughput

The figures in Table II are independent of client and server load, provided there is no need for processor reallocation. Two other important metrics, call latency and call throughput, are load dependent, though. Both latency and throughput depend on the number of client processors, $C$, the number of server processors, $S$, and the number of runnable threads in the client's address space, $T$.

To evaluate latency and throughput, we ran a series of tests using different values for $T$, $C$, and $S$ in which we timed how long it took the client's $T$ threads to make 100,000 "Null" procedure calls into the server inside a tight loop. The Null procedure call takes no arguments, computes nothing, and returns no results; it exposes the performance characteristics of a round-trip cross-address space control transfer.

Figures 3 and 4 graph call latency and throughput as a function of $T$. Each line on the graphs represents a different combination of $S$ and $C$. Latency is measured as the time from when a thread calls into the Null stub to when control returns from the stub, and depends on the speed of the message primitives and the length of the thread ready queues in the client and server. When the number of caller threads exceeds the total number of processors ($T > C + S$), call latency increases since each call must wait for a free processor in the client or the server or both. Throughput, on the other hand, is less sensitive to $T$. Except for the special case where $S = 0$, as long as the call processing times in the client and server are roughly equal (see Table II), then throughput, as a function of $T$, improves until $T \geq C + S$. At that point, processors are kept completely busy, so there can be no further improvement in throughput by increasing $T$.

When the number of calling threads, client processors, and server processors is 1 ($T = C = S = 1$), call latency is 93 $\mu$secs. This is "pure" latency, in the sense that it reflects the basic processing required to do a round-trip message transfer.[1] Unfortunately, the latency includes a large amount of wasted processing time due to idling in the client and server while waiting

[1] The 14 $\mu$sec discrepancy between the obseved latency and the expected latency (54 + 53, cf., Table II) is due to a low-level scheduling optimization that allows the caller thread's context to remain loaded on the processor if the ready queue is empty and there is no other work to be done anywhere in the system. In this case, the idle loop executes in the context of the caller thread, enabling a fast wakeup when the reply message arrives. At $T = C = S$, the optimization is enabled. The optmization is also responsible for the small "spike" in the throughput curves shown in Figure 4.

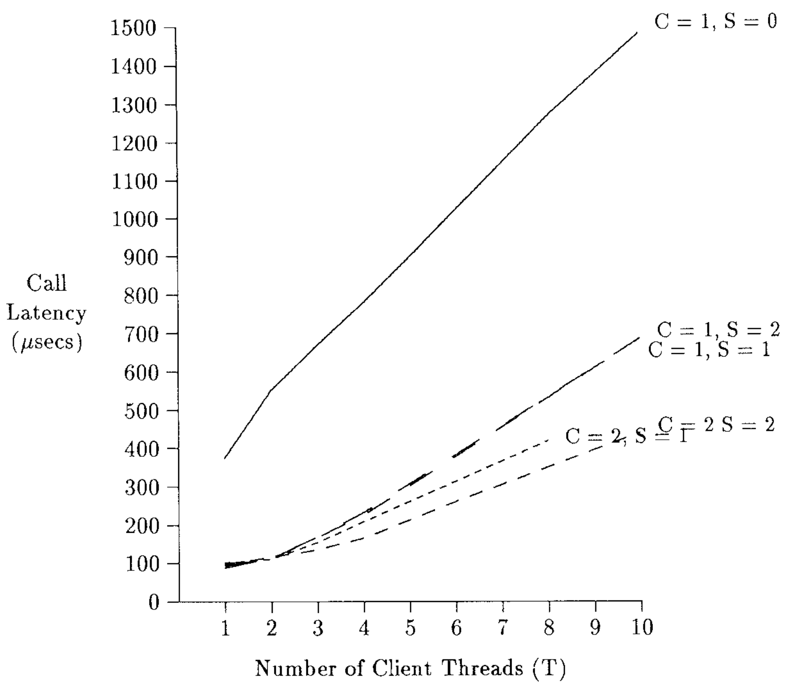

Fig. 3. URPC call latency.

for the next call or reply message. At $T = 2$, $C = 1$, $S = 1$, latency increases by only 20 percent (to 112 μsecs), but throughput increases by nearly 75 percent (from 10,300 to 17,850 calls per second). This increase reflects the fact that message processing can be done in parallel between the client and server. Operating as a two-stage pipeline, the client and server complete one call every 53 μsecs.

For larger values of $S$ and $C$, the same analysis holds. As long as $T$ is sufficiently large to keep the processors in the client and server address spaces busy, throughput is maximized.

Note that throughput for $S = C = 2$ is not twice as large as for $S = C = 1$ (23,800 vs. 17,850 calls per second, a 33 percent improvement). Contention for the critical sections that manage access to the thread and message queues in each address space limits throughput over a single channel. Of the 148 instructions required to do a round-trip message transfer, approximately half are part of critical sections guarded by two separate locks kept on a per-channel basis. With four processors, the critical sections are therefore nearly always occupied, so there is slowdown due to queueing at the critical sections. This factor does not constrain the aggregate rate of URPCs between multiple clients and servers since they use different channels.

When $S = 0$ throughput and latency are at their worst due to the need to reallocate processors frequently between the client and server. When $T = 1$, round-trip latency is 375 μsecs. *Every call requires two traps and two processor reallocations*. At this point, URPC performs worse than LRPC (157 μsecs).

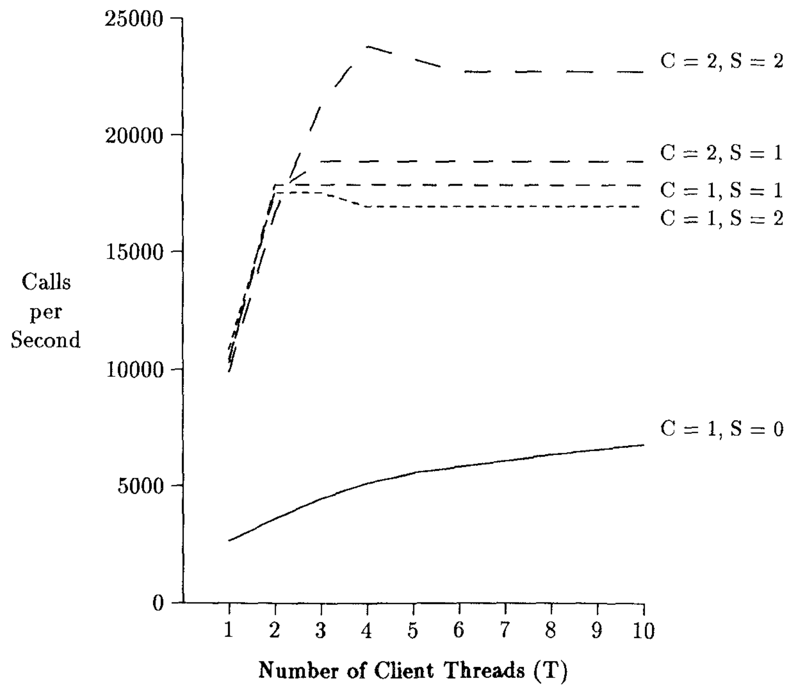

Fig. 4. URPC call throughput

We consider this comparison further in Section 5.5. Although call performance is worse, URPC's thread management primitives retain their superior performance (see Table I). As $T$ increases, though, the trap and reallocation overheads can be amoritized over the outstanding calls. Throughput improves steadily, and latency, once the initial shock of processor reallocation has been absorbed, worsens only because of scheduling delays.

### 4.4 Performance Summary

We have described the performance of the communication and thread management primitives for an implementation of URPC on the Firefly multiprocessor. The performance figures demonstrate the advantages of moving traditional operating system functions out of kernel and implementing them at the user level.

Our approach stands in contrast to traditional operating system methodology. Normally, operating system functionality is pushed *into* the kernel in order to improve performance at the cost of reduced flexibility. With URPC, we are pushing functionality *out* of the kernel to improve both performance *and* flexibility.

## 5. WORK RELATED TO URPC

In this section we examine several other communication systems in the light of URPC. The common theme underlying these other systems is that they reduce the kernel's role in communication in an effort to improve performance.

### 5.1 Sylvan

Sylvan [14] relies on architectural support to implement the Thoth [15] message-passing primitives. Special coprocessor instructions are used to access these primitives, so that messages can be passed between Thoth tasks without kernel intervention. A pair of 68020 processors, using a coprocessor to implement the communication primitives, can execute a *send-receive-reply* cycle on a zero-byte message (i.e., the Null call) in 48 $\mu$secs.

Although Sylvan outperforms URPC, the improvement is small considering the use of special-purpose hardware. Sylvan's coprocessor takes care of enqueueing and dequeueing messages and updating thread control blocks when a thread becomes runnable or blocked. Software on the main processor handles thread scheduling and context switching. As Table II shows, though, thread management can be responsible for a large portion of the round-trip processing time. Consequently, a special-purpose coprocessor can offer only limited performance gains.

### 5.2 Yackos

Yackos [22] is similar to URPC in that it strives to bypass the kernel during communication and is intended for use on shared memory multiprocessors. Messages are passed between address spaces by means of a special message-passing process that shares memory with all communicating processes. There is no automatic load balancing among the processors cooperating to provide a Yackos service. Address spaces are single threaded, and clients communicate with a process running on a single processor. That process is responsible for forwarding the message to a less busy process if necessary. In URPC, address spaces are multithreaded, so a message sent from one address space to another can be fielded by any processor allocated to the receiving address space. Further, URPC on a multiprocessor in the best case (no kernel involvement) outperforms Yackos by a factor of two for a round-trip call (93 vs. 200 $\mu$secs).[2] The difference in communication performance is due mostly to the indirection through the interposing Yackos message-passing process. This indirection results in increased latency due to message queueing, polling, and data copying. By taking advantage of the client/server pairing implicit in an RPC relationship, URPC avoids the indirection through the use of pair-wise shared message queues. In the pessimistic case (synchronous processor allocation on every message send), URPC is over thirteen times more efficient (375 vs. 5000 $\mu$secs) than Yackos.

### 5.3 Promises

Argus [28] provides the programmer with a mechanism called a *Promise* [27] that can be used to make an asynchronous cross-address space procedure call.

[2] Yackos runs the Sequent Symmetry, which uses 80386 processors that are somewhat faster than the C-VAX processors used in the Firefly.

The caller thread continues to run, possibly in parallel with the callee, until the time that the call's results are needed. If a thread attempts to "collect" on a Promise that has not yet been fulfilled (i.e., the cross-address space call has not yet completed), the collecting thread blocks.

A Promise is used to compensate for the high cost of thread management in the Argus system. Although an asynchronous call can be simulated by forking a thread and having the forked thread do the cross-address space call synchronously, the cost of thread creation in Argus [29] (around 20 procedure call times) precludes this approach. In URPC, the cost of thread creation is on the order of six procedure call times, and so the issue of whether to provide explicit language support for asynchronous cross-address space calls becomes one largely of preference and style, rather than performance.

## 5.4 Camelot

Camelot [19] is a high-performance distributed transaction processing system. Key to the system's performance is its use of recoverable virtual memory and write-ahead logging. Because of the cost of Mach's communication primitives [18], data servers use shared memory queues to communicate virtual memory control information and log records to the disk manager on each machine. When the queues become full, data servers use Mach's kernel-level RPC system to force a processor reallocation to the disk manager so that it can process the pending messages in the queue.

URPC generalizes the ad hoc message-passing approach used by Camelot. Camelot's shared memory queues can only be used between the data servers and the disk manager. URPC's channels are implemented beneath the stub layer, allowing any client and any server to communicate through a standard RPC interface. Camelot's thread management primitives are not integrated with those that access the shared memory queues. Finally, Camelot's shared memory queues support only unidirectional communication, whereas communication in URPC is bidirectional, supporting request-reply interaction.

At the lowest level, URPC is simply a protocol by which threads in separate address spaces access shared memory in a disciplined fashion. Programmers have often used shared memory for high-bandwidth cross-address space communication, but have had to implement special-purpose interfaces. URPC formalizes this interface by presenting it in the framework of standard programming abstraction based on procedure call and inexpensive threads.

## 5.5 LRPC

LRPC demonstrates that it is possible to communicate between address spaces by way of the kernel at "hardware speed." Like URPC, LRPC passes parameters in shared memory that is pair-wise mapped between the client and server. Unlike URPC, though, LRPC uses kernel-level threads that pass between address spaces on each call and return. LRPC greatly reduces the amount of thread management that must be done on each call, but is still performance limited by kernel mediation.

In the worst case, when there are no processors allocated to the server, one to the client, and there is one client thread, the latency for the Null URPC is 375 $\mu$secs. On the same hardware, LRPC takes 157 $\mu$secs. For both URPC and LRPC, each call involves two kernel invocations and two processor reallocations. There are two reasons why LRPC outperforms URPC in this case; the first is an artifact of URPC's implementation, while the second is inherent in URPC's approach:

—*Processor reallocation in URPC is based on LRPC.* URPC implements processor allocation using LRPC, which is a general cross-address space procedure call facility for kernel-level threads. We used LRPC because it was available and integrated into the Firefly operating system. The overhead of LRPC-based processor reallocation is 180 $\mu$secs, but we estimate that a special-purpose mechanism would be about 30 percent faster.

—*URPC is integrated with two-level scheduling.* Processor reallocation occurs after two distinct scheduling decisions are made: (1) Is there an idle processor, and (2) is there an underpowered address space to which it can be reallocated? In contrast, a call using LRPC makes no scheduling decisions (one of the main reasons for its speed). Making these two decisions is more expensive than not making them, but is necessary to take advantage of the inexpensive synchronization and scheduling functions made possible by user-level thread management. The overhead of these two scheduling decisions is about 100 $\mu$secs per round-trip call (only in the case where processor reallocation is required, of course).

It should not be surprising that indirecting through a second-level scheduler, which manages threads, increases the cost of accessing the scheduler at the first level, which manages processors. The trick is to infrequently interact with the first-level scheduler. When the first-level scheduler must be used, the overhead of having to pass through the second-level mechanism will inevitably degrade performance relative to a system with only a single level of scheduling.

## 6. CONCLUSIONS

This paper has described the motivation, design, implementation, and performance of URPC, a new approach that addresses the problems of kernel-based communication by moving traditional operating system functionality out of the kernel and up to the user level.

We believe that URPC represents the appropriate division of responsibility for the operating system kernels of shared memory multiprocessors. While it is a straightforward task to port a uniprocessor operating system to a multiprocessor by adding kernel support for threads and synchronization, it is another thing entirely to design facilities for a multiprocessor that will enable programmers to fully exploit the processing power of the machine. URPC demonstrates that one way in which a multiprocessor's performance potential can be greatly increased is by designing system facilities for a multiprocessor in the first place, thereby making a distinction between a multiprocesssor operating system and a uniprocessor operating system that happens to run on a multiprocessor.

## REFERENCES

1. United States Department of Defense. *Reference Manual for the Ada Programming Language*, July 1980.
2. AGARWAL, A. Performance tradeoffs in multithreaded processors. Tech. Rep. MIT VLSI Memo 89-566, Massachusetts Institute of Technology, Laboratory for Computer Science, Sept. 1989.
3. AGARWAL, A., HENNESSY, J., AND HOROWITZ, M. Cache performance of operating system and multiprogramming workloads. *ACM Trans. Comput. Syst. 6*, 4 (Nov. 1988), 393–431.
4. AGARWAL, A., LIM, B.-H., KRANZ, D., AND KUBIATOWICZ, J. APRIL: A processor architecture for multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture.* May 1990, 104–114.
5. ANDERSON, T. E., LAZOWSKA, E. D., AND LEVY, H. M. The performance implications of thread management alternatives for shared memory multiprocessors. *IEEE Trans. Comput. 38*, 12 (Dec. 1989), 1631–1644. To appear in *Proceedings of the 13th ACM Symposium on Operatıng Systems Principles.*
6. ANDERSON, T. E. BERSHAD, B. N., LAZOWSKA, E. D., AND LEVY, H. M. Efficient user-level thread management for shared memory multiprocessors. Tech. Rep. 90-04-02, Dept. of Computer Science and Engineering, Univ. of Washington, April 1990
7. BBN Laboratories. *Butterfly Parallel Processor Overvıew.* BBN Labs., Cambridge, Mass., June 1985.
8. BERSHAD, B. N. High performance cross-address space communication. Ph.D. dissertation, Dept. of Computer Science and Engineering, Univ. of Washington, June 1990 Tech. Rep. 90-06-02.
9. BERSHAD, B. N., LAZOWSKA, E. D., AND LEVY, H. M. PRESTO: A system for object-oriented parallel programming. *Softw. Pract. Exper. 18*, 8 (Aug. 1988), 713–732.
10. BERSHAD, B. N., ANDERSON, T. E., LAZOWSKA, E. D., AND LEVY, H M. Lightweight remote procedure call. *ACM Trans. Comput. Syst. 8*, 1 (Feb. 1990), 37–55. Also appeared in *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, Dec. 1989.
11. BIRRELL, A. D. AND NELSON, B. J. Implementing remote procedure calls *ACM Trans. Comput. Syst. 2*, 1 (Feb. 1984), 39–59.
12. BIRRELL, A. D. An introduction to programming with threads. Tech. Rep. 35, Digital Equipment Corporation Systems Research Center, Palo Alto, Calif., Jan. 1989.
13. BLACK, D. L. Scheduling support for concurrency and parallelism in the Mach operating system. *IEEE Comput. Mag. 23*, 5 (May 1990), 35–43.
14. BURKOWSKI, F. J., CORMACK, G., AND DUECK, G. Architectural support for synchronous task communication. In *Proceedıngs of the 3rd ACM Conference on Architectural Support for Programming Languages and Operating Systems.* April 1989, 40–53.
15. CHERITON, D. R. Thoth: A portable real-time operating system. *Commun. ACM 2* (Feb. 1979), 105–115.
16. CHERITON, D. R. The V distributed system. *Commun. ACM. 31*, 3 (Mar. 1988), 314–333.
17. COOK, D. The evaluation of a protection system. Ph.D. dissertatıon, Cambridge Univ., Computer Lab., April 1978.
18. DUCHAMP, D. Analyis of transaction management performance. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles.* Dec. 1989, 177–190.
19. EPPINGER. J. L. Virtual memory management for transaction processing systems. Ph.D. dissertation, Dept. of Computer Science, Carnegie Mellon Univ., Feb. 1989.
20. GUPTA, A., TUCKER, A., AND URUSHIBARA, S. The impact of operating system scheduling policies and synchronization methods of the performance of parallel applications. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (May 1991).
21. HALSTEAD, R. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst. 7*, 4 (Oct. 1985), 501–538.

22. Hensgen, D., and Finkel, R. Dynamic server squads ın Yackos. Tech. Rep 138-89, Dept of Computer Scıence, Univ. of Kentucky, 1989.
23. Jones, M. B. and Rashid, R. F. Mach and Matchmaker: Kernel and language support for object-oriented distributed systems In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applıcations (OOPSLA '86).* (Sept. 1986), 67-77.
24. Karger, P. A. Usıng registers to optımize cross-domain call performance. In *Proceedings of the 3rd ACM Conference on Archıtectural Support for Programming Languages and Operatıng Systems* (April 3-6 1989), 194-204.
25. Lenoski, D., Laudon, J., Gharachorloo, K., Gupta, A., and Hennessy, J. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedıngs of the 17th Annual Internatıonal Symposium on Computer Archıtecture.* (May 1990), 148-159.
26 Levy, H. M. and Eckhouse, R. H. *Computer Programmıng and Archıtecture: The VAX-11. 2nd Ed.* Digıtal Press, Bedford, Mass., 1989.
27. Liskov, B. and Shrira, L. Promises. Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Desıgn and Implementation* (June 1988), 260-267.
28. Liskov, B. Distributed programming in Argus. *Commun. ACM 31*, 3 (Mar. 1988), 300-312
29. Liskov, B., Curtis, D., Johnson, P., and Scheifler, R. Implementatıon of Argus. In *Proceedings of the 11th ACM Symposium on Operatıng Systems Princıples* (Nov. 1987), 111-122.
30. Nelson, B J. Remote procedure call. Ph.D. dissertatıon, Dept. of Computer Science, Carnegie Mellon Unıv., May 1981
31. Ousterhout, J K. Partitioning and cooperation in a distributed operating system. Ph D. dissertation, Dept of Computer Science, Carnegie-Mellon Univ., April 1980.
32. Ritchie, D. and Thompson, K. The Unix time-sharing system. *Commun. ACM 17*, 7 (July 1974), 365-375.
33. Roberts, E. S. and Vandevoorde, M. T. WorkCrews: An abstraction for controlling parallelism Tech. Rep. 42, Digital Equipment Corporation, Systems Research Center, Palo Alto, Calif., April 1989.
34 Royner, P , Levin, R., and Wick, J. On extending Modula-2 for building large, integrated systems. Tech Rep. 3, Digital Equipment Corporation, Systems Research Center, Palo Alto, Calif., Jan. 1985.
35. Sequent Computer Systems, Inc. *Symmetry Technical Summary*, 1988.
36. Tevanian, A., Rashid, R. F., Golub, D. B., Black, D. L., Cooper, E , and Young, M. W Mach threads and the Unix kernel: The battle for control. In *Proceedıngs of the 1987 USENIX Summer Conference.* (1987), 185-197.
37. Thacker, C. P., Stewart, L. C., and Satterthwaite, Jr., E H Firefly: A multıprocessor workstation. *IEEE Trans. Comput. 37*, 8 (Aug 1988), 909-920.
38. Weiser, M. Demers, A., and Hauser, C. The portable common runtıme approach to interoperability In *Proceedings of the 12th ACM symposıum on Operatıng Systems Prıncıples.* (Dec. 1989), 114-122.

Received July 1990; revised February 1991; accepted February 1991