Runtime Mechanisms for Efficient Dynamic Multithreading

Vijay Karamcheti, John Plevyak and Andrew A. Chien*
Concurrent Systems Architecture Group
Department of Computer Science
University of Illinois at Urbana-Champaign

*E-mail: {*vijayk,jplevyak,achien*}@*cs.uiuc.edu*

Running Head Runtime Mechanisms for Dynamic Multithreading

Contact Author Vijay Karamcheti
2233 Digital Computer Laboratory
University of Illinois at Urbana-Champaign
1304 W. Springfield Ave.
Urbana, IL 61801
phone: (217) 244-7116, fax: (217) 333-3501
E-mail: *vijayk@cs.uiuc.edu*

Abstract

High performance on distributed memory machines for programming models with dynamic thread creation and multithreading requires efficient thread management and communication. Traditional multithreading runtimes, consisting of few general-purpose, bundled mechanisms that assume minimal compiler and hardware support, are suitable for computations involving coarse-grained threads but provide low efficiency in the presence of small granularity threads and irregular communication behavior.

We describe two mechanisms of the Illinois Concert runtime system which address this shortcoming. The first, *hybrid stack-heap execution*, exploits close coupling with the compiler to dynamically form coarse-grained execution units; threads are lazily created as required by runtime situations. The second, *pull messaging*, exploits hardware support to implement a distributed message queue with receiver-initiated data transfer, delivering robust performance across a wide range of dynamic communication characteristics. We measure their performance impact based on a Cray T3D implementation of the Concert system. Individually, the mechanisms increase absolute execution efficiency by up to 50%. Together, they increase the feasible space of efficient computations, enabling compute granularities an order of magnitude smaller. Performance results for two large irregular applications demonstrate that expressing programs using dynamic multithreading need not compromise on performance.

List of Symbols

%	percent
μ	mu (Greek)
τ	tau (Greek)
σ	sigma (Greek)

1 Introduction

Irregular and dynamic problems are challenging to express and program efficiently on distributed memory machines. This is because they are not well matched to the two predominant parallel programming models. Their computation structure often does not fit into data parallel models, and message passing requires the programmer to deal explicitly with the complexities of data placement, addressability, and concurrency control. Consequently, programming models based on dynamic thread creation and multithreading [21, 3, 12, 41, 8, 17, 28, 16] are increasingly popular for expressing such problems. Such models involve user-defined computation units (hereafter referred to as logical threads) which are dynamically created to reflect the natural concurrency structure of the program; multithreading maps these onto the physical machine improving processor utilization. These models form the basis for several concurrent object-oriented languages [20, 11, 49] and message-driven systems [28], and simplify program expression by supporting flexible computation and synchronization structures.

Several researchers have investigated multithreading runtime systems which provide mechanisms for communication and thread creation, synchronization and scheduling. For the most part, these systems consist of a few general-purpose, bundled mechanisms which assume minimal compoler and hardware support. Portable runtime systems [21, 16, 40] build these mechanisms on top of vendor-supported, standardized lightweight thread management [27] and communication [15, 44] interfaces; however, these incur relatively large overheads, requiring coarse-granularity threads for efficiency. While systems with specialized runtimes [3, 8, 28] can provide efficient primitives supporting finer grained threads, they still incur large thread management and communication overheads for irregular, dynamic computations:

- Such computations exhibit wide variations in thread granularity and are typically not amenable to compile-time analyses which can coalesce logical threads into sufficiently coarse-grained physical threads.
- Such computations exhibit unbalanced communication traffic which is unsynchronized across the processors, inefficiently supported by traditional communication mechanisms.

In this paper, we demonstrate that the key to delivering efficient performance for irregular applications involving dynamically created threads is for the runtime to exploit closer coupling with the compiler on one end and the hardware on another. We describe two mechanisms of the Illinois Concert runtime system — *hybrid stack-heap execution* and *pull messaging* — which enhance the basic multithreading mechanisms (shown in Figure 1) to address the above problems.

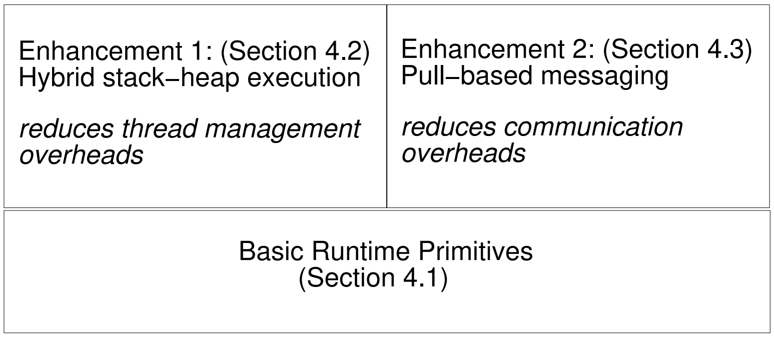

Figure 1: Structure of the Illinois Concert runtime system.

Hybrid stack-heap execution dynamically coalesces logical threads into coarser-grained physical threads based on runtime data location and available parallelism. Our technique provides a flexible runtime interface, enabling the compiler to generate code which optimistically executes a logical thread sequentially on its caller's stack, *lazily* creating a different heap-allocated thread only if it suspends or need be scheduled separately. The technique executes sequential program portions (where all accessed data is local) with the efficiency of static procedure calls, and parallel portions using efficient multithreading among the heap-allocated threads.

Pull-messaging provides robust high-performance communication even in the presence of irregular, unbalanced traffic, and unsynchronized processor communication and computation phases. Exploiting hardware support for remote memory access, synchronization and prefetching, our mechanism builds a distributed queue implementation of messaging with lazy, receiver-initiated data transfer. Distributed message queuing decouples the sending processor from the activity of other processors. Lazy, receiver-initiated data transfer "pulls" the message from the sender's memory, eliminating output contention because data is moved only when the receiver is ready to process it.

Detailed performance studies for a synthetic compute-communicate microkernel based on a Cray T3D implementation, demonstrate that individually, the mechanisms each increase microkernel execution efficiency by up to 50 percentage points; hybrid stack-heap execution reduces thread management overheads for small compute granularities and high runtime locality, while pull-messaging improves communication performance for low runtime locality. Together, they increase the space of efficient computations, enabling computations with an order of magnitude smaller granularity and decreased runtime locality. For example, the mechanisms enable 70% microkernel execution efficiency, previously requiring compute granularities larger than 300μs and greater than 80% local accesses, to be achieved with significantly lower granularities of 10μs (with 100% local accesses) and 150μs (with 0% local accesses). Measurements for two large irregular applications — hierarchical radiosity and macromolecular protein dynamics — show that performance is comparable to that achieved by explicitly optimized versions, demonstrating that expressing programs using dynamic multithreading need not compromise on performance.

The rest of the paper is organized as follows. We summarize the relevant background in Section 2 describing irregular computation structures as well as our specific programming and execution models. In Section 3, we present a microkernel program used as a running example. Section 4 presents the individual components of the Concert runtime system. In Section 5, we quantify the performance advantages of each mechanism using a synthetic microkernel as well as two large irregular applications. Related work is discussed in Section 6, and we conclude in Section 7.

2 Programming and Computational Model

Irregular application programs (such as molecular dynamics, particle simulations, adaptive mesh refinement, etc.) are characterized by irregularly sized units of work, which may be created dynamically, and data access patterns which are unpredictable. In addition, modern algorithms often make use of complex data structures to achieve high efficiency [2, 19]. Thus, such computation structures are not easily amenable to expression either in a regular data-parallel model, or in a message passing model that requires the programmer to map the computation into a fixed number of threads synchronizing using matching communication primitives.

Programming models based on a dynamic thread pool operating against shared data provide several tools which simplify the expression of irregular parallel computations. A *shared name space* for the data objects allows programmers to build sophisticated distributed data structures without explicit name management. *Dynamic thread creation* frees programmers from explicit thread management and synchronization and allows the irregular concurrency in the application to be expressed in a non-binding manner, leaving the implementation free to adapt the concurrency to available parallelism.

In this work, we assume a fine-grained concurrent object-oriented model where objects reside in a global namespace and each method invocation corresponds to a logical thread. Synchronization between threads is achieved via *futures* [22]: if the caller thread *touches* the future, i.e. it attempts to read its value, before the thread responsible for writing the value is finished, then the caller thread blocks. When the thread holding the *continuation* (the right to determine the future) finishes its computation, it writes the future's value and restarts any blocked threads. This programming model supports a wide variety of synchronization and communication structures including: synchronous (RPC), data (object) parallel, reactive, etc. In addition, continuations can be forwarded to another thread or stored in data structures, allowing construction of application-specific communication and synchronization structures. Our programming model provides implicit object-level concurrency control for preventing inconsistent concurrent accesses from several threads, implying that threads may suspend awaiting access control even if data objects are locally available.

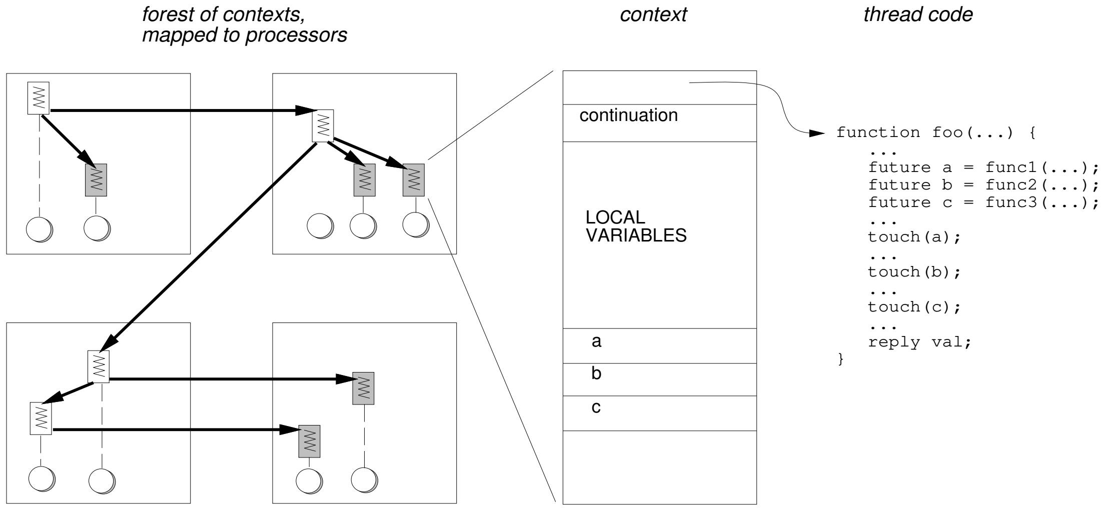

Figure 2: Computation model showing processors (large boxes), contexts (small boxes), objects (circles), and thread creation structure (dark lines). A context includes space for the thread local variables, future locations for pending requests, and the continuation (linkage to the requester). The shaded contexts are active (or ready); on each processor, these contexts are linked into a scheduling structure (not shown).

Efficient execution of the above programming model on distributed memory parallel machines requires mapping the logical threads into physical threads according to the computation model shown in Figure 2. Creating a thread involves allocating a *context*, depositing arguments into it, and initiating execution of the thread code relative to the newly allocated context. The context provides storage for the local variables of the thread (similar to a procedure stack frame), and futures and continuations required for synchronization. Thread creation does not suspend the caller; thus, dynamic thread creation produces a forest of contexts. Threads access data objects in a location-independent fashion: the underlying system synthesizes a global namespace by detecting accesses to remote data objects and translating them into communication operations. Several contexts can be mapped to the same processor and are managed by a scheduler which interleaves the processor resources among ready threads. A thread suspends either when it accesses remote data or when it touches the value of one of its futures, causing the scheduler to activate the next ready thread. A blocked thread becomes ready when the future it is blocked on is filled by another thread holding the continuation.

The above programming and computation models are supported by the Illinois Concert system [9] which is an implementation platform for the Concurrent Aggregates [11] and ICC++ [10] concurrent object-oriented languages. The Concert system consists of an optimizing compiler [39] and the runtime system described in the rest of the paper. The compiler is capable of resolving interprocedural control and data flow information [37], enabling it to specialize the generated code based on synchronization and communication features required by the computation. The compiler generates C code which links with a library of runtime primitives to produce the executable.

3 Example: A Commmunicate-Compute Microkernel

In this section, we describe a microkernel program used as a running example in the rest of the paper. The microkernel serves to highlight the key performance advantages of the two runtime mechanisms, later corroborated for whole applications in Section 5.2. The code for the microkernel is shown in Figure 3. The `main` routine creates a set of independent threads, `WorkThread`, on each processor. `conc` specifies that all the threads can be processed concurrently. The computation terminates when all threads on all processors complete. Each `WorkThread` thread consists of a data access phase where it gathers data values from (potentially remote) objects, followed by a phase where it computes using these values. Since data accesses are subject to the object-level concurrency control described earlier, each access creates a logical thread for processing the service request.

```
1   main() {
2      conc for (i=0; i<nprocs; i++)
3         conc for (j=0; j<nthreads; j++)
4            WorkThread();
5   }
6
7   WorkThread() {
8      // data access: obj is potentially remote
9      conc { x = valx(obj); y = valy(obj); z = valz(obj); }
10     // compute with data
11     compute(x, y, z);
12  }
```

Figure 3: Code structure of the communicate-compute microkernel (shown using ICC++ syntax). The execution behavior of the microkernel is characterized by the fraction and distribution of remote accesses in line 9 and the compute granularity and variance in line 11 (shown boxed).

This microkernel exhibits the general communication and computation characteristics of several irregular applications such as molecular dynamics, hierarchical radiosity calculation, etc. where a thread first gathers data from a set of objects and then operates on it. Its specific behavior can be controlled by four parameters:

- Percentage of remote accesses (line 9): This determines the amount of communication required by the microkernel. Many irregular applications have statically unpredictable communication behavior; consequently, the computation must adapt to runtime locality for efficiency.
- Distribution of remote accesses (line 9): Unlike regular applications, irregular ones are characterized by unbalanced traffic patterns with varying degrees of output contention. Controlling the location of `obj` produces various output contention situations.
- Compute granularity (line 11): This parameter affects the ratio of compute grain size to communication overhead. Natural expression of irregular applications often produces small granularity threads.
- Variance in compute granularity (line 11): This parameter creates execution scenarios where threads exhibit wide variations in compute granularities. Irregular computations apply effort where most effective, creating large granularity variations.

Corresponding to the three components shown in Figure 1, we consider microkernel executions for four runtime versions (described in detail in Section 4):

1. Base, consisting of basic runtime primitives.
2. Base+Stack, consisting of Base enhanced with hybrid stack-heap execution mechanisms.
3. Base+Pull, consisting of Base enhanced with pull-messaging mechanisms.
4. Base+Stack+Pull, which includes both hybrid stack-heap execution and pull-messaging mechanisms.

4 Runtime Mechanisms

In this section, we describe the three Concert runtime system components shown in Figure 1, limiting our attention to thread management and communication mechanisms. While all multithreading runtime systems provide base mechanisms for data transfer and thread creation, scheduling and synchronization, the Concert runtime includes two additional mechanisms which address the major shortcomings of dynamic multithreaded computations: increased thread management overheads resulting from fine-grained threads, and increased communication overheads because of unbalanced, unsynchronized communication.

In this section, we first describe our implementation of the basic communication and thread management primitives. These primitives provide a baseline for evaluating the performance advantages of the other two mechanisms. We then describe the hybrid stack-heap execution and the pull-messaging mechanisms.

4.1 Basic Runtime Primitives

The thread management mechanisms provide primitives for creation and deletion, scheduling, and synchronization. The communication mechanisms provide primitives to send and receive data. The base Concert primitives represent a high-performance implementation of traditional multithreading runtime systems providing compiler-oblivious bundled primitives. For example, on the IBM SP/2, the Nexus runtime system [16] which is layered on top of a standard thread package (pthreads [27]) and IBM's implementation of MPI [15], incurs thread creation and communication overheads (0-byte latency) of 32.0μs and 44.0μs respectively. As we shall see later, the Concert primitives described below incur an order of magnitude less cost (see Tables 1 and 2).

4.1.1 Thread Management

Thread Creation and Deletion Two primitives, `thread_create` and `thread_delete`, are provided. The `thread_create` operation allocates a context (the thread's activation frame), deposits the supplied arguments into it, and initiates execution of the thread code (pointed to by `function_ptr`) relative to this newly allocated context. The `thread_delete` operation reclaims the context for the thread `t`.

```
Thread* thread_create(function_ptr, args, ...);
void    thread_delete(Thread* t);
```

In our base implementation, a context is a heap-allocated structure, whose allocation and deletion is performed efficiently using a free list. Use of a heap-allocated context is enabled by a specialized compiler which generates code for saving and restoring thread state (into the context) at suspension points. The compiler can also optimize thread switching cost by managing live state at suspension points. More importantly, heap-allocated contexts enable creation and deletion costs of less than 1μs, an order of magnitude cheaper than that for portable multithreading systems such as Chant [21], Nexus [16], etc. These latter systems require a dedicated stack per thread because they rely on a sequential compiler infrastructure which does not provide any support for saving and restoring thread state across suspensions.

Operation		Cost cycles	Cost μs
Creation	`thread_create`	150	1.00
	`thread_delete`	100	0.67
Scheduling	`thread_enqueue`	40	0.27
	`thread_dequeue`	40	0.27
Synchronization	`make_future`	20	0.14
	`touch_future`	11	0.07
	(if empty)	22	0.14
	`resolve_future`	10	0.07
	(if waiting)	50	0.35

Table 1: Costs of basic Concert thread management primitives on the T3D.

Thread Scheduling Two primitives: `thread_enqueue` and `thread_dequeue` are provided: `thread_enqueue` attaches the thread to the scheduler, while `thread_dequeue` selects the next ready thread and initiates execution.

```
void thread_enqueue(Thread* t);
void thread_dequeue();
```

In our base implementation, the scheduler maintains the list of ready threads as a singly-linked list. Making a thread ready is a simple matter of adding the context to the tail of the list. Scheduling the next ready thread involves dequeuing the head of the list and initiating execution of the thread code via an indirect function call. More sophisticated thread scheduling policies (e.g., priority-based queuing) can be built on top of this scheme.

Thread Synchronization As described in Section 2, thread synchronization is achieved via the future mechanism. Primitives are provided to make, resolve and touch futures. `make_future` tags the context location as a future and returns a continuation. `touch_future` involves a tag check and returns TRUE if the future has already been resolved, else FALSE. `resolve_future` stores a value into the future and enqueues any blocked threads.

```
Continuation make_future(return_location);
int          touch_future(return_location);
void         resolve_future(Continuation contin, value);
```

In addition, the Concert runtime provides support for counting futures as well as a general `touch` primitive permitting synchronization on multiple futures, amortizing the cost of thread restart.

4.1.2 Communication

Communication operations arise whenever a thread accesses remote data or interacts with a remote thread. The communication primitives in the Concert runtime system exploit a lean messaging interface to incur 5-10 times lower latency as compared to vendor communication libraries.

The Concert communication primitives use the low-level Fast Messages [35] interface. The interface consists of two send primitives and one receive primitive:

```
typedef void function_ptr(...);
void    send_4(int rnode, function_ptr *fptr, arg1, arg2, arg3, arg4);
void    send(int rnode, function_ptr *fptr, void *buf, int size);
int     extract();
```

Each message send is associated with execution of a handler at the destination node (similar to active messages [46]). The `send_4` is optimized for transferring a small number of register arguments (up to 4) to the destination node, while the `send` primitive is more general, accepting an arbitrary sized buffer and the message length as arguments. Message reception requires use of the `extract` primitive which executes the handlers for all pending messages and returns TRUE if any messages were processed. Similar to active messages, this interface assumes a certain usage discipline. Specifically, one must ensure that protocols involving sends from within message handlers must be deadlock free. This is another place where the availability of a compiler helps reduce the cost of primitive runtime mechanisms: a compiler can enforce the required discipline.

The T3D implementation of the Fast Messages interface [30] makes use of hardware support for fetch-and-increment and remote memory access [34] to perform buffer management and data transfer without involving the destination processor. This decouples the sending processor from destination processor activity, improving communication performance. The fetch-and-increment register is used to index a preallocated set

Operation (all costs in μs)		Message Size (in bytes) 16	32	64	128	256
Overhead	send	1.84	1.85	2.03	2.71	3.53
	extract	0.73	0.73	0.73	0.73	0.73
One-way Latency		6.16	6.25	7.19	8.07	8.65
Vendor Library (PVM) Latency		34.31	34.33	34.95	36.49	40.45

Table 2: Latency and software overheads of Concert communication primitives on the T3D for various message sizes. The vendor communication library latency numbers are shown for comparison.

of message buffers. The source node obtains a valid index by performing a fetch-and-increment operation with respect to the destination node, then transfers data using remote stores. Message completion at the receiver is detected by a tag at the end of the message buffer. This implementation of the lean messaging interface incurs an order of magnitude lower latency (shown in Table 2) as compared to vendor communication libraries. In addition, the Concert primitives overlap communication and computation; consequently, the non-overhead portion of latency can be effectively eliminated.

4.1.3 Summary: Basic Runtime Primitives

The Concert runtime system provides efficient thread management and communication operations, utilizing the existence of a specialized compiler to incur an order of magnitude less cost than corresponding vendor-supplied mechanisms. In addition the compiler can customize the bundling of the runtime mechanisms affecting procedure call boundary crossings (e.g., the compiler can inline some of the runtime primitive calls), further reducing cost in situations where it has static information about thread interactions [29].

Corresponding to these basic primitives, we define a version of the runtime, Base, which serves as a competitive baseline for comparing the performance advantages of the other mechanisms. The Base runtime provides bundled versions of thread creation and scheduling primitives. Specifically, for the microkernel described in Section 3, since the compiler cannot statically determine that all data accesses in line 9 of the microkernel will complete without blocking for concurrency control, the generated code creates a physical thread for each of the data access requests. Thus, each data access request involves a `make_future`, any required communication (if the thread needs to be executed remotely[1]), creation, scheduling, and deallocation of the callee context, and touching the future along with suspension and restart (if required) of the caller. For the same reason, each logical `WorkThread` thread also results in the creation of a separate physical thread. The compiler can perform one optimization though: the synchronization for all thread requests in lines 4 and 9 can be grouped, ensuring that the caller context suspends and restarts only once. The performance of the microkernel code using the Base runtime is described in Section 5.

4.2 Enhancement 1: Hybrid Stack-Heap Execution

While the basic runtime primitives in Section 4.1 enable efficient execution of finer granularity computations (particularly in comparison with portable runtime systems with more expensive primitives), as we shall see later, they still incur sizable thread management costs for dynamic computations. These computations are less amenable to static analyses, so the compiler is typically unable to coalesce logical threads into sufficiently coarse-grained physical threads. This section describes a hybrid stack-heap execution mechanism which addresses this shortcoming.

The hybrid stack-heap mechanism constructs coarser-grained physical threads "on the fly" from fine-grained logical threads, achieving high sequential efficiency when the thread accesses only local data objects and incurring minimal thread management overhead when the thread interacts with remote data objects or threads. Our technique provides a flexible interface for the runtime primitives to the compiler, enabling

[1] We assume that the thread is executed local to the data object that it accesses.

it to generate code which optimistically executes a logical thread sequentially on its caller's stack, *lazily* creating a different thread only when the callee computation needs to suspend or be scheduled separately. To separately optimize for the two modes of operation – sequential and parallel – the compiler generates two code versions for each thread body. One version executes off a stack-allocated activation frame and is optimized for sequential efficiency, accruing the advantages of procedure-call like efficiency for thread interactions when only local data is accessed. The other version operates from a heap-allocated context and is optimized to minimize thread scheduling and synchronization overheads. However, since a thread may not complete on the stack because of various blocking situations, the stack code version detects these situations at runtime, lazily creates a heap context (corresponding to a physical thread) and resumes execution from the corresponding point in the heap code version. We define a hierarchy of interaction schemas for the stack versions of the thread code, which the compiler can optimize for efficiency. Table 3 summarizes these interaction schemas which are described below in detail.

Version		Basic Operation
Heap		Most general schema, thread arguments/linkage through heap-allocated contexts
Stack	Non-blocking	Regular C call/return
	May-block	Regular call; detect and lazily create heap context on block
	Continuation-passing	Extension of may-block which allows forwarding on the stack

Table 3: Various thread interaction schemas in the hybrid stack-heap execution model.

4.2.1 Heap Version: Optimized for Parallel Execution

This version operates against a heap-allocated activation frame (context) and is a conservative implementation supporting thread suspension. Suspension may occur while interacting with remote data objects and threads, or while waiting for results from another blocked thread. Thread creation, scheduling and synchronization is identical to that described in Section 4.1. The compiler optimizes the code to minimize the thread management overhead: several requests are issued in parallel and the touching of the corresponding futures is grouped so as to minimize the number of required restarts. In addition, the compiler carefully manages the amount of state that needs to be saved and restored across suspension points.

4.2.2 Stack Versions: Optimized for Sequential Execution

Stack versions optimistically execute the thread code using a stack-allocated activation frame, lazily "falling back" to a heap context only when the thread needs to suspend. Thus, logical thread creation is essentially similar to a procedure call, incurring negligible overhead when the thread completes without blocking. However, when the thread blocks, we need to create the callee's heap context and set up the linkage between the caller and callee threads to allow thread execution to continue using the heap version. Depending on the thread interaction scenario, these operations require different information to be passed across the initial procedure call boundary and incur different fallback costs. To allow the compiler to optimize these costs based on available information, we define a hierarchy of stack schemas: Non-blocking, May-block, and Continuation-passing.

The *Non-blocking* version is used when the compiler can prove that the called thread and all of its descendent calls will complete without blocking. When the compiler cannot prove this but knows that the callee thread does not require the caller's continuation (except to return a value), the *May-block* version is used. If the thread blocks, a heap context is lazily allocated to continue execution once it resumes. Finally, the *Continuation-passing* version is used if the callee thread may require the continuation of a future in the (as yet uncreated) caller thread's context. In this case, we create both the callee's context as well as the continuation lazily. Our compiler selects the appropriate schema for each thread interaction based on a global flow analysis [37] which conservatively determines the blocking and continuation requirements of each thread body [38]. A novel aspect of our hybrid stack-heap execution model is that it is implemented entirely in C, and consequently, is portable across a variety of parallel platforms. The discussion below describes how detection of successful completion and setting up of thread linkages in the event of fallback are handled by building on top of the traditional C procedure call/return linkage.

Non-blocking: Straight C Call When the compiler can ascertain that a thread will not block, all interactions are handled using a standard C call. This enables entire non-blocking subgraphs of the thread call tree to execute with no thread management overhead. The code sequences for the caller and callee threads are shown in Figure 4.

```
...
rval = thread_code(...);
...
```

```
ret_type thread_code(...) {
   ...
}
```

Figure 4: Caller (left) and Callee (right) code sequences for the Non-blocking schema.

May-block: Lazy Context (Thread) Creation This stack version handles the case when a logical thread call may block. The calling schema, shown in Figure 5, distinguishes between the two outcomes – successful completion and a blocked callee thread. If the callee runs to completion, a NULL value is returned, and the caller thread extracts the actual return value from `rval`, a pointer to which is passed in as an argument to the call. If the callee thread blocks, the callee context (which itself was just created) is returned (in response to the C procedure call), enabling the caller to set up a linkage to the callee context. This is necessary because the linkage between caller and callee was implicit in the stack structure, and the caller must insert a continuation for the callee's return value into the callee context to preserve that linkage. Subsequently, the caller will, if necessary, create its own context, reverting to the parallel code version, save its state for restart from the heap version, and return its own context to its caller. Figure 6 shows the unwinding of stack frames when logical callee threads cannot complete their execution on the stack. In this case, the fallback code creates the callee's context, saves local state into it, and propagates the fall back by returning this context to its caller which then sets up the linkage.

```
...
callee_context = thread_code(&rval,...);
if (callee_context != NULL) {
   callee_context->contin =
      make_continuation(rval);
   // propagate blocking
}
...
```

```
Context* thread_code(rval_ptr,...) {
   ...
   if (non_blocking_arm) {
      *rval_ptr = val;
      return NULL;
   } else {        // fallback
      own_context = create_context();
      // save state to heap
      return own_context;
   }
   ...
}
```

Figure 5: Caller (left) and Callee (right) code sequences for the May-block schema.

Thus, the may-block calling schema allows a sequence of may-block threads to run to completion on the stack, creating physical threads only as required by the runtime situations.

Continuation-passing: Lazy Continuation (and Context) Creation Explicit continuation passing can improve the composability of concurrent programs [49, 11]. However, when continuation passing occurs, invocations on the stack are complicated because the callee may want its continuation. If the call is being

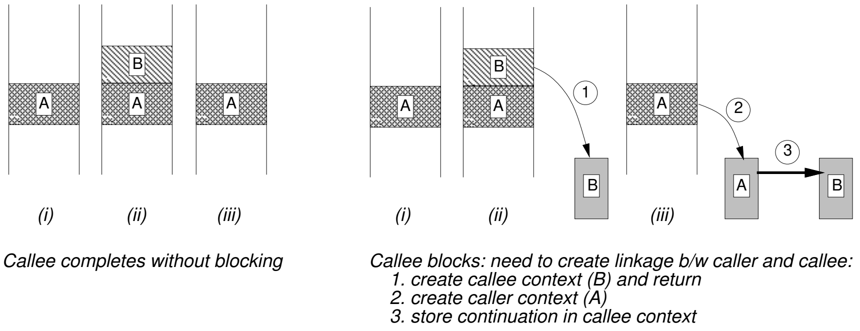

Figure 6: **May-block** interaction schema. The left figure shows successful completion, while the right figure shows the stack unwinding when the call cannot be completed.

executed on the stack, the callee's continuation is implicit. Since one of our goals is to execute forwarded invocations [25] on the stack, lazy allocation of the continuation is essential. As we shall see, allocation of a continuation also implies creation of the context in which the returned value will be stored.

The calling schema for the continuation passing case (see Figure 7) uses an additional parameter, **cinfo**, which, along with the **rval_ptr** encodes information necessary to determine what to do should the continuation be needed. The **cinfo** information is simply passed along to support local forwarding, but if a thread tries to store the continuation or forward it off-node, it must be created. **cinfo** contains information indicating whether or not the context containing the continuation's future has already been created, and sufficient information to create both the context and the callee's continuation in case it has not.[2]

```
...
caller_context =
   thread_code(&rval,cinfo,...);
if (caller_context != NULL) {
   // save state to heap
   // propagate blocking
}
...
```

```
Context* thread_code(rval_ptr,cinfo,...) {
   ...
   if ( need_continuation ) {      fallback
      caller_context =
         create_context(rval_ptr,cinfo);
      own_context = create_context();
      own_context->contin =
         make_continuation(caller_context,cinfo);
      // save state to heap
      return caller_context;
   } else {
      *rval_ptr = val;
      return NULL;
   }
   ...
}
```

Figure 7: Caller(left) and Callee(right) code sequences for the Continuation-passing schema.

When the continuation is not created, the thread which resolves the future simply stores the result through **rval_ptr**, and passes NULL return values back to its caller. The caller of the first continuation-passing method receives this NULL value and looks in **rval** for the result, thus executing the forwarded continuation completely on the stack. On the other hand, if the continuation is required by the callee, **cinfo** is consulted. The fallback code first obtains the caller context (creating it if necessary), and then the continuation for a new future at a location in the caller's context corresponding to **rval_ptr** and the return value offset stored in **cinfo**. The callee may now do whatever is desired with the continuation, finally passing the continuation's future's context back to its caller. The callee code in Figure 7 shows a situation

[2] This typically requires information about the size of the caller context, and the return location within it.

where the callee also suspends, causing it to create its own context, save any state necessary, and insert the created continuation. Figure 8 shows the stack unwinding and linkages which need to be set up as part of the fallback. The reader is referred to [38] for implementation details.

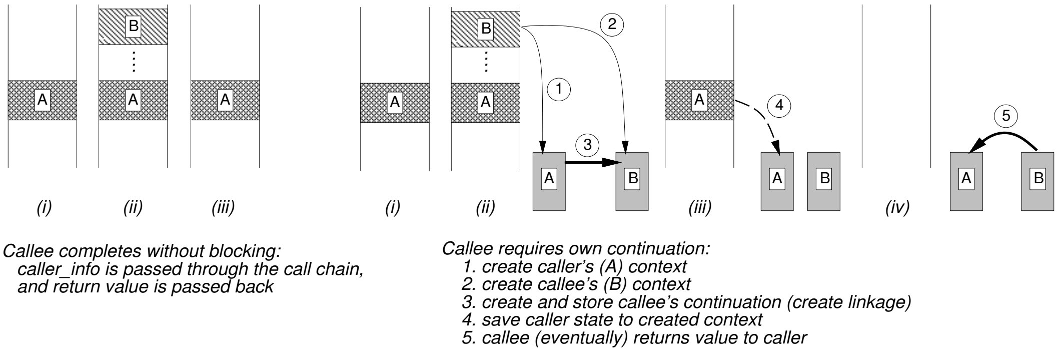

Figure 8: Continuation-passing interaction schema. The left figure shows callee completion without blocking, while the right figure shows the fallback when the callee requires the caller's continuation.

4.2.3 Overheads of Hybrid Stack-Heap Execution

Call Overhead (in Alpha instructions)

Calling schema for Caller		Calling schema for Callee: Parallel	Sequential		
			NB	MB	CP
Parallel		320	0	6	8
Seq.	NB	–	0	–	–
	MB	–	0	6	8
	CP	–	0	6	8

Fallback Overhead (in μs)

Calling schema for Caller		Calling schema for Callee: Parallel	Sequential		
			NB	MB	CP
Parallel		–	0	0.15	0
Seq.	NB	–	0	–	–
	MB	–	0	2.00	0.30
	CP	–	0	3.50	1.80

Table 4: Call and fallback costs (at the caller) for different caller-callee scenarios on the Cray T3D, in terms of overhead required in addition to a C function call. NB, MB and CP stand for Non-blocking, May-block and Continuation Passing sequential calling schemas respectively.

Table 4 presents the cost of the hybrid stack-heap invocation mechanisms for various caller-callee scenarios as overhead required in addition to the cost of a basic C function call on the Cray T3D.[3] There are two components to this overhead: the first (shown in the left table) corresponds to the situation when the callee thread completes on the stack, and the second (shown in the right table) indicates the additional fallback cost when the callee stack frame must be unwound into the heap. Stack calls which complete without blocking have cost comparable to a basic C function call and two orders of magnitude less overhead than the parallel (heap-based) thread interaction (320 instructions). The 6–8 additional instructions for sequential calls are due to call arguments, and passing the return value through memory, rather than in a register.

The fallback overheads vary from 0.15 – 3.50μs depending on the specific caller-callee scenario. The unwinding costs are different for different caller and callee combinations because the different schemas place the responsibility for context creation and state saving at different places. These fallback overheads make explicit the tradeoff in using the sequential and parallel versions. The maximum fallback cost for any

[3] On an Alpha (the node processor of the T3D), a C funtion call costs 25-40 instructions.

caller-callee pair is comparable to the basic heap-based invocation (320 instructions = $2.1\mu s$),[4] so optimistic execution using a sequential invocation first is effective in almost all cases. The same numbers also show that a sequential thread version can incur substantial overhead if it blocks repeatedly incurring multiple fallbacks; thus, reverting to the parallel method after the first fallback is a good strategy, especially if several synchronizations are likely.

4.2.4 Summary: Hybrid Stack-Heap Execution

The hybrid stack-heap runtime execution mechanisms permit the compiler to generate code which dynamically coalesces logical threads into larger-grained physical threads based on runtime locality and parallelism situations. By optimistically executing logical threads on the caller's stack using a specialized sequential code version, and lazily creating a physical thread only when the callee code needs to block, we minimize thread management overheads and accrue advantages of good sequential performance as appropriate.

The addition of the hybrid stack-heap execution mechanisms to the Base runtime described in Section 4.1 produces a new version Base+Stack. This runtime version exposes a flexible interface enabling the generated code to take advantage of runtime locality situations. Specifically, for the microkernel program, the compiler optimistically inlines the data accesses in line 9 predicated on locality and concurrency control checks. Remote accesses are handled efficiently by avoiding thread creation and scheduling overheads when the access can be completed without blocking for concurrency control reasons. The `WorkThread` code itself is invoked via a may-block stack schema. This has the implication that threads which end up accessing only local objects at runtime (i.e., those for whom `obj` is a local object pointer) execute completely on the stack, incurring only the cost of a C function call. Those threads which access truly remote objects lazily create heap contexts to complete their execution. The performance advantages of using the hybrid stack-heap execution mechanism are described in Section 5.

4.3 Enhancement 2: Pull-based Communication Mechanisms

While hybrid stack-heap execution reduces thread management overheads, it does not address the increased communication overheads that arise from unbalanced traffic and unsynchronized communication. This section describes a pull-based messaging implementation which delivers performance robust over a range of dynamic communication characteristics. The rationale for these mechanisms is that traditional messaging implementations inefficiently support the above characteristics because a slow receiver (busy computing or unable to accept data fast enough because of output contention) can back up the network to other processors. This back up increases the send overheads for processors which are now unable to inject messages into the network. This degradation is present even when buffer management and data transfer for individual message sends can be accomplished without destination participation (as with our T3D fetch-and-increment based implementation described in Section 4.1); this is because the destination needs to eventually participate to manage finite buffering resources (e.g., by reclaiming consumed buffers) and any delay in this participation holds up the senders. The performance degradation due to unresponsive receivers and output contention even with modest fan-in can be severe, increasing send overheads by up to an order of magnitude [30].

Our solution exploits hardware support on the T3D (also present in several current and likely future machines) to build a distributed message queue with lazy receiver-initiated data transfer which decouples senders from receivers and eliminates output contention. Using the T3D's atomic-swap hardware to link messages into a distributed message queue, sending can go at the maximum swap rate with the receiving processor "pulling" messages from the sender's memory. Receives incur a round-trip latency, but the T3D's prefetch queue can be used to mask this latency. Pull messaging eliminates output contention and sender-receiver coupling since data is only moved when the receiver is ready to process it. An additional advantage is that buffer overflow happens rarely, only when a node has sent many messages that have not yet been consumed. This type of fan-out problem is much easier to solve than fan-in. Figure 9 shows this distributed message queue as implemented by the pull messaging primitives in the Fast Messages library. Each processor holds a tail pointer for its distributed message queue in a local memory location. Message transfer involves

[4] The CP-MB fallback corresponds to context creations and scheduling for both caller and callee threads.

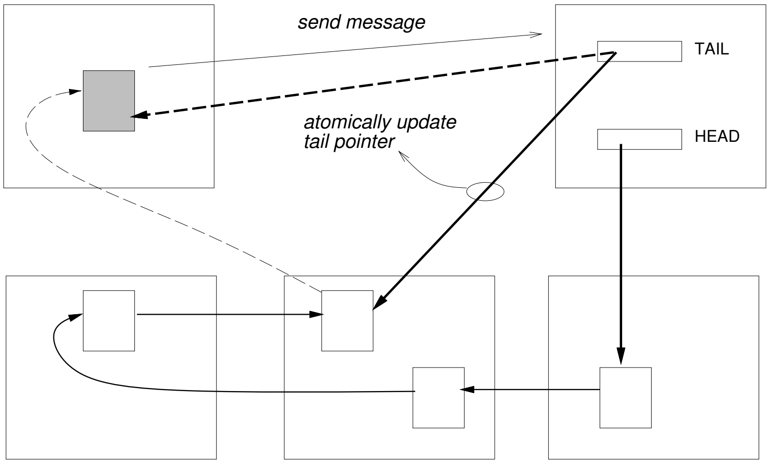

Figure 9: Distributed message queue implementation of pull-messaging. A message send links in the source-buffered message into a distributed queue (by atomically updating the tail pointer). A processor receives by "pulling" messages along its head pointer.

four steps: First, the source writes its message data into a local buffer and swaps a pointer to the buffer with the contents of the tail pointer on the destination (Step 1). This makes the local buffer the last entry in the destination's distributed message queue. The source then stores a pointer to the buffer into the memory address swapped out of the tail pointer (Step 2), linking the previous end of the queue to the local buffer. To receive, the destination uses prefetch operations to retrieve the message data (Step 3). On completion of the transfer it invalidates the source buffer (Step 4), permitting its reuse. There are no races between steps 1 and 2 at the source and step 3 at the destination, because the destination can detect the end of the distributed message queue and simply wait (or go and do some other work) until more messages have been added.

4.3.1 Robustness to Output Contention

To examine the performance advantages of pull-based messaging versus push-based messaging, we consider how the messaging overheads vary for four multi-party traffic distributions which load the network and generate output contention spanning from none to extreme. *all-to-all* is a permutation pattern which generates moderate network contention but no output contention (fan-in), and is used as a baseline to evaluate the effect of output contention. In *random*, each processor picks a random destination and sends to it. Since several processors may pick the same destination, the pattern generates moderate fan-in. *hotspot* is the same as random, except that one node is designated a hotspot (processor 0) and receives a constant multiple of the traffic of the other nodes (we consider hotspot,x4 and hotspot,x16). The hotspot intensity is proportional to the amount of fan-in. *all-to-one* represents the extreme hotspot situation where all the traffic is sent to a single processor.

For each pattern, we measure the steady-state send and receive overheads with all processors executing a kernel which consists of a fixed size message send, a receive, and a computation part (modeled by a stochastic variable delay). The overheads for push- and pull-based mechanisms are shown in Figure 10 for four message sizes. Note that this is a log-log scale and each point corresponds to the harmonic mean of source overheads over all nodes for a 64-node T3D. The point-to-point costs have been included for comparison. These graphs correspond to a specific compute granularity (13.2μs), but the performance trend remains essentially unchanged as compute grain size and its variance are modified.

Considering the push messaging case first, the effect of network loading is as expected, increasing source overheads for 1024 byte messages from 13.5μs for the point-to-point case to 30μs for the all-to-all case. However, the results show that the push messaging scheme is not robust over output contention. As the

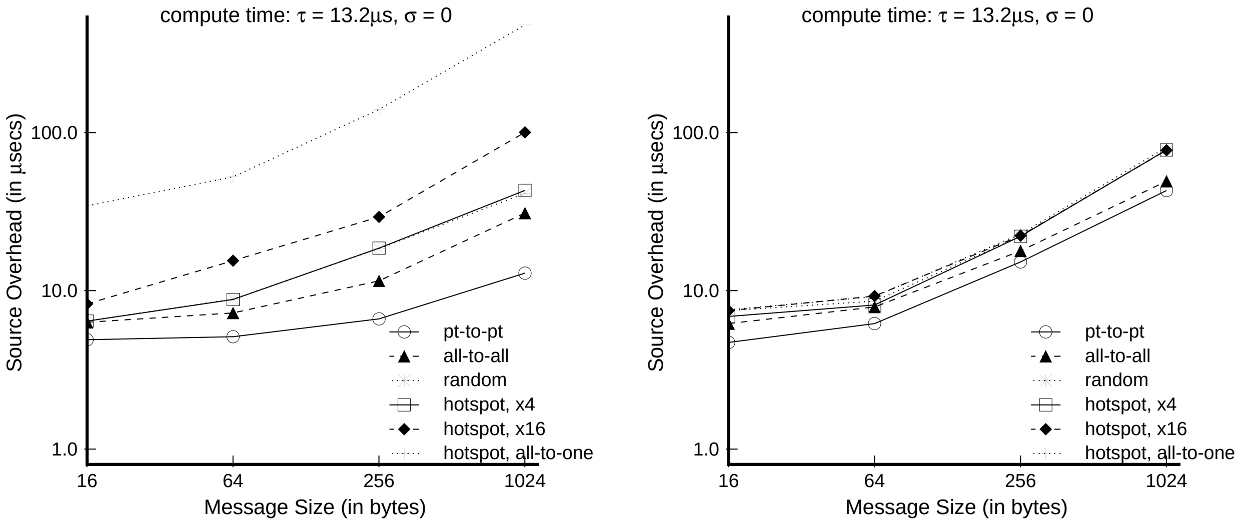

Figure 10: Source overheads for push (**left**) and pull (**right**) versions of messaging on the T3D for four message sizes.

hotspot intensity increases (greater imbalance in message destinations or larger messages), source overheads increase by up to an order of magnitude as compared to the all-to-all case. For the message size considered above, the source overheads are 95μs for the hotspot,x16 pattern (3:1 degradation), and 455μs for the all-to-one pattern (15:1 degradation). This performance loss is due to classic hot spot saturation [36] which backs up the network, slowing the injection of all messages into the network. Such hot spot contention will occur in any messaging layer which eagerly pushes messages to their destination.

In contrast, pull messaging can eliminate output contention by enqueuing messages onto a distributed message queue and moving data to the destination only upon request. As seen in Figure 10, pull messaging mechanisms exhibit robust performance irrespective of the traffic pattern as indicated by the close clustering of the curves in the graph. As the intensity of the hot spot increases, there is no noticeable degradation in performance (the linear increase in send overheads with message size occurs because of the source copy). The major reason for pull messaging's robustness is the benefit of distributed queueing and lazy data movement. Distributed queuing can sustain large fan-in into hot nodes, supporting even all-to-one communication for extremely large machines with no degradation in performance. The hot node only "pulls" in message data as required, matching the data transmission rate to that feasible for the node. Thus, there is neither excessive data traffic nor enqueue traffic to interfere with other communication. In fact, we have found it effectively impossible to generate traffic loads which cause any input or output contention using pull-based messaging.

4.3.2 Costs of Pull-based Messaging

Source and destination overheads for the push (fetch-and-increment based) and pull (atomic swap based) messaging implementations are shown in Table 5. The pull mechanisms are competitive for small messages, but pulling the message data results in increased overheads as compared to push messaging for messages larger than 64 bytes. This overhead difference is largely due to the high cost of interaction with the T3D prefetch queue: 0.1μs (15 cycles) to issue a single word fetch, and 0.15μs (20 cycles) to extract a word from the prefetch queue. Note however, that simple architectural improvements [30] can make the costs of pull messaging competitive with push messaging for all message sizes.

Implementation		Message size (in bytes)						
		16	32	64	128	256	512	1024
Push Messaging	Src	1.84	1.85	2.03	2.84	3.53	8.26	14.65
(Fetch-and-increment based)	Dest	0.73	0.73	0.73	0.73	0.73	0.73	0.73
Pull Messaging	Src	1.49	1.56	1.67	1.93	2.49	4.13	6.31
(Atomic-swap based)	Dest	1.30	1.70	2.46	5.49	9.53	17.17	30.18

Table 5: Base source and destination overheads of the push (fetch-and-increment based) and pull (atomic-swap based) messaging schemes on the T3D. (all times in μs).

4.3.3 Summary: Pull-based Messaging

Pull messaging mechanisms enable the generated code to deliver communication performance robust over irregular, unbalanced traffic, and unsynchronized processor communication and computation phases. A distributed message queue is a form of source buffering which eliminates the detrimental coupling between sender and receiver, while lazy receiver-initiated data transfer eliminates output contention by matching the data transfer rate to that sustainable by the node. Exploiting hardware for remote memory access, synchronization, and prefetch which is available on the T3D and present on several current and likely future machines, a pull-messaging implementation can deliver performance comparable to push-based messaging.

The addition of pull messaging to the runtime primitives described in Sections 4.1 and 4.2 produces the Base+Pull and Base+Stack+Pull runtime versions. Note that these versions perform *all* communication using the pull-based primitives. While there are benefits of using a hybrid push- and pull-based messaging approach (e.g., for regular traffic with large messages), it is currently not used in our compiler because the analysis required to automate this selection is complicated, requiring analysis of the temporal nature of asynchronous communication operations in a multithreaded setting. Additionally, by first quantifying the performance of an all pull-messaging implementation, we can ascertain the situations in which a hybrid scheme would be beneficial. Section 5 presents the performance impact of pull-messaging mechanisms, evaluated using the microkernel and application programs. As we shall see, pull messaging is clearly beneficial in situations where traffic patterns are irregular and unpredictable. Even in cases where communication overhead is critical to performance, the robustness of pull messaging helps minimize the performance loss.

5 Performance Results

In this section we describe results of experiments quantifying advantages of the runtime mechanisms described in Section 4. All experiments were conducted using the Concert system (compiler and runtime) implementation for the Cray T3D. Two sets of experiments are described. The first uses the synthetic microkernel (Section 3) to perform a controlled study of the individual and aggregate performance impact of the runtime mechanisms, while the second characterizes their aggregate impact on two large irregular application programs.

5.1 Microkernel Performance

We measure the performance of the microkernel program corresponding to the four runtime versions described earlier. Base (Section 4.1) serves as a baseline and corresponds to the use of general-purpose bundled runtime mechanisms. Base+Stack (Section 4.2) and Base+Pull (Section 4.3) examines the individual contributions from the hybrid stack-heap execution and pull-messaging mechanisms respectively. Base+Stack+Pull represents the aggregate effect of both mechanisms.

The performance of these four runtime versions is measured on a 32-processor Cray T3D for various values of the microkernel parameters 3(summarized in Table 6). The metric of interest is the execution efficiency which corresponds to the fraction of useful (user) work in the total execution time. We describe results of three experiments. First, we examine the individual performance advantages of hybrid stack-heap execution and pull-messaging as the compute granularity and fraction of remote accesses are varied with the

Parameter	Value(s)
Percentage of Remote Accesses	0%, 20%, 40%, 60%, 80%, 100%
Distribution of Remote Accesses	all-to-all, random, hotspot(4x), hotspot(16x)
Compute Granularity (τ)	10μs, 50μs, 100μs, 500μs, 1000μs, 5000μs
Variation in Compute Granularity (σ)	0.0τ, 0.5τ, 1.0τ

Table 6: Values of microkernel parameters used in the experiments.

other two parameters kept constant. The second experiment examines the advantages of pull messaging in more detail by examining the effect of variations in compute granularity and distribution of remote accesses. Finally, we characterize constant efficiency regions for the runtime versions in terms of two parameters: the compute granularity and fraction of local accesses, demonstrating that the runtime mechanisms increase the region of parameter space where high performance is achievable.

5.1.1 Experiment 1: Impact of Runtime Mechanisms

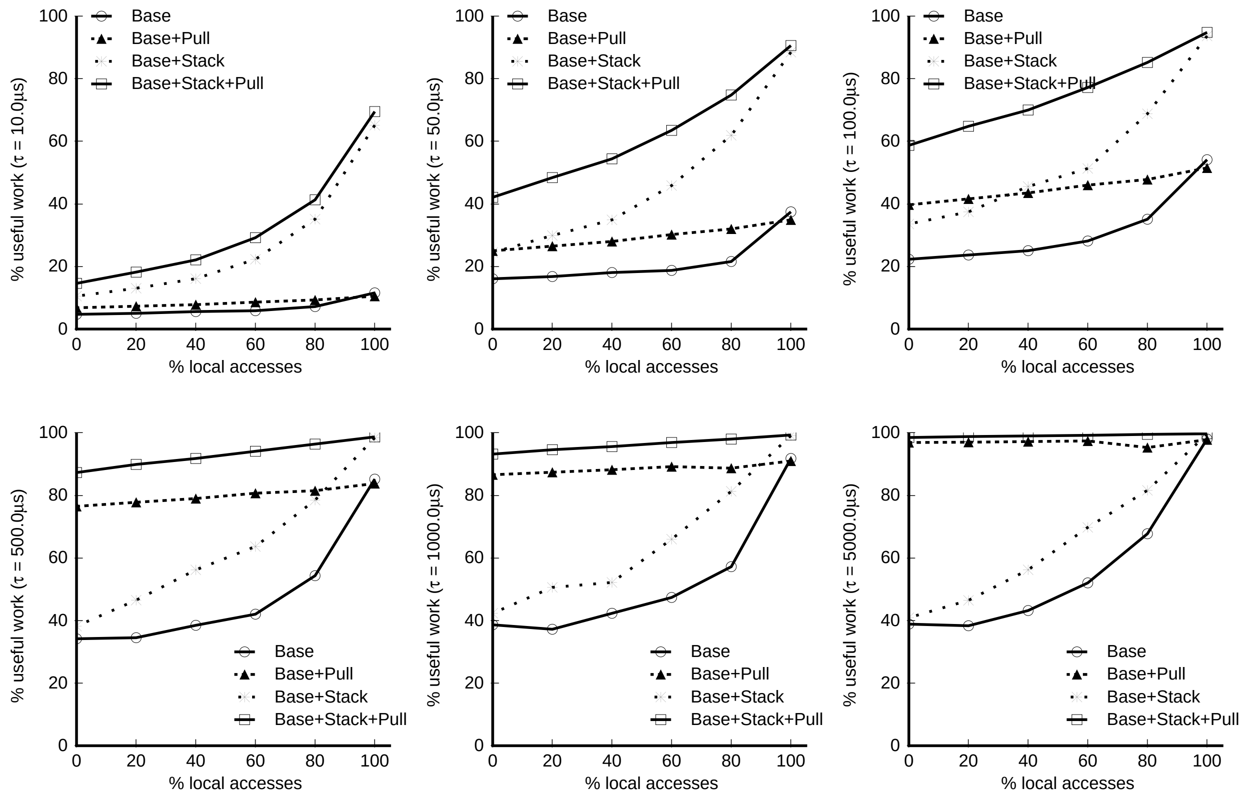

Figure 11: Impact of runtime mechanisms: microkernel execution efficiency for four runtime versions as a function of the percentage of local accesses, parameterized with the compute granularity (τ).

Figure 11 shows the microkernel performance for four versions of the runtime as the percentage of local accesses is varied from 0% (all remote) to 100% (all local). The six graphs correspond to different values of the compute granularity. The following conclusions can be drawn from the results:

1. Impact of Hybrid Stack-Heap Execution Model: The performance increment between the Base+Stack and Base versions and between the Base+Stack+Pull and Base+Pull versions, shows that hybrid stack-heap execution mechanisms can *increase execution efficiency by up to 50%*. For a specific compute granularity, the advantages from hybrid execution mechanisms increase with runtime locality, validating our earlier claim that the mechanism enables higher performance by adapting to runtime situations. The mechanism enables high performance for a broader range of compute granularities; for example, 70% efficiency can be achieved with granularities as low as 10μs. Since the runtime can only decrease overheads, the Base scheme catches up for larger compute granularities.

2. Impact of Pull Messaging: The performance increment between the Base+Pull and Base versions and between the Base+Stack+Pull and Base+Stack versions, shows that pull-based messaging *can increase execution efficiency by up to 50%* by providing increased decoupling between the sender and receiver processors. This decoupling becomes less important as the amount of communication decreases, so the advantages from pull messaging decrease with increased runtime locality. For a fixed fraction of local accesses, the advantages from pull-messaging increase with increases in compute granularity. This behavior arises from the microkernel structure where increasing compute granularity also increases the likelihood that a destination processor will be unresponsive to service a communication request. Performance degrades for push messaging because of sender-receiver coupling, while pull messaging eliminates this coupling to deliver robust high performance.

3. Aggregate Impact of the Runtime Mechanisms: In addition to the specific trends described above, one trend can be observed across all the graphs. The hybrid stack-heap execution model is effective in improving the efficiency of computations with small granularity and a high percentage of local accesses. The pull messaging mechanisms targets the other end of the parameter space, improving efficiency for large granularity computations which have a significant fraction of remote accesses. Thus, for the microkernel, the two mechanisms enable computations with significantly smaller compute granularity and larger amounts of communication to be efficiently executed.

5.1.2 Experiment 2: Robustness of Pull-Messaging

Figure 12 shows the performance of the Base+Stack and Base+Stack+Pull versions as the distribution of remote accesses is varied. Each row of plots is parameterized with different values of the standard deviation of compute granularity. The percentage of remote accesses is fixed at 80%. In all the plots, varying the compute granularity has only a second-order effect when compared with the effect of the traffic patterns. As can be seen from Figure 12, the performance of the Base+Stack+Pull version is robust over output contention, showing only the effects of an increase in the computation's critical path arising from the hotspot processors having to do more work. This robust behavior is in contrast to the Base+Stack version whose performance is unpredictable and poor both because of unresponsive receivers and because of unbalanced traffic distributions. These effects interact with the push-messaging's sender-receiver coupling, decreasing performance because of increased overheads at senders waiting to send messages.

5.1.3 Experiment 3: Regions of Acceptable Efficiency

Figure 13 shows the parameter space regions, defined by compute granularity and the fraction of local accesses, where microkernel execution for the four runtime versions achieves $\geq 70\%$ efficiency. Note that the compute granularity is plotted on a log scale. The $\geq 70\%$ region with the Base version is showed in black at the upper-right corner of the graph, requiring compute granularities greater than 300μs and greater than 80% local accesses.[5] The addition of the hybrid stack-heap mechanisms increases the acceptable efficiency region to include the hashed portions, marked +Stack and +Stack or +Pull. This enables 70% execution efficiency with granularities as low as 10μs and fewer local accesses (60%). The addition of the pull communication mechanisms increases the region further (shown by the hashed portion marked +Pull) allowing microkernel executions with even 0% local accesses to achieve $\geq 70\%$ efficiency. Finally, the two mechanisms together add

[5] The 70% efficiency region requires a large fraction of local accesses even for relatively large compute granularities because the sender-receiver coupling worsens as compute granularity is increased, increasing the likelihood that a receiver is unresponsive.

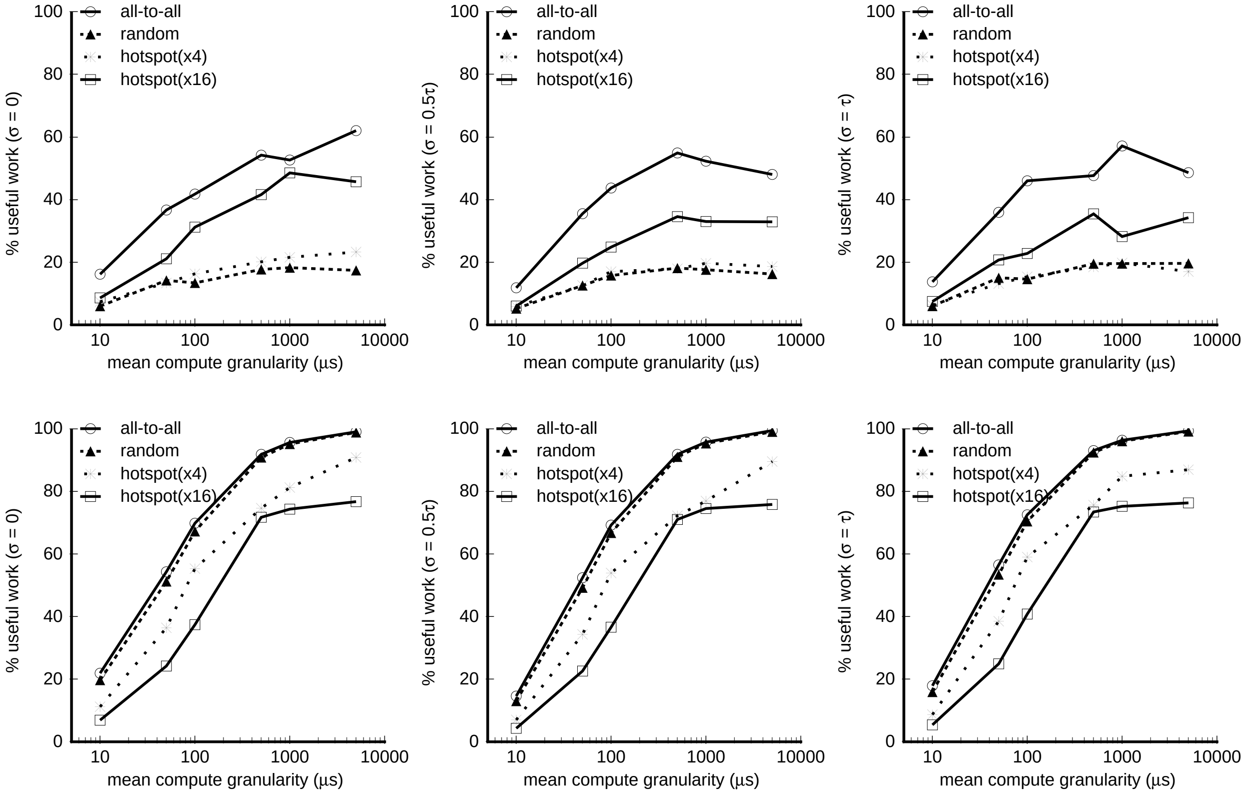

Figure 12: Robustness of pull-messaging: microkernel execution efficiency of Base+Stack (top) and Base+Stack+Pull (bottom) versions for several output-contention traffic patterns as compute granularity (τ) is varied (80% of accesses are to remote objects). Each row of plots correspond to variations in the compute granularity (σ).

the solid gray region marked +Pull+Stack, allowing 70% execution efficiency to be achieved with granularities as low as $10\mu s$ (for 100% local accesses) and $150\mu s$ (for 0% local accesses).

Figure 13 clearly demonstrates that the two runtime mechanisms increase the flexibility of achieving high performance, enabling even computations with smaller compute granularities and a larger fraction of remote accesses to achieve high efficiency levels. The plot also highlights the relative merits of the two runtime mechanisms. Hybrid stack-heap execution enables small granularity threads to be executed efficiently as long as the execution has sufficient runtime locality. On the other hand, pull-messaging mechanisms increase the locality region which can be executed efficiently for a particular compute granularity.

5.2 Application Programs

In this section, we examine how the performance advantages observed on a synthetic microkernel translate to two large irregular application programs expressed using dynamic multithreading. For each application, we first briefly describe the application program and its code structure, and then its performance corresponding to the four runtime versions. We demonstrate that the mechanisms enable a high-level dynamic multithreading expression of these applications to achieve performance comparable to explicitly optimized versions.

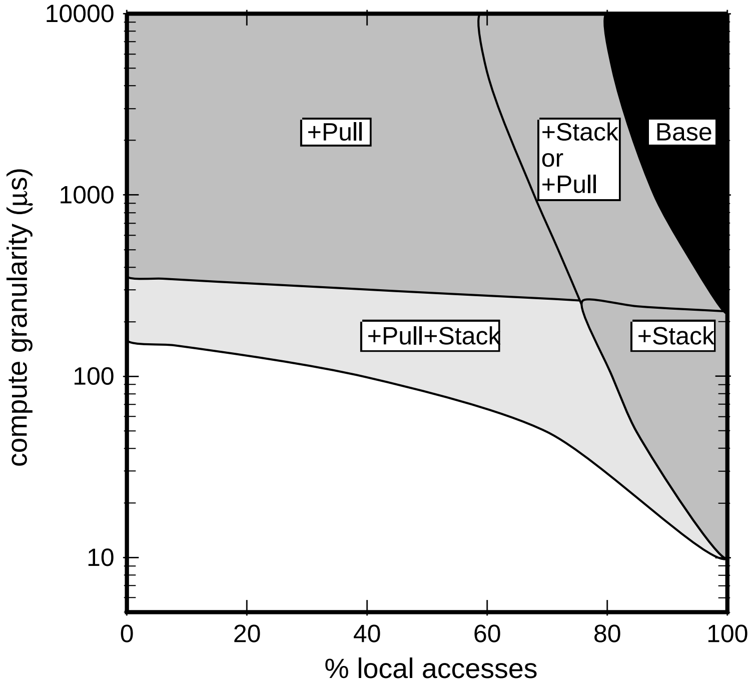

Figure 13: Regions of the compute granularity and fraction of local accesses parameter space where the microkernel achieves $\geq 70\%$ efficiency for the four runtime versions. The efficiency region achieved by only the Base runtime is shown in black; adding the mechanisms increases this region (the increments correspond to the appropriately labeled hashed and gray portions).

5.2.1 Hierarchical Radiosity

The radiosity method computes the global illumination in a scene containing diffusely reflecting surfaces by expressing the radiosity of an elemental surface patch as a linear function of the radiosities of all other patches, weighted by the distance and occlusion between the patches. We use an algorithm due to Hanrahan [23], modeled after hierarchical N-body methods. The method starts with the initial patches comprising the scene and computes light transport between pairs of patches, hierarchically subdividing each patch as needed to ensure accuracy. Each patch maintains interaction lists of potentially visible neighbors. Computation proceeds in iterations, where for each patch, we compute its radiosity due to all patches on its interaction list, subdividing it or the other patches hierarchically as required. Subdivided patches acquire their own interaction lists and contribute a weighted term to their parent's radiosity. The visibility calculation dominates the computation time.

Our parallel algorithm is derived from the radiosity application in the SPLASH-2 benchmark suite [48]. There are three levels of parallelism in each iteration: across all input patches, across child patches of a subdivided patch, and across neighbor patches stored in the interaction list. Unlike the SPLASH-2 code which explicitly creates tasks and manages task queues, our Concert program merely exposes the visibility calculations as concurrent logical threads. Each thread has a structure similar to the microkernel: it first accesses data from potentially remote patches, and then computes upon them. Our implementation caches some of these accesses in software so as to reduce the total communication required by the program. This application exhibits irregularity in how deep each input patch can be refined, in how many interactions need to be computed for each patch, and even how long a particular interaction calculation takes. Additionally, the work across the iterations is not uniform with $> 90\%$ of the computation occuring in the first iteration.

Radiosity: Impact of Runtime Mechanisms Figure 14 shows code performance for the four runtime versions, expressed as speedup with respect to the fastest single node execution time. The mechanisms make

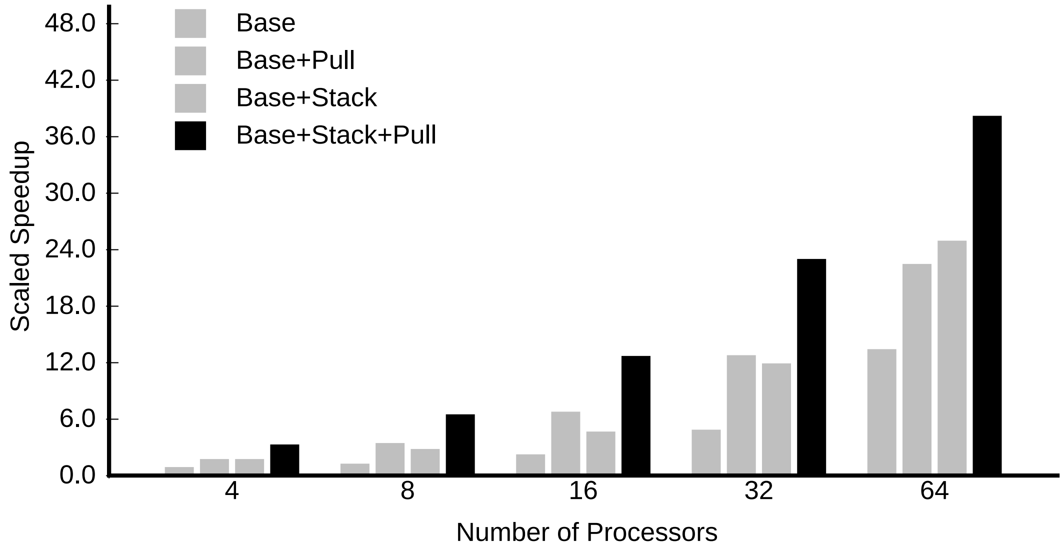

Figure 14: Speedup achieved by the radiosity code for four runtime versions: base, base+pull, base+stack, and base+pull+stack, for various T3D configurations. The BF refinement threshold was set to 0.015 and the area refinement threshold to 2000 units. Pull messaging and hybrid stack-heap execution model each improve performance by up to two times.

good speedups possible even with a fine-grained dynamic multithreading specification. Hybrid stack-heap execution and pull messaging individually improve performance over the Base case by up to 2 times. For example, in the 32-processor case, these mechanisms enable the speedup to increase from 6x to 12x. Hybrid stack-heap execution causes visibility calculations to complete on the stack when the patches are co-located (or found in the software cache), unwinding to a heap context otherwise. The outer thread responsible for computing all interactions for a particular patch unwinds to the heap when one of the interaction threads itself suspends. Pull-messaging helps provide robust communication performance in the presence of unpredictable communication which is unsynchronized with ongoing computation and dictated by patch placement. The two mechanisms together have a cumulative effect across the range of processors, increasing the achieved speedup by up to 2 times over either of the mechanisms used alone, and up to 4 times over the **Base** version.

Radiosity: Performance in Context To place the above speedups in context, we compared our absolute sequential run time with that achieved by the SPLASH-2 version of the program. Our sequential performance is within 20% of the SPLASH-2 code on a SparcStation 20 and within 80% of it on a single T3D node. The performance degradation on the T3D arises from the inability of the generated code to effectively exploit the small L1 cache (along with the lack of an L2 cache) on the DEC Alpha 21064. Our parallel performance can be compared with previously published application speedup numbers (obtained from [43]) based on a hand-tuned implementation running on the DASH [31], a cache-coherent shared-memory machine. Although, the T3D and the DASH are architecturally quite different — the DASH has hardware support for cache-coherent shared memory and has relatively faster communication (in terms of processor clocks), while the T3D is a distributed memory machine requiring an order of magnitude higher cost to access remote data — our implementation achieves a speedup of 23 on 32 T3D processors which compares well with a speedup of 26 on 32 processors of the DASH.

5.2.2 Molecular Dynamics: IC-Cedar

IC-CEDAR is a medium-sized protein molecular dynamics (MD) simulation program that models the motion of individual protein and surrounding solvent atoms using Newton's equations of motion. The application is challenging to parallelize on distributed memory machines because of its irregular computation and communication structure. IC-CEDAR is based on CEDAR [24], a sequential MD program written in C and FORTRAN. Simulations are carried out in discrete time steps with four computation phases: calculation of neighbor lists, computation of bonded and non-bonded forces, integration of motion, and SHAKE (an iterative coordinate correction phase). The neighbor list phase constructs a list of interacting atoms for each atom based on a cutoff radius and requires irregular data accesses. The force-calculation phase includes both short-range bonded forces, involving groups of 3-4 atoms, and long-range Coulombic and Lennard-Jones forces involving atom pairs determined by each atom's neighbor list. The force calculation on each group of 2 to 4 atoms requires first reading the position fields and then updating the force fields of each atom.

Our expression of the above algorithm makes use of a spatial partitioning of atoms along with software caching of data accesses to reduce the amount of required communication. Additionally, each major phase is separated into a subphase which fetches all remote data values (implemented as a single thread which performs several data accesses to potentially remote objects), and another subphase which operates against these fetched values. Thus, although IC-CEDAR relies on the dynamic multithreading model to concurrently initiate and synchronize among data accesses to remote objects, it resembles a message-passing program in its explicit separation of communication and computation phases. As we shall see later, this separation has an important ramification on the performance advantages of pull messaging mechanisms.

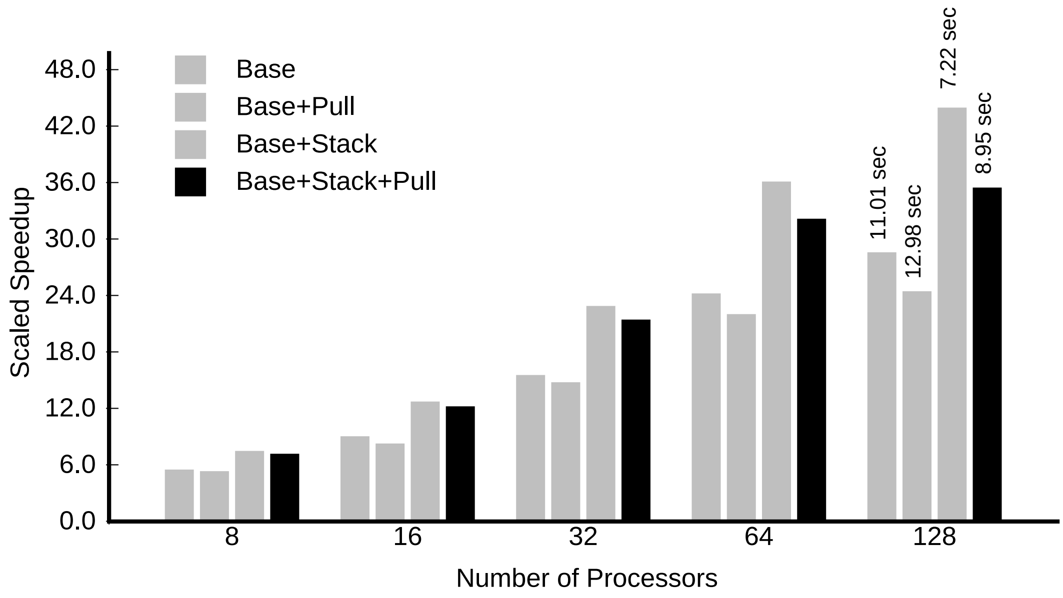

Figure 15: Scaled speedup achieved by four versions of the IC-CEDAR code: **base**, **base+pull**, **base+stack**, and **base+pull+stack** for various T3D configurations. Pull messaging does not yield additional advantages due to the explicit decoupling between communication and computation in the code. Hybrid heap-stack execution improves performance by up to 1.5 times.

IC-CEDAR: Impact of Runtime Mechanisms Figure 15 shows the code performance for four runtime versions on the myoglobin data set, expressed as speedup measured with respect to the compute portion of the 8 node execution time. As with the Radiosity application, the mechanisms enable good speedups to be achieved despite using a fine-grained natural application specification. Individually, hybrid stack-heap execution improves performance by up to 1.5 times. For example, for the 64 processor case, Base gets a speedup of 24x in contrast to Base+Stack which achieves a speedup of 36x. This mechanism yields benefit both for accesses which complete in the local cache (these complete on the stack) and for remote accesses which complete without blocking for concurrency control reasons (these complete in the message handler avoiding thread creation). However, unlike the Radiosity application, pull messaging yields little additional advantages over push messaging mechanisms. Communication in IC-CEDAR involves larger message sizes for which pull messaging incurs higher per-message overheads. Additionally, the explicit separation of communication and computation phases is well-suited to push messaging because all processors are responsive and participating to complete the communication phase. Thus, this is a situation where a hybrid scheme consisting of both push and pull messaging can yield benefit. However, as the difference in absolute runtimes for the 128-processor case shows, this degradation is less severe than the speedup bars indicate.

IC-CEDAR: Performance in Context To place the above speedups in context, we measured the absolute sequential times for the same data set with for the sequential C/FORTRAN version of CEDAR and a one-processor run of IC-CEDAR. On a SparcStation 20, the sequential performance of IC-CEDAR is within 75% of the performance achieved by the C/FORTRAN version of CEDAR. To compare our parallel performance, we looked at the performance achieved by CHARMM [26], a highly tuned SPMD implementation built on top of sequential FORTRAN code with manually inserted calls to the CHAOS runtime library responsible for communication and load-balancing. Despite differences in simulation models, both IC-CEDAR and CHARMM exhibit similar computation and communication behavior, and perform similar amounts of work. For similar myoglobin data sets on 64 nodes on the T3D, IC-CEDAR achieves comparable absolute performance coming within 33% of the CHARMM performance (23.49s versus 18.46s). The remaining is very likely a result of differences in quality of code produced by the Fortran and C compilers.

6 Discussion and Related Work

We have described two runtime mechanisms which overcome the excessive thread management and communication overheads for dynamic computations resulting both from the lack of compile-time information and the unbalanced, unsynchronized nature of communication and computation. While these mechanisms were developed in the context of fine-grained concurrent object-oriented languages, we believe they are applicable to other multithreaded programming models where the compute and communicate operations of dynamically created threads cannot be explicitly managed and scheduled statically. The biggest advantage of the hybrid stack-heap execution and pull-based messaging mechanisms are that they enable applications exhibiting a much larger range of compute granularities and communication characteristics to be executed with high efficiency. As a compiler target, these mechanisms produce robust, high performance despite imperfect compile-time information, irregular computation structure, and even poor programmer data distribution decisions.

Our work is related to previous research which has looked at efficient support for thread management and communication; however, the primary distinction is its emphasis on delivering high performance for dynamic, fine-grained applications without specialized hardware support.

Efficient Thread Management Specialized hardware approaches have suggested providing multiple hardware contexts [1, 13] and integrating thread creation with message reception [42, 33]. In contrast, hybrid stack-heap execution can be supported entirely on stock hardware benefiting from advances in commodity microprocessor architectures.

Portable multithreading runtime systems such as Chant [21], Nexus [16] and PORTS [40], built on top of vendor-supported, standardized light-weight thread [27] and communication [15, 44] interfaces, incur large thread and communication overheads (30-50μs) requiring large granularity threads for efficiency. Finer grained threads can be supported by programming systems such as Mentat [20], Cilk [3], COOL [8], and Charm++ [28]. However, such systems typically assume minimal compiler support, relying on the programmer to control thread granularity and mapping for performance. Automating these decisions requires compile-time knowledge of how the execution unfolds; consequently, such systems achieve low efficiency for irregular, dynamic computations.

The close compiler-runtime coupling which characterizes hybrid stack-heap execution is also found in other dynamic, fine-grained programming systems. The Threaded Abstract Machine (TAM) [12] model provides a cost hierarchy, enabling the compiler to manage synchronization, scheduling and storage at the activation-frame level. Our work complements TAM's by providing mechanisms which enable robust performance in spite of statically unpredictable situations. The idea of lazily creating threads as required by runtime situations can also be found in the work on Lazy Task Creation [32] and Leapfrogging [47] in the context of shared-memory machines, and Olden [41], Stacklets [18], and StackThreads [45] in the context of distributed-memory machines. The former two schemes were developed in the context of parallel languages with explicit futures [22], and allow stealing previously deferred stack frames to adaptively control execution granularity and work distribution. Hybrid stack-heap execution differs in that it allows eager work distribution. The latter three schemes all allocate new threads whenever the current one blocks due to a remote operation. Thus, unlike hybrid stack-heap execution, they use the same mechanism to support both work distribution and latency-hiding and cannot optimize their code for either situation.

Efficient Communication Previous research has largely focused on reducing point-to-point messaging costs and paid less attention to multi-party communication characteristics. Hardware approaches [13, 4] have argued for a closer integration of the network interface with the processor. Software approaches have investigated the design of messaging layers with minimal kernel interaction [14, 6], and the active messages [46] approach of offloading all but the essential operations from the messaging layer. These approaches alone are inadequate to prevent performance degradation arising from sender-receiver decoupling in the presence of unbalanced and unsynchronized communication traffic.

Two recent approaches address the above shortcoming. Callahan [7] proposes a network interface which limits the number of outstanding messages between pairs of processors to improve network performance even for unbalanced traffic. Pull messaging achieves the same effect with general-purpose hardware: remote memory access and synchronization hardware is available in several current parallel machines. Brewer [5] has looked at software solutions for improving performance for all-pairs communication traffic on the CM-5. However, their solutions consider only permutation traffic patterns, requiring all processors to cooperate for achieving high performance. In contrast, our pull-based messaging scheme provides robust performance for asynchronous and varied communication behaviors.

7 Conclusions

We have described two runtime mechanisms which can enable stock scalable parallel machines to support dynamic fine-grained multithreaded computations efficiently, enabling them to achieve performance comparable to hand-tuned approaches. Hybrid stack-heap execution dynamically coalesces fine-grained logical threads into larger-grained physical threads, overcoming lack of compile-time information which otherwise increases thread management overheads. Closer coupling with the compiler enables logical threads to be optimistically executed on the caller's stack; threads are lazily created as required by runtime situations. Pull messaging provides performance robust over unbalanced and unsynchronized communication traffic. The shared address space hardware on the Cray T3D is used to implement a distributed message queue with lazy receiver-initiated data transfer which improves performance by providing sender-receiver decoupling.

Performance studies for a synthetic compute-communicate microkernel and two irregular applications based on a Cray T3D implementation demonstrate that these mechanisms significantly enhance the range of computation granularities and communication characteristics which can be efficiently supported. In particular, each increases microkernel execution efficiency by up to 50 percentage points; hybrid stack-heap execution is effective for executions with small compute granularity and high runtime locality, while pull messaging improves performance for executions with low runtime locality. In addition, the mechanisms enable 70% microkernel execution efficiency with compute granularities of 10μs (for 100% local accesses) and 150μs (for 0% local accesses); in contrast, reaching the 70% efficiency level originally required the microkernel to have compute granularity larger than 300μs and greater than 80% local accesses. Furthermore, measurements for two irregular applications – hierarchical radiosity and macromolecular dynamics – show that performance is comparable to that achieved by explicitly optimized versions, demonstrating that expressing programs using dynamic multithreading need not compromise on performance.

These mechanisms were developed in the context of the Illinois Concert system – an implementation platform for executing fine-grained concurrent object-oriented languages on distributed memory machines. Ongoing work builds upon the fine-grained mechanisms described here by exploiting application information to proactively control the computation, ensuring increased effectiveness from the mechanisms.

Acknowledgements

The authors would like to thank Xingbin Zhang, Julian Dolby, and other members of the Concurrent Systems Architecture Group for helpful discussions and help building the Concert System.

The research described in this paper was supported in part by NSF grants CCR-92-09336, MIP-92-23732 and CDA-94-01124, ONR grants N00014-92-J-1961 and N00014-93-1-1086, NASA grant NAG 1-613, and a special-purpose grant from the AT&T Foundation. Andrew Chien is supported in part by NSF Young Investigator Award CCR-94-57809. Vijay Karamcheti is supported in part by an IBM Computer Sciences Fellowship. The experiments described in this paper were conducted on the Cray T3Ds at the Pittsburgh Supercomputing Center and the Jet Propulsion Laboratory.

References

[1] Agarwal, A., Kubiatowicz, J., Kranz, D., Lim, B.-H., Yeung, D., D'Souza, G., and Parkin, M. Sparcle: An evolutionary processor design for large-scale multiprocessors. *IEEE Micro 13*, 3 (June 1993), 48–61.

[2] Barnes, J., and Hut, P. A hierarchical O(N log N) force calculation algorithm. Tech. rep., The Institute for Advanced Study, Princeton, New Jersey, 1986.

[3] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., Shaw, A., and Zhou, Y. Cilk: An efficient multithreaded runtime system. In *Proceedings of Principles and Practice of Parallel Programming* (1995).

[4] Borkar, S., Cohn, R., Cox, G., Gross, T., Kung, H. T., Lam, M., Levine, M., Moore, B., Moore, W., Peterson, C., Susman, J., Sutton, J., Urbanski, J., and Webb, J. Supporting systolic and memory communication in iWarp. In *Proceedings of the 17th International Symposium on Computer Architecture* (1990), IEEE Computer Society, pp. 70–81.

[5] Brewer, E. A., and Kuszmaul, B. C. How to get good performance from the CM-5 data network. In *Proceedings of the International Parallel Processing Symposium* (1994), pp. 858–867.

[6] C. A. Thekkath, H. L., and Lazowska, E. D. Separating data and control transfer in distributed operating systems. In *Proceedings of the Sixth Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)* (1994).

[7] Callahan, T., and Goldstein, S. NIFDY: A low overhead, high throughput network interface. In *Proceedings of the International Symposium on Computer Architecture* (1995).

[8] Chandra, R., Gupta, A., and Hennessy, J. L. Data locality and load balancing in COOL. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (1993).

[9] Chien, A., Karamcheti, V., and Plevyak, J. The Concert system—compiler and runtime support for efficient fine-grained concurrent object-oriented programs. Tech. Rep. UIUCDCS-R-93-1815, Department of Computer Science, University of Illinois, Urbana, Illinois, June 1993.

[10] Chien, A., and Reddy, U. ICC++ language definition. Concurrent Systems Architecture Group Memo, Also available from http://www-csag.cs.uiuc.edu/, February 1995.

[11] Chien, A. A. *Concurrent Aggregates: Supporting Modularity in Massively-Parallel Programs.* MIT Press, Cambridge, MA, 1993.

[12] Culler, D., Sah, A., Schauser, K. E., von Eicken, T., and Wawrzynek, J. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages an Operating Systems* (1991), pp. 164–75.

[13] Dally, W. J., Chien, A., Fiske, S., Horwat, W., Keen, J., Larivee, M., Lethin, R., Nuth, P., Wills, S., Carrick, P., and Fyler, G. The J-Machine: A fine-grain concurrent computer. In *Information Processing 89, Proceedings of the IFIP Congress* (August 1989), pp. 1147–1153.

[14] Druschel, P., and Peterson, L. L. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of Fourteenth ACM Symposium on Operating Systems Principles* (December 1993), ACM SIGOPS, ACM Press, pp. 189–202.

[15] Forum, M. P. I. The MPI message passing interface standard. Tech. rep., University of Tennessee, Knoxville, April 1994.

[16] Foster, I., Kesselman, C., Olson, R., and Tuecke, S. Nexus: An interoperability layer for parallel and distributed computer systems. Tech. Rep. Version 1.3, Argonne National Laboratory, Dec. 1993.

[17] Freeh, V. W., Lowenthal, D. K., and Andrews, G. R. Distributed filaments: Efficient fine-grain parallelism on a cluster of workstations. In *First Symposium on Operating Systems Design and Implementation* (Monterey, CA, Nov. 1994), pp. 201–212.

[18] Goldstein, S. C., Schauser, K. E., and Culler, D. Lazy threads, stacklets, and synchronizers: Enabling primitives for parallel languages. In *Proceedings of POOMA'94* (1994).

[19] Greengard, L., and Rokhlin, V. A fast algorithm for particle simulations. *Journal of Computational Physics 73* (1987), 325–48.

[20] Grimshaw, A. Easy-to-use object-oriented parallel processing with Mentat. *IEEE Computer 5*, 26 (May 1993), 39–51.

[21] Haines, M., Cronk, D., and Mehrotra, P. On the design of Chant: A talking threads package. In *Proceedings of Supercomputing'94* (November 1994), pp. 350–359.

[22] Halstead Jr., R. H. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems 7*, 4 (October 1985), 501–538.

[23] Hanrahan, P., Salzman, D., and Aupperle, L. A rapid hierarchical radiosity algorithm. *Computer Graphics (Proc Siggraph) 25*, 4 (July 1991), 197–206.

[24] Hermans, J., and Carson, M. Cedar documentation. Unpublished manual for CEDAR, 1985.

[25] Horwat, W., Chien, A., and Dally, W. Experience with CST: Programming and implementation. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation* (1989), ACM SIGPLAN, ACM Press, pp. 101–9.

[26] Hwang, Y.-S., Das, R., Saltz, J., Brooks, B., and Hodošček, M. Parallelizing molecular dynamics programs for distributed memory machines: An application of the CHAOS runtime support library. Tech. Rep. CS-TR-3374 and UMIACS-TR-94-125, University of Maryland, Nov. 1994. to appear in IEEE Computational Science and Engineering.

[27] IEEE. Thread extensions for portable operating systems. (Draft 7), Feb. 1992.

[28] Kale, L. V., and Krishnan, S. CHARM++: A portable concurrent object oriented system based on C++. In *Proceedings of OOPSLA'93* (1993), pp. 91–108.

[29] Karamcheti, V., and Chien, A. Concert – efficient runtime support for concurrent object-oriented programming languages on stock hardware. In *Proceedings of Supercomputing'93* (1993).

[30] Karamcheti, V., and Chien, A. A. A comparison of architectural support for messaging on the TMC CM-5 and the Cray T3D. In *Proceedings of the International Symposium on Computer Architecture* (1995).

[31] Lenoski, D., and et al. The Stanford DASH Multiprocessor. *IEEE Computer* (Mar 1992), 63–79.

[32] Mohr, E., Kranz, D., and Halstead Jr., R. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems 2*, 3 (July 1991), 264–280.

[33] Nikhil, R. S., Papadopoulos, G. M., and Arvind. *T: A multithreaded massively parallel architecture. In *The 19th Annual International Symposium on Computer Architecture* (1992), Association for Computing Machinery, pp. 156–167.

[34] Numrich, R. W. The Cray T3D address space and how to use it. Tech. rep., Cray Research, Inc., August 1994.

[35] Pakin, S., Lauria, M., and Chien, A. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing* (December 1995).

[36] Pfister, G. F., and Norton, V. A. Hot spot contention and combining in multistage interconnection networks. *IEEE Transactions on Computers C-34*, 10 (October 1985), 943–948.

[37] Plevyak, J., and Chien, A. A. Precise concrete type inference of object-oriented programs. In *Proceedings of OOPSLA'94, Object-Oriented Programming Systems, Languages and Architectures* (1994), pp. 324–340.

[38] Plevyak, J., Karamcheti, V., Zhang, X., and Chien, A. A hybrid execution model for fine-grained languages on distributed memory multicomputers. In *Proceedings of Supercomputing'95* (1995).

[39] Plevyak, J., Zhang, X., and Chien, A. A. Obtaining sequential efficiency in concurrent object-oriented programs. In *Proceedings of the ACM Symposium on the Principles of Programming Languages* (January 1995), pp. 311–321.

[40] PORTS Consortium. The PORTS0 interface. Tech. Rep. Version 0.3, Argonne National Laboratory, January 1995.

[41] Rogers, A., Carlisle, M., Reppy, J., and Hendren, L. Supporting dynamic data structures on distributed memory machines. *ACM Transactions on Programming Languages and Systems* (1995).

[42] Sakai, S., Yamaguchi, Y., Hiraki, K., Kodama, Y., and Yuba, T. An architecture of a dataflow single chip processor. In *International Symposium on Computer Architecture* (1989).

[43] Singh, J. P., Gupta, A., and Levoy, M. Parallel visualization algorithms: Performance and architectural implications. *IEEE Computer 27*, 7 (July 1994), 45–56.

[44] Sunderam, V. S. PVM: A framework for parallel distributed computing. *Concurrency, Practice and Experience 2*, 4 ([12] 1990), 315–340.

[45] Taura, K., Matsuoka, S., and Yonezawa, A. StackThreads: An abstract machine for scheduling fine-grain threads on stock CPUs. In *Joint Symposium on Parallel Processing* (1994).

[46] von Eicken, T., Culler, D., Goldstein, S., and Schauser, K. Active Messages: a mechanism for integrated communication and computation. In *Proceedings of the International Symposium on Computer Architecture* (1992).

[47] WAGNER, D. B., AND CALDER, B. G. Leapfrogging: A portable technique for implementing efficient futures. In *Proceedings of the Fifth ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming* (1993), pp. 208–217.

[48] WOO, S. C., OHARA, M., TORRIE, E., SINGH, J. P., AND GUPTA, A. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the International Symposium on Computer Architecture* (1995), pp. 24–36.

[49] YONEZAWA, A., Ed. *ABCL: An Object-Oriented Concurrent System.* MIT Press, 1990. ISBN 0-262-24029-7.