To appear in the Proceedings of the Fifth International Symposium on High-Performance Computer Architecture.

Parallel Dispatch Queue: A Queue-Based Programming Abstraction To Parallelize Fine-Grain Communication Protocols

Babak Falsafi
School of Electrical & Computer Engineering
Purdue University
1285 EE Building
West Lafayette, IN 47907
babak@ecn.purdue.edu

David A. Wood
Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton Street
Madison, WI 53706
david@cs.wisc.edu

Abstract

This paper proposes a novel queue-based programming abstraction, Parallel Dispatch Queue (PDQ), that enables efficient parallel execution of fine-grain software communication protocols. Parallel systems often use fine-grain software handlers to integrate a network message into computation. Executing such handlers in parallel requires access synchronization around resources. Much as a monitor construct in a concurrent language protects accesses to a set of data structures, PDQ allows messages to include a synchronization key protecting handler accesses to a group of protocol resources. By simply synchronizing messages in a queue prior to dispatch, PDQ not only eliminates the overhead of acquiring/releasing synchronization primitives but also prevents busy-waiting within handlers.

In this paper, we study PDQ's impact on software protocol performance in the context of fine-grain distributed shared memory (DSM) on an SMP cluster. Simulation results running shared-memory applications indicate that: (i) parallel software protocol execution using PDQ significantly improves performance in fine-grain DSM, (ii) tight integration of PDQ and embedded processors into a single custom device can offer performance competitive or better than an all-hardware DSM, and (iii) PDQ best benefits cost-effective systems that use idle SMP processors (rather than custom embedded processors) to execute protocols. On a cluster of 4 16-way SMPs, a PDQ-based parallel protocol running on idle SMP processors improves application performance by a factor of 2.6 over a system running a serial protocol on a single dedicated processor.

This work is supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525 and ARPA order no. B550, NSF PYI Award CCR-9157366, NSF Grants MIP-9225097 and MIP-9625558, an IBM graduate fellowship, and donations from A.T.&T. Bell Laboratories, Hewlett Packard, IBM Corporation, and Sun Microsystems. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the U.S. Government. This research also greatly benefited from computing resources purchased by NSF Institutional Infrastructure Grants No. CDA-9623632.

1 Introduction

Clusters of symmetric multiprocessors (SMPs), have emerged as a promising approach to building large-scale parallel machines [23,22,15,14,5]. The relatively high volumes of small- to medium-scale SMP servers make them cost-effective as building blocks. By connecting SMPs using commodity off-the-shelf networking fabric, system designers hope to construct large-scale parallel machines that scale with both cost and performance.

To program these clusters, researchers are studying a variety of parallel programming abstractions. Some of these abstractions—such as shared virtual memory [5]—communicate data at coarse granularity (e.g., a 4-Kbyte page) using conventional high-overhead legacy TCP/IP protocols. Many abstractions, however, rely on low-overhead messaging—as in Active Messages [25]—and employ fine-grain protocols to exchange small amounts of data (e.g., 8~256 bytes) over the network [23,22,15]. Protocol handlers in such systems, typically execute a small number of instructions to move data between the application's data structures and the network message queues, and optionally perform a small amount of computation and book-keeping and send a reply message.

Fine-grain parallel abstractions traditionally targeted uniprocessor-node parallel computers. As such, the protocol handlers either executed on the node's commodity processor along with computation or an embedded processor on the network interface card. Multiple SMP processors, however, increase the demand on fine-grain protocol execution on a node [23,16,14,6]. To maintain the balance between computation and communication, protocol execution performance must increase commensurate to the number of SMP processors.

One approach to increase software protocol performance is to execute protocol handlers in parallel. Legacy stack protocols (e.g., TCP/IP) have long been parallelized to execute on multiple SMP processors [21,1,10,11]. These protocols synchronize and coordinate handler accesses to system resources—e.g., messaging queues, and protocol and application data structures—using software spin-locks. Fine-grain parallel systems, however, have short handler running times. Acquiring and releasing

software locks around individual resources would incur prohibitively high overheads and may result in busy-waiting in these systems, thereby offsetting the gains from parallel handler execution. As such, executing fine-grain protocol handlers in parallel requires efficient synchronization support [15].

Recent research on networking technology for clusters has primarily focused on virtualizing the network interface with no support for fine-grain handler synchronization. Both U-Net [24] and the Virtual Interface Architecture [4] provide multiple pairs of message send/receive queues per node with protected low-overhead access to the network. To avoid fine-grain synchronization, many fine-grain parallel systems [15,22] using these networks partition the node's resources (e.g., memory) among the SMP processors treating each processor as a stand-alone node in a uniprocessor-node parallel computer.

A significant shortcoming of the multiple protocol queues model is that individual processors do not take advantage of the tight coupling of resources within an SMP. Michael et al., recently observed that static partitioning of messages into two protocol queues leads to a significant load imbalance [16]. Rather than partition the resources among protocol queues, processors on one node can collaborate handling messages from a single queue. It follows from a well-known queueing theory result that single-queue/multi-server systems inherently outperform multi-queue/multi-server systems [13].

In this paper we propose *Parallel Dispatch Queue (PDQ)*, a set of mechanisms that allow protocol handlers to synchronize in a single queue prior to dispatch. Much as a monitor synchronization variable in a concurrent language provides mutual exclusion for a set of data structures [7], PDQ allows a message to specify a synchronization *key* corresponding to a *group* of resources a protocol handler accesses when handling the message. By synchronizing messages prior to dispatch and executing handlers in parallel only for messages with distinct keys, PDQ obviates the need for explicit fine-grain synchronization around individual resources within protocol handlers. PDQ can significantly improve performance by not only eliminating the overhead of acquiring/releasing a synchronization primitive but also preventing busy-waiting within handlers. Busy-waiting offsets the gains from parallel handler execution and wastes processor cycles that could otherwise contribute to handling messages.

To fully exploit the potential for parallel handler execution, PDQ requires a protocol programmer/designer to organize resources into fine-grain groups so that frequently executing handlers can access resources in mutual exclusion. Fine-grain protocol handlers, however, occasionally may require access to a larger group of resources—e.g., to migrate an entire application data structure from one node to another. PDQ provides mechanisms to temporarily serialize handler execution so that a handler can access all of the available resources at the cost of a lower protocol performance.

We evaluate PDQ's impact on fine-grain protocol performance in the context of fine-grain distributed shared memory (DSM) [23,22]. As a programming abstraction, however, PDQ has potential for much wider applicability and can be used to efficiently synchronize threads in any fine-grain parallel computing environment. We propose

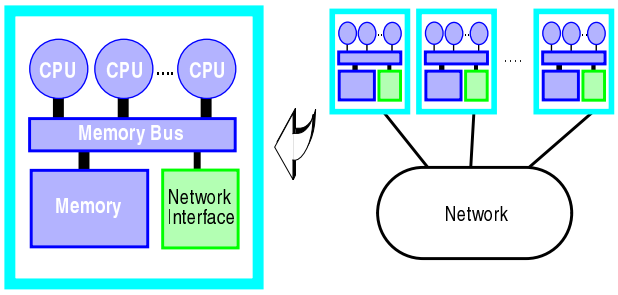

FIGURE 1. An SMP cluster.

and evaluate three fine-grain DSM systems loosely derived from the Wisconsin Typhoon family of DSMs [20]:

- *Hurricane*, is a high-performance system that tightly integrates PDQ, fine-grain sharing hardware, and multiple embedded protocol processors into a single custom device interfacing the network,
- *Hurricane-1*, is a less hardware-intensive system that integrates PDQ with fine-grain sharing hardware into a custom network interface device, but uses dedicated SMP-node processors for protocol execution,
- *Hurricane-1 Mult*, is most cost-effective and is much like *Hurricane-1* but schedules the protocol handlers on idle SMP processors thereby eliminating the extra dedicated SMP processors.

To gauge the performance impact of PDQ on software DSM protocol performance, we compare the Hurricane systems to Simple COMA (S-COMA) [8], an all-hardware DSM protocol implementation. Our model for S-COMA conservatively assumes that all protocol actions are executed in one processor cycle and only accounts for handler memory access times. Results from simulating the Hurricane and S-COMA systems running shared-memory applications indicate that:

- parallel protocol execution using PDQ significantly increases software protocol performance,
- Hurricane with multiple embedded processors offers performance competitive to or better than S-COMA,
- Hurricane-1 Mult benefits most from parallel protocol execution especially for clusters of fat SMPs (i.e., SMPs with a large number of processors), in which many idle processors contribute to protocol execution. Hurricane-1 Mult on a cluster of 4 16-way SMPs increases application performance on average by a factor of 2.6 over a system with a single dedicated SMP protocol processor per node.

The rest of the paper is organized as follows. Section 2 presents an overview of the fine-grain parallel systems we study. Section 3 describes fine-grain synchronization using PDQ in detail and discusses the implementation issues. Section 4 presents an application of PDQ in the context of fine-grain software DSM. Section 5 presents a discussion of the performance results. Finally, Section 6 concludes the paper.

```
fetch&add(int src, int *valptr,
          int inc)
{
    int oldval = *valptr;
    *valptr += inc;
    send(src, fetch&addresp,
         valptr, oldval);
}
```

```
locked_fetch&add(int src, val *valptr,
                 int inc)
{
    lock(valptr->lock);
    int oldval = *vaptr->val;
    *valptr->val += inc;
    unlock(valptr->lock);
    send(src,fetch&addresp,
         valptr, oldval);
}
```

FIGURE 2. Fine-grain communication protocols: (left) a simple fine-grain protocol handler, (right) protecting handler data structures using locks.

2 Fine-Grain Communication Protocols

Figure 1 illustrates the general class of parallel machines that we study in this paper. Each node consists of an SMP connected to a low-latency, high-bandwidth network via a network interface card residing on the memory bus. A high-level parallel programming abstraction—such as Split-C [3] or coherent distributed shared memory—provides fine-grain communication among processors. Low-level communication occurs *within* a node using the snoopy cache coherent memory bus. A software protocol implements communication *across* SMP nodes using a fine-grain messaging abstraction—such as Active Messages [25]. The software protocol executes either on embedded network interface processors [16,14] or on the SMP commodity processors [15,23].

Figure 2 (left) depicts an example of a simple active-message based protocol which performs a fetch&add operation on a memory word. The handler takes as input parameters the message source node id (src), the address of the memory word (valptr), and the fetch&add increment (inc). The handler reads the current content of the memory word, increments it, and subsequently sends the appropriate reply message. In general, fine-grain protocol handlers are more complex than our simplistic fetch&add handler and access more than just a single resource. For instance, coherent DSM protocols maintain a memory block's sharing status in a directory which is additionally accessed by the protocol handlers.

In this paper, we study parallel execution of fine-grain protocol handlers. Parallel handler execution requires synchronization around the protocol resources. A simple approach to synchronizing handler accesses is to use software spin-locks (e.g., as in parallel TCP/IP protocols [21,1,10,11]). Figure 2 (right) depicts the use of a spin-lock in our fetch&add handler to prevent simultaneous fetch&add operations on the same memory word by parallel handlers. The figure indicates that the lock doubles the number of memory accesses in the fetch&add handler. Moreover, synchronizing within the handler may result in busy-waiting if multiple processors simultaneously access the same memory word. Instead, we propose a technique to synchronize handlers prior to dispatch to obviate the need for synchronizing within handlers.

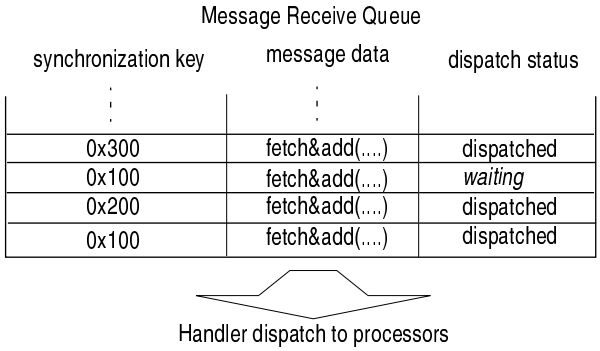

FIGURE 3. Parallel fetch&add handler dispatch.

3 PDQ: Synchronization in a Queue

Parallel dispatch queue (PDQ) is a set of mechanisms that unify access synchronization to protocol resources (e.g., data in memory) with protocol handler dispatch. Rather than perform fine-grain synchronization (using software spin-locks) around individual resources, PDQ allows a message to request access to a *group* of resources and performs the synchronization at handler dispatch time. A PDQ message specifies a synchronization *key* indicating the set of resources accessed by the message much as a monitor variable in a concurrent language [7] provides mutual exclusion around a group of data structures. A PDQ implementation then dispatches handlers with distinct synchronization keys in parallel while serializing handler execution for a specific synchronization key.

Figure 3 illustrates an example of how a PDQ implementation may dispatch our fetch&add handler (from Figure 2 (left)) in parallel. Assume that the protocol programmer specifies the fetch&add memory word address as the PDQ synchronization key. There are four (active) messages in the queue requiring invocation of the fetch&add handler. The PDQ implementation dispatches fetch&add handlers for messages with memory addresses 0x100, 0x200, and 0x300 in parallel. A fetch&add handler for a second message with memory address 0x100 can not execute due to a first message being handled with the same key. However, other incoming messages with distinct keys will continue to dispatch as long as there are processors waiting to execute handlers.

Fine-grain synchronization in a dispatch queue has two advantages. First, it obviates the need for explicit synchronization within handlers and thereby eliminates the corresponding overhead. Because of the fine-grain nature of resource accesses (e.g., reading a small memory word or block), the overhead of acquiring and releasing a lock may prohibitively increase handler execution time.

Second, without in-queue synchronization, multiple handlers may dispatch and contend for the same resource. Because only one handler succeeds in acquiring the lock, the rest of the dispatched handlers must wait spinning on the lock. An example of handler synchronization after dispatch is address interlocks in the Stanford FLASH [12]. The interlocks guarantee mutual exclusion among messages entering the handler execution pipeline and freeze the pipeline (i.e., busy wait) when two messages have the same address. Busy-waiting, however, wastes cycles that

could otherwise contribute to handling messages for which resources are available.

Rather than busy wait, some fine-grain messaging systems (like Optimistic Active Messages (OAM) [26]) postpone handling a message and invoke a light-weight thread to re-execute the handler after the lock is released. Resource contention in such a system, however, may severely impact performance due to the high thread management overhead. Synchronization in a queue eliminates busy waiting by only dispatching and executing handlers that access mutually exclusive resources.

To fully realize the potential for parallel handler execution, PDQ only requires protocol programmers to organize the resources into fine-grain groups which frequently executing handlers can access in mutual exclusion. Fine-grain parallel systems, however, may occasionally execute protocol handlers that require access to a large set of protocol resources. In fine-grain DSM, for instance, the majority of executed protocol handlers implement coherence on fine-grain shared-memory blocks. Occasionally, a protocol handler may manipulate multiple blocks—e.g., to migrate a page of shared memory from one node to another. PDQ also provides mechanisms by which the system can temporarily revert back to sequential handler execution at the cost of lower protocol performance.

3.1 PDQ Programming Model

Much like other queue-based abstractions (e.g., Remote Queues [2]), PDQ provides a programming interface to store and remove entries into a queue. An *enqueue* operation includes at least three parameters specifying a queue name, a synchronization key, and a pointer to a buffer containing the handler dispatch address and the message data. As a high-level synchronization abstraction, the PDQ programming interface does not associate a specific syntax with the queue name. The queue name, for instance, can be an id corresponding to the machine node on which the queue resides, or a global address in a shared-memory system. A *dequeue* operation specifies the queue from which a processor wishes to handle messages. The dequeue operation either returns successfully with a synchronization key and a pointer to a buffer with the message data, or with a flag indicating there are no messages to be dispatched.

Besides synchronization keys used by the protocol programmer to allow in-queue synchronization, PDQ also provides two pre-defined synchronization keys. A *sequential* synchronization key indicates that a handler must execute in isolation. The implementation simply stops dispatching handlers, waits for all handlers to complete, and subsequently dispatches the handler for the message with the sequential key. Once the handler completes, PDQ can resume dispatching handlers in parallel.

PDQ also provides a pre-defined *nosync* key that indicates handler synchronization is not required. A message with a nosync key may invoke a handler at any time. Such a message can be used in a fine-grain parallel system to access remote read-only data structures. Similarly, applications with inherent data races (e.g., using SOR-based algorithms [27]) in which data coherence is not a requirement for correct execution can use nosync messages to perform remote writes to data.

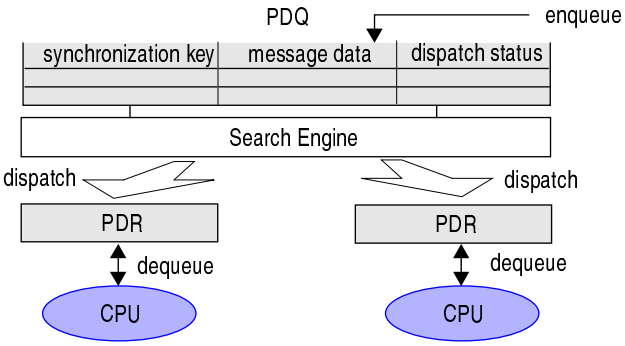

FIGURE 4. Parallel handler dispatch in PDQ.

3.2 PDQ Implementation

In the simplest form, a PDQ implementation (Figure 4) provides a queue of incoming messages, a search engine to perform an associative search through the messages to match synchronization keys, and a per-processor protocol dispatch register (PDR) through which the protocol processors receive dispatched PDQ entries. The implementation also provides per-processor message send queues (not shown) so that processors can send messages without requiring synchronization. The PDQ also keeps track of an entry's dispatch status so that the search engine can determine which keys are currently dispatched.

In this paper, we only consider PDQ implementations in which every machine node includes a single PDQ for incoming messages, and a single PDR and message send queue per protocol processor. Virtualizing the PDQ hardware to provide multiple protected message queues per processor is an active area of research and beyond the scope of this paper [4,24].

Message queues can generally be large and may sometimes spill to memory to remove back pressure from the network [17]. As such, an associative search on the entire queue may not be feasible. The search, however, can be limited to a small number of entries in the PDQ. The implementation can use a small buffer to store PDQ entries that are ready to be dispatched and perform the search in the background while the handlers are executing. The search engine can continue inserting entries from the PDQ into the buffer. The dispatch logic can meanwhile simply remove entries from the small buffer and store them in the PDRs upon demand.

Similarly, for message queues that partially reside in memory, a PDQ implementation may use a buffer to cache several entries at the head of the PDQ. The buffer allows the search engine to proceed without frequently accessing the memory. Such a scheme allows the entire PDQ to spill to memory much like a cachable queue [17]. Buffer entries can be prefetched from memory upon a message dispatch to hide the latency of the memory access.

4 Hurricane: Parallelizing Fine-Grain DSM

We evaluate the impact of parallel handler execution on software protocol performance in the context of fine-grain DSM. Our results, however, are applicable to a wide

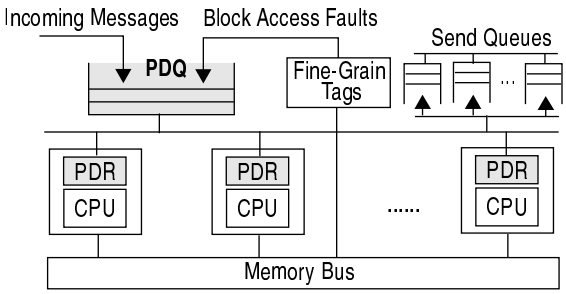

FIGURE 5. A Hurricane custom device.

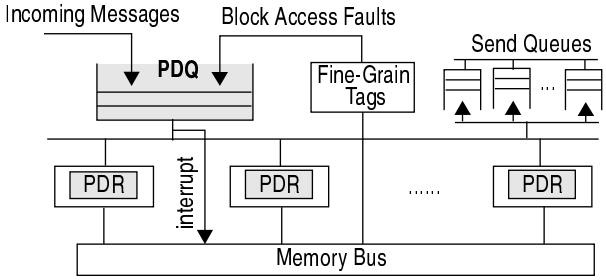

FIGURE 6. A Hurricane-1 custom device.

variety of fine-grain parallel systems and programming abstractions that require efficient fine-grain synchronization and can take advantage of PDQ to execute in parallel. We propose and evaluate three fine-grain DSM systems—*Hurricane*, *Hurricane-1*, and *Hurricane-1 Mult*—loosely derived from the Wisconsin Typhoon family of DSMs [20]. The systems vary in the level of hardware support for fine-grain sharing and result in a spectrum of system cost and performance.

All systems execute a modified version of the Stache coherence protocol [19] written to comply with the PDQ protocol programming interface. Stache is a full-map invalidation-based cache coherence protocol that caches remote data into the node's main memory. To implement caching, Stache allocates memory at page granularity but maintains coherence at a fine-grain cache block granularity (e.g., 32~128 bytes).

Our modified protocol uses cache block shared-memory addresses as the PDQ synchronization key and organizes all protocol data structures (e.g., DSM directory) so that data structure accesses for distinct cache blocks are mutually exclusive. Protocol handlers manipulating data structures for a group of cache blocks—such as page allocation/deallocation handlers—use a special PDQ synchronization key corresponding to an invalid address to guarantee serialization semantics (Section 3.1).

In the rest of this section, we describe in detail the Hurricane fine-grain DSMs. Section 5 presents the performance evaluation and the results of this study.

4.1 Hurricane

Hurricane, like Typhoon [20], is a high-performance hardware-centric implementation that integrates the fine-grain access control logic, messaging queues, and embedded protocol processors into a single custom device that interfaces the network. A Hurricane custom device differs from Typhoon in that it includes multiple embedded processors (rather than one) and provides the PDQ hardware to dispatch protocol handlers to the protocol processors.

Figure 5 illustrates the architecture of a Hurricane custom device. In fine-grain DSM, there are two types of events that invoke protocol handlers. A *block access fault* (generated locally on a node) corresponds to a request to fetch and access a remote shared-memory block. A *message* (from other nodes) typically carries fetched shared data or coherence information such as an invalidation. A PDQ collects all block access faults and incoming messages. Both protocol event types use a global shared-memory address (corresponding to a cache block) as the PDQ synchronization key. The PDQ dispatches block access faults and messages into a per-processor PDR upon demand. A protocol processor indicates the completion of a handler by writing into its PDR. Upon handler completion, the PDQ dispatches a new entry into a processor's PDR. As in Typhoon, the PDRs reside on the cache bus and can be accessed in a single cycle [20].

4.2 Hurricane-1 & Hurricane-1 Mult

Hurricane-1 is a less hardware-intensive implementation and combines the fine-grain access control logic with the messaging queues on a single device but uses SMP commodity processors to run the software protocol. As in Hurricane, a single PDQ gathers information about all block access faults generated on the node and all of the incoming messages. Hurricane-1 also provides a PDR per SMP processor to implement handler dispatch.

Figure 6 illustrates the architecture of a Hurricane-1 custom device. To provide efficient polling between SMP processors and the PDRs across the memory bus, each PDR is implemented as a cachable control register [17]. By allowing a PDR to be cached in the processor cache, a cachable control register turns polling into a cache hit in the absence of protocol events, thereby eliminating polling traffic over the memory bus. Upon dispatching a protocol event, a Hurricane-1 device invalidates the cached copy of the PDR forcing a protocol processor to read the new PDR contents. A protocol processor indicates completion of a handler by performing an uncached write into its PDR.

A Hurricane-1 device allows both dedicated and multiplexed protocol scheduling on SMP processors [6]. Multiple SMP processors can be dedicated to only execute protocol handlers for the duration of an application's execution. Dedicated protocol processors save overhead by not interfering with the computation, result in a lower protocol occupancy [9]—i.e., the time to execute a protocol handler, and consequently increase protocol performance. Dedicated processors, however, waste processor cycles that could otherwise contribute to computation. In the rest of this paper, we use the term *Hurricane-1* to refer to a fine-grain DSM with the Hurricane-1 custom device and dedicated SMP protocol processors.

A Hurricane-1 device also supports multiplexed protocol scheduling where all SMP processors perform computation and execute protocol handlers whenever idle. To guarantee timely message handling when all processors are busy computing, a Hurricane-1 device provides a mechanism for invoking interrupts on the memory bus (Figure 6). Whenever an SMP processor resumes compu-

tation, it signals the device by performing an uncached write into its PDR. An interrupt arbiter on the memory bus distributes interrupts round-robin among SMP processors. To reduce the interrupt frequency and eliminate extra scheduling overhead, a Hurricane-1 device only delivers an interrupt when all SMP processors are busy. Such a policy assumes that interrupts are infrequent and are only invoked to prevent long protocol invocation delays. In the rest of this paper, we use the term *Hurricane-1 Mult* to refer to a fine-grain DSM with the Hurricane-1 custom device and multiplexed SMP processor scheduling.

5 Performance Evaluation

We use the Wisconsin Wind Tunnel II (WWT-II) [18] to simulate SMP cluster implementations of the Hurricane systems (Figure 1). Each node consists of 400 MHz dual-issue statically scheduled processors—modeled after the Ross HyperSPARC—interconnected by a 100 MHz split-transaction bus. We model a highly-interleaved memory system, characteristic of high-performance SMP servers. A snoopy MOESI coherence protocol—modeled after SPARC's MBus protocol—keeps the caches within each node consistent. Our fine-grain DSM software protocol (Section 4) extends the SMP shared memory abstraction across a cluster. Unless specified otherwise, we assume a 64-byte DSM protocol.

WWT-II assumes perfect instruction caches but models data caches and their contention at the memory bus accurately. WWT-II further assumes a point-to-point network with a constant latency of 100 cycles but models contention at the network interfaces. Interrupt overheads are 200 cycles, characteristic of carefully tuned parallel computers.

To gauge the impact of PDQ on software protocol performance we compare the Hurricane systems to a simple all-hardware protocol implementation, Simple-COMA (S-COMA) [8]. S-COMA is an invalidation-based full-map directory protocol much like Stache. The simulation model for S-COMA assumes minimum protocol occupancies accounting for only memory access times. As such, S-COMA's performance numbers in this study are optimistic, making the comparison to S-COMA conservative.

In the rest of this paper, we use the term *protocol processor* to refer to either S-COMA's finite-state-machine (FSM) hardware protocol implementation, an embedded processor on Hurricane, or a commodity SMP processor in Hurricane-1 and Hurricane-1 Mult. Following this terminology, S-COMA is a single-processor device, and Hurricane and Hurricane-1 are either single-processor or multiprocessor devices.

5.1 Protocol Occupancy

Parallel protocol execution improves communication performance by reducing queueing at the protocol processor. Queueing is a function of both application communication characteristics and protocol occupancy [9]. Latency-bound applications primarily benefit from low-occupancy implementations (such as hardware DSM) because a lower occupancy directly reduces roundtrip miss times and thereby communication time. Bandwidth-bound applications, however, may eventually saturate a single protocol processor even in a low-occupancy implementation due to a large number of outstanding protocol events which lead to queueing. Such applications will most likely benefit from parallel protocol execution.

	Action	S-COMA	Hurricane	Hurricane-1
	detect miss, issue bus transaction	5	5	5
Request	dispatch handler	12	16	87
Request	get fault state, send	0	36	141
	network latency	100	100	100
Reply	dispatch handler	1	3	51
Reply	directory lookup	8	61	121
Reply	fetch data, change tag, send	136	140	205
	network latency	100	100	100
Response	dispatch handler	1	4	50
Response	place data, change tag	8	50	63
Response	resume, reissue bus transaction	6	6	178
	fetch data, complete load	63	63	63
	Total	440	584	1164

TABLE 1. Remote read miss latency breakdown (in 400-MHz cycles) for a 64-byte protocol.

Table 1 compares the minimum protocol occupancies in S-COMA, Hurricane, and Hurricane-1 (Hurricane-1 Mult). The table depicts the breakdown of time for various system events on a simple remote read of a 64-byte block. The table groups the system events into three categories. A request category on the caching node accounts for all the events from the arrival of the bus transaction upon a block access fault to sending a request message to the home node. A reply category on the home node consists of all events from the detection and dispatch of the request message to sending the 64-byte block to the caching node. A response category on the caching node accounts for all events from the dispatch of the reply message to resuming the computation.

Hurricane and S-COMA both tightly integrate the protocol resources on a single custom device. The key source of overhead in Hurricane as compared to S-COMA is the time to execute the handler instructions. Instruction execution overhead in Hurricane results in a significant increase in request/response protocol occupancies of 315%, but only increases the total roundtrip miss time by 33% as compared to S-COMA. Therefore, applications which rely on high request/response bandwidth—e.g., processors on one SMP node access data on several other nodes—can significantly benefit from parallel protocol execution in Hurricane.

In addition to software execution overhead, Hurricane-1 (Hurricane-1 Mult) also incurs the overhead of traversing the memory bus to receive handler dispatch information (i.e., to access the PDR), and moving fine-grain data blocks between the protocol processor caches

and message queues [20]. These additional overheads increase the request/response occupancies and total roundtrip latency in Hurricane-1 by 518% and 165% as compared to S-COMA respectively. Because of the large overall protocol occupancies, applications executing on Hurricane-1 can benefit from both higher request/response bandwidth and reply bandwidth through parallel protocol execution.

5.2 Results

Comparing minimum protocol occupancies and roundtrip times helps analyze the latency and bandwidth characteristics of S-COMA and the Hurricane systems for simple DSM operations (e.g., a remote read miss). Real applications, however, exhibit more complex interactions between the memory system and the protocol resulting in an increase in protocol occupancies and roundtrip times. Remote misses, for instance, can result in messages among three nodes in a producer/consumer relationship if neither the producer nor the consumer are the home node for the data. Real data sets also typically do not fit in caches and produce additional memory traffic on the bus. The performance impact of parallel protocol execution also highly depends on how much queueing there is at the protocol processor. Moreover, parallel execution using PDQ is only beneficial when there are multiple independent protocol events (i.e., corresponding to distinct memory blocks) in the queue. In this section, we evaluate our DSMs' performance using shared-memory applications.

Table 2 presents the applications we use in this study and the corresponding input parameters. *Barnes*, *cholesky*, *fft*, *fmm*, *radix* and *water-sp* are all from the SPLASH-2 [27] benchmark suite. *Em3d* is a shared-memory implementation of the Split-C benchmark [3].

The table also depicts application speedups for a cluster of 8 8-way SMPs interconnected by S-COMA hardware. The speedups are calculated with respect to application running times on a uniprocessor. *Water-sp* is primarily computation-intensive and achieves near-linear speedups. *Cholesky* is primarily communication-bound, suffers from a severe load imbalance [27], and does not speed up much. *Barnes*, *fmm*, and *em3d* have moderate communication-to-computation ratios and achieve a 50% efficiency with 64 processors. *Fft* and *radix* are communication-bound and exhibit poor speedups. In the rest of the section, we present performance results normalized to our base S-COMA system.

Baseline System Performance

Figure 7 depicts a performance comparison of the base case systems. Our baseline system corresponds to a cluster of 8 SMPs. The S-COMA and Hurricane, and Hurricane-1 Mult systems use 8-way SMPs. The Hurricane-1 systems use 8 SMP processors per node for computation and extra dedicated SMP processors for protocol execution. Performance results are normalized to S-COMA. The figure indicates that S-COMA improves performance over a software protocol running on an embedded processor (Hurricane 1pp) on average by 32%. The figure also indicates that S-COMA significantly improves performance (by up to 89%) over a software protocol implementation running on a commodity SMP processor (Hurricane-1 1pp). These results are consistent with those of Reinhardt et al. comparing Typhoon and Typhoon-1 (which are similar to single-processor Hurricane and Hurricane-1) against S-COMA [20].

Benchmark	Description	Input Set	Speedup
barnes	Barnes-Hut N-body simulation	16K particles	31
cholesky	Sparse cholesky factorization	tk29.O	5
em3d	3-D wave propagation	76K nodes, 15% remote	34
fft	Complex 1-D radix-$\sqrt{n}$ FFT	1M points	19
fmm	Fast Multipole N-body simulation	16K particles	31
radix	Integer radix sort	4M integers	12
water-sp	Water molecule force simulation	4096 molecules	61

TABLE 2. Applications, input sets, and S-COMA speedups on a cluster of 8 8-way SMPs.

The graphs indicate that there are three classes of applications. The first class is *water-sp* which is primarily computation-intensive and not sensitive to protocol execution speed. All systems perform within 91% of S-COMA for *water-sp*.

The second class consists of *barnes* and *fmm* which are primarily latency-bound and do not substantially benefit from parallel protocol execution. In these applications, much of the execution time is spent in a force calculation phase between bodies in a galaxy. Communication in this phase is sporadic and evenly distributed among the nodes. These applications benefit more from a lower protocol occupancy than parallel protocol execution.

A single-processor Hurricane system performs well (within 90% of S-COMA) running *barnes* and *fmm.* Two-processor and four-processor Hurricane systems improve performance over a single-processor configuration by at most 11% and 13% respectively. A single-processor Hurricane-1 system reduces the performance to approximately within 60% of S-COMA making room for performance improvement. Nevertheless, adding protocol processors to Hurricane-1 increases the performance to at most within 84% of S-COMA. Furthermore, a Hurricane-1 with four dedicated protocol processors improves performance over Hurricane-1 Mult; handler scheduling and the resulting cache interference in Hurricane-1 Mult incur overhead and increase protocol occupancy (Section 4.2). The parallelism in handler execution is not high enough to offset the extra scheduling overhead.

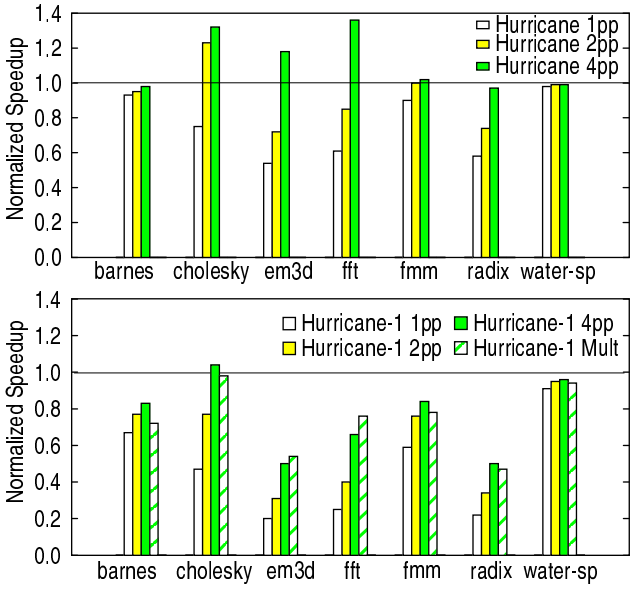

FIGURE 7. Comparing baseline system performance.

The figure compares Hurricane's (above) and Hurricane-1's (below) performance with S-COMA on a cluster of 8 SMPs. The Hurricane, Hurricane-1 Mult, and S-COMA systems use 8-way SMPs. The Hurricane-1 systems use additional dedicated protocol processors per SMP. The graphs plot application speedups in one- (1pp), two- (2pp), and four-processor (4pp) Hurricane and Hurricane-1 systems, and Hurricane-1 Mult system. The speedups are normalized to S-COMA. Values appearing under the horizontal line at 1 indicate a better performance under S-COMA.

The third class consists of *cholesky*, *em3d*, *fft*, and *radix* which are all bandwidth-bound applications. *Cholesky* incurs a large number of compulsory misses to data that is not actively shared. As such, the reply handlers in *cholesky* frequently involve reading data from memory and have high occupancies. Multiprocessor Hurricane devices substantially improve performance over single-processor devices by parallelizing the memory accesses thereby increasing the reply bandwidth. A two-processor Hurricane actually improves performance over S-COMA by 23%. Limited parallelism in protocol execution, however, limits Hurricane's performance improvement over S-COMA to at most 32% with four protocol processors.

In *cholesky*, Hurricane-1's performance also extensively benefits from multiple protocol processors. Adding protocol processors significantly improves performance even up to four processors. The high protocol occupancy in Hurricane-1 results in large queueing delays at the protocol processor. Parallel protocol processors reduce queueing delays and thereby improve performance. The four-processor Hurricane-1 outperforms S-COMA, and the Hurricane-1 Mult system both performs very close to S-COMA and improves cost by eliminating the extra dedicated protocol processors.

Communication and computation in *em3d*, *fft*, and *radix* proceed in synchronous phases. Communication in these applications is highly bandwidth-intensive, bursty, and of a producer/consumer nature. In *em3d*, communication involves reading/writing memory blocks from/to neighboring processors. *Fft*, and *radix* both perform all-to-all communication with every processor exchanging its produced data with other processors. The large degrees of sharing in *em3d*, *fft*, and *radix*, result in frequent coherence activity. Coherence events often involve executing protocol handlers that only modify state and send control messages (e.g., an invalidation). Because the handlers do not transfer data between the memory and the network, the handlers' occupancy in a software protocol is primarily due to instruction execution. Software protocol implementations, therefore, have a much higher occupancy for control messages than hardware implementations. The figure indicates that the single-processor Hurricane systems at best perform within 61% of S-COMA. The single-processor Hurricane-1 systems exhibit extremely poor performance and at best reach within 25% of S-COMA's performance.

Multiprocessor Hurricane systems help mitigate the software protocol execution bottleneck in *em3d*, *fft*, and *radix*. The two-processor Hurricane systems improve performance over a single-processor system by at most 40% because parallel protocol execution at a minimum incurs the additional overhead of protocol state migration among the protocol processor caches. The four-processor Hurricane systems' performance ranges from slightly worse than S-COMA (in *radix*) to 36% better than S-COMA (in *fft*). Hurricane-1's performance also significantly improves with multiple protocol processors but at best reaches within 76% of S-COMA (in *fft* under Hurricane-1 Mult).

To summarize the results, a four-processor Hurricane system on average increases speedups by 12% over S-COMA, and a four-processor Hurricane-1 on average performs within 76% of S-COMA. More importantly, the most cost-effective Hurricane-1 Mult system performs within 74% of an all-hardware S-COMA system without requiring extra dedicated protocol processors. Previous research indicated that static partitioning of resources among protocol processors results in a load imbalance rendering parallel protocol execution less beneficial [16]. These results indicate that fine-grain handler synchronization using PDQ can realize the full potential of parallelism in fine-grain protocol execution.

Impact of Clustering Degree

This section evaluates the impact of clustering degree—i.e., the number of processors in every SMP node—on the relative performance of the systems while maintaining the number of processors and the total amount of memory in the system constant.

Clustering typically increases the total amount of protocol traffic generated per machine node [23]. The increase in protocol traffic, however, depends on an application's sharing patterns. On the one hand, clustering allows processors to share a single cached copy of remote data, reducing protocol traffic generated per processor. On the other hand, in the absence of sharing, clustering may linearly increase protocol traffic in/out of a node with the increase in the number of processors per node. Clustering also reduces the number of network interfaces in the system, placing higher demands on the protocol processors favoring parallel protocol execution.

Figure 8 compares Hurricane's performance against S-COMA for a cluster of 16 4-way SMPs (above) and a cluster of 4 16-way SMPs (below). The graphs indicate that a higher clustering degree increases the performance

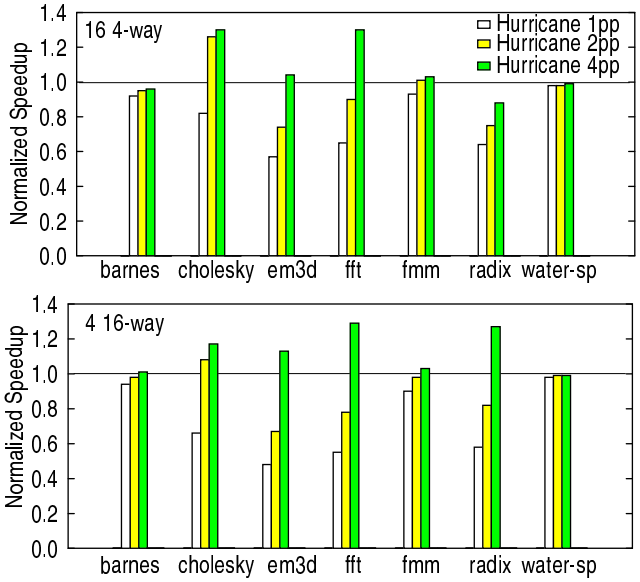

FIGURE 8. Impact of clustering degree on Hurricane's performance.

The figure compares performance in S-COMA and Hurricane on a cluster of 16 4-way SMPs (above), and a cluster of 4 16-way SMPs (below). The graphs plot application speedups in one- (1pp), two- (2pp), and four-processor (4pp) Hurricane systems. The speedups are normalized to S-COMA. Values appearing under the horizontal line at 1 indicate a better performance under S-COMA.

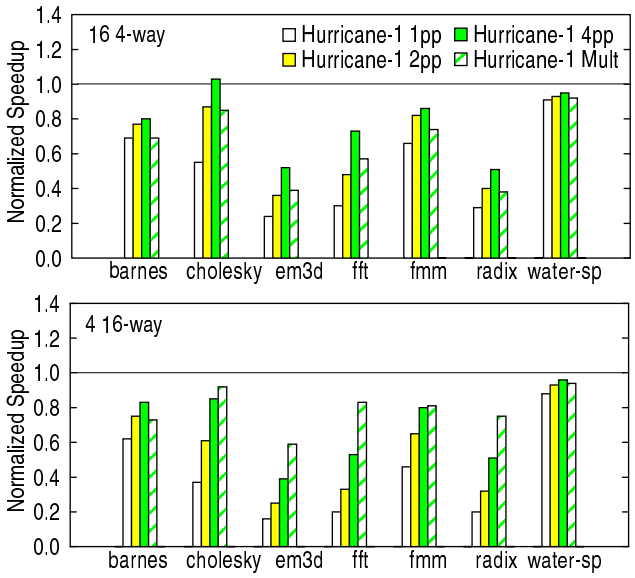

FIGURE 9. Impact of clustering degree on Hurricane-1's performance.

The figure compares performance in S-COMA and Hurricane-1 on a cluster of 16 (above) and 4 (below) SMPs. The S-COMA and Hurricane-1 Mult systems use 4-way (above) and 16-way (below) SMPs respectively. The rest of the Hurricane-1 systems use additional dedicated protocol processors per SMP. The graphs plot application speedups in one- (1pp), two- (2pp), and four-processor (4pp) Hurricane-1, and Hurricane-1 Mult systems. The speedups are normalized to S-COMA. Values appearing under the horizontal line at 1 indicate a better performance under S-COMA.

gap between the single-processor Hurricane systems and S-COMA in most of the applications. This result indicates that queueing delays due to a smaller number of network interface devices in the system has a higher impact on performance than the gains from sharing remote data.

Multiple protocol processors in Hurricane systems help close the performance gap between software and hardware implementations. With a clustering degree of 16, a four-processor Hurricane system outperforms S-COMA in all the applications except for *water-sp*; Hurricane's performance in *water-sp* is within 99% of S-COMA. An increase in the clustering degree from 4 to 16 increases a four-processor Hurricane's performance on average from 7% to 13% over S-COMA's.

Figure 9 illustrates the impact of clustering degree on Hurricane-1's performance. A high clustering degree has a large impact on the single-processor Hurricane-1's performance. Because of the poor performance of the single-processor system, even the large performance improvements due to four protocol processors fail to make Hurricane-1 competitive with S-COMA. Not surprisingly, Hurricane-1 Mult substantially benefits from a high clustering degree and outperforms a four-processor Hurricane-1 system in all bandwidth-bound applications. Increasing the clustering degree from 4 to 16 also allows Hurricane-1 Mult to improve performance from 65% to 80% of S-COMA.

Impact of Block Size

An increase in the protocol block size increases the overall protocol bandwidth out of a node. Large block sizes also increase the fraction of protocol occupancy due to data transfer time between memory and the network. Amortizing the software protocol overhead over a larger overall occupancy reduces the performance gap between software and hardware protocol implementations.

Large blocks, however, result in false sharing in applications with very fine sharing granularity thereby increasing protocol activity. Higher protocol activity intensifies queueing at the protocol processors and results in a larger performance gap between software and hardware protocol implementations. Parallelizing protocol execution alleviates the performance loss due to false sharing by reducing queueing at the protocol processors.

Figure 10 compares Hurricane's performance against S-COMA's for a 32-byte protocol (above) and a 128-byte protocol (below). The graphs corroborate the intuition that an increase in the block size reduces the performance gap between the single-processor Hurricane systems and S-COMA in some applications, and increases the gap in others. With a 128-byte block, *cholesky*, *em3d*, *fft*, *radix*, *water-sp* all exhibit better performance under single-processor Hurricane systems relative to S-COMA. *Barnes* and *fmm* share data at very fine granularity, suffer from false sharing with 128-byte blocks, and therefore experience a larger performance gap between the single-processor Hurricane and S-COMA.

The graphs also indicate that a large block size not only favors the single-processor Hurricane system, but also the multiprocessor systems. Two protocol processors make a Hurricane system competitive with S-COMA in all the applications. A four-processor Hurricane system on average speeds up application execution time by 20% over

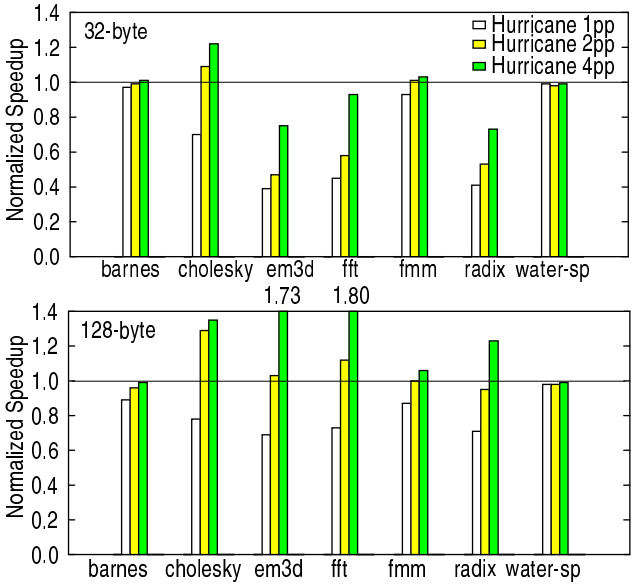

FIGURE 10. Impact of block size on Hurricane's performance.

The figure compares performance in S-COMA and Hurricane for a 32-byte (above) and a 128-byte (below) block protocol. The graphs plot application speedups in one- (1pp), two- (2pp), and four-processor (4pp) Hurricane systems. The speedups are normalized to S-COMA. Values appearing under the horizontal line at 1 indicate a better performance under S-COMA.

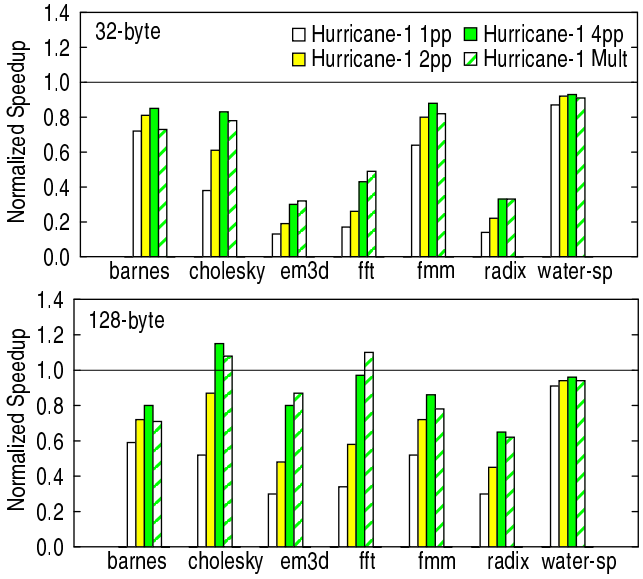

FIGURE 11. Impact of block size on Hurricane-1's performance.

The figure compares performance in S-COMA and Hurricane-1 for a 32-byte (above) and a 128-byte (below) block protocol. The graphs plot application speedups in one- (1pp), two- (2pp), and four-processor (4pp) Hurricane-1, and Hurricane-1 Mult systems. The speedups are normalized to S-COMA. Values appearing under the horizontal line at 1 indicate a better performance under S-COMA.

S-COMA. These results, indicate that pipelining protocol handler execution to allow for multiple outstanding memory requests may enable single-processor devices to achieve a high protocol bandwidth with large blocks.

Figure 11 illustrates the impact of protocol block size on Hurricane-1's performance. A large block size has a higher impact on a single-processor Hurricane-1's performance as compared to Hurricane. Large blocks benefit systems with high software protocol overheads (as in Hurricane-1) allowing the system to amortize the overhead over a larger protocol occupancy. Much as in the Hurricane systems, multiprocessor Hurricane-1 systems close the performance gap between Hurricane-1 and S-COMA. A four-processor Hurricane-1 system, and a Hurricane-1 Mult system both reach approximately within 88% of S-COMA's performance.

6 Conclusions

Many parallel applications and programming abstractions rely on low-overhead messaging and employ fine-grain communication protocols to exchange small amounts of data over the network. Traditionally, fine-grain parallel systems targeted uniprocessor-node parallel computers and executed the fine-grain protocol handlers on either the node's single commodity processor or an embedded network interface processor. With the emergence of cluster of SMPs, however, multiple SMP processors increase the demand on software protocol execution. One approach to provide communication performance commensurate to the number of SMP processors, is to execute fine-grain protocol handlers in parallel.

In this paper, we proposed a novel set of mechanisms, Parallel Dispatch Queue (PDQ), to efficiently execute fine-grain protocol handlers in parallel. Much as a monitor synchronization variable protects a set of data structures in a concurrent programming language, PDQ requires protocol programmers/designers to partition protocol resources into mutually exclusive groups and annotate protocol messages with a corresponding synchronization key. A PDQ implementation then dispatches and executes handlers for messages with distinct synchronization keys in parallel and only serializes handler execution for a given key. In-queue synchronization at handler dispatch time obviates the need for explicit fine-grain synchronization around individual resources within handlers, eliminates busy-waiting, and increases protocol performance.

We studied PDQ's impact on software protocol performance in the context of fine-grain DSM implemented on a cluster of SMPs. Simulation results running shared-memory applications indicated that: (i) parallel protocol execution using PDQ significantly improves the communication performance in a software fine-grain DSM, (ii) tight integration of PDQ and embedded processors in a single custom device can offer performance competitive or better than an all-hardware DSM, and (iii) PDQ best benefits cost-effective systems that use idle SMP processors (rather than custom embedded processors) for protocol execution. Application performance on a cluster of 4 16-way SMPs using PDQ and idle SMP processors for protocol execution on average improved by a factor of 2.6 over a system with a single dedicated SMP protocol processor.

References

[1] M. Bjoerkman and P. Gunningberg. Locking effects in multiprocessor implementations of protocols. In *SIGCOMM '93*, pages 74–83, September 1993.

[2] E. A. Brewer, F. T. Chong, L. T. Liu, S. D. Sharma, and J. Kubiatowicz. Remote queues: Exposing message queues for optimization and atomicity. In *Proceedings of the Seventh ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 42–53, 1995.

[3] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, Nov. 1993.

[4] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd. The virtual interface architecture. *IEEE Micro*, 18(2):66–76, 1998.

[5] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy. SoftFLASH: Analyzing the performance of clustered distributed virtual shared memory supporting fine-grain shared memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, Oct. 1996.

[6] B. Falsafi and D. A. Wood. Scheduling communication on an SMP node parallel machine. In *Proceedings of the Third IEEE Symposium on High-Performance Computer Architecture*, pages 128–138, Feb. 1997.

[7] C. Ghezzi and M. Jazayeri. *Programming Language Concepts*. Wiley, 2/e edition, 1987.

[8] E. Hagersten, A. Saulsbury, and A. Landin. Simple COMA node implementations. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, Jan. 1994.

[9] C. Holt, M. Heinrich, J. P. Singh, E. Rothberg, and J. Hennessy. The effects of latency, occupancy, and bandwidth in distributed shared memory multiprocessors. Technical Report CSL-TR-95-660, Computer Systems Laboratory, Stanford University, January 1995.

[10] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan. 1991.

[11] M. Kaiserswerth. The parallel protocol engine. *IEEE/ACM Transactions on Networking*, 1(6):650–663, December 1993.

[12] J. Kuskin et al. The stanford FLASH multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, Apr. 1994.

[13] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice Hall, 1984.

[14] T. Lovett and R. Clapp. STiNG: A CC-NUMA compute system for the commercial marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[15] S. S. Lumeta, A. M. Manwaring, and D. E. Culler. Multiprotocol active messages on a cluster of SMP's. In *Proceedings of Supercomputing '97*, November 1997.

[16] M. Michael, A. K. Nanda, B.-H. Lim, and M. L. Scott. Coherence controller architectures for SMP-based CC-NUMA mulitprocessors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, May 1997.

[17] S. S. Mukherjee, B. Falsafi, M. D. Hill, and D. A. Wood. Coherent network interfaces for fine-grain communication. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 247–258, May 1996.

[18] S. S. Mukherjee, S. K. Reinhardt, B. Falsafi, M. Litzkow, S. Huss-Lederman, M. D. Hill, J. R. Larus, and D. A. Wood. Wisconsin Wind Tunnel II: A fast and portable parallel architecture simulator. In *Workshop on Performance Analysis and Its Impact on Design (PAID)*, June 1997.

[19] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, Apr. 1994.

[20] S. K. Reinhardt, R. W. Pfile, and D. A. Wood. Decoupled hardware support for distributed shared memory. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[21] J. D. Salehi, J. F. Kurose, and D. Towsley. The effectiveness of affinity-based scheduling in multiprocessor networking. *IEEE Transactions on Networking*, 4(4), Aug. 1996.

[22] D. J. Scales, K. Gharachorloo, and A. Aggarwal. Fine-grain software distributed shared memory on SMP clusters. In *Proceedings of the Fourth IEEE Symposium on High-Performance Computer Architecture*, Feb. 1998.

[23] I. Schoinas, B. Falsafi, M. D. Hill, J. Larus, and D. A. Wood. Sirocco: Cost-effective fine-grain distributed shared memory. In *Proceedings of the Sixth International Conference on Parallel Architectures and Compilation Techniques*, October 1998.

[24] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 40–53, Dec. 1995.

[25] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrating communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[26] D. A. Wallach, W. C. Hsieh, K. L. Johnson, M. F. Kaashoek, and W. E. Weihl. Optimistic active messages: A mechanism for scheduling communic ation with computation. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, July 1995.

[27] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, July 1995.