Support for Recoverable Memory in the Distributed Virtual Communication Machine

Marcel-Cătălin Roşu
IBM T.J.Watson Research Center
mrosu@watson.ibm.com

Karsten Schwan
Georgia Institute of Technology
schwan@cc.gatech.edu

Abstract

*The Distributed Virtual Communication Machine (*DVCM*) is a software communication architecture for clusters of workstations equipped with programmable network interfaces (NIs) for high-speed networks. DVCM is an extensible architecture, which promotes the transfer of application modules to the NI. By executing 'closer' to the network, on the NI CoProcessor, these modules can communicate with significantly higher message rates and lower latencies than achievable at the CPU-level.*

*This paper describes how DVCM modules can be used to enhance the performance of the Cluster Recoverable Memory system (*CRMem*), a transaction-processing kernel for memory-resident databases. By using the NI CoProcessor for CRMem's remote operations, our implementation achieves more than 3,000 $trans/sec$ on a simplified TpcB benchmark.*

1 Introduction

Background. Clusters of workstations are becoming the platforms of choice for resource-intensive applications. In a cluster, the aggregate processing capacity, memory, and storage scale well at close to linear costs. Moreover, the use of high-speed networks like Gigabit Ethernet, Myrinet, or ATM, enables data transfer rates in the Gigabit range and latencies in the single- or low double-digit microsecond range. However, application-level message rates and latencies are limited by the inherent inability of communication code to scale with the increases in the CPU clock rates. Programmable network interfaces (NIs), typical for high-speed network cards, allows us to address this drawback.

The DVCM Approach. This paper presents the Distributed Virtual Communication Machine (*DVCM*), a software communication architecture which exploits the two layers of the resource hierarchy of a cluster equipped with programmable NIs (see Figure 1) – NI and host resources. The DVCM architecture is extensible and it leverages the NI CoProcessor

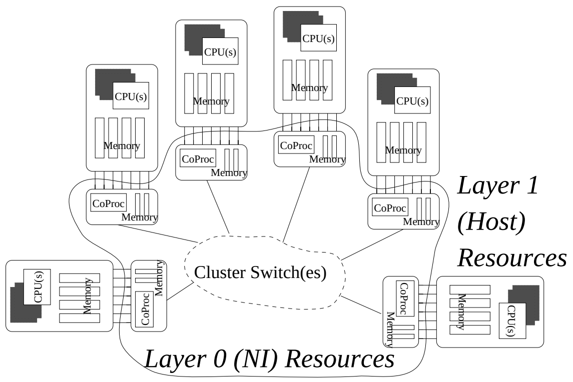

Figure 1. Two-layer resource hierarchy

and memory to improve the performance of cluster applications. DVCM departs from existing CPU-centric communication architectures by enabling cluster applications to implement their communication-intensive operations as DVCM extension modules run on the NI, and achieve message rates one order of magnitude higher and latencies significantly lower than achievable at CPU-level. Using FORE ATM cards and an 8 μsec-latency switch, we attain intermodule latencies as low as 20 $\mu secs$ and message rates as high as 200,000 $msgs/sec$.

Recoverable Memory in DVCM. DVCM enables efficient implementations of cluster applications like the Cluster Recoverable Memory system (*CRMem*), a transaction-processing kernel for memory-resident databases. The transactional model of CRMem is very similar to that introduced by Lightweight Recoverable Virtual Memory (LRVM) [17]. CRMem is implemented as a collection of processes running on separate cluster nodes and sharing a database. Each database element is replicated on two or more nodes and replicas are kept consistent across transaction boundaries. The CRMem implementation that uses DVCM extension modules to support its state-sharing operations attains processing rates of more than $3000 trans/sec$ on a simplified TpcB benchmark.

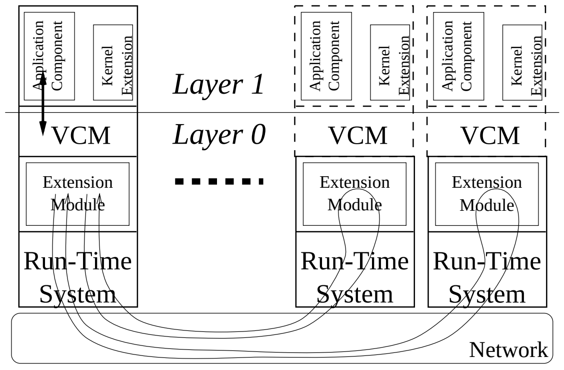

Figure 2. Cluster application using DVCM

DVCM's extensibility is motivated by our belief that no collection of remote operations can provide the best performance for *all* cluster applications. Therefore, DVCM is designed as an infrastructure that applications can extend with communication primitives, implemented as extension modules, that best match their communication patterns (see Figure 2). The DVCM components interface extension modules with host-resident applications, NI resources, and remote extension modules. DVCM provides basic protection mechanisms, such as key-based protection of the application state.

DVCM runs on both the host and the NI. The host-resident components are an application-level library and a kernel extension module. The NI-resident components include the DVCM infrastructure and the extension modules.

Fundamental for our approach is the existance of unused NI resources. Programmable NIs built using commodity CoProcessors are characterized by a slack between the available and the required processing capabilities and memory on the card. Technologies like system-on-a-chip [10, 14] contribute to increasing these resource slacks.

Experimental Platform. The DVCM implementation described in this paper uses Fore SBA-200E ATM cards in a cluster of Sun Ultra I 170 workstations running Solaris 2.5. The Fore cards are ATM adapters for Sun's proprietary SBUS I/O architecture, and they feature an Intel i960CA microprocessor clocked at 25MHz, 256 KBytes of SRAM, and special-purpose hardware for ATM-related operations.

Paper Overview. The next section presents the NI-resident components of the DVCM. Section 3 describes the CRMem system and how DVCM extension modules are used to improve its performance. Section 4 discusses related work.

2 NI-resident DVCM

The NI-resident DVCM is organized in three layers (see Figure 2): (1) the VCM-based interface, (2) the DVCM extension modules, and (3) the DVCM run-time system. In this section, we briefly describe the first two layers and thoroughly describe and evaluate the run-time system; a detailed description of the first two layers is included in [16, 15].

VCM-based Interface Layer. On each workstation in the cluster, a local NI may be abstracted to the local processes as a Virtual Communication Machine (*VCM*). The main components of the VCM are the address space and the extensible instruction set. The VCM address space is the union of the memory pages registered with the NI by local applications. Host CPU and NI CoProcessor use the pages in the VCM address space to communicate with each other. The VCM instruction set consists of core instructions and application-specific instructions. The VCM-based layer implements the core instructions and dispatches the application-specific instructions for execution to the DVCM extension modules. The union of the VCM instruction sets and address spaces in the cluster composes the DVCM instruction set and address space, respectively.

DVCM Extension Modules. A DVCM extension module consists of state variables, event handlers, and initialization code. State variables are allocated in NI memory. Extension modules include three types of event handlers: instruction handlers, control message handlers, and timeout handlers. The module's initialization code registers the application-specific instructions and the types of control messages implemented by the module with the DVCM.

2.1 Run-Time System

The DVCM run-time system serves two purposes. First, it provides a collection of NI resource abstractions to be used by the extension modules, thereby isolating them from the details of the NI hardware. Second, the run-time system implements a set of services necessary to a wide range of extension modules. This obviates the need for replicating functionality in multiple extension modules and facilitates the development of new extension modules. Note that given our focus on experimenting with and evaluating the novel capabilities of the DVCM, the current implementation does not address all of the security issues generally associated with an extensible distributed system. To account for the overheads induced by the protection mechanisms, the current implementation checks protection keys before memory accesses; however, it does not include any key generation or exchange algorithms.

Architecture. The DVCM run-time system integrates two sets of primitives. A set of low-level primitives assists extension modules in using the NI resources (i.e., CoProcessor, memory, and link-level registers), and in signaling local applications. The high-level primitives implement timeout services and low-overhead, reliable communication between CoProcessors.

Implementation and Performance The implementation of the DVCM run-time system is strongly influenced by two factors. First, the NI has significantly fewer computational and memory resources than the host. Second, on some NI cards, the CoProcessor performs several compute-intensive tasks related to the link-level messaging layer such as AAL5 segmentation and reassembly.

Addressing these limitations, we made the following implementation decisions: the run-time system and the extension modules share the same protection domain, the run-time system code is single-threaded, and the extension module handlers are never preempted. The unique control thread alternates between dispatching extensions to handle incoming messages and executing instructions issued by local applications. This approach eliminates the overheads present in multithreaded implementations or in implementations featuring multiple protection domains.

The rest of this section describes the implementation of the most important run-time system elements and their performance characteristics. The measurements are taken with a 40 $nsec$-resolution clock.

CPU-CoProcessor Synchronization. CPU and CoProcessor may synchronize using interrupts or polling. The overhead of using interrupts is relatively high, namely the latency of a null interrupt handler averages $\approx 9\mu secs$ and the latency of a null application-level signal handler is $\approx 49\mu secs$. To hide this relatively high latency, the run-time system does not wait for the signal acknowledgment but checks for the acknowledgement of the previous signal before sending a new one. Polling-based synchronization is faster: the CPU acknowledges a CoProcessor update of an NI memory variable in $\approx 2\mu secs$ and the CoProcessor acknowledges a CPU update of a host memory variable in $\approx 5\mu secs$.

NI Heap. For improved predictability, the heap is implemented as a buddy system. The average costs of heap $alloc$ and $free$ are $\approx 3.7\mu secs$ and $\approx 5\mu secs$, respectively.

Timeout Services. The implementation of timeout services trades accuracy for overhead and predictability. The underlying data structure is similar to the Timing Wheel. Scheduling and canceling computations are done in constant time. The DVCM run-time system checks the timeout data structure between handler executions rather than using interrupts. This is because the interrupt is likely to occur while it is accessing NI registers and, on most NIs, independent sequences of NI register accesses cannot be interleaved.

Control Message Layer. Control messages are short, typed, point-to-point messages. In the current implementation, control messages are no larger than four ATM cells (i.e., 176 byte payload). The run-time system ensures that control messages are delivered reliably and in-order, and dispatches them based on their type. Control message reliability is implemented using a sliding-window protocol and retransmissions. The window size is only a few dozen messages because large bursts of messages between the same pair of NIs are unlikely and because round-trip times are short. Acknowledgements indicate received and lost messages and they are sent after a certain number of control messages are received or immediately, when requested by the sender.

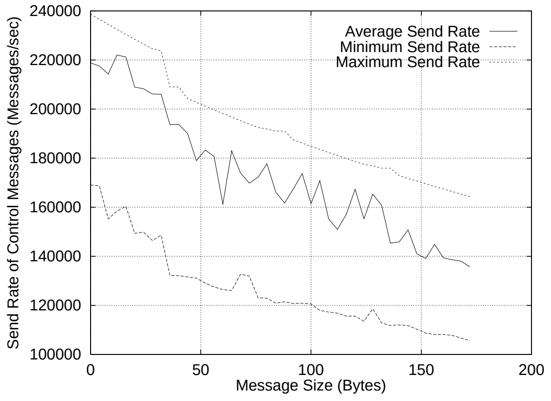

Figure 3. Control Messages: Send Rate

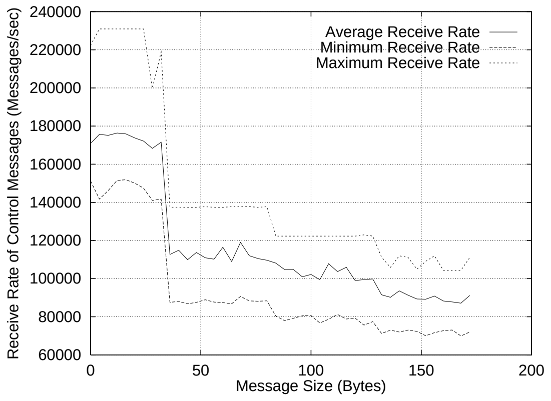

Figure 4. Control Messages: Receive Rate

Figures 3 and 4 present the rates of sending and receiving control messages. The message burst is equal to the size of the sliding window. As expected, send overheads are smaller than receive overheads, as reflected in the difference between the achievable send and receive rates. Send rates decrease smoothly because the cost of padding outgoing messages to the next ATM cell boundary is relatively insignificant. In contrast, receive rates decrease in steps because the receiver has to handle an entire ATM cell before determining how much of its content is useful data. Both send and receive rates are an order of magnitude higher than the rates achievable at application level with the same hardware [16]. This is because (1) CoProcessor-to-CoProcessor messaging requires lower CoProcessor overhead than application-level messaging, and (2) I/O bus

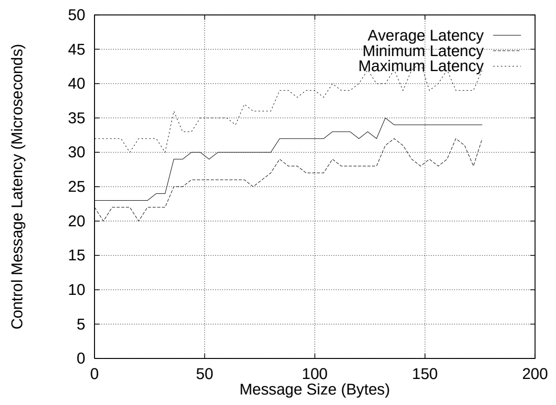

Figure 5. Control Message Latency

traversal is removed from the message path at both source and destination nodes.

Figure 5 plots the one-way latency of control messages, i.e., half of the measured round-trip latency. For these measurements, we use an extension module that defines a DVCM instruction and two control message types, PING and PONG. The instruction parameters indicate the number of PING-PONG messages to be exchanged, the message size, and the remote host.

Control message latencies are lower than half the application-level latencies, which start at $60 \mu secs$ for single ATM cell messages [16]. The average latencies of a one-cell control messages is 23 $\mu secs$. Separately, we measured the average latency of a one-cell datagram to be 18 $\mu secs$. Therefore, the cost of ensuring reliable, in-order delivery by the CoProcessor of a one-cell messages is $\approx$ 5 $\mu secs$.

These results validate our approach of transferring communication intensive application modules to the NI.

3 Cluster Recoverable Memory

The Cluster Recoverable Memory (CRMem) system offers simple transactional guarantees on virtual memory segments in the cluster. CRMem extends the LRVM model [17] with application-configurable support for concurrent transaction execution. Each CRMem segment is replicated on two or more nodes to protect its content from local failures. One of these hosts, called the segment's *home node*, coordinates cluster-wide accesses to the segment's memory locations. CRMem uses a memory-resident *commit log*, which is stored in a circular buffer, substantially larger than the maximum number of concurrent transactions. Each node has its own commit log, replicated on one of the other nodes.

CRMem can perform in two configurations. In the *symmetric configuration*, transactions are executed concurrently on any of the cluster nodes, and all of the segment's replicas backup each other. In the *mirror configuration*, transactions are performed on a single node, called *primary host*. The segment replicas on the other hosts act only as backup for the replica on the primary host. The primary host is the home node for all the segments. The CRMem transactional model is the same in both configurations.

This section describes a CRMem implementation that leverages the extensibility of the DVCM architecture by implementing performance critical operations as DVCM extension modules.

3.1 CRMem Operations

CRMem transactions consist of read and write memory operations on segments of recoverable virtual memory. CRMem segments are either temporary storage, allocated by the application, or memory image of a data file, read from a permanent storage medium. Host characteristics determine the maximum number of CRMem segments and their maximum size.

An application component registers with CRMem using the *InitCRMem* call. The $\{$*Begin,Commit,Abort*$\}$ *Transaction* calls have the usual semantics. All of the memory segments used in a transaction must be mapped or initialized locally before the transaction starts. Temporary recoverable memory segments are allocated with the *InitSegment* call; storage-backed segments are mapped into the main memory with the *MapSegment* call. Before executing a memory operation inside a CRMem transaction, the application must use the *SetRange* call to specify the range of memory locations affected by the operations and the operation type (read or write). The latter parameter is an extension to the LRVM model introduced to support concurrent transaction execution.

The main interface between an application and CRMem is a routine that executes one transaction at a time using the $\{$*Begin,Commit,Abort*$\}$*Transaction* calls. CRMem can execute concurrently several instances of this routine, in separate user-level threads. Distributed transactions can be built based on CRMem primitives. CRMem does not support nested transactions and it does not control the I/O operations performed inside a transaction.

3.2 Concurrency Control

Each CRMem segment is associated a set of locks and a *lock-mapping function* at initialization time. The function maps a segment range into the subset of locks that have to be acquired before accessing the memory locations in the range. All of the segment replicas are associated the same number of locks and the same lock-mapping function.

Transaction synchronization is *transparent* to the application components. The *SetRange* procedure invokes the segment's mapping function and acquires the corresponding locks on the local node and on the segment's home node, if different.

Cluster applications using CRMem transactions can implement *application-specific policies* for concurrency control by defining lock-mapping functions that exploit the semantics of data structures stored in the associated segments. For instance, an application that updates a search tree may use different concurrency control mechanisms when rebalancing disjoint vs. overlapping subtrees or when updating data vs. link fields of the tree nodes. Alternatively, an application can disable CRMem concurrency control mechanisms by using lock-mapping functions that map every segment to the empty set. Applications are responsible for ensuring that the enabled concurrency control policy is deadlock- and livelock-free.

3.3 Implementations

To evaluate the benefits of leveraging DVCM extensibility, we built several CRMem implementations, which are identical except for the implementations of the communication-related operations: the *remote 'atomic' write*, the *remote lock acquire*, and the *remote lock release*. *Remote 'atomic' write* is issued by the *CommitTransaction* call to update remote replicas of the modified segments and of the commit log. The backup node executes the update only after receiving all of the messages describing the operation. *Remote-lock acquire* is issued by *SetRange* to acquire locks on the home node of a segment. The operation is issued after all of the necessary locks are acquired on the local node. *Remote-lock release* is issued by *CommitTransaction* and *AbortTransaction*. *Remote-lock acquire/release* are not used in the mirror configuration.

The implementation called *CoProcessor* uses two DVCM-extension modules to implement the three operations. The implementations called *CPU Polling* and *CPU Interrupt* use the CoProcessor only for messaging; the three operations are implemented entirely by the host CPU. In the *CPU Polling* implementation, the application continuously checks for new messages while in the *CPU Interrupt* implementation, the NI interrupts the CPU and triggers a DVCM signal for each message received. The DVCM module supporting *CPU Polling/Interrupt* implements reliable, user-level messaging. The DVCM modules supporting the *CoProcessor* implementation are described in Section 3.4.

CRMem Locks. CRMem locks are implemented as memory locations in the application address space mapped in the local VCM address space. A lock's memory must be in the VCM address space because both CPU, on behalf of the local application component, and NI CoProcessor, on behalf of a remote application component, access the lock.

The size and alignment of a lock's memory representation should be selected such that the 'closest' processor (CPU or CoProcessor) can execute atomic operations on the representation and the other processor can write into a *proper subset* of the representation. Host memory should be used for locks that are accessed most often locally, while NI memory should be used for locks that are accessed most often remotely.

For instance, our CRMem implementations use an eight-byte word to represent a host-resident lock because the UltraSPARC CPU features 64-bit atomic operations and the i960 CoProcessor can write into the first or the last four bytes of the lock. More specifically, the *lower half* of the lock representation is used for local synchronization and it is set/reset by the CPU using atomic memory instructions. This location contains the ID of the local thread holding the lock, if any. The *upper half* of the lock representation is used for cluster-wide segment synchronization and it is set/reset by the local CoProcessor using DMA operations. This location is used only on the segment's home node and it contains the ID of the remote node holding the lock, if any. A free lock is represented by a null value.

3.4 DVCM Support for CRMem

In this section, we describe the two DVCM extension modules that implement the remote atomic write and remote-lock operations in the *CoProcessor* CRMem.

Remote Atomic Write. The DVCM extension module implementing the remote atomic write defines one instruction and one control message types. The instruction handler packs protection key, destination addresses, and update content into successive control messages. The last control message generated by this operation requires immediate acknowledgement. On the source, i.e., primary node, the write operation completes when the source node receives the acknowledgement for the last control message. On the backup node, the processing of incoming control messages is postponed until the last control message is received.

This instruction gathers/scatters updates from/to the origin/backup node. Before reading from or writing into any CRMem segments, the CoProcessor checks the protection keys passed as parameters.

Remote-Lock Acquire/Release. The DVCM extension module implementing the remote-lock operations defines two instructions and two control message types. The protection key and addresses of the remote locks to be acquired or released are sent to the segment's home node using control messages. Our implementations limit the number of locks handled by an operation to 39, which is the maximum number of remote lock addresses that fit in one control message.

The local execution of a lock operation packs the protection key and the addresses of the locks to be acquired/released into a control message. To simplify the description, we assume that the lock operation targets one lock only. The remote execution of an acquire-lock operation first reads the entire lock representation (eight bytes). Second, if all are zero the ID of the requesting node is written into the *upper half*. Third, the CoProcessor reads the *lower half* of the lock to verify that no local acquire operation was successfully executed before the write of the *upper half*. If the *lower half* is non-zero, the *upper half* is cleared and failure is reported to the requesting node. Otherwise, success is reported. The remote execution of a release-lock operation clears the appropriate half of the lock representation.

Multiple-Lock Acquire Optimization. A remote-lock acquire operation has all-or-none semantics. The implementation exploits this characteristic to hide the relatively high latency of DMA read operations. More specifically, rather than acquiring the requested locks one at a time, the implementation reads all their representations in one step. If all locks are free, it writes their *upper halves* in the second step. Finally, it verifies all their *lower halves* in the third step. This optimization is enabled by the fact that the CoProcessor reads from host memory are non-blocking. Batching memory accesses helps hiding some of the I/O bus latency.

3.5 Transaction Processing Applications

The benefits of using DVCM extension modules for CRMem remote operations are evaluated with two transaction-processing applications. The first application, called 'synthetic transaction application', handles 10,000 three-field records grouped in one recoverable memory segment. The segment is associated 1,000 locks and the uniform lock mapping function[1]. Each transaction selects a random record and stores the sum of the first two fields into the third one. The application executes 5,000 transactions.

The second application, called 'simplified TpcB benchmark', is similar to the TPC-B transaction-processing benchmark [1]. This application handles 10,000 account records, 32 branch records, 256 teller records, and 100 history records grouped in four separate segments. Every ten account records share one lock, each branch record is protected by its own lock, every two teller records share a lock, and each history record is protected by its own lock. Each transaction selects a random account record and teller; the teller selection determines the branch record. The transaction adds or subtracts a random amount from the account, teller, and bank balances and it records the operation in a history record. Only the most recent 100 operations are maintained. The application executes 5,000 transactions.

[1]This mapping function distributes records uniformly to the locks.

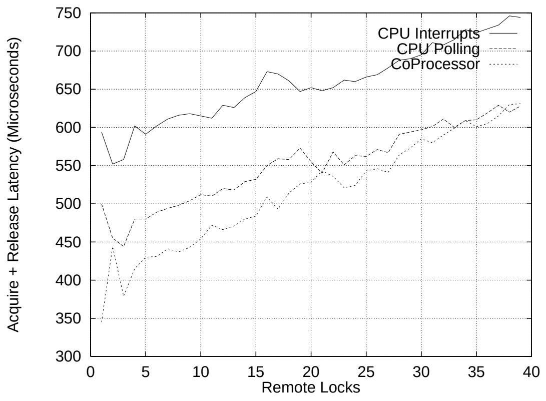

Figure 6. Remote Lock Implementations

3.6 Performance Characteristics

This section evaluates the four CRMem implementations focusing on remote-lock latencies and transaction rates. First, we evaluate the implementations of CRMem locks. Second, the synthetic transaction application is used to compare the three implementations. Third, we evaluate the *CoProcessor* implementation with the simplified TpcB benchmark.

CRMem Locks. Figure 6 shows the combined latency of a pair of remote-lock operations (acquire + release) in the *CPU Interrupt*, *CPU Polling* and *CoProcessor* implementations. In all of these implementations, the amount of processing on the remote node grows with the number of locks handled. The combined latency of the remote acquire/release operations can be approximated by the formula $K + O * locks$, where O and K are implementation-specific constants and $locks$ is the number of remote locks handled. The O factor is primarily determined by the speed of the processor handling the locks. The K factor represents the network latency and the cost of passing the message from CoProcessor to CPU and getting the results back.

The *CPU Interrupt* and *CPU Polling* implementations have equal O factors and different K factors. The difference is twice the latency of an application-level signal handler ($49 \mu secs$). The *CoProcessor* implementation has the smallest K factor because it is not affected by the CoProcessor-CPU synchronization. However, this implementation has the highest O factor because the CoProcessor is slower than the CPU and because the per-lock CoProcessor overhead is higher than the CPU overhead. As a result, the *CoProcessor* implementation has a lower latency than the *CPU Polling* implementation when handling a small number of locks; as the number of locks increases, the latency difference gets smaller.

The CRMem lock implementation is one example of functionality that can be executed more efficiently on the

CoProcessor. Similar experiments, not included in this paper, show that the *CoProcessor* implementation of the remote atomic write has lower latencies than the other two implementations.

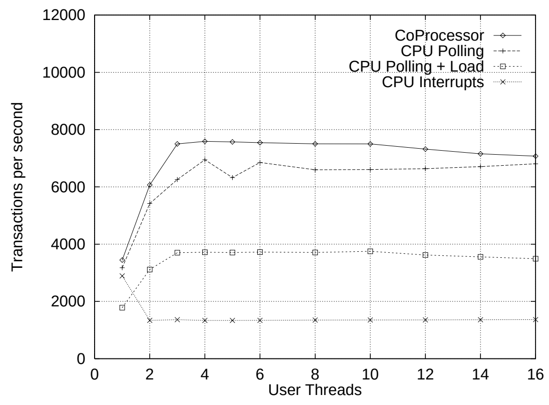

Figure 7. Synthetic Transaction Application

Synthetic Transaction Application. Figure 7 plots the transaction rates of the synthetic transaction application in mirror configuration with one backup node. The network latency is hidden with a degree of parallelism larger or equal to four. Among the three implementations using the NI, the *CoProcessor* implementation performs best. As expected, the *CPU Interrupts* implementation has the lowest transaction rates because of the signal and interrupt overheads.

Adding a CPU-intensive load on the backup node reduces by half the transaction rates of the *CPU Polling* implementation. In contrast, similar loads (not shown in the Figure) do not affect the *CoProcessor* implementation because this implementation does not use the CPU on the backup node for transaction processing.

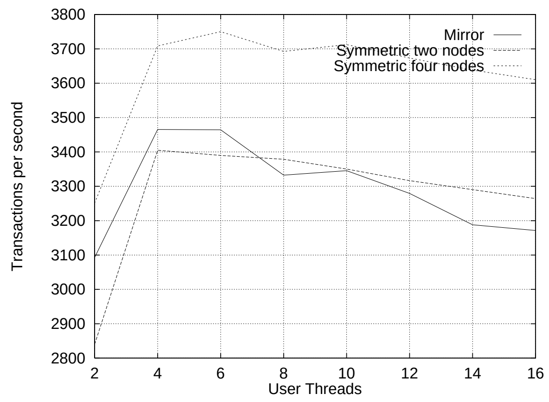

Figure 8. Simplified TpcB Benchmark

Simplified TpcB Benchmark. Figure 8 plots the transaction rates of the simplified TpcB benchmark in mirror configuration with one backup node and in two- and four-node symmetric configurations. Because of the higher transaction overheads, this application has significantly lower transaction rates than the synthetic transaction application. Running four or more transactions in parallel hides the inter-node latencies in our cluster.

The transaction rates in mirror configuration and in two-node symmetric configuration are similar despite the fact that the latter configuration uses an additional CPU. The gain enabled by additional processing capacity is offset by the need for cluster-wide synchronization. The four-node configuration achieves only $\approx 10\%$ higher transaction rates than the two-node configuration although it uses twice as many resources. We believe that the scalability of this application is limited by its inability to divide the CRMem locks uniformly among the home nodes in the cluster.

4 Related Work

Over the past several years, research efforts in operating systems and network adapter architectures have focused on improving communication performance over high-speed interconnects such as ATM, FDDI, HIPPI, and Myrinet. The main interest is in building faster and lighter messaging primitives or distributed shared-memory systems for clusters of workstations [3, 4, 13, 19]. While we are also interested in achieving better communication performance, our focus is on identifying an application-NI interface that allows efficient implementations of application-specific mechanisms for state sharing across the cluster. The DVCM abstraction is our solution to this challenge. The DVCM focus on application extensibility makes it different from CoProcessor-based I/O architectures for storage controllers and NIs on mainframes and high-end file servers, which only control the attached I/O devices or run a customized TCP/IP stack.

The benefits of extending and configuring the resource-management substrate have been studied for a wide variety of platforms, ranging from OS kernels [2, 5, 20], to NI cards [7, 9], and node controllers for distributed-memory parallel machines [8, 18]. The DVCM architecture builds upon this research. However, DVCM is unique in its approach to assembling a *tightly coupled* computing platform out of commodity components by leveraging current trends in the architecture of these components.

Previous research has explored lightweight kernels for transaction processing, like LRVM [17] and Rio Vista [12], and systems leveraging remote cluster memory, like Harp [11] and GMS [6]. CRMem shares many concepts with these projects. However, transferring state sharing and global-synchronization functionality 'closer' to the network has not been previously explored for systems like these.

By leveraging the DVCM capabilities, CRMem outperforms and extends LRVM and Rio Vista on several accounts. CRMem enables higher transaction rates than

LRVM because accessing remote cluster memory is faster than accessing a locally attached disk. Transaction overheads in CRMem are higher than in Rio Vista, but its fault-protection mechanisms are superior. Namely, the CRMem helps ensure reliable transaction processing even in the presence of natural disasters by 'mirroring' transactions at remote, 'safe' backup facilities. This is in contrast to efforts like LRVM and Rio Vista, which rely on 'local' backup media. In addition, CRMem allows for concurrent transaction executions and its concurrency control mechanisms can take advantage of the application's data structure semantics.

5 Conclusions

This paper presents the DVCM, a software communication architecture for clusters of workstations using programmable NIs. The DVCM advocates the transfer of selected application functionality from the host to the NI, where it can be executed 'closer' to the network. DVCM clusters appear to the applications with NI-resident modules as being more tightly coupled than clusters using traditional, CPU-centric communication mechanisms.

We demonstrate how DVCM can be used to support CRMem, a transaction-processing kernel for memory-resident databases. Our experiments demonstrate that using the CoProcessor for CRMem's remote operations is beneficial. Together with a group of computer architecture researchers, we explore alternative NI designs and implementations, and evaluate them with other applications.

References

[1] TPC Benchmark B Standard Specification. *Technical report, Transaction Processing Performance Council*, Aug. 1996.

[2] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety, and performance in the SPIN operating system. *Proceedings of the 15th ACM Symposium on Operating System Principles*, Dec. 1995.

[3] G. Buzzard, D. Jacobson, M. Mackey, S. Marovich, and J. Wilkes. An implementation of the Hamlyn sender-managed interface architecture. *Proceedings of the 2nd Symposium on Operating Systems Design and Implementations*, Oct. 1996.

[4] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: Efficient Support for Reliable, Connection-Oriented Communication. *Proceedings of Hot Interconnects V*, Aug. 1997.

[5] D. Engler, M. Kaashoek, and J. O'Toole. Exokernel: an operating system architecture for application-level resource management. *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, Dec. 1995.

[6] M. J. Feeley, W. E. Morgan, F. P. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing Global Memory Management in a Workstation Cluster. *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 325–336, Dec. 1995.

[7] M. E. Fiuczynski, B. N. Bershad, R. P. Martin, and D. E. Culler. SPINE - An Operating System for Intelligent Network Adapters. *University of Washington, Department of Computer Science and Engineering, TR-98-08-01*, Aug. 1998.

[8] M. Heinrich, J. Kuskin, D. Ofelt, J. Heinlein, J. Baxter, J. P. Singh, R. Simoni, K. Gharachorloo, D. Nakahira, M. Horowitz, A. Gupta, M. Rosenblum, , and J. Hennessy. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 274–285, Oct. 1994.

[9] Y. Huang and P. K. McKinley. Efficient Collective Operations with ATM Network Interface Support. *Proceedings of the 1996 International Conference on Parallel Processing*, pages 34–43, Aug. 1996.

[10] P. M. Kogge. EXECUBE – A New Architecture for Scalable MPPs. *Proceedings of the 1994 International Conference on Parallel Processing*, 1994.

[11] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the HARP File System. *Proceedings of the 13rd ACM Symposium on Operating Systems Principles*, pages 226–238, Oct. 1991.

[12] D. E. Lowell and P. M. Chen. Free Transactions with Rio Vista. *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 92–101, Oct. 1997.

[13] S. Pakin, M. Laura, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. *Supercomputing*, Dec. 1995.

[14] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A Case for Intelligent RAM: IRAM. *IEEE Micro*, Apr. 1997.

[15] M.-C. Roşu and K. Schwan. Sender Coordination in the Distributed Virtual Communication Machine. *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, pages 322–329, Jul. 1998.

[16] M.-C. Roşu, K. Schwan, and R. Fujimoto. Supporting Parallel Applications on Clusters of Workstations: The *Virtual Communication Machine*-based Architecture. *Cluster Computing*, 1:1–17, Jan. 1998.

[17] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight Recoverable Virtual Memory. *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages 146–160, Dec. 1993.

[18] P. Steenkiste. Network-based multicomputers: a practical supercomputer architecture. *IEEE Transactions on Parallel and Distributed Systems*, 8(7):861–875, Aug. 1996.

[19] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Dec. 1995.

[20] D. A. Wallach, D. R. Engler, and M. F. Kaashoek. ASHs: Application-specific handlers for high-performance messaging. *Proceedings of ACM SIGCOMM'96 Conference on Applications, Technologies, Architectures and Protocols for Computer Communication*, pages 40–52, Aug. 1996.