Submitted to the *2002 ACM International Symposium on Computer Architecture (ISCA'02)*

Experiences with VI Communication for Database Storage

Yuanyuan Zhou, Cezary Dubnicki, Suresh Jagannathan, and James F. Philbin
StorageNetworks, Inc., Princeton, NJ–08540, USA
Angelos Bilas
Department of Electrical and Computer Engineering,
University of Toronto, Toronto Ontario M5S3G4, Canada
Kai Li
Department of Computer Science,
Princeton University, Princeton, NJ–08544, USA

ABSTRACT

This paper examines how VI–based interconnects can be used to improve I/O path performance between a database server and a storage system. We design and implement a software layer, DSA, that is layered between the application and VI. DSA takes advantage of specific VI features and deals with many of its shortcomings. We provide and evaluate one kernel–level and two user–level implementations of DSA. These implementations trade transparency and generality for performance at different degrees, and unlike research prototypes are designed to be suitable for real–world deployment. We present detailed measurements using a commercial database management system with both microbenchmarks and industrial database workloads on a mid–size (4 CPU) and a large (32 CPU) database server.

Our results show that VI–based interconnects and user–level communication can improve all aspects of the I/O path between the database system and the storage backend. We also find that to make effective use of VI in I/O intensive environments we need to provide substantial additional functionality than what is currently provided by VI. Finally, new storage APIs that help minimize kernel involvement in the I/O path are needed to fully exploit the benefits of VI–based communication.

1 INTRODUCTION

User–level communication architectures have been the subject of recent interest because of their potential to reduce communication related overheads. Because they allow direct access to the network interface without going through the operating system, they offer applications the ability to use customized, user–level I/O protocols. Moreover, user–level communication architectures allow data to be transferred between local and remote memory buffers without operating system and processor intervention, by means of DMA engines (RDMA). These features have been used with success in improving the performance and scalability of parallel scientific applications. The VI (Virtual Interface) architecture is a well–known, industry–wide standard for system area networks based on these principles that has spawned a number of initiatives such as Infiniband [18] and DAFS (Direct File Access File Storage) [10] that target other important domains.

In this paper, we study the feasibility of leveraging VI–based communication to improve I/O performance and scalability for storage–centric applications and in particular database applications executing real–world online transaction processing loads. An important issue in database systems is the overhead of processing I/O operations. For realistic on–line transaction processing (OLTP) database loads, I/O can account for a significant percentage of total execution time. Achieving high transaction rates, however, can only be realized by reducing I/O overheads, providing more CPU cycles for transaction processing. For this reason, storage architectures strive to reduce costs in the I/O path between the database system and the disk subsystem.

In this work we are primarily interested in examining the effect of user–level communication on block I/O performance for database applications. We focus on improving the I/O path from the database server to storage by using VI–based interconnects. High–performance database systems typically use specialized storage area networks (SANs), such as Fibre Channel (FC), for the same purpose. We propose an alternative storage architecture called VI–attached Volume Vault (V3) that consists of one or more database servers which communicate with a storage server cluster using VI. Each V3 storage node in our cluster is a commodity PC consisting of a collection of low–cost disks, large memory, a VI–enabled network interface, and one or more processors. V3 is designed as a new generation database storage product, and as such addresses issues dealing with reliability, fault-tolerance, and scalability that would not necessarily be considered in a research prototype. V3 has been deployed and tested in customer sites for more than 6 months.

VI does not deal with a number of issues that are important for storage applications. For this reason, we design a new block-level I/O module, DSA, that is layered between the application and VI (Figure 1). DSA uses a custom protocol to communicate with the actual storage server, the details of which are beyond the scope of this paper. DSA takes advantage of specific VI features and addresses many of its shortcomings with respect to I/O–intensive applications and in particular databases. DSA uses RDMA capabilities and utilizes low latency overheads for initiating I/O operations and accessing the network interface. More importantly, DSA deals with issues not addressed by VI. For example, VI does not provide flow control, is not suited for applications with large numbers of communication buffers or large numbers of asynchronous events, and most existing VI implementations do not provide strong reliability guarantees. In addition, although certain VI features such as RDMA can benefit even kernel–level, legacy APIs, the current VI specification is not well–suited for this purpose.

We evaluate three different implementations of DSA, one kernel–level implementation and two user-level, that trade transparency and generality for performance at different degrees. We present detailed measurements using a commercial database management system, Microsoft *SQL Server 2000*, with both microbenchmarks and industrial database workloads (*TPC–C*) on a mid–size (4 CPU) and a large (32 CPU) database server with up to 12 Terabytes of disk storage.

Our results show that:

1. VI–based interconnects and user–level communication can improve all aspects of the I/O path between the database system and the storage backend, and result in transaction rate improvements of up to 20% for large database configurations.

2. To make effective use of VI in I/O intensive environments requires substantial modifications and enhancements to flow control, reconnection, interrupt handling, memory registration, and lock synchronization.

3. New storage APIs that help minimize kernel involvement in the I/O path are needed to fully exploit the benefits of VI–based communication.

The rest of the paper is organized as follows. Section 2 presents the V3 architecture and the various DSA implementations. Section 3 presents our performance optimizations for DSA. Section 4 presents our experimental platforms and Sections 5 and 6 discuss our results. Finally, we present related work in Section 7 and draw our conclusions in Section 8.

2 SYSTEM ARCHITECTURE

To study feasibility and design issues in using user–level communication for database storage, we define a new storage architecture that allows us to attach a storage backend to database systems through a VI interconnect. This section provides a brief overview of the VI-attached Volume Vault (V3) architecture and then focuses on DSA and its implementations.

2.1 V3 Architecture

Figure 1 shows the overall V3 architecture. A V3 system consists of database servers (V3 clients) and storage nodes (V3 servers). Client configurations can vary from small–scale uniprocessor and SMP systems to large–scale multiprocessor servers. Clients connect to V3 storage nodes through the VI interconnect.

Each V3 server provides a virtualized view of a disk (V3 disk or volume). To support industrial workloads, each V3 server node has redundant components (power supplies, system disks, network interfaces, etc.) to avoid single points of failure. Each V3 disk consists one or many physical disks attached to V3 storage nodes. With current disk technologies,

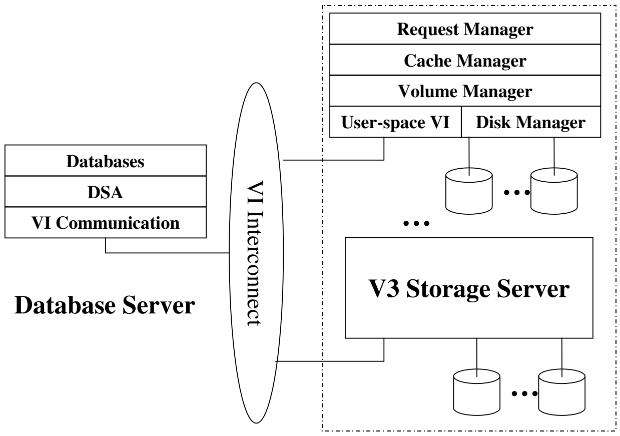


Figure 1: V3 Architecture overview.

a single V3 disk can provide more than 2TB of storage. Since existing VI networks support a large number of nodes (up to 128), a multi–node V3 backend can provide more than 250TB of storage. V3 uses large main memories as disk buffer caches to help reduce disk latencies [24]. While the V3 backend can provide storage to any application, it is designed specifically with database applications in mind. A V3 server is structured differently from a typical disk subsystem. It is controlled by custom software (Figure 1) that includes several modules: a request manager, cache manager, a volume manager, and a disk manager. The server employs a lightweight pipeline structure for the I/O path that avoids synchronization overheads and allows large numbers of disk requests to be serviced concurrently. Since the V3 server software runs at user level, it uses user level VI to communicate with its clients. Because the focus of this paper is on feasibility issues in using VI for database storage, we do not elaborate further on V3's internal structure, although we refer to its properties when appropriate.

2.2 DSA Implementations

DSA is a client-side, block-level I/O module specification that is layered between the application and VI. DSA deals with issues not supported in VI but are necessary for supporting storage I/O intensive applications and takes advantage of VI-specific features.

The new features provided by DSA include flow control, retransmission and reconnection, which are critical for industrial–strength systems, and performance optimizations for alleviating high-cost operations in VI. DSA's flow control mechanism allows for large numbers of outstanding requests and manages client and server buffers appropriately to avoid overflow errors. DSA includes optimizations for memory registration and deregistration, interrupt handling, and lock synchronization issues to minimize their performance impact. These issues are discussed in greater detail in Section 3.

Moreover, DSA takes advantage of a number of VI features. It uses direct access to remote memory (RDMA) to reduce overheads in transferring both data and control information. I/O blocks are transferred between the server cache and application buffers without copying. Issuing I/O operations can be done with a few instructions directly from the application without kernel involvement. Moreover, I/O request completion can happen directly from the V3 server with RDMA. Finally, DSA takes advantage of the low packet processing overhead in the NIC and allows for large numbers of overlapping requests to maximize I/O throughput even at small block sizes.

We provide one kernel and two user–level implementations of DSA. Our kernel–level implementation is necessary to support storage applications on top of VI–based interconnects by using existing operating system APIs. Our user–level implementations allows storage applications to take advantage of user–level communication. We briefly discuss each implementation below.

Kernel–level Implementation: To leverage the benefits of VI in kernel–level storage APIs we provide a kernel–level implementation of DSA (*kDSA*). *kDSA* is implemented on top of a preliminary kernel–level version of the VI specification [9] provided by Giganet [16]. The API exported by *kDSA* is the standard I/O interface defined by Windows for kernel storage drivers. Thus, our kernel–level implementation for DSA can support any existing user–level or kernel–level application without any modification. *kDSA* is built as a thin monolithic driver to reduce the overhead of going through multiple layers of software. Alternative implementations, where performance is not the primary concern, can layer existing kernel modules, such as SCSI miniport drivers, on top of *kDSA* to take advantage of VI–based interconnects.

In our work, we optimize the kernel VI layer for use with DSA. Our experience indicates that a number of issues, especially event completions are different from user–level implementations. In particular, we find that although kernel–level VI implementations can provide optimized paths for I/O completions, the user–level specification of the VI API [9] does not facilitate this approach.

User–level Implementations: To take advantage of the potential provided by user–level communication we also provide two implementations of DSA at user level. These implementations differ mainly in the API they export to applications.

wDSA is a user–level implementation of DSA that provides the Win32 API and replaces the Windows standard system library, `kernel32.dll`. *wDSA* filters and handles all I/O calls to V3 storage and forwards other calls to the native `kernel32.dll` library. *wDSA* supports the standard Windows I/O interface and therefore can work with applications that adhere to this standard API without modifications.

wDSA communicates with the V3 server at user–level and it eliminates kernel involvement for issuing I/O requests. Requests are directly initiated from application threads with standard I/O calls. *wDSA* still requires kernel involvement for I/O completion due to the semantics of `kernel32.dll` I/O calls. Since *wDSA* is unaware of an application's I/O semantics, it must trigger an application–specified event or schedule an application–specific callback function to notify the application that may be waiting for this thread to complete upon an I/O request completion. Interrupts are used to receive notifications from VI for V3 server responses. Upon receiving an interrupt, *wDSA* completes the corresponding I/O request and notifies the application. Support for these mechanisms may involve extra system calls, eliminating many of the benefits of initiating I/O operations directly from user–level. Moreover, we find that implementing `kernel32.dll` semantics is non–trivial and makes *wDSA* prone to portability issues across different versions of Windows.

cDSA is the user–level implementation of DSA that provides a new I/O API to applications to fully exploit the benefits of VI–based communication. It primarily consists of 15 calls to handle synchronous or asynchronous read/write operations, I/O completions, and scatter/gather I/Os. Similarly to any customized approach, this implementation trades off transparency for performance. The new *cDSA* API avoids the overhead of satisfying the standard Win32 I/O semantics and hence is able to minimize the amount of kernel involvement, context switches, and interrupts. However, this approach requires cognizance of the database application's I/O semantics, and modification of the application, if necessary, to adhere to this new API.

The main feature of *cDSA* relevant to this work is application–controlled I/O completion mode. Using the *cDSA* interface, applications choose either polling or interrupts as the completion mode for I/O requests. In polling mode, an I/O completion does not trigger any event or callback function, and the application explicitly polls the I/O request completion flag. *cDSA* updates the flag using RDMA directly from the storage node. By doing this, it can effectively reduce the number of system calls, context switches and interrupts associated with I/O completions. An application can switch from polling to interrupt mode before going to sleep, causing I/O completions to be handled similarly to *wDSA*.

To evaluate *cDSA* we use a slightly modified version of Microsoft *SQL Server 2000*[1]. This version of *SQL Server 2000* replaces the Win32 API with the new API provided by *cDSA*. Since the completion flags in the *cDSA* API are also part of the Win32 API, the modifications to *SQL Server 2000* are minor. We note that *cDSA* also supports more advanced features, such as caching and prefetching hints for the storage server. These features are not used in our experiments and are beyond the scope of this paper.

3 SYSTEM OPTIMIZATIONS

In general, our experience shows that VI can be instrumental in reducing overheads in the I/O path. However, we have also encountered a number of challenging issues in using VI–based interconnects for database storage systems including memory registration and deregistration overhead, interrupt handling, and lock synchronization. To deal with these issues, we explore various optimizations for DSA and quantify their impact on performance in Section 6. Because *wDSA* must precisely implement the semantics defined by the Win32 API, opportunities for optimizations are severely limited and not all optimizations are possible.

[1]Note to the reviewers: This modified version of *SQL Server 2000* is provided by Microsoft.

3.1 VI Registration and Deregistration

Memory registration and deregistration are expensive operations that, when performed dynamically, impact performance dramatically. I/O buffers in VI need to remain pinned until the transfer finishes. Current VI–enabled NICs have a limitation on how much memory they can register. For instance, the Giganet cLan card [16] we use allows 1GByte of outstanding registered buffers, and takes about $10\mu s$ to register and deregister an 8K buffer. When the number of registered buffers exceeds this limit, the application needs to deregister memory and free resources on the NIC. The simple solution of pre–registering all I/O buffers at application startup cannot be applied in database systems, since they use large caches and require large numbers of I/O buffers. Given that we need to dynamically manage registered memory, previous work for user–level communication [4, 7] has shown how the NIC can collaborate with host–level software (either kernel or user–level) to manage large amounts of host memory. These solutions have been evaluated in cases where the working set of registered memory is small. However, database systems use practically all available I/O cache for issuing I/O operations so the expected hit ratio on the NIC register table would be low. Moreover, in *SQL Server 2000* virtual to physical mappings can be changed by the application, providing no simple way to invalidate cached registrations without access to source code or interception of system calls.

In our work, we first optimize the existing VI code and we eliminate pinning and unpinning from the registration and deregistration paths. In *kDSA* the Windows I/O Manager performs these operations and handles pinned buffers to *kDSA*. In *cDSA* we use AWE (Address Windowing Extensions) [20] memory to allocate the database server cache. AWE is a feature of Windows for systems that have lagre amounts of physical memory. When memory is allocated as AWE memory it is always pinned.

Finally, we use a new optimization, called *batched deregistration* to reduce the average cost of deregistering memory. VI–enabled NICs usually register consecutive I/O buffers in successive locations in a NIC table. DSA uses extensions to the VI layer (kernel or user–level) to divide this NIC table into small regions of one thousand consecutive entries (4MBytes worth of host memory). Instead of deregistering each I/O buffer when the I/O completes, we postpone buffer deregistration for a short period of time and deregister full regions with one operation when all buffers in the region have been completed. Thus, we perform one deregistration every one thousand I/O operations, practically eliminating the overhead of deregistration when I/O buffers are short–lived. Note that if a single buffer in a region is not used then the whole region will not be deregistered, consuming resources on the NIC. We note that registration of I/Os buffer cannot be avoided, unless we delay issuing the related I/O operation. For this reason, we register I/O buffers dynamically on every I/O.

3.2 Interrupts

Interrupts are used to notify the database system when an asynchronous I/O request completes. To improve performance, database systems issue large numbers of asynchronous I/O requests. In V3, when the client (i.e., database server) receives a response from the server (i.e., V3 storage), DSA notifies the database application. DSA uses interrupts to receive notifications for I/O responses from the NIC. As in most operating systems, interrupt cost is high on Windows, in the order of 5–$10\mu s$ on our platforms. When there is a large number of outstanding I/Os, the total interrupt cost can be prohibitively high, in excess of 20–30% of the total I/O overhead on the host CPU. Interrupt batching is used to reduce this cost.

kDSA uses a novel scheme to batch interrupts. It observes the number of outstanding I/O requests in the I/O path and if this number exceeds a specified threshold it disables explicit interrupts for server responses. Thus, instead of using interrupts, *kDSA* checks synchronously for completed I/Os during issuing new I/O operations. Interrupts are re–enabled if the number of outstanding requests falls below a minimum threshold; this strategy avoids unnecessarily delaying the completion of I/O requests when there is a small number of outstanding I/Os. This method works extremelly well in benchmarks where there is a large number of outstanding I/O operations, as is the case with most large–scale database workloads.

As we described earlier, *cDSA* takes advantage of features in its API to reduce the number of interrupts required to complete I/O requests. In particular, completion flags associated with each outstanding I/O operation are set via RDMA by the storage server upon I/O completion. This flag is polled by the application. If the flag is not set within the polling interval, *cDSA* waits for the interrupt upon I/O completion and signals the database appropriately. Under heavy database workloads this scheme significantly reduces the number of interrupts generated for I/O completions.

3.3 Lock Synchronization

Using run–time profiling we find that a significant percentage of the database CPU is spent on lock synchronization in the I/O path. To reduce the lock synchronization time we optimize the I/O request path to reduce the number of lock/unlock operations, henceforth called synchronization pairs. In *kDSA* we perform a single synchronizaion pair in the send path and a single synchronization pair in the receive path. However, besides *kDSA*, the Windows I/O Manager uses at least two more synchronization pairs in the send and receive paths and VI uses two more, one for registration/deregistration and one for queueing/dequeueing. Thus, there is a total of about 8–10 locks involved in the path of processing a single I/O request. *cDSA* has a clear advantage with respect to locking, since it has control over the full path between the database server and the actual storage. The only locks that are not in our control are the four locks in the VI layer. We carefully optimize the I/O path to minimize synchronization contention, which results in only six locks in the I/O path. In *cDSA* we also lay out data structures carefully to minimize cache misses. Although it is possible to reduce the number of synchronization pairs in VI by replacing multiple fine–grain locks with fewer, coarser–grain locks, preliminary measurements show, that the benefits are not significant. The main reason is that VI locks are per connection and DSA uses many connections to connect to V3 storage nodes in realistic experiments.

Platform	Component	Mid–size	Large
Database Host	CPU	4 x 700 MHz Pentium II	32 x 800 MHz Pentium II
	Memory	4 GB	32 GB
	Hardware Cache (per CPU)	8KB L1I/L1D, 1MB L2	8KB L1I/L1D, 2MB L2, 8 x 32MB L3
	PCI slots	2 x 66MHz, 64Bit	96 x 66MHz, 64Bit
	NICs	4 cLan cards	8 cLan cards
	# Disks in local case	176	640
	OS	Windows 2000 Advanced Server	Windows XP
	Database	Microsoft *SQL Server 2000*	Microsoft *SQL Server 2000*
	Database Size	1,625 warehouses	10,000 warehouses
V3 Server	# V3 Nodes	4	8
	CPU	Dual 700 MHz Pentium II	Dual 700 Mhz Pentium II
	Hardware Cache (per CPU)	8KB L1I/L1D, 1MB L2	8KB L1I/L1D, 1MB L2
	Memory (per node)	2 GB	3 GB
	V3 Cache (per node)	1.6 GB	2.4 GB
	Disk Type	18 GB 10K RPM SCSI	18 GB 15K RPM Fibre Channel
	Disk Controller	UltraSCSI 320	Mylex eXtreme RAID 3000[21]
	Total # Disks	60	640
	Total disk capacity	1 TeraByte	11.5 TeraBytes
	OS	Windows 2000 Workstation	Windows 2000 Workstation

Table 1: Configuration summary for the mid–size and large database setups.

4 EXPERIMENTAL PLATFORM

To cover a representative mix of configurations we use two types of platforms in our evaluation: a mid-size, 4–way and an aggressive, 32–way Intel–based SMP as our database servers. Table 1 summarizes the hardware and software configurations for the two test platforms. The database system we use is Microsoft *SQL Server 2000*. Our mid-size database server configuration uses a system with four, 700MHz Pentium II Xeon processors. Our large database server configuration uses an aggressive, 32–way SMP server with 800MHz Pentium II Xeon processors. The server is organized in eight sub–pods, each with four processors and 32 MBytes of third–level cache (total of 256 MBytes). The server has a total of 32 GBytes of memory organized in four memory modules. The four memory modules are organized in a uniform memory access architecture. Every pair of nodes and all memory modules are connected with a crossbar interconnect with a total of four crossbar interconnects for the four pairs of nodes.

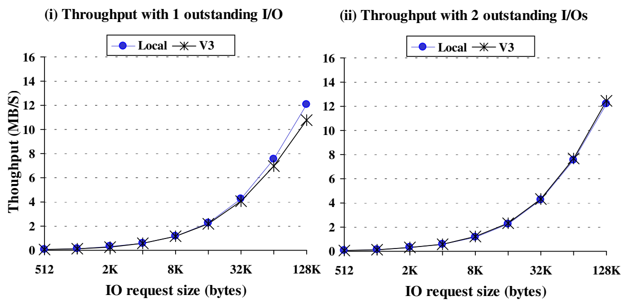
Each V3 storage server in our experimental setup contains two, 700 MHz Pentium II Xeon processors and 2–3 GBytes of memory. Each database server connects to a number of V3 nodes as specified in each experiment. In all our experiments, the same disks are either connected directly to the database server (in the local case), or to V3 storage nodes. Both the mid-size and the large configurations use a Giganet network [16] with one or more network interface cards (NICs) that plug in PCI–bus slots. The user–level communication system we use is an implementation of the VI Specification [12] provided by Giganet [16]. For communicating in the kernel, we use a kernel level implementation of a subset of the VI specification that was initially provided by Giganet and which we optimize for our system. The maximum end-to-end user-level bandwidth of Giganet is about 110 MBytes/s and the one-way latency for a small message (64 bytes) is 7 microseconds.

5 MICRO–BENCHMARK RESULTS

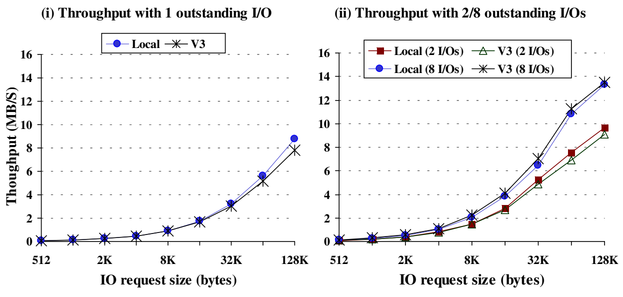
We use various workloads to investigate the base performance of V3 and the different DSA implementations. In the micro–benchmark measurements, we compare the performance of V3 against a local configuration. The V3 configuration uses two nodes, a single application client that runs our micro-benchmark and a single storage node that presents a virtual disk to the application client. The disk appears to the client as a local disk. The local case uses a locally-attached disk without any V3 software. We use *kDSA* as representative of the DSA implementations, and comment on the others where appropriate. For convenience, to distinguish among different V3 configurations when necessary we use the name of the corresponding DSA implementation used; recall all configurations use the same server implementation. Thus, references to *kDSA* means a V3 system that implements DSA in the kernel, references to *wDSA* means a V3 system that implements DSA using a *kernel32.dll* interceptor, and references to *cDSA* means a V3 system that uses a customized user-level block I/O API.

We present three types of statistics for various I/O request sizes: throughput, latency, and execution time breakdowns. The I/O request size is usually fixed in a given database configuration but may vary across database configurations. We use request sizes between 512 and 128K bytes, which cover all realistic I/O request sizes in databases. For these measurements, the cache block size is always set to 8 KBytes.

I/O Throughput: Figure 2 presents throughput for 100% read and write workloads with one or more outstanding I/O operations. In each workload all I/O operations are random. Reads to a V3 disk have 0% cache hit ratio and writes are write–through. With one outstanding I/O, our implementation adds overhead, especially for large I/O request sizes and has throughput 3-10% worse than the local case (Figure 2). However, when the number of outstanding I/Os increases, the throughput difference between a V3 disk and a local disk decreases because requests are pipelined. For



(a) Read Throughput



(b) Write Throughput

Figure 2: Read (left) and write (right) throughput for workloads with 100% random I/O requests.

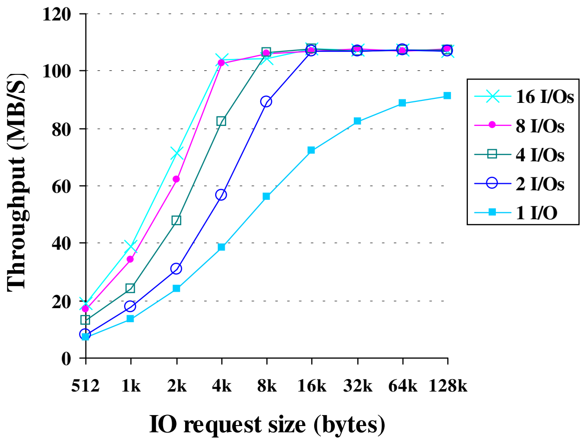


Figure 3: *kDSA* read throughput with 100% cache hit ratio with different number of outstanding I/Os.

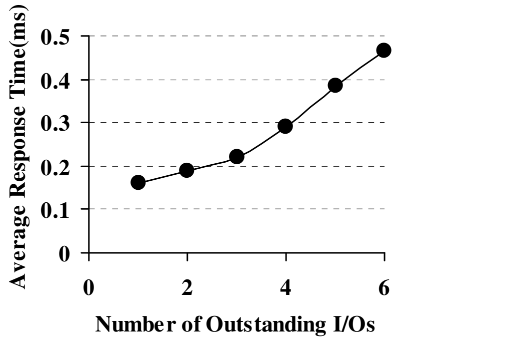


Figure 4: Average response time for an 8 KB read from a V3 cache with different number of outstanding I/Os.

example, V3 can achieve the same read throughput as a local disk when there are 2 outstanding I/O requests, and the same write throughput with 8 outstanding I/O requests (Figure 2b). Since databases always generate more than one outstanding request to tolerate I/O latency, V3 can provide the same throughput as local disks even with a 0% cache hit ratio for realistic database loads.

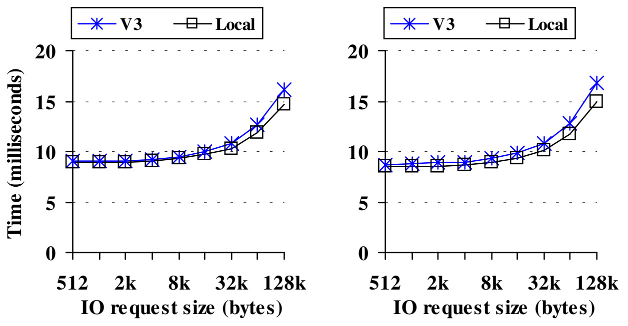
Figure 3 gives the V3 read throughput with 100% cache hit rate. With 1 outstanding read, the throughput reaches a maximum of about 90 MB/s with 128 KB requests. However, when there are more outstanding requests, the peak VI throughput (about 110 MB/s) can be reached at small reqeust sizes. For example, with 4 outstanding reads, the VI interconnect is saturated even with 8KB request. Lower cache hit ratios require higher numbers of outstanding I/Os to saturate the VI interconnect.

Figure 4 shows the average response time for reading an 8KB block from a V3 cache. When the number of outstanding requests is less than 4, the average response time increases slowly. With more than 4 outstanding I/O requests, the VI interconnect is saturated (see Figure 3) with 8K requests. Above this threshold, the average response time increases linearly and is a function of network queuing.

I/O Latency: Figure 5 shows that a V3 server that exhibits a 0% cache hit ratio has similar random read latency as the local case when the read size is less than 64 KBytes. The extra overhead introduced by V3 is less than 3%. For larger I/O request sizes, however, V3 introduces higher overheads, which are proportional to the request size. For example, V3 has around 10% overhead for 128 KBytes reads because of increased data transfer time. In addition, the packet size in the cLan VI implementation is 64KB minus 64 bytes. Therefore, to transfer 128 KBytes requires breaking the data to 3 VI RDMAs. Write latency behaves similarly, For request sizes up to 32 KBytes, V3 has latencies similar to the local case. For larger request sizes, V3 is up to 10% slower due to network transfer time.

Figure 6 presents random read latency with a 100% cache hit ratio. Reading from the V3 storage cache is two orders of magnitude faster than reading from a SCSI disk. It takes about 80 μs to read a 512-bytes block from the V3 cache compared to roughly 9 ms for a local disk read.

We also plot round-trip response time using raw VI calls. The VI test includes several steps: (1) the client registers a receive buffer; (2) the client sends a 64 bytes request to the server; (3) the server receives the request; (4) the server



(a) Read (b) Write

Figure 5: Read (left) and write (right) latency for workloads with 100% random requests.

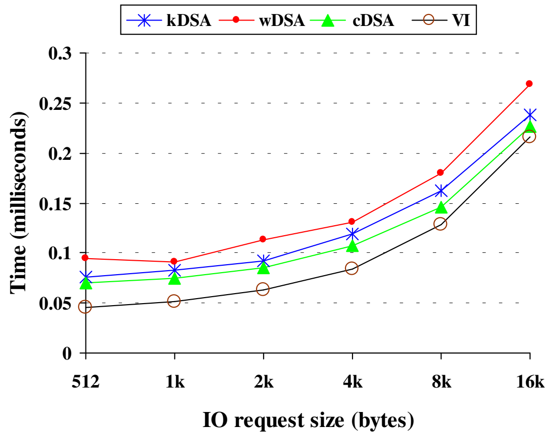


Figure 6: Read latency for workloads with 100% random I/O read requests and 100% cache hit ratio.
.

sends the data of requested sizes from a preregistered send buffer to the client using RDMA; (5) the client receives an interrupt on the VI completion queue; (6) the client deregisters the receive buffer, and repeats. All these steps are necessary to use VI in the I/O path for database storage. In this experiment, we always use polling for incoming messages on the server and interrupts on the client. The reason is that, in general, polling at this level will occupy too much CPU time on the client (the database server) and can only be used in collaboration with the application. Therefore, besides the typical message packaging overhead and wire latency, the VI numbers shown in Figure 6 also include registration/deregistration cost (5-10 microseconds each) and interrupt cost on the client (5-10 microseconds). We see that V3 adds about 15–50 μs overhead on top of VI to handle flow control and block I/O request processing on both the client and server side. These additional overheads vary among the different client implementations. *cDSA* has the least overhead, up to 15% better than the *kDSA*, and up to 30% than *wDSA* because it incurs no kernel overhead in the I/O path. *wDSA* has up to 20% higher latency than *kDSA*. Since there is only one outstanding request in the latency test, optimizations like batching of deregistrations and interrupts are not helpful here.

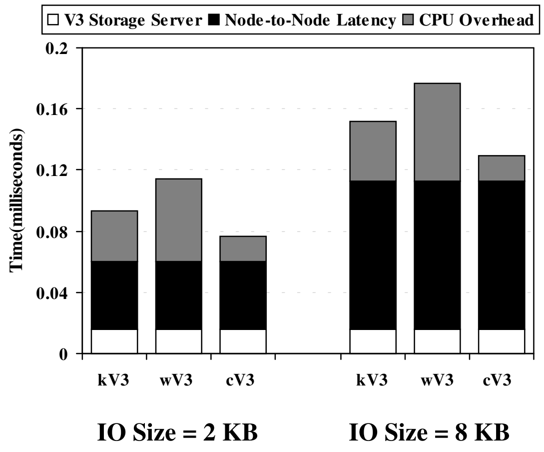


Figure 7: Response time breakdown for a read I/O request.
.

I/O Overhead Breakdown: Differences in simple latency and throughput tests cannot directly be translated to differences in database transaction rates since most commercial databases are designed to issue multiple concurrent I/Os and to tolerate high I/O latencies and low–throughput disks. To better understand the effect of using VI–based interconnects on database performance, we need to examine the source of overhead in the different implementations. Figure 7 provides a breakdown of the DSA overhead as measured on the client-side. This is the round–trip delay (response time) for a single, uncontended read I/O request as measured in the application. We instrument our code to provide a breakdown of the round–trip delay to the following components:

- *CPU overhead* is the overhead spent by the CPU to initiate and complete I/O operations. This overhead is incurred in full on the host CPU and is one of the most important factors in determining overall database system performance. Since database servers tend to issue a large number of I/O operations, high CPU overheads reduce the number of cycles available to handle client requests and degrade overall performance.

- *Node–to–node latency* includes the processing overhead at the NIC, the data transfers from memory over the PCI bus to the NIC at the sender the transfer over the network link, and the transfer from the NIC to memory at the receiver.

- *V3 server overhead* includes all processing related to caching, disk I/O, and communication overhead on the V3 server.

For smaller I/O sizes (2 KBytes), the cost of I/O processing on the storage server is about 20% of the total cost. For larger I/O sizes (8 KBytes), where communication latency becomes more pronounced, storage server overheads as a percentage of the total cost decreases to about 9%. *cDSA* has the lowest CPU overhead among the three, with *wDSA* incurring nearly three times more overhead than *cDSA*. The primary reason is that *cDSA* does not have the requirement of satisfying the standard Windows I/O semantics and hence

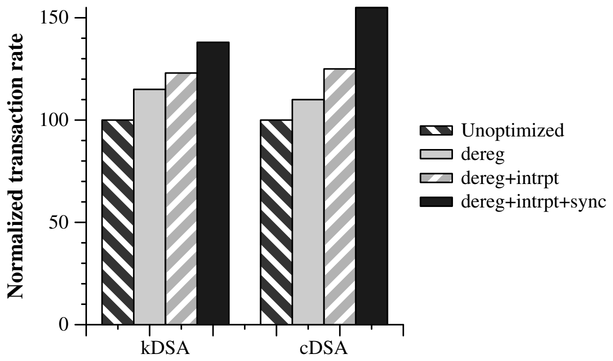


Figure 8: Effect of optimizations on tpmC for the large configuration. *dereg* refers to our batching deregistration technique, *intrpt* to interrupt bathing, and *sync* to synchronization optimizations. All results are normalized to the unoptimized case.

.

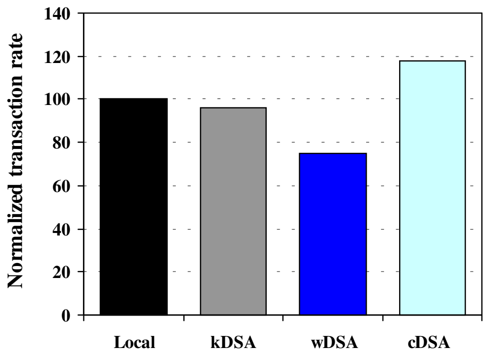


Figure 9: Normalized *TPC–C* transaction rates for the large configuration.

.

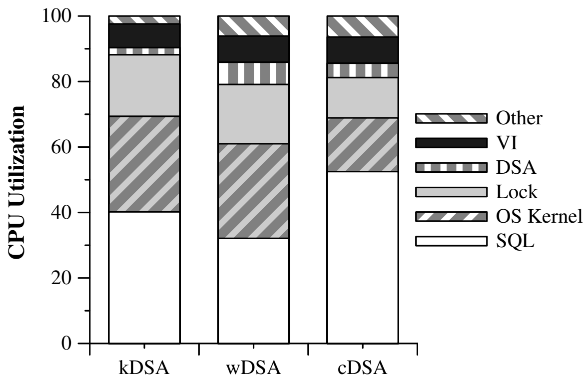


Figure 10: CPU utilization breakdown for *TPC–C* for the large configuration.

is able to minimize the amount of kernel involvement, context switches and interrupts.

6 OLTP RESULTS

To investigate the impact of VI-attached storage on realistic applications we also use *TPC–C*, a well known on-line transaction processing benchmark. *TPC–C* simulates a complete computing environment where a population of users executes transactions against a database. The benchmark is centered around the principal activities of an order–entry environment. *TPC–C* is more complex than previous OLTP benchmarks because of its multiple transaction types, the complex database it uses and its overall execution structure. *TPC–C* involves a mix of concurrent transactions of different types and complexity, either executed on-line or queued for deferred execution. The I/O requests generated by *TPC–C* are random and they have a 70% reads–30% writes distribution. The performance of *TPC–C* is measured in transactions per minute (tpmC). Due to restrictions imposed by the TPC council, we present relative tpmC numbers, normalized to the local case.

6.1 Large Database Configuration

To evaluate the impact of our approach on absolute performance and scalability of databases, we run *TPC–C* on an aggressive, state-of-the-art 32–processor database server. To keep up with CPU capacity we also use large database sizes and numbers of disks as shown in Table 1. The database working set size is around 1 TByte, much larger than the aggregate cache size (less than 56 GBytes) of the database server and the V3 storage server. In our experiments, there are 8 Giganet cLan NICs on the server, each connected to a single V3 storage node. Each V3 storage node has 80 physical disks locally attached, for a total of 640 disks (11.2 TBytes total).

We first consider the impact of various optimizations on the *TPC–C* transaction rates for the large configuration for *kDSA* and *cDSA*. Figure 8 shows that these optimizations result in significant improvements on performance. Batched deregistration increases the transaction rate by about 15% for *kDSA* and 10% for *cDSA*. These benefits are mainly due to the fact that deregistration requires locking pages, which becomes more expensive as the number of processors increase. Batching interrupts improves system performance by about 8% for *kDSA* and 15% for *cDSA*. The improvement for *cDSA* is larger because *SQL Server 2000* mostly uses polling for I/O completions under *cDSA*. Finally, reducing lock synchronization shows an improvement of about 15% for *kDSA* and about 30% for *cDSA*. Reducing lock synchronization has the largest performance impact in *cDSA* because *cDSA* can optimize the full I/O path from the database to the communication layer.

We next consider absolute database performance. Figure 9 shows the normalized *TPC–C* transaction rate for V3 and a fibre–channel storage system; the results for V3 reflect optimizations in the *kDSA* and *cDSA* implementations. The fibre–channel device driver used in the local case is a highly optimized version provided by the disk controller vendor. We find that *kDSA* has competitive performance to the local case. *cDSA* performs 18% better than the local case. *wDSA* performs the worst, with a 22% tpmC lower than *kDSA*. Figure 10 shows the execution time breakdown on the database host for the three client implementations[2]. CPU utiliza-

[2]Note to reviewer: We do not present the exact break-

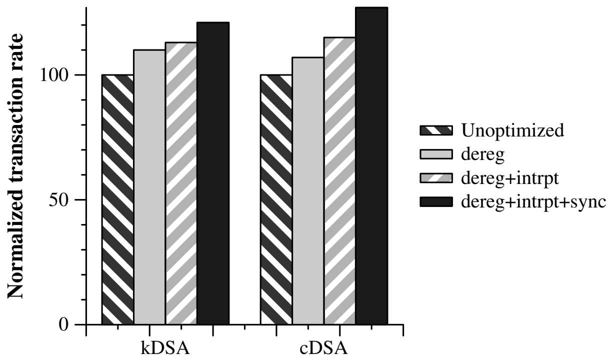


Figure 11: Effect of optimizations on tpmC for the mid-size configuration. *dereg* refers to our batching deregistration technique, *intrpt* to interrupt batching, and *sync* to synchronization optimizations. All results are normalized to the unoptimized case.
.

tion is divided into six categories: (1) *SQL Server 2000*, (2) OS kernel processing, (3) locking overhead, (4) DSA, (5) VI overhead (library and drivers), and (6) other overhead including sockets, etc. The lock time is a component of either *kDSA* and the OS kernel, or *wDSA* and *cDSA* that we single out and present separately, due to its importance. Lock synchronization and kernel times include overheads introduced by *SQL Server 2000*, such as context switching, that are not necessarily related to I/O activity. First, we see that *cDSA* spends the least amount of time in the OS and lock synchronization. This leaves more cycles for transaction processing and leads to higher tpmC rates. We also note that *kDSA* spends the least amount of time in the DSA layer, since most I/O functionality is handled by the OS kernel. *wDSA* spends the highest time in the kernel and the DSA layer due to the complex `kernel32.dll` semantics. Finally, VI overhead remains relative constant across all implementations. Second, we notice that the portion of the CPU time not devoted to transaction processing is high even in *cDSA*. For *kDSA* and *wDSA*, the time spent executing database instructions is below 40%, whereas in *cDSA* this percentage increases to about 50%. This difference is the primary reason for the higher transaction rates in *cDSA*. In this configuration, locking and kernel overheads account for about 30% of the CPU time. About 15% is in the DSA layer (excluding locking) and about 5% is unaccounted. Our results indicate that the largest part of the 30% is due to non-I/O related activity caused by *SQL Server 2000*. We believe further improvements would require restructuring of *SQL Server 2000* and Windows XP code.

6.2 Mid-size Database Configuration

In contrast to the large configuration which aims for absolute performance, the mid-size configuration is representative of systems that target cost–performance.

down for the local case because it was measured by the hardware vendor and we are limited in the information we can divulge by licensing agreements. However, *kDSA* is representative of the local case.

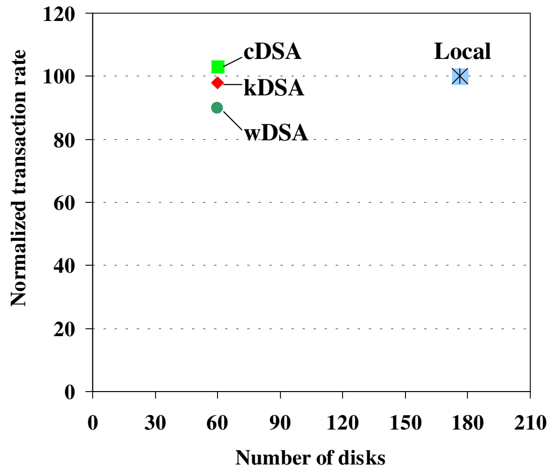


Figure 12: Normalized *TPC–C* transaction rate for the mid-size configuration.

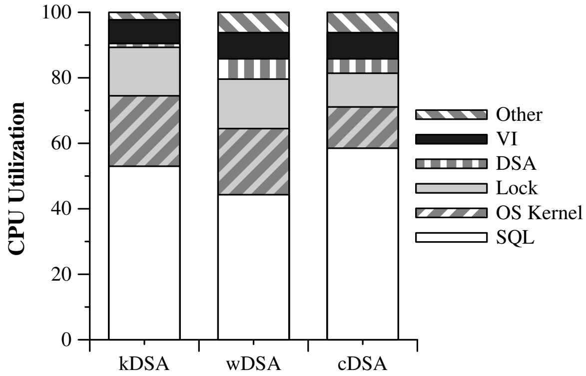


Figure 13: CPU utilization breakdown for *TPC–C* for the mid-size configuration.
.

We first consider the impact of each optimization on tpmC rates. Figure 11 shows that batched deregistration results in a 10% improvement in *kDSA* and 7% in *cDSA*. Batching interrupts increases the transaction rate by an additional 3% in *kDSA* and 8% in *cDSA*. The reason for the small effect of batching interrupts, especially in *kDSA* is that under heavy I/O loads, many replies from the storage server tend to arrive at the same time. These replies can be handled with a single interrupt, resulting in implicit interrupt batching. Finally, as in our previous experiment, lock synchronization shows the highest additional improvement, about 8% in *kDSA* and 12% in *cDSA*.

Next, we look at how V3 performs compared to local SCSI disks. Figure 12 shows the tpmC rates for the local and different V3 configurations. We see that *kDSA*, *cDSA*, and the local case are comparable in the total tpmC achieved. *kDSA* is about 2% worse than the local case and *cDSA* is about 3% better that the local case. Finally, *wDSA* is 10% slower compared to the local case.

Note that the different DSA implementations are competitive with the local case, but uses only one third of the total number of disks (60 as opposed to 176), with the addition of 8GB of memory (6.4GB used for caching) on the V3 server. Although, it is not straight forward to calculate system prices, preliminary calculations of cost components show that the V3 based system has a lower price leading to a better $/tpmC ratio. The reduction in the total number of disks is possible due to the following reasons: VI-based communication reduces host CPU overhead. VI has very low latency and requires no memory copying. This allows more I/O operations to be issued by the database server. Also, VI's low latency magnifies the effect of V3 server caching, which has 40-45% hit ratio for reads in this experiment. With 1,625 warehouses, the database working set size is 100 GBytes, greater than the aggregate cache size on the V3 server. We observe that the same experiment executed on a V3 server configuration with half the cache size (3.2GB) reduces the tpmC rate for *cDSA* by only 7% indicating that while effective use of the cache is important, it is less critical than reducing CPU overhead.

Figure 13 shows the execution time breakdown for the different implementations. The breakdowns are similar to the large configuration. However, we note that the kernel and lock overheads for *kDSA* and *wDSA* are much less pronounced on the mid-size than the large size configuration, and the maximum CPU utilization (in *cDSA*) is about 60%, compared to 50% in the large configuration.

6.3 Summary

We find that VI-attached storage incurs very low overheads, especially for requests smaller than 64 KBytes. In comparing the three different approaches of using VI to connect to database storage, we see that *cDSA* has the best performance because it incurs the least kernel and lock synchronization overhead. The differences among our client implementations become more apparent under large configurations due to increased kernel and synchronization contention. However, our CPU utilization breakdowns indicate that the I/O subsystem overhead is still high even with *cDSA*. Reducing this overhead requires effort at almost every software layer including the database server, the communication layer and operating system.

Finally, contrary to common belief, our results show that high cache hit rates can be achieved in *TPC–C*. Traditional systems usually achieve a close–to–zero read hit ratio for I/O operations issued by the database server. In our large configuration setup, we the read cache hit ratio is consistently as high as 18.7%, with 2.4 GBytes of cache in each V3 server and eight V3 storage nodes (for a total of 19.2 GBytes of storage cache). This high cache hit ratio is in addition to the database caching performed by the database server, which has 32 GBytes of main memory. The mid-size configuration exhibits hit rates between 40-45% for read operations using 6.4GBytes of aggregate cache.

7 RELATED WORK

Our work shares similarity with previous research in the following areas: (i) user–level communication, (ii) using VI for databases, and (iii) storage protocols and storage area networks.

User–level communication: Our work draws heavily on previous research on system area networks and user–level communication systems. Other work has examined issues related to communication processing in parallel applications and the use of commodity interconnects to provide low–latency and high–bandwidth communication at low–cost. Besides VI, specific examples of user–level communication systems include Active Messages [8], BDM [17], Fast Messages [23], PM [26], U-Net [3], and VMMC [11]. Previous work in user–level communication has also addressed the issue of dynamic memory registration and deregistration [4, 7]. These solutions target applications with small working sets for registered memory and require modifications either in the NIC firmware or the operating system kernel.

Using VI for databases: VI–based interconnects have been used previously by database systems for purposes other than improving storage I/O performance. In particular, they have been used to reduce database server CPU cycles devoted to servicing client requests over a LAN. Traditionally, clients in a LAN connect to database servers through IP–based networks. Using VI–based communication between the server and the clients reduces CPU cycles needed to handle client requests by reducing TCP/IP stack processing on the server. Results indicate that a user–space communication mechanism between database clients and servers can improve application transaction rates by up to 15% [5, 15, 22]. VI–based networks have also been used to provide parallel database execution on top of clusters of commodity workstations, e.g. in [5, 22], as opposed to tightly integrated database servers. Recent work has also examined the use of VI in other storage applications, such as clustered web servers [6].

Storage protocols and storage area networks: The Direct Access File System (DAFS) [10] collaborative is an attempt to define a file access and management protocol designed for local file sharing on clustered environments connected using VI–based interconnects. Similar to our work, DAFS provides a custom protocol between clients and servers in a storage system over VI. An important difference between DAFS and the communication protocol used in *DSA* is that DAFS operates at the file system level, while our focus has been on block level I/O.

Traditional storage area interconnects are mostly SCSI and Fiber Channel (FC). Recent efforts in the area have concentrated on optimizing these networks for database applications. For instance, SCSI controllers and drivers are optimized to reduce the number of interrupts on the receive path, and to impose very little overhead on the send path [21]. This requires offloading storage related processing, e.g. the SCSI protocol to customized hardware on the storage controller. However, this approach still requires going through the kernel and incurs relatively high CPU overheads. One can view the use of VI–based interconnects as an alternative approach to providing scalable, cost–effective storage area networks. Recent industry–sponsored efforts try to unify storage area networks with traditional, IP–based LANs. Examples include SCSI over IP (iSCSI), FC over IP (FC/IP), or VI over IP (VI/IP). Since most of these protocols are in the prototyping stage, their performance characteristics have not yet been studied.

Finally, Infiniband [18] is a new interconnection network technology that targets the scalability and performance issues in connecting devices, including storage devices, to host CPUs. It aims at addressing scalability issues by using switched technologies and the performance problems by reducing host CPU overhead and providing Gbit–level bandwidth using ideas from past research in user–level communication systems. It also provides RDMA I/O support. Our work is directly applicable to systems that attempt to use Infiniband as a storage area network.

There has also been a lot of work in the broader area of databases and storage systems. For instance, [1, 19] study the interaction of OLTP workloads and various architectural features, focusing mostly on workloads that fit in main memory. Also, work in disk I/O [2, 13, 14, 25, 27, 28] has explored many directions in the general area of adding intelligence to the storage subsystem and providing enhanced features. Although our work shares broad similarity with these efforts, it differs in significant ways, both in terms of goals and techniques used.

8 CONCLUSIONS

In this work, we study how VI–based interconnects can be used to reduce overheads in the I/O path between a database system and a storage backend. We design a block–level storage architecture (V3) that take advantage of features found in VI–based communication systems. In addition, we provide and evaluate three different client-side implementations of a block-level I/O module (DSA) that sits between an application and VI. These different implementations trade transparency and generality for performance at different degrees. We perform detailed measurements using Microsoft *SQL Server 2000* with both microbenchmarks and real–world database workloads on a mid–size (4–CPU) and a large (32–CPU) database server. We find that on large database configurations *cDSA* provides a 20% transaction rate improvement over a well–tuned fibre–channel implementation. On mid–size configurations, all three DSA implementations are competitive with optimized fibre–channel implementations, but provide substantial potential for better price–performance ratios.

Our experience shows that VI–based interconnects have a number of advantages compared to more traditional storage area and IP–based networks. (i) RDMA operations available in VI help in two significant ways: To avoid copying in I/O write operations and to transfer I/O buffers directly from application memory to the V3 server cache. Also, to use flags in application memory and directly notify it for completion of I/O requests without any application processor intervention. The application still needs to poll these flags, however, their value is updated without the application loosing the CPU. (ii) The low overhead in initiating I/O operations can reduce application CPU utilization to a bear minimum. (iii) Third, VI interconnects can reach their peak throughput at relatively small message sizes (smaller than the size of I/O buffers). This allows for one NIC to service heavy I/O rates, reducing overall system cost and complexity.

However, VI–based interconnects also pose a number of challenges to higher layers, especially in I/O–intensive environments, such as database system. In particular, dealing with flow control, memory registration and deregistration, synchronization, interrupts, and reducing kernel involvement are significant challenges for storage systems. Also, new storage APIs that help minimizing kernel involvement in the I/O path are needed to fully exploit the benefits of user–level communication. *cDSA* exhibits the best performance characteristics among the implementations we study due to the specialized I/O interface it provides. More aggressive strategies than *cDSA* are also possible. Storage protocols can provide new, even database–specific, I/O interfaces, that take full advantage of user–level communication and allow applications to provide hints and directives to the I/O system. Such interfaces are facilitated by the availability of low–cost CPU cycles on the storage side as found in the V3 architecture.

Finally, there is currently a number of efforts to incorporate VI–like features to other interconnects as well. Most notably, Infiniband and iSCSI, target to address many of the issues related to storage area networks and include (or there is discussion about including) features such as RDMA capabilities. Our work is directly applicable to systems with Infiniband interconnects and relevant for systems with future iSCSI interconnects. Our results show that the CPU I/O overhead in the database system is still high (about 40–50% of CPU cycles) even in our best implementation. Reducing this overhead requires efforts, not only at the interconnect level, but at almost every software layer including the database server, the communication layer, and the operating system, and especially on the interfaces among these components.

REFERENCES

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, editors, *Proceedings of the Twenty-fifth International Conference on Very Large Databases*, 1999.

[2] D. C. Anderson, J. S. Chase, S. Gadde, A. J. Gallatin, K. G. Yocum, and M. J. Feeley. Cheating the I/O bottleneck: Network storage with Trapeze/Myrinet. In *Proceedings of the USENIX 1998 Annual Technical Conference*, 1998.

[3] A. Basu, V. Buch, W. Vogels, and T. von Eicken. U-net: A user-level network interface for parallel and distributed computing. *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP), Copper Mountain, Colorado*, December 1995.

[4] A. Basu, M. Welsh, and T. von Eicken. Incorporating memory management into user-level network interfaces. http://www2.cs.cornell.edu/U-Net/papers/unetmm.pdf, 1996.

[5] B. C. Bialek. Leading vendors validate power of clustering architecture, detail of the tpc-c audited benchmark. http://wwwip.emulex.com/ip/pdfs/performance/IBM_TPC-C_Benchmark.pdf, Jul. 2000.

[6] E. V. Carrera, S. Rao, L. Iftode, and R. Bianchini. User-level communication in cluster-based servers. In *Proceedings of the 8th IEEE International Symposium on High-Performance Computer Architecture (HPCA 8)*, 2002.

[7] Y. Chen, A. Bilas, S. N. Damianakis, C. Dubnicki, and K. Li. UTLB: A mechanism for address translation on network interfaces. In *Proceedings of the Eighth International Conference Architectural Support for Programming Languages and Operating Systems ASPLOS*, pages 193–203, San Jose, CA, Oct. 1998.

[8] B. N. Chun, A. M. Mainwaring, and D. E. Culler. Virtual network transport protocols for myrinet. In *Hot Interconnects Symposium V*, Stanford, CA, August 1997.

[9] Compaq/Intel/Microsoft. *Virtual Interface Architecture Specification, Version 1.0*, Dec. 1997.

[10] DAFS Collaborative. *DAFS: Direct Access File System Protocol Version: 1.00*, Sept. 2001.

[11] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: efficient support for reliable, connection-oriented communication. In *Proceedings of Hot Interconnects*, Aug. 1997.

[12] D. Dunning and G. Regnier. The Virtual Interface Architecture. In *Proceedings of Hot Interconnects V Symposium*, Stanford, Aug. 1997.

[13] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[14] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka. File server scaling with network-attached secure disks. In *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 1997.

[15] Giganet. Giganet whitepaper: Accelerating and scaling data networks microsoft sql server 2000 and giganet clan. http://wwwip.emulex.com/ip/pdfs/performance/-sql2000andclan.pdf, Sept. 2000.

[16] Giganet. Giganet cLAN family of products. http://www.emulex.com/products.html, 2001.

[17] H. Gregory, J. Thomas, P. McMahon, A. Skjellum, and N. Doss. Design of the BDM family of myrinet control programs, 1998.

[18] InfiniBand Trade Association. Infiniband architecture specification, version 1.0. http://www.infinibandta.org, Oct. 2000.

[19] K. Keeton, D. Patterson, Y. He, R. Raphael, and W. Baker. Performance characterization of a Quad Pentium Pro SMP using OLTP Workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-98)*, 1998.

[20] Microsoft. Address windowing extensions and microsoft windows 2000 datacenter server. Windows Hardware Engineering Conference: Advancing the Platform. Also available at: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnw2k/html/awewindata.asp, March 30 1999.

[21] Mylex. eXtremeRAID 3000 High Performance 1Gb Fibre RAID Controller. http://www.mylex.com.

[22] ORACLE. Oracle net vi protocol support, a technical white paper. http://www.vidf.org/Documents/whitepapers/Oracle_VI.pdf, February 2001.

[23] S. Pakin, V. Karamcheti, and A. A. Chien. Fast Messages: Efficient, portable communication for workstation clusters and massively parallel processors (MPP). *IEEE Concurrency*, 5(2):60–73, April-June 1997. University of Illinois.

[24] Reference omitted for blind review process. *This entry will expand to multiple references in the final version of the paper.*

[25] E. Riedel and G. Gibson. Active disks - remote execution for network-attached storage, 1997.

[26] H. Tezuka, A. Hori, and Y. Ishikawa. PM: A high-performance communication library for multi-user parallel environments. Technical Report TR-96015, Real World Computing Partnership, Nov. 1996.

[27] M. Uysal, A. Acharya, and J. Saltz. Evaluation of active disks for decision support databases. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, pages 337–348, Toulouse, France, Jan. 8–12, 2000. IEEE Computer Society TCCA.

[28] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, 14(1):108–136, Feb. 1996.