VI-Attached Database Storage *

Yuanyuan Zhou[1], Angelos Bilas[2], Suresh Jagannathan[3], Dimitrios Xynidis[2],
Cezary Dubnicki[4], and Kai Li[5]

[1]Department of Computer Science, University of Illinois, Urbana-Champaign,Urbana, IL–61822
[2]Department of Computer Science, University of Crete, Greece
[3]Department of Computer Science, Purdue University, West Lafayette, IN–47907
[4]NEC Laboratories, 4 Independence Way,Princeton, NJ–08540
[5]Department of Computer Science, Princeton University, Princeton, NJ–08540

Abstract

This article presents a VI–attached database storage architecture to improve database transaction rates. More specifically, we examine how VI–based interconnects can be used to improve I/O path performance between a database server and a storage subsystem. To facilitate the interaction between client applications and a VI-aware storage system, we design and implement a software layer called DSA, that is layered between applications 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 have also investigated many design tradeoffs in the storage cluster.

We present detailed measurements using a commercial database management system with both micro-benchmarks and industrial database workloads on a mid–size, 4 CPU, and a large, 32 CPU, database server. We also compare the effectiveness of VI-attached storage with an iSCSI configuration, and conclude that storage protocols implemented using DSA over VI have significant performance advantages. More generally, 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 back-end. 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.

*An earlier version of this paper [40] appears in the 29th ACM International Symposium on Computer Architecture (ISCA2002), focusing on the effects of VI network on the database side. This paper expands on those results, presenting details about the storage architecture, and furnishing additional experiments.

1 Introduction

The rapid decline in the cost of storage per Gbyte, and the growing demand for storage (projected to double every year [12]) has led to increased interest in storage system design. Recent interest in this area has targeted the following areas: (i) performance, (ii) scalability, (iii) reliability, (iv) availability, and (v) manageability. Database systems are particularly sensitive to storage system performance and scalability issues. The performance of a database system is typically measured by how many transactions it can process per second. Because database applications impose serious I/O requirements, achieving high transaction rates can only be realized by reducing I/O overhead. For realistic database loads, however, I/O can account for between 20% to 40% of total execution time [2]. Therefore, to maximize CPU utilization and improve overall transaction rates, database systems need to ameiloriate I/O overhead.

While database management systems perform a number of optimizations to minimize unnecessary I/O, storage architectures also strive to reduce I/O–related costs on the I/O path between the database management system and the disk. Various techniques are used for this purpose: (i) optimizing data layout on disks to reduce disk seek times (at the expense of lower disk utilization); (ii) using caching on the storage side to reduce average disk I/O latency; and, (iii) optimizing the data path from the server to storage both to incur lower CPU overheads in initiating I/O operations as well as to eliminate data copies.

These techniques address I/O overheads generated by the database application and costs incurred on the storage system. However, they do not adequately address operating system overheads for preparing I/O on the database server and network overheads introduced in communicating with a storage server in a SAN environment. Both these overheads negatively impact database server performance. Furthermore, scaling database storage systems to larger configurations using conventional architectures is extremely costly due to the high cost of storage interconnects and storage controllers. Thus, for most real database applications, high I/O-related overheads remain a major limiting factor for scalability and performance improvements.

User–level communication architectures have been the subject of recent interest because of their potential to reduce communication related overheads. Because they allow applications 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, reducing host overhead. These features have been used with success in improving the performance and scalability of parallel scientific applications [5, 35, 16, 32, 11, 23]. The Virtual Interface (VI) architecture [13] is a well–known, industry–wide standard for system area networks based on these principles that has spawned a number of initiatives, such as the Direct Access File Systems (DAFS) [14], 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 [34]. Achieving high transaction rates, however, can only be realized by reducing I/O overheads on the host CPU and providing more CPU cycles for transaction processing.

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, for the same purpose. We propose an alternative storage architecture called VI–attached Volume Vault (V3) that consists of a storage cluster and one or more database servers, all using the same VI-based network. 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. tested in customer sites.

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 (Direct Storage Access), 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. 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 modules, legacy APIs and the current VI specification are not well–suited for this purpose. DSA makes extensive use of RDMA capabilities and low overhead operations for initiating I/O requests. More importantly, DSA deals with issues not addressed by VI.

Recently, there has been significant recent interest in examining alternative solutions that would both be able to reduce the cost of the underlying equipment and management as well as better follow the technology curves of commodity networking components. One such approach is using IP-type networks and tunneling storage protocols on top of IP to leverage the installed base of equipment as well as the increased bandwidth available, especially of local area networks. It is currently projected that 10 Gigabit Ethernet interconnects will soon reach commodity status, providing at least as much bandwidth as available)in current storage area networks. iSCSI [24] is an IP-based storage protocol based on SCSI [4].

It provides a transport layer for SCSI commands over TCP/IP. Thus, it functions as a bridge between the popular SCSI storage protocol and the popular TCP/IP local area network family of protocols. Although this approach has obvious intuitive appeal, its benefit on system performance remains unclear. The introduction of both a new type of interconnect as well as a number of protocol layers in the I/O protocol stack may introduce significant overheads.

In our work, we evaluate three different implementations of DSA, one kernel–level implementation and two user-level ones that trade transparency and generality for performance at different degrees. We also contrast DSA with a commodity iSCSI storage system. We present detailed DSA measurements using a commercial database management system, Microsoft *SQL Server 2000*, with both micro-benchmarks and industrial database workloads (*TPC–C*) on a mid–size (4 CPU) and a large (32 CPU) database server with up to 12 TB of disk storage. Our results show that:

1. Effective use of VI in I/O intensive environments requires substantial enhancements in flow control, reconnection, interrupt handling, memory registration, and lock synchronization.

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

3. By employing commodity storage nodes and PCs to implement the storage back–end, the storage system can leverage low memory prices to provide very large caches at low costs. Moreover, the clustered nature of the V3 architecture and the use of VI–based interconnects allow V3 to easily scale in a cost–effective manner and provide attractive disk utilization characteristics.

4. VI–based interconnects and user–level communication can improve all aspects of the I/O path between the database system and the storage back-end and result in transaction rate improvements of up to 18% for large database configurations.

5. Commodity iSCSI systems, in contrast to DSA, introduce significant overheads compared to directly attached storage subsystems. Thus, although they have significant potential, existing incarnations do not offer a viable replacement for optimized storage subsystems.

The rest of the article is organized as follows. Section 2 presents the V3 architecture and the various DSA implementations. Section 4 presents our performance optimizations for DSA. Section 5 presents our experimental platforms and Sections 6 , 7, and 8 discuss our results. Finally, we present related work in Section 9 and draw our conclusions in Section 10.

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 back-end to database systems through a VI interconnect. This section provides a brief overview of the VI-attached Volume Vault (V3) architecture.

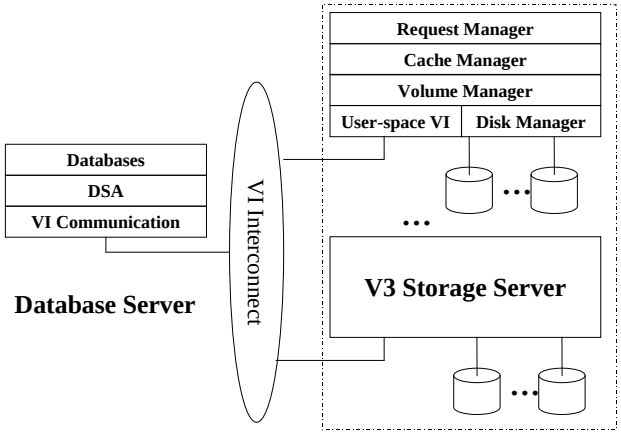


Figure 1: V3 Architecture overview.

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 volume). To support industrial workloads and to avoid single points of failure, each V3 server node has redundant components (power supplies, system disks, network interfaces, etc.). Each V3 volume consists of one or more physical disks attached to V3 storage nodes. V3 volumes can span multiple V3 nodes, using combinations of RAID, concatenation and other disk organizations. With current disk technologies, a single V3 volume can provide more than 2 TB of storage. Since existing VI networks support a large number of nodes (up to 128), a multi–node V3 back-end can provide more than 250 TB of storage. V3 uses large main memories as disk buffer caches to help reduce disk latencies [41].

2.1 V3 Server Overview

The internal architecture of a V3 node is structured differently than a traditional disk subsystem. Large memory serves as a tertiary disk cache, and sophisticated cache management algorithms are employed to ensure the cache's effective use in reducing latency induced by disk interaction. Application requests are handled using an optimized multithreaded pipeline organization that further reduces I/O latency by allowing different disk requests to be serviced concurrently.

A V3 node acts as an "intelligent" storage node because general-purpose processors are used to manage a collection of virtual disks. In our design, compute cycles on a V3 node can be used to implement caching

algorithms, reorganize disk layout, and provide various forms of software RAID and RAID across multiple nodes. This capability can be provided without the need to steal processor cycles from applications. Providing a virtual disk abstraction in conjunction with significant caching makes it possible for multiple disk spindles to work by processing requests from different internal V3 processors concurrently. V3 client drivers can take advantage of this increased intelligence by supporting batch writes and streamlined protocols to further bolster throughput and reduce CPU usage on the client. Finally, because V3 units are network-aware, they support a high-degree of scalable storage since new V3 nodes can be freely added to provide additional storage as warranted.

V3's performance advantage derives from two key features of the system architecture: (1) the use of VI as network interface to provide reduced latency and improved throughput, and (2) the use of a multithreaded architecture that exploits pipelining, maximizes use of memory, and minimizes unnecessary disk access. In the next subsection, we briefly describe V3 cache management.

2.2 Cache Management

An efficient replacement algorithm is critical for cache performance. However, determining such an algorithm depends upon how cache data is accessed. Access patterns are influenced by where the cache resides in the storage hierarchy. For example, accesses to second-level server buffer caches (of the kind managed by a V3 node) are misses from a first-level buffer cache; a first-level cache is the cache maintained by a client application. First-level caches typically employ a least recently used (LRU) replacement algorithm so that recently accessed blocks will be available in the cache if the data they contain are frequently accessed by the application. Although an LRU strategy favors access patterns with strong temporal locality, such a strategy may be inappropriate for second-level buffer caches since these caches are accessed only when temporal locality techniques fail at first-level.

To elaborate further, let's consider access frequency to the second-level buffer cache and how it is correlated to misses from the first-level cache. Hot and cold blocks are not likely to be referenced frequently in the second-level cache: hot blocks remain resident on the primary cache, and cold blocks are accessed infrequently by definition. Indeed, access frequency distribution in second-level caches is uneven, with most accesses being made to a small set of blocks. A good replacement algorithm will selectively keep frequently accessed *warm* blocks for a long period of time. Warm blocks are blocks that are likely to be flushed from the first-level cache (because of the LRU algorithm), but that are likely to be frequently referenced from the second-level cache. Warm blocks should stay resident in second-level caches to reduce access to slow disks.

The cache replacement algorithm used in V3 was developed based on extensive studies of access patterns over a range of real-world benchmarks [41]. We have designed a second-level buffer cache replacement algorithm called Multi-Queue (MQ) based on our access pattern study. It maintains blocks with different

access frequencies in different queues. Queues are ordered with respect to the expected lifetime of the blocks they hold. Blocks in Q_i are expected to have a longer lifetime in the cache than blocks in Q_j ($j > i$). MQ also uses a history buffer to remember access frequencies of recently evicted blocks for some period of time. Blocks are demoted from higher to lower level queues wheen the life time of this block in its current queue expires.

3 DSA Implementations

Direct Storage Access (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. It interacts with V3 storage nodes using a storage protocol that in addition to block-level read and write datapath operations, also provide various control-path and administrative operations which are not critical to system performance.

The new features provided by DSA include flow control, retransmission, and reconnection that 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 4.

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 (database) 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 allow storage applications to take advantage of user–level communication. Figure 2 shows the I/O path for each DSA implementation. The I/O path includes three basic steps: (i) register memory, (ii) post read or write request, and (iii) transfer data. Next, we briefly discuss each implementation.

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 [13] provided by Giganet [21]. The API exported by *kDSA* is the standard

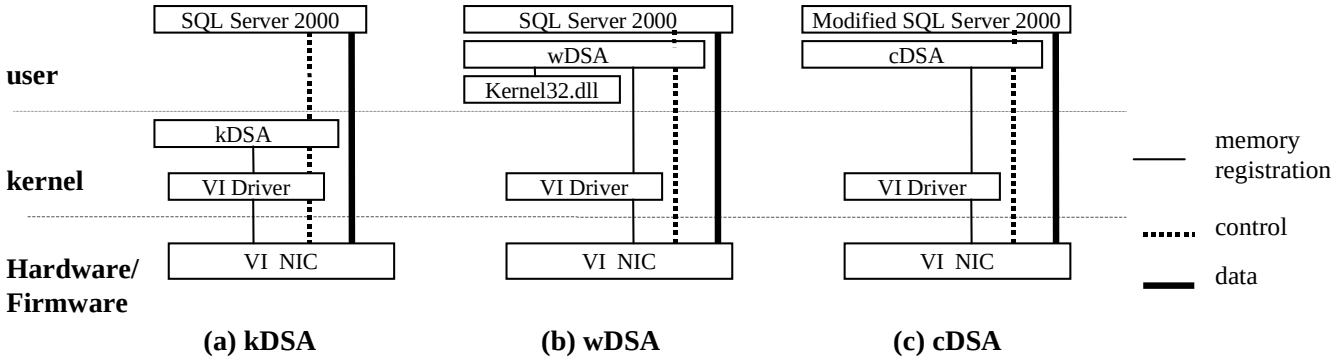


Figure 2: The I/O path in each of our three DSA implementations.

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 can benefit from techniques different from the ones used in 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 [13] 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 application I/O semantics, it must trigger an application–specific event or schedule an application–specific callback function to notify the application thread for completions of I/O requests. 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 a user–level implementation of DSA that provides a new I/O API to applications to exploit the benefits of VI–based communication. The new API consists primarily 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 I/O semantics, and, in some cases, modification of the application to adhere to this new API.

The main feature of *cDSA* relevant to this work is an 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*. 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.

4 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. Because *wDSA* must precisely implement the semantics defined by the Win32 API, opportunities for optimizations are severely limited and not all optimizations are possible. For this reason, we focus mostly on optimizations for *kDSA* and *cDSA*.

4.1 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 a transfer finishes. Current VI–enabled NICs have a limitation on how much memory they can register. For instance, the Giganet

cLan card [21] we use allows 1 GB of outstanding registered buffers and takes about 10μs 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 [10, 6] 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 translation 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 the Address Windowing Extensions (AWE) [28] to allocate the database server cache on physical memory. AWE is a feature of Windows for systems that have large amounts of physical memory. The AWE extensions provide a simple API that applications can use to allocate physical memory and map it to their virtual address space. Applications can then access large amounts of physical memory by manually mapping the regions of interest to their virtual address space with low-overhead calls. Application memory allocated as AWE memory 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 (4 MB 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.

4.2 Interrupts

Interrupts are used to notify the database host when an asynchronous I/O request completes. To improve performance, database systems issue large numbers of asynchronous I/O requests. In V3, when the

database server receives a response from a V3 storage node, the DSA layer needs to notify the database application with a completion event. 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 5–10μs 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. In this work, we use interrupt batching to reduce this cost as described next.

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, and 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. Our method works extremely well in benchmarks where there is a large number of outstanding I/O operations, as is the case with most large–scale database workloads.

cDSA takes advantage of features in its API to reduce the number of interrupts required to complete I/O requests. Upon I/O completion, the storage server sets via RDMA a completion flag associated with each outstanding I/O operation. The database server polls these flags for a fixed interval. If the flag is not set within the polling interval, *cDSA* switches to waiting for an interrupt upon I/O completion and signals the database appropriately. Under heavy database workloads this scheme almost eliminate the number of interrupts for I/O completions.

4.3 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 synchronization 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 both the send and receive paths and VI uses two more, one for registration/deregistration and one for queuing/dequeuing. Thus, there is a total of about 8–10 synchronization pairs 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 synchronization pairs that are not in our control are the four pairs in the VI layer. In *cDSA* we also lay out data structures carefully to minimize processor 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

Component	Mid–size	Large
CPU	4 x 700 MHz PII	32 x 800 MHz PII
L1 Cache per CPU	8KB Per CPU	8KB per CPU
L2 /L3 Unified	1MB/ No	2MB / 32MB
Memory	4 GB	32 GB
PCI slots	2 x (66MHz, 64bit)	96 x (66MHz, 64bit)
NICs	4 cLan cards	8 cLan cards
# Local Disks	176	640
Windows Ver.	2000 AS	XP
Database Size (warehouses)	1,625 (1 TB)	10,000 (10 TB)

Table 1: Database host configuration summary for the mid–size and large database setups.

Component	Mid–size	Large
# V3 Nodes	4	8
CPU	2 x 700 MHz PII	2 x 700 MHz PII
l1 / L2	8KB / 1MB	8KB / 1MB
Memory/V3 Cache per Node	2GB/1.6GB	3GB/2.4GB
Disk Type	SCSI, 18GB, 10K RPM	FC, 18GB, 15K RPM
Disk Controller	UltraSCSI 320 (RAID0)	Mylex eXtreme RAID 3000 [29] (RAID5)
T# Disks/disk space	60/1TB	640/11.5TB
Windows Version	2000 Workstation	2000 Workstation

Table 2: V3 server configuration summary for the mid–size and large database setups.

not significant. The main reason is that VI synchronization pairs are private to a single VI connection. Since DSA uses multiple VI connections to connect to V3 storage nodes, in realistic experiments this minimizes the induced contention.

5 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. Tables 1 and 2 summarize the hardware and software configurations for the two 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 nodes, each with four processors and 32 MB of third–level cache (total of 256 MB). The server has a total of 32 GB 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 GB 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. For the mid-size configuration, disks are organized using software RAID0, whereas the large configuration uses hardware RAID5. Both the mid-size and the large configurations use a Giganet network [21] 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 [17] provided by Giganet [21]. 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. In our setups, the maximum end-to-end user-level bandwidth of Giganet is about 110 MB/s and the one-way latency for a 64–bytes message is about 7 μs.

6 Micro–benchmark Results

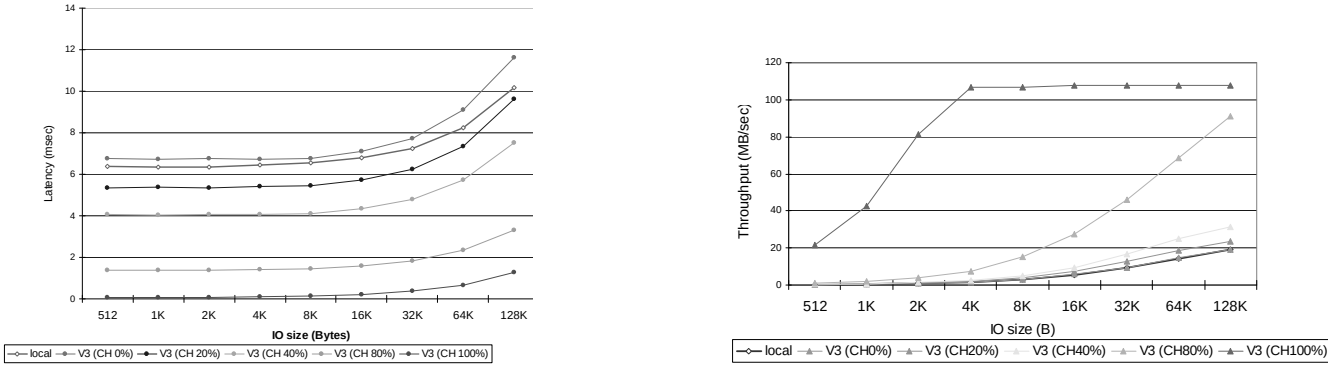
We use various workloads to investigate the base performance of V3 and the different DSA implementations. We examine the overhead introduced by DSA compared to raw VI. We then examine the performance V3–based I/O when data is cached. We also compare the performance of V3 against a configuration with local disks. Finally, to better quantify

In our experiments, 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. All configurations use the same server implementation.

We present five types of statistics for various I/O request sizes: response times, throughput, execution time breakdowns,effects of communication parameters, and effects of multiprocessor vs. uniprocessor machines. 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 8K bytes.

6.1 Server Performance

Figure 3(a) shows latency results that reflect different cache hit rates. These results are compared against a configuration that only uses local disks. The tests perform a sequence of random reads on the disk. All configurations use Ultra 160 SCSI disks. The cache size on the V3 node was 64MB, and the cache block size was 8K bytes. With caching disabled, the overhead of using a V3 system is exposed; this overhead includes the cost of VI communication and processing costs on the V3 node. Note, however, that latency overhead reduces significantly as cache hits on the V3 node increase. With a cache-hit rate of 40%, latency is significantly reduced by over 30% over the local case on 8K message blocks, the typical block size used by SQL server. As cache utilization increases, improvements become even more pronounced.



(a) Latency

(b) Throughput

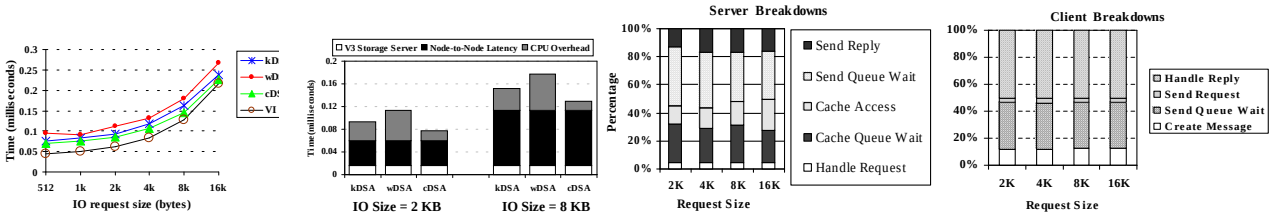
Figure 3: Bssic VI and V3 read latencies and server read latency and throughput as a function of different cache hit rates (kDSA).

The next figure (Figure 3(b) shows throughput effects introduced by the V3 cache. This experiment measures throughput of random reads of various sizes on a RAID 0 configured system using five Ulta 160 SCSI disks. When cache utilization is low, overall throughput from a V3 system is roughly the same as the throughput achievable using local disks. When cache utilization is 100%, throughput quickly saturates the maximum bandwidth achievable by Giganet's cLAN VI NIC, roughly 110 MB/sec. When the cache hit rate is at 80% throughput scales with message size, for all practical sizes. Across different block sizes, the throughput realized using a V3-based storage system is at worst competitive with local disk throughput, and in most cases significantly superior.

6.2 DSA Overhead

We first examine the overhead introduced by DSA compared to the raw VI layer (Figure 4(a)). The raw VI latency 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 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 3(a) also include registration/deregistration cost (5-10 microseconds each) and interrupt cost on the client (5-10 microseconds).

The V3 latency tests are measured by reading a data block from the storage server using each of the three DSA implementations. We see that V3 adds about 15–50 μs overhead on top of VI. This additional overhead varies among the different client implementations. *cDSA* has the least overhead, up to 15%



(a) V3 and VI latency (b) End-to-end breakdown (c) Client and server breakdown

Figure 4: Read I/O request time breakdown (kDSA).

.

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.

To better understand the effect of using VI–based interconnects on database performance, we next examine the source of overhead in the different implementations. Figure 4(b) 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: (1)*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. (2)*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. (3) *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 KB), the cost of I/O processing on the storage server is about 20% of the total cost. For larger I/O sizes, e.g. 8 KB, 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 is able to minimize the amount of kernel involvement, context switches and interrupts.

Figure 4(c) shows more detailed breakdowns on both the server-side and client-side for a random read request of various sizes with *kDSA*. On the server side, more than 50% of the server overhead is spent waiting inside either the cache queue or the send queue. The queue waiting is caused by the pipeline server architecture. Cache management and communication are handled by different threads and these

threads communicate with each other using task queues. Therefore, once the communication thread receives a request, the request is inserted into the cache queue. When the cache thread is scheduled, it removes the request from the queue and looks for corresponding disk blocks in the buffer cache. Once the data is located or loaded from disks, the request is inserted into the send queue and then completed by the send thread. Therefore, with only one outstanding request in the system, the request needs to wait in the queue for the system to switch from the cache thread to the communication thread. With more outstanding requests, the context switch time is amortized and does not affect the overall system throughput. The situation can be improved with multiprocessor machines as V3 servers, the performance impact of which is discussed in Section 6.5.

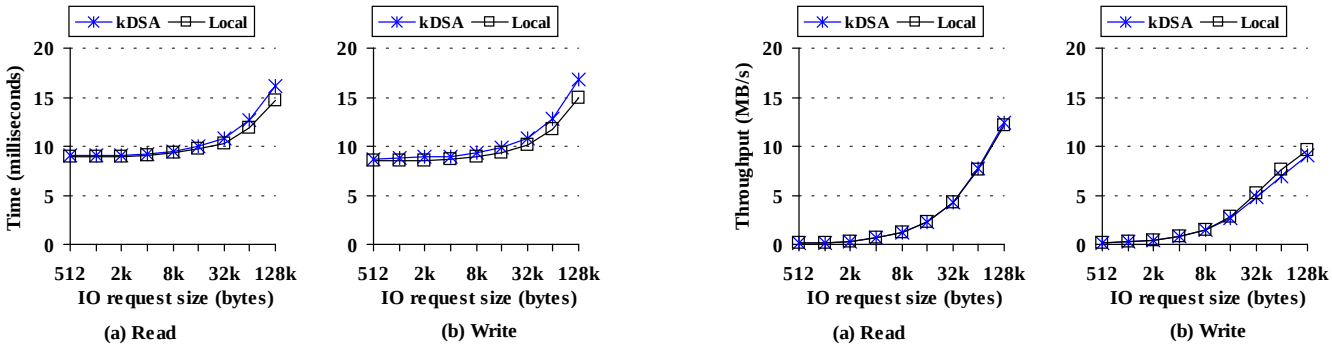
On the client side, around 50% of the client overhead is spent in handling server reply. Since the breakdowns are measured using *kDSA*, the disk driver needs to decode the reply, find the corresponding I/O block, change the request status, set user specified return values, and, most importantly, trigger events or schedule callback functions to wake up the user-level thread. Comparing to *kDSA*, *wDSA* has to spend more time in handling server reply because it needs to make a system call to trigger the corresponding event or callback function to nofity the application thread about the I/O completion. *cDSA* spends the least time in handling server reply because it only needs to set the corresponding memory flag. Similar to the server side, the client also spends significant time waiting in a queue, send queue. The main reason is that the request is created by the disk driver but is sent out by the communication VI driver.

6.3 V3 vs. Local Case

Next, we compare the performance of a V3–based I/O subsystem to a local configuration in which disk I/O does not involve network communication. To make the comparison meaningful, these experiments set the V3 server cache size to zero and all V3 I/O requests are serviced from disks in the V3 server. In each workload all I/O operations are random.

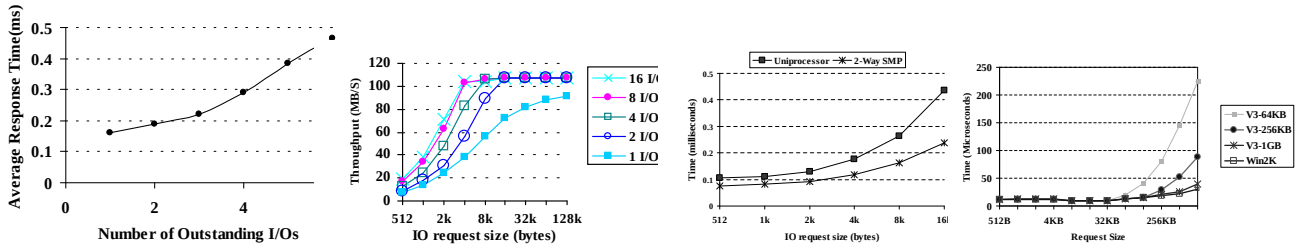
Figure 5(a) shows that the V3 setup has similar random read response time as the local case when the read size is less than 64 KB. The extra overhead introduced by V3 is less than 3%. For larger I/O request sizes, however, V3 introduces higher overheads, proportional to the request size. For example, V3 has around 10% overhead for 128 KB reads because of increased data transfer time. In addition, the packet size in the cLan VI implementation is $64K - 64$ bytes. Therefore, to transfer 128 KB requires breaking the data to three VI RDMAs. Write response time behaves similarly, For request sizes up to 32 KB, V3 has response times similar to the local case. For larger request sizes, V3 is up to 10% slower due to network transfer time.

Figure 5(b) presents our results for 100% read and write workloads with two outstanding I/O operations. As mentioned above, V3 adds 3-10% overhead compared to the local case. However, when the



(a) Response time (b) Throughput

Figure 5: V3 and local read and write response time (one outstanding request) and throughput (two outstanding requests).



(a) Cached-block respon time (b) Cached-block throughpu (c) Effects of SMP node (d) Effects on RAID5

Figure 6: Effects of caching, multiprocessor architecture, and implicaitons on RAID5.

number of outstanding I/O requests increases, the throughput difference between a V3 volume and a local disk decreases due to pipelining. V3 can achieve the same *read* throughput as a local disk with two outstanding requests and the same *write* throughput with eight outstanding requests. Since databases always generate more than one outstanding requests 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.

6.4 V3 Cached–block Performance

Next, we examine the overhead of V3–based I/O with caching turned on. Since the amount of cache resident on disk controllers in the local case is negligible, we only present numbers for the V3 setup.

Figure 6.3(a) shows the average response time for reading an 8 KB block from the V3 cache. When the number of outstanding requests is less than four, the average response time increases slowly. With more than four outstanding I/O requests, the VI interconnect is saturated (see Figure 6.3(b)) with 8 KB requests. Above this threshold, the average response time increases linearly and is a function of network queuing.

Figure 6.3(b) shows the V3 *read* throughput with a 100% cache hit ratio for different numbers of outstanding requests. With one outstanding read request, the throughput reaches a maximum of about

90 MB/s with 128 KB requests. However, with more outstanding requests, the peak VI throughput (about 110 MB/s) can be reached at small request sizes. With four outstanding read requests, the VI interconnect is saturated even with 8 KB requests. Lower cache hit ratios require higher numbers of outstanding I/Os to saturate the VI interconnect.

6.5 Effect of Implementing a V3 Node on a Multiprocessor

We have evaluated the effects of multiprocessor configurations on V3 server performance. Since the V3 server does not perform any computation-intensive operations, CPU utilization is likely to be low unless it is configured with many disk spindles. Using an SMP as a V3 server can improve overall system throughput by allowing multiple I/O requests to be serviced in parallel. This becomes especially relevant when I/O workloads become significant.

To see how multiprocessor systems can help improving the average response time, Figure 6.3(c) compares the response time between a uniprocessor machine and a 2-way SMP machine as the V3 server. The numbers are measured running the random read microbenchmark with a 100% storage cache hit ratio. At any time, only one I/O request is outstanding. The client configuration is the same for both experiments. As shown on the figure, the 2-way SMP V3 server consistently outperforms the uniprocessor machine for all request sizes. A multiprocessor machine provides better response time because it allows two or more server threads to run simultaneously. Since the V3 server employs a pipeline architecture, having two threads (for example, the receive thread and the cache thread) running simutaneously can reduce context switch time. Moreover, it also shortens the time for a request waiting in VI's completion queue to be picked up by the communication thread. As a result, client requests are serviced faster in the multiprocessor server configuration than in the uniprocessor one.

6.6 Effect of Communication Parameters on RAID5 Performance

V3 uses OS-based, software implementations of RAID organizations [33]. Due to resource limitations of the communication layer, V3 has to impose maximum request sizes. I/O requests larger than the maximum size are broken into multiple subrequests, each of which is smaller than the maximum request size allowed[1]. These subrequests are sent to the V3 server separately. To maximize parallelism, the V3 server does not group all the related subrequests into one. Instead, the V3 server treats each subrequest as an independent request. This design can improve the average response time if the volume is organized using either sequential organizations or simple striping (RAID0) organization. In general, the smaller the maximum request size, the higher degree the parallelism and thereby the better the response time.

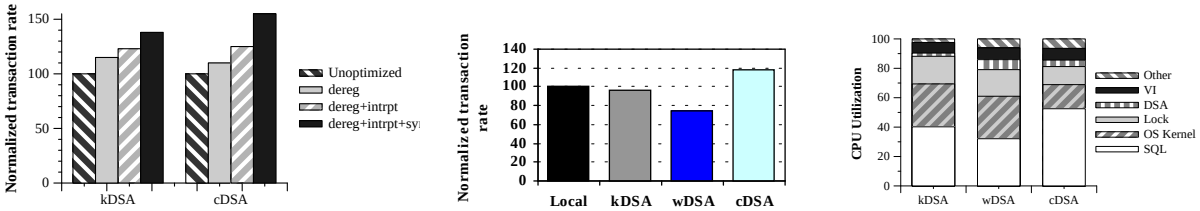
[1]Due to the underlying VI implementation, I/O requests with sizes equal to the maximum request size also need to be broken into smaller requests. The reason is that the VI communication layer needs several bytes in the packet to send VI specific control information

However, on RAID5 organizations, the impact of the maximum request size is quite different. Figure 6.3(d) shows the response time for a random write request in a V3 virtualized volume using a RAID5 organization over 10 physical disks with a striping block size of 64-Kbytes. We compare V3 with a local RAID5 organization implemented by the Windows 2000 volume manager. We vary the maximum request size from 64 KBytes to 1 Gigbytes to examine the impacts on RAID5 performance. Since V3 employs a write-through policy, writes are not acknowledged until data is safely written to disk. In RAID5, each block write consists of three steps: (i) read the other blocks in the same striping group into memory; (ii) calculate the parity block; (iii) write the parity block and the updated block into disks. So if the write request contains all the blocks in the same striping group, step (i) can be skipped. In other words, large I/O writes outperforms multiple of small I/O writes.

For small I/O requests, V3 performs similarly to the local RAID5 organization, since both the local case and V3 need to perform all three steps to complete a write. However, for large I/O requests, V3 performs worse than the local case, especially if the maximum request size is small compared to the striping size (the size of a striping group). This is because the local case can skip step (i) in the write operation, whereas V3 needs to break large I/O requests into small ones (as a result of the underlying VI implementation), and thereby may not be able to skip this step. For example, if the maximum request size is 64 KBytes (V3-64KB), the response time for a write request of size much greater than 64 KBytes in V3 can be upto 4X slower compared to a local RAID5 configuration. For the V3-256KB configuration, response times can be upto 2X slower for request sizes greater than 256KBytes. When the maximum request size is 1 Gigabytes (V3-1GB), the V3 performs similarly to the local case. In OLTP and DSS workloads, most write requests are smaller than 128 KBytes. Therefore, either 256 KBytes or 1GB would be appropriate values for the maximum I/O request size.

7 OLTP Results

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 response times and low–throughput disks. To investigate the impact of VI-attached storage on realistic applications we use *TPC–C*, a well known on-line transaction processing (OLTP) benchmark [36]. *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* 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



(a) Effects of optimizations (b) Normalized transaction rate (c) CPU utilization breakdown

Figure 7: Effects of optimizations (a), normalized transaction rates (tpmC) (b), and CPU utilization breakdowns (c) for the large configuration. *dereg* refers to our batching deregistration technique, *intrpt* to interrupt batching, and *sync* to synchronization optimizations.

present relative tpmC numbers, normalized to the local case.

7.1 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 2. The database working set size is around 1 TB, much larger than the aggregate cache size (less than 56 GB) 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 TB total).

Although the aggregate main memory cache size of the V3 configuration is significantly smaller than the total database working set, the architecture has nonetheless significant advantages over typical storage area networks whose customized disk controllers cannot easily support cache sizes of this magnitude; V3's reliance on commodity components allow it to leverage mainstream technology improvements without incurring prohibitive costs. In our experiments, we measured cache hit rates on the order of 18%-20%, a significant number, given the typical access patterns for *TPC–C*.

We first consider the impact of various optimizations on the *TPC–C* transaction rates for the large configuration for *kDSA* and *cDSA*. Figure 7(a) 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 at larger processor counts. Batching interrupts improves system performance by about 7% for *kDSA* and 14% 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 12% for *kDSA* and about 24% 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 7(b) shows the normalized *TPC–C* transaction rate for V3 and a Fibre Channel (FC) storage system; the results for V3 reflect optimizations in

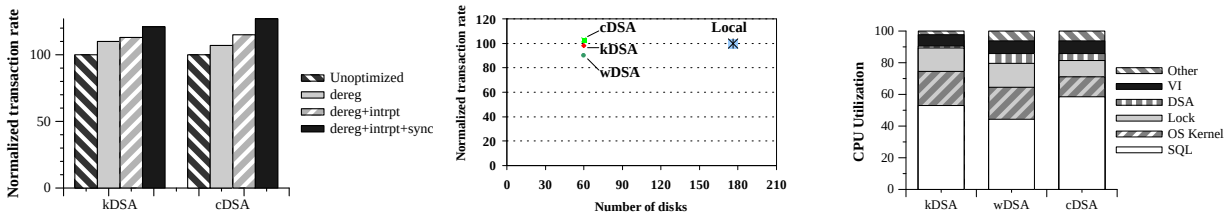
the *kDSA* and *cDSA* implementations. The FC device driver used in the local case is a highly optimized version provided by the disk controller vendor. We find that *kDSA* has competitive performance to the local case. *cDSA* performs 18% better than the local case. *wDSA* performs the worst, with a 22% tpmC lower than *kDSA*.

Figure 7(c) shows the execution time breakdown on the database host for the three client implementations. We do not present the exact breakdown 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. CPU utilization 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 time spent inside the socket library and other standard system libraries. The lock time is a component of either *kDSA* and the OS kernel, *wDSA*, or *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.

7.2 Mid-size Database Configuration

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

We first consider the impact of each optimization on tpmC rates. Figure 8(a) shows that batched deregistration results in a 10% improvement in *kDSA* and 7% in *cDSA*. Batching interrupts increases the transaction rate by an additional 2% 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 nodes 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 large configuration, lock synchronization shows the highest



(a) Effects of optimizations (b) Normalized transaction rate (c) CPU utilization breakdown

Figure 8: Effects of optimizations (a), normalized transaction rates (tpmC) (b), and CPU utilization breakdowns (c) for the mid-size configuration. *dereg* refers to our batching deregistration technique, *intrpt* to interrupt batching, and *sync* to synchronization optimizations.

additional improvement, about 7% in *kDSA* and 10% in *cDSA*.

Next, we look at how V3 performs compared to local SCSI disks. Figure 8(b) 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 than the local case. Finally, *wDSA* is 10% slower compared to the local case.

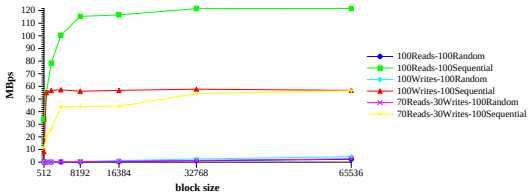
Note that the different DSA implementations are competitive with the local case, but use only one third of the total number of disks (60 as opposed to 176), with the addition of 8GB of memory (6.4GB used for caching) on the V3 server. Although it is not straight forward to calculate system prices, preliminary 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 GB, greater than the aggregate cache size on the V3 server.

Figure 8(c) 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.

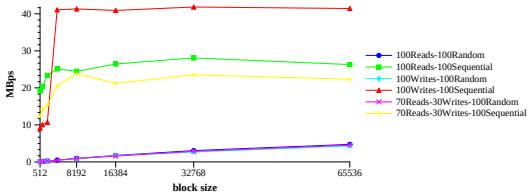
8 Comparision with iSCSI

In this section, we first examine the overheads associated with storage systems built out of commodity PCs and commodity Gigabit Ethernet inteconnects, which has emerged as a promising approach in building network attached storage systems. We then contrast these results with DSA.

Our iSCSI testbed consists of 16, dual processor, AMD-based systems. Each system is equiped with an Athlon MP2200 processor running at 1.8GHz, with 512 MBytes of main memory and a 64bit/66MHz PCI bus. The systems are connected with both 100MBit/s (Intel 82557/8/9 adapter) and 1GBit/s (DLink

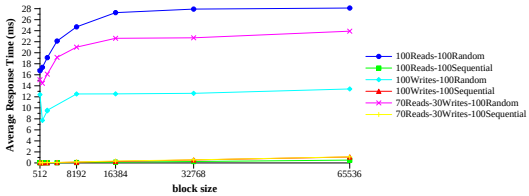


(a) Local

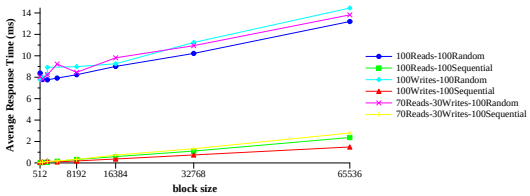


(b) iSCSI

Figure 9: IOmeter throughput.



(a) Local



(b) iSCSI

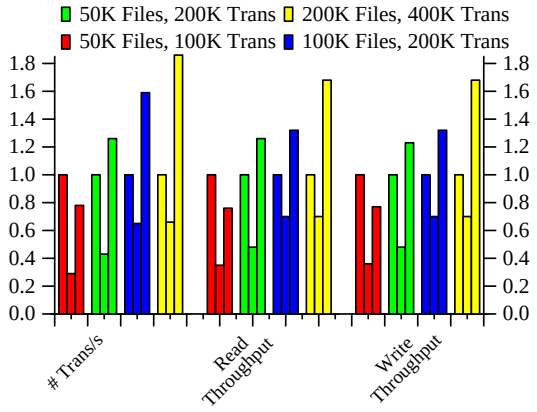
Figure 10: IOmeter average response time.

DGE550T adapter) Ethernet networks. All nodes are connected on a single 24-port switch (DLink DGS-1024) with a 48 GBit/s backplane. The 100MBit network is used only for management purposes. All traffic related to our storage experiments use the GBit Ethernet network. There are two on board IDE controllers on each node – a system IDE controller with two ATA66/100 channels for up to four devices and an IDE Promise PDC20276 RAID controller with two ATA66/100 channels for up to four devices. Each node has an 80-GByte system disk (WD800BB-00CAA1) connected to the system IDE controller. Three of the system nodes are equipped with five additional disks of the same type. All disks (except the system disk) are configured in RAID-0 mode using the Linux MD driver.

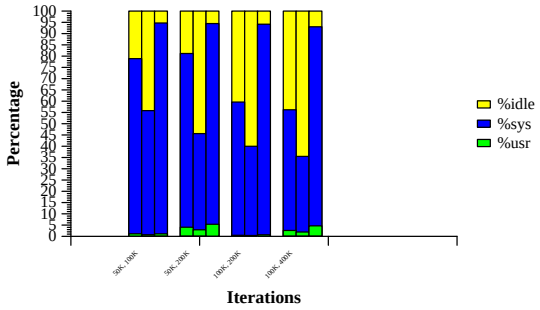
The operating system we use is Linux RedHat 9.0, kernel version 2.4.23-pre5. The iSCSI target and initiator are based on a reference implementation from Intel. Because Intel's iSCSI impelementation supports only non-SMP kernels, our kernel is built with no SMP support. The iSCSI target runs at user level, whereas the iSCSI initiator runs in the kernel. For this work we had to extend the iSCSI target to support block devices larger than 4 GBytes.

In our experiments we examine three configurations: *local,* where disks are attached directly to the application server, *iSCSI,* where one storage node exports a single iSCSI volume, and *iSCSIx3,* where three storage nodes export one volume each and these volumes are concatenated with software RAID0 at the application server.

Figures 9 and 10 compare our local and iSCSI configurations. We see that the maximum throughput achieved with iSCSI is significantly lower than with local disks, for various synthetic workloads. In particular the maximum achieved throughput with iSCSI is about one third of the local case (the maximum throughput path from the initiator to the server is limited by the PCI/GigE hardware to about 120

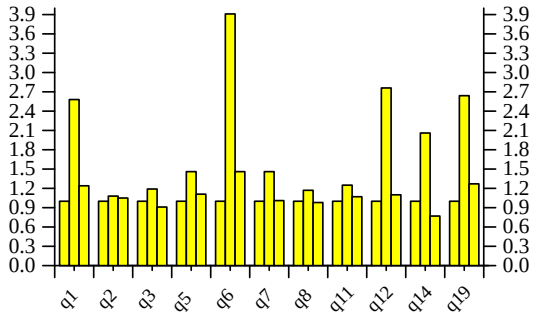


(a) Performace

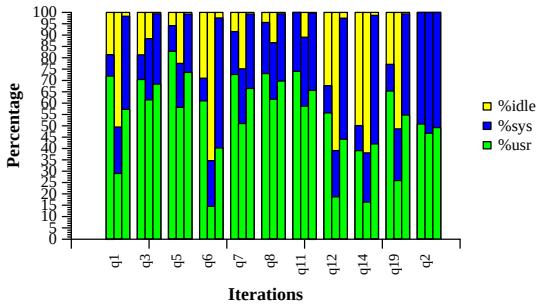


(b) CPU Utilization

Figure 11: Postmark performance and CPU utilization. For each input there are three bars, one for each system configuration: local (left), iSCSI (middle), and iSCSIx3 (right).



(a) Performance



(b) CPU Utilization

Figure 12: MySQL performance and CPU utilization. For each TPC-H query there are three bars, one for each system configuration: local (left), iSCSI (middle), and iSCSIx3 (right).

MBytes/s).

We also consider two applications, Postmark with four different inputs, and MySQL with a subset of TCP-H queries, to examine the impact of iSCSI on CPU utilization (Figures 11 and 12). Our results show that iSCSI introduces additional overhead compared to the local configuration, where disks are directly attached to the application server. This is true, not only for latency, which is to be expected, even in the case of DSA, but also for bandwidth and CPU utilization as well. Using more disks, in the iSCSIx3 configuration closes the performance gap with the local configuration, but does so at the cost of significant increase in CPU utilization (Figures 11(b) and 12(b)).

A direct comparision between iSCSI and DSA is difficult because of the significant differences in the underlying architecture, network fabric, and protocols used between the two. Nonetheless, our experiments reveal that DSA improves CPU utilization and bandwidth significantly compared to the local case, while iSCSI does significantly worse. This suggests that custom storage protocols on top of optimized interconnects, such as DSA over VI, currently have a significant advantage in terms of performance.

8.1 Summary

We find that VI-attached storage incurs very low overheads, especially for requests smaller than 64 KB. 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 the operating system.

9 Related Work

Our work bears similarities with previous research in the following areas: (i) user–level communication, (ii) using VI for databases, and (iii) storage protocols and storage area networks. Next we consider each of these areas separately.

User–level communication: Our work draws heavily on previous research on system area networks and user–level communication systems. Previous 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 [11], BDM [23], Fast Messages [32], PM [35], U-Net [5], and VMMC [16]. Previous work in user–level communication has also addressed the issue of dynamic memory registration and deregistration [10, 6]. 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. 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, which can improve application transaction rates by up to 15% [31, 7, 20]. VI–based networks have also been used to enable parallel database execution on top of clusters of commodity workstations, e.g. in [7, 31], as opposed to tightly integrated database servers.

Storage protocols and storage area networks: The Direct Access File System (DAFS) [14] 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.

Traditionally storage area networks are mostly implemented with SCSI or FC interconnects. 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 [29]. This requires offloading storage related processing 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. Currently there are efforts underway to examine the overhead of iSCSI in various contexts [1, 30, 39, 27, 15]. Since there is no clear understanding of the overheads associated with iSCSI and to better control various system parameters, in this work we perform our own experiments with a commodity iSCSI setup.

Our work is directly applicable to systems that attempt to use Infiniband as a storage area network. Infiniband [25] 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 and RDMA I/O support.

There has also been a lot of work in the broader area of databases and storage systems. For instance, Compaq's TruCluster systems [8] provide unified access to remote disks in a cluster, through a memory channel interconnect [22], which bears many similarities with VI interconnects. Recent work has also examined the use of VI in other storage applications, such as clustered web servers [9]. The authors in [34, 26, 2] study the interaction of OLTP workloads and various architectural features, focusing mostly on smaller workloads than ours. Also, work in disk I/O [38, 19, 18, 37, 3] has explored many directions in the general area of adding intelligence to the storage devices and providing enhanced features. Although our work shares similarities with these efforts, it differs in significant ways, both in terms of goals and techniques used.

Finally, there is currently a number of efforts to incorporate VI–type features in other interconnects as well. Most notably, Infiniband and iSCSI, address many of the issues related to storage area networks and include (or will likely include) features such as RDMA. Our work is directly applicable to systems with Infiniband interconnects and relevant for systems with future iSCSI interconnects.

10 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 back-end and constrast it with a commodity iSCSI implementation. We design a block–level storage architecture (V3) that takes 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 micro-benchmarks and real–world database workloads on a mid–size (4–CPU) and a large (32–CPU) database server.

Our work shows that new storage APIs that help minimize kernel involvement in the I/O path are needed to fully exploit the benefits of user–level communication. We find that on large database configurations *cDSA* provides a 18% 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 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, *cDSA*, implementation. Reducing this overhead requires effort, not only at the interconnect level, but at almost every software layer including the database server, the communication layer, the operating system, and especially the interfaces among these components. More aggressive strategies than *cDSA* are also possible. Storage protocols can provide new, even database–specific, I/O APIs, that take full advantage of user–level communication and allow applications to provide hints and directives to the I/O system. Such protocols and APIs are facilitated by the availability of low–cost CPU cycles on the storage side as found in the V3 architecture.

Our experience shows that VI–based interconnects have a number of advantages compared to more traditional storage area and IP–based networks. (1) RDMA operations available in VI help avoid copying in I/O write operations and allow the contents of I/O buffers to be transfered directly from application memory to the V3 server cache. They also allow the use of flags in application memory that can be directly set by V3, providing a means to notify applications when I/O requests have been serviced, *without* application processor or operating system intervention. Although applications still need to poll these flags, their values are updated without the application losing control of the processor. (2) I/O operations can be initiated with low overhead, reducing application CPU utilization for servicing I/O requests to a bear minimum. (3) 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. (4) Moreover, we examine the behavior of commodity iSCSI systems and find that iSCSI introduces significant overheads compared to directly attached disks. As such, more customized remote storage access protocols, such as DSA, currently have a significant

performance advantage.

Despite these positive features, VI–based interconnects also pose a number of challenges to making them useful to higher layers, especially in I/O–intensive environments, such as database systems. In particular, techniques to deal with flow control, memory registration and deregistration, synchronization, interrupts, and reducing kernel involvement are significant challenges for storage systems, and are not directly handled by VI; such functionality must be added as an intervening layer between the application and VI.

11 Acknowledgments

We would like to thank the reviewers for their insightful comments. We thankfully acknowledge the support of GSRT, Greece, for supporting our iSCSI investigation.

References

[1] S. Aiken, D. Grunwald, and J. W. Andrew R. Pleszkun. A performance analysis of the iscsi protocol. In *11th NASA Goddard, 20st IEEE Conference on Mass Storage Systems and Technologies (MSST2003)*, Apr. 2003.

[2] 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.

[3] 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.

[4] ANSI. Scsi-3 architecture model (sam), x3.270:1996. In *11 West 42nd Street, 13th Floor, New York, NY 10036*.

[5] 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.

[6] 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.

[7] 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.

[8] W. M. Cardoza, F. S. Glover, and W. E. Snaman, Jr. Design of the TruCluster multicomputer system for the Digital UNIX environment. *Digital Technical Journal of Digital Equipment Corporation*, 8(1):5–17, 1996.

[9] 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.

[10] 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.

[11] 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.

[12] K. G. Coffman and A. M. Odlyzko. Growth of the internet. In *In Optical Fiber Telecommunications IV, I. P. Kaminow and T. Li, eds., Academic Press*, 2001.

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

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

[15] I. Dalgic, K. Ozdemir, R. Velpuri, and U. Kukreja. Comparative performance evaluation of iscsi protocol over metro, local, and wide area networks. In *12th NASA Goddard & 21st IEEE Conference on Mass Storage Systems and Technologies (MSST2004)*, Apr. 2004.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

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

[22] R. Gillett, M. Collins, and D. Pimm. Overview of network memory channel for PCI. In *Proceedings of the IEEE Spring COMPCON '96*, Feb. 1996.

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

[24] I. E. T. F. (IETF). iSCSI, version 08. In *IP Storage (IPS), Internet Draft, Document: draft-ietf-ips-iscsi-08.txt*, Sept. 2001.

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

[26] 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.

[27] Y. Lu, Farrukh, Noman, and D. H. Du. Simulation study of iscsi-based storage system. In *12th NASA Goddard & 21st IEEE Conference on Mass Storage Systems and Technologies (MSST2004)*, Apr. 2004.

[28] 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.

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

[30] W. T. Ng, H. Sun, B. Hillyer, E. Shriver, E. Gabber, and B. Ozden. Obtaining high performance for storage outsourcing. In *Proc. of the 1st USENIX Conference on File and Storage Technologies (FAST02)*, pages 145–158, Jan. 2002.

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

[32] 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.

[33] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays for inexpensive disks (raid). In *Proceedings of ACM SIGMOD Conference*, pages 109–116, June 1988.

[34] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. The Impact of Architectural Trends on Operating System Performance. In *Proceedings of the 15^{th} ACM symposium on Operating Systems Principles*, pages 285–298. ACM Press, 1995.

[35] 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.

[36] Transaction Processing Performance Council. *TPC Benchmark C*. Shanley Public Relations, 777 N. First Street, Suite 600, San Jose, CA 95112-6311, May 1991.

[37] 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.

[38] 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.

[39] H. Xiong, R. Kanagavelu, Y. Zhu, and K. L. Yong. An iscsi design and implementation. In *12th NASA Goddard & 21st IEEE Conference on Mass Storage Systems and Technologies (MSST2004)*, Apr. 2004.

[40] Y. Zhou, A. Bilas, S. Jagannathan, C. Dubnicki, J. Philbin, and K. Li. Experiences with vi communication for database storage. In *Proc. of the 29th International Symposium on Computer Architecture (ISCA29)*, May 2002.

[41] Y. Zhou, J. F. Philbin, and K. Li. The multi-queue replacement algorithm for second level buffer caches. In *USENIX Annual Technical Conference*, pages 91–104, June 2001.