Performance Measurements of a User-Space DAFS Server with a Database Workload

Samuel A. Fineberg and Don Wilson
NonStop Labs
Hewlett-Packard Company
19333 Vallco Parkway, M/S 4402
Cupertino, CA 95014
fineberg@hp.com

ABSTRACT

We evaluate the performance of a user-space Direct Access File System (DAFS) server and Oracle Disk Manager (ODM) client using two synthetic test codes as well as the Oracle database. Tests were run on 4-processor Intel Xeon-based systems running Windows 2000. The systems were connected with ServerNet II, a Virtual Interface Architecture (VIA) compliant system area network. We compare the performance of DAFS/ODM and local-disk based I/O, measuring I/O bandwidth and latency. We also compare the runtime and CPU utilization of the Oracle database running the TPC-H benchmark over DAFS/ODM and local disk.

Categories and Subject Descriptors

B.4.m [Input/Output and Data Communications]: Miscellaneous; C.4 [Performance of Systems]: Performance attributes; D.4.3 [Operating Systems]: File Systems Management – *Distributed file systems*; D.4.8 [Operating Systems]: Performance – *Measurements*;

General Terms

Experimentation, Measurement, Performance.

Keywords

DAFS, Database, File Systems, I/O, Networks, Performance Evaluation, RDMA.

1. INTRODUCTION

The Direct Access File System (DAFS) is a new standard for building high-performance network-attached file servers using RDMA-enabled networks. DAFS is designed to enhance the performance of applications within a datacenter, including databases. Oracle has recently introduced the Oracle Disk Manager (ODM) file system API to allow its database to take advantage of advanced file systems like DAFS. Hewlett-Packard's NonStop Labs group has built a prototype user-space DAFS server and a user-space ODM client in order to evaluate DAFS.

In this work we evaluate our DAFS/ODM prototype using two synthetic test codes as well as the Oracle database. These tests were run on 4-processor Intel Xeon-based servers running Windows 2000, connected with the ServerNet II system area network. We make comparisons between the performance of DAFS/ODM and local-disk based I/O, measuring I/O bandwidth and latency. We also compare the runtime and CPU utilization of Oracle running TPC-H over DAFS/ODM and local disk.

2. DAFS SERVER

The server described in this paper has been built to conform to the Direct Access File System (DAFS) [4, 6] protocol. DAFS is a new file-access protocol, based on NFS Version 4 [16]. DAFS is designed to enable high-performance file sharing in data center environments using RDMA-enabled interconnect technologies. These include network technologies such as the Virtual Interface Architecture (VIA) [3, 9], iWarp [1], and InfiniBand [10].

DAFS was developed by the DAFS Collaborative, a group of over 80 organizations including industry, academics, and research labs. DAFS is a file access protocol, like NFS [2] or CIFS [17], that can be accessed either through a kernel-based or user-library based client. The DAFS collaborative has defined an API that takes advantage of DAFS's advanced features, but other interfaces can be supported by the DAFS protocol as well.

2.1 DAFS Details

DAFS is a client-server distributed *file* access protocol. Therefore, data requested from a DAFS server is specified as bytes at a offset in a file, not raw disk blocks. DAFS builds on the NFS4 specification, but adds enhanced features ideal for *local file sharing* environments. By local file sharing, we mean systems that are in close proximity and connected by a high-speed RDMA-enabled private network.

Because DAFS is simply a messaging protocol it can be implemented at several levels in an operating system. VIA is by its nature a user-space networking model, so user-space implementations of both DAFS clients and servers are possible as demonstrated in this work. DAFS clients and servers can also be implemented inside an operating system kernel. For example, Magoutis et. al. [11, 12] implemented a user-space client and a kernel-space server. In addition, Broadband Storage, Fujitsu, and Network Appliance demonstrated kernel based servers and clients at the first DAFS Developers Conference. The DAFS protocol is flexible enough to allow clients to deliver many different file APIs. This could include the familiar Windows and UNIX file APIs, as well as new

APIs such as the DAFS API [5], Oracle's ODM API [14, 15], or MPI2-I/O [13].

2.1.1 Session Management and Request/Response Model

A DAFS client initially connects to a server by establishing a connection to a well known network address on the server. For example, on VIA based systems, the client connects to a VI with a known remote address and a known discriminator (which is like a TCP/IP port number). DAFS requests and responses are simply send/receive style messages with a specified format. DAFS messages include a generic header that is the same for any type of request/response, a fixed-size portion unique to the specific request/response, and a "heap" containing variable-sized fields and data. Request messages are sent from a client to the server, and responses are sent from the server to a client. The client and server must pre-post "receives" before they can accept a request or response messages.

A DAFS server will pre-post a number of receives to each of its active client connections. The number of pre-posted receives will determine the number of outstanding requests the server can accept from each client connection (this parameter is negotiated as part of the session establishment protocol). The server will then pre-post an additional receive before responding to each client request.

Clients pre-post a receive to be used for the response before sending each server request. In addition, clients will need to pre-post additional receives if they want to use the optional back-channel request features of DAFS. Since our server did not support back-channel requests, clients only received messages in response to a request.

All DAFS server state is stored as part of a session. Session state includes credentials, open files, locks, etc. When a session is lost, all open filehandles must be closed, all locks are released, etc. Note that a session does not necessarily correspond to a network connection. If a connection is lost, the client can attempt to re-attach to the pre-existing session. However, for the server implemented in this paper, sessions only exist for the life of a connection.

2.1.2 Non-file Access DAFS Commands

DAFS includes commands to browse files and directories. A DAFS file or directory is referenced by using a *filehandle*. The root of a server's filesystem can be obtained using the `DAFS_PROC_GET_ROOTHANDLE` request. Then, directories can be browsed using `DAFS_PROC_LOOKUP` (to look up the filehandle of a directory or file) and `DAFS_PROC_LOOKUPP` (to look up the parent directory of a directory or file). Directory contents can be read using `DAFS_PROC_READDIR_x` where `x` is either `INLINE` or `DIRECT` (the difference between inline and direct requests will be described later in this section).

Other non-file access DAFS requests include commands for creating directories, deleting files, managing locks, managing the server's response cache (if it has one), providing hints about file usage, reading/writing file attributes, etc. Each of these requests has a header and heap format outlined in the DAFS specification, and DAFS also specifies the header and heap format for the server's response.

2.1.3 File Access Commands

Before any data can be read or written, DAFS files must be explicitly opened using the `DAFS_PROC_OPEN` request. The open request takes the filehandle of the containing directory, as well as any other desired parameters, and the response returns a filehandle to the specified file. Open files can be closed using the `DAFS_PROC_CLOSE` request. Once a file is open, the client can read from or write to the file using the `DAFS_PROC_READ_x` or `DAFS_PROC_WRITE_x` requests where `x` is either *`INLINE`* or *`DIRECT`*.

2.1.3.1 Inline I/O

Inline requests carry their data in the actual request or response messages. Therefore, for `DAFS_PROC_READ_INLINE`, the request packet will specify the filehandle, offset, and number bytes to be read. The response packet will include status, number of bytes successfully read, and the actual data from the file. A `DAFS_PROC_WRITE_INLINE` request will contain the filehandle, offset, the number of bytes to be written, and the data to be written to the file. Then, the response will return status and the actual number of bytes written. Because all data must fit in a single message, inline I/O data sizes are limited by the amount of space available in the maximum size request/response message.

2.1.3.2 Direct I/O

Direct requests separate the data transfer from the request/response messages, transferring the data with RDMA reads or RDMA writes. For example, a `DAFS_PROC_READ_DIRECT` request message specifies a filehandle, file offset, and requested number of bytes to read. In addition, it will include the virtual address of the user's data buffer on the client machine and a "memory handle" for that buffer. The server then reads the file bytes and copies them directly to the user buffer on the client system with a RDMA write. Finally, after the RDMA write has completed, a response will be sent specifying status and the amount of data successfully read.

Note that all buffers must be "registered" prior to issuing any DAFS request. Registering memory causes the operating system to lock the memory pages into RAM and it sets up virtual to physical address translation tables in the NIC. This allows an RDMA-enabled NIC to perform user-space DMA using the user's virtual address instead of a physical address. Inline request/response buffers can be pre-registered when the client library is initialized. Therefore, for inline I/O there is usually no per-I/O memory registration overhead, but data must be copied from the user buffer to one of these "system" buffers. For direct I/O, there is usually no copying needed since I/O can occur directly to/from the user buffer. However, the user buffer must be registered before it can be used in any direct I/Os.

2.1.3.3 Inline vs. Direct I/O Trade-offs

There are numerous trade-offs between direct and inline I/O. In general, direct I/O is considered better because the client saves the overhead of copying data to or from a message buffer. This can save a significant amount of CPU overhead, improving the performance of CPU-bound applications. In addition, with direct I/O RDMA data transfer lengths can be up to the network's Maximum Transfer Unit (MTU) size (ServerNet II's MTU is 4GB, though a more typical MTU size for other VIA networks is 64KB). To limit buffer usage, most DAFS implementations will limit the maximum single transfer size (our server limits this to 1MB by default). However, when a DAFS server processes a single direct read or write request, it may issue one or more RDMA operations. This means that a direct operation can be of virtually unlimited size, reducing the number of request/response messages. Finally, bulk data movement is server-driven, i.e., the clients do not initiate RDMA transfers. Since the server initiates all RDMA, it can schedule data movement to avoid overloading its data links. The server can also use this control over RDMA to efficiently utilize its buffer memory and ensure fairness.

Inline I/Os require copying from system buffers to user buffers. In addition, they must fit into a single request or response packet, so they are limited in size. However, direct I/O actually requires additional steps over inline I/O. As with inline I/O, direct I/O operations still must send a request and response message. In addition, direct I/O clients must register user data buffers so that they can be used as targets of server-initiated RDMA. Inline message buffers also need to be registered, however, in most cases DAFS clients will pre-register all message buffers at client initialization time. This pre-registration eliminates any per-I/O memory registration overhead. While the extra operations required for direct I/O should take less time than additional request/response messages, for short I/Os that fit in a single request/response message, the copying overhead of inline I/O may be less than the additional overhead needed for direct I/O.

2.2 Prototype DAFS Server Details

The DAFS server used in this paper was developed by HP's NonStop Labs. It supports a subset of the DAFS 1.0 specification. The server is a user space service/daemon for Windows 2000 and Linux (however, all performance testing was performed on Windows). It uses the Virtual Interface Provider Library (VIPL) 1.0 [9] for all communication operations. To ensure compatibility, messages are constructed using the protocol stubs from the DAFS 1.0 SDK[1] (we do not use any of the SDK's transport functions).

All of the DAFS server's buffers are pre-allocated and pre-registered. This includes both a pool of send/receive buffers for request/response messages as well as a pool of direct I/O buffers used for RDMA. Memory registration only occurs once at program initialization. Further, disk I/O is performed directly to/from the server's message or direct I/O buffers, so no additional data copies are required.

The prototype DAFS server is heavily multithreaded and utilizes an I/O driven architecture to enhance request pipelining. The server is designed to handle many concurrent requests such that the utilization of the system's functional units will be maximized. This approach optimizes total server throughput, but it may increase the latency of a single DAFS request.

Files are stored on the native filesystem (ext2 for Linux and NTFS for Windows 2000) and file access is performed with the standard filesystem APIs. The server does optionally use several performance features of Win32, including asynchronous and unbuffered I/O. Asynchronous I/O eliminates contention for file pointers. When the DAFS server opens a file for a client, it only actually opens it once. If the server utilizes a synchronous I/O interface, each operation must consist of a seek followed by a read or write. The seek and the read/write must act as a single atomic operation, so the file must be locked. This means that only a single I/O may be operating on each file at a time. With asynchronous I/O, called "overlapped" in Win32, the offset is included in the read or write command and the file pointer is ignored. This means that multiple reads and writes can be issued simultaneously, without locking the file.

[1] The DAFS SDK 1.0 is available from the DAFS Collaborative web site (http://www.dafscollaborative.org).

The second Windows feature we utilized is "unbuffered I/O". This option is selected when a file is opened. Access to files open in this mode bypasses the Windows buffer cache. This has two disadvantages, first it means that Windows will do no caching of data. However, this works well since both Oracle and the RAID controller already do their own caching. The second disadvantage is that all data must be sector (512 bytes in most cases) aligned. This is a problem for general I/O, but ODM allows you to easily impose this restriction on Oracle (by setting ODM's "Physical Block Size" parameter to 512). Unbuffered I/O tends to be much faster than buffered I/O when data is read only once and accesses are random. The data copying overhead generated by buffered I/O can be very large, and it may consume a significant amount of a system's CPU cycles.

3. ORACLE DISK MANAGER

The Oracle Disk Manager (ODM) API [14, 15] is a file access interface specification for use with Oracle databases. It is a built-in feature of the Oracle 9i database. ODM provides an alternate interface for creating, deleting, and accessing files. The ODM interface is a stripped down file API with several enhancements over traditional file APIs. Oracle attempts to create and open all files with ODM. If the files are not available through the ODM library an error is returned, and Oracle will revert to the normal filesystem APIs.

3.1 ODM Commands

ODM files are created with the `odm_create` command. `odm_create` includes a file size parameter so files can be pre-allocated. In addition, ODM files include a file type specifying the database's intended use for the file. After the file has been created, it can be accessed by the Oracle instance that created it. However, the file does not become persistent and is not accessible by any other Oracle instances until it is committed with the `odm_commit` command. If Oracle crashes before calling `odm_commit`, or the file is intentionally aborted using the `odm_abort` command, the file and all changes will be lost.

Once a file has been committed, it can be opened by one or more Oracle instances. To open a file, Oracle uses the `odm_identify` command. `odm_identify`, like `odm_commit`, includes a file type parameter. `odm_identify` also includes a "key" parameter to prevent the accidental opening of a file by an Oracle instance that is not part of the same cluster.

Once a file has been identified, all I/O is performed using the `odm_io` command. `odm_io` uses an asynchronous descriptor-based interface. It specifies what I/O requests to initiate and what to wait for. When a thread wants to wait for I/O to complete, it can tell `odm_io` to either wait on specific I/O requests or wait for some number of I/O requests to complete.

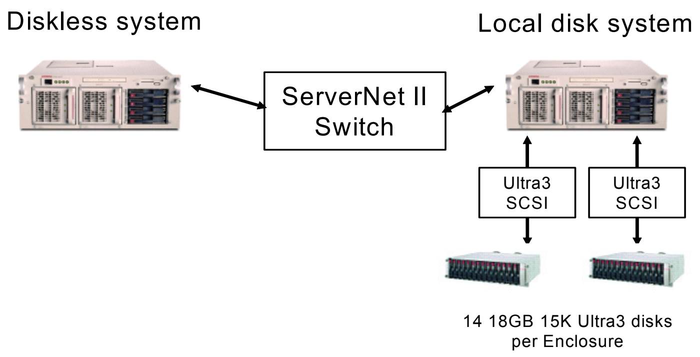

Figure 1. Test system configuration

Any parameter can be set to NULL, so `odm_io` can be used to queue I/O without waiting or `odm_io` can wait for I/O without queuing anything. This interface provides a lot of flexibility for different I/O usage patterns with a minimum of command syntax. I/Os are segregated by thread, so only the thread that issued an I/O request can wait for it. Odm also has provisions for batch I/O using a "`ODM_RELATED`" flag, however this was ignored in the prototype server.

Odm also includes several other functions including a close command (`odm_unidentify`), commands for managing mirrored volumes (which were not implemented in the prototype), and a command for resizing a file (which is used when extending database files).

3.2 Prototype ODM Client

The prototype ODM client was implemented by NonStop Labs. It is implemented as a user-space dynamic-link library (DLL) for Oracle 9i on Windows 2000. The client library uses the DAFS SDK 1.0 protocol stubs for compatibility, but like our DAFS server, all data movement operations were re-implemented directly on top of VIPL. Currently our ODM library only supports a single DAFS server, specified in a file that is read when the DLL is initialized. The DAFS filesystem appears as a special drive letter, also set in the configuration file. Oracle attempts to open all files using ODM. When Oracle uses an ODM command on a non-DAFS filesystem the ODM library will return an error indicating that the file is not an odm file.

The ODM library is heavily multithreaded, and decouples asynchronous I/Os from the Oracle threads. The number of outstanding I/Os supported by the prototype ODM is tunable. Small I/Os are implemented using the inline read/write commands and pre-allocated/pre-registered data buffers. Larger I/Os are implemented using the direct read and write commands.

Our ODM library registers/deregisters direct read and write buffers "on the fly." This creates a lot of memory registration overhead, however, it was less than we originally expected, and it was necessary. While we could attempt to cache client memory registrations, it would lead to problems. ODM has no concept of memory registration, and there are no restrictions on the system calls that a client can perform. Unfortunately, some system calls can modify the virtual-to-physical address mappings without notifying the ODM library or the network driver. For example, consider what happens if a program "malloc"s a chunk of memory, uses it for a direct I/O, then frees it. At this point there is no guarantee that in a subsequent "malloc" the OS will not re-map a different chunk of physical memory to that same virtual address. This problem was observed previously when implementing MVICH[1] (an implementation of the Message Passing Interface over VIA) as well as in DSA [19] (a VIA-based storage library for SQL server). While we could implement appropriate OS hooks that would perform memory deregistration as part of the "free" command, this is outside the scope of the user-space DLL that we implemented. In addition, we could have forced the client to notify ODM when it does anything that could affect the virtual to physical address mappings of its memory, but this would have required changes to Oracle.

4. DAFS TESTBED SYSTEM

Our goal in testing DAFS/ODM was to compare its performance with locally connected disks. To run our "local disk" experiments we used a system configuration that was reasonable for running Oracle. Therefore, the system had multiple Intel Xeon CPUs, lots of memory, high performance 15000RPM disks, and high performance RAID controllers. This "local disk" system became our baseline for measurement, and we ran all "local disk" experiments on this system alone.

To test DAFS/ODM, we added a second multiprocessor Xeon system with a configuration similar to the first, but with no high performance disks. In addition, we added a fast system area network to connect the two systems. For the DAFS/ODM tests, the second "diskless" system ran the DAFS/ODM client software (e.g., Oracle), and the "local disk" system ran the DAFS server software.

4.1 System Configuration

The system configuration used for our tests is illustrated in Figure 1. The specific system configurations were as follows:

- HP Proliant 6400R System:
  - four Xeon 500MHz processors with 1MByte L2 cache
  - 3GB RAM
  - dual bus PCI 64/33
- ServerNet II system area network
- Windows 2000 Server

[1] For information on MVICH see: http://www.nersc.gov/research/FTG/mvich/.

In addition, the "local disk" system also had the following storage hardware:

- Two Proliant Smart Array 5304 Controllers
  - 128 MBytes of battery-backed disk cache per controller
  - 4 ultra3 SCSI ports per controller (only one was used)
- Two Proliant 4314R disk enclosures
  - 1 ultra3 SCSI Bus with 14 disk slots (per enclosure)
  - 14 18GB 15000 RPM ultra3 disks (Seagate) per enclosure

The Smart Array controller was configured for RAID 1/0 (i.e., mirrored striped disks) with one entire enclosure (14 disks) per RAID partition. We formatted each RAID partition with a separate NTFS filesystem. Both systems also had some lower performance SCSI drives that were used for their operating system and Oracle binaries.

4.2 ServerNet II SAN

All DAFS commands were sent over the ServerNet II system area network (SAN). ServerNet II [7, 8] is a hardware-based VIA 1.0 compliant network, developed by the HP's NonStop Enterprise Division. ServerNet and ServerNet II were originally developed for HP (Tandem) NonStop servers, and they were marketed as a general system area network for PC clusters from 1997-2000. In 2000 ServerNet was discontinued as a separate product, but it is still utilized as the internal fabric on NonStop Kernel based servers. ServerNet II has a VIPL 1.0 library, and it supports several optional features of VIA. These include RDMA read and Reliable Reception (both of which were used for the server). The ServerNet II NICs used for this work were 64-bit/66MHz PCI cards. ServerNet II networks can be built in a variety of topologies using 12-port switches. For this work, however, only a single switch was needed.

Each ServerNet NIC has two ports that can be attached into separate network fabrics. In a dual-fabric network ServerNet can tolerate a fabric failure by switching all traffic to the alternate. In addition, ServerNet II can stripe network traffic across both fabrics to achieve better performance. The ServerNet network used for these experiments was set up with dual fabrics. However, both ports were attached to the same ServerNet switch. This configuration does not provide the same level of fault tolerance as dual-switch configuration, but it does have similar performance.

In theory ServerNet II should perform at 1.25 Gbit/sec per link per direction, i.e., 5 Gbits/sec, which translates to over 400 MBytes/sec (after protocol overhead). However, our test system's PCI 64/33 bus limits performance to only 250 MBytes/sec. In addition, ServerNet II bandwidth is affected by many factors including the system's PCI implementation, driver tuning parameters, the type of data transfer operation, the number of network fabrics used, and the number of Virtual Interfaces (VIs) used. These performance factors are outside the scope of this paper. For the configuration and usage model utilized in this paper's experiments, we measured the maximum ServerNet II bandwidth to be 110 MBytes/sec.

5. RAW I/O BENCHMARK RESULTS

We used three benchmarks to measure the performance of the test system. The first two of these were raw I/O tests. *Odmblast* was an ODM based I/O stress tests and *Odmlat* was an ODM based I/O latency tester. The final test was an Oracle based TPC-H [18] benchmark, which will be discussed in Section 6.0. All of the tests were run on local disk as well as over DAFS. We utilized the ODM library for all tests except the local disk TPC-H test. For the DAFS tests, ODM communicated with our DAFS server over ServerNet II. For local disk *Odmblast* and *Odmlat*, a special ODM library was used that emulated the ODM commands using direct Win32 calls. The local disk TPC-H test accessed the local filesystem directly from Oracle (i.e., it did not use ODM).

5.1 Odmblast

Odmblast is an odm-based stress test. It was designed to create a high load on the DAFS server using the ODM client interface. The *odmblast* code consists of a closed loop in which up to 32 I/Os are outstanding. Each call to `odm_io` issues 16 I/O requests and waits on 16 requests (from the previous call to `odm_io`). *Odmblast* has the ability to spread its requests across two files. In the reported experiments we map these files to two separate filesystems, each with a separate RAID controller and disk set. This improves performance and mimics the Oracle layout we utilized in Section 6. We issued enough I/Os to ensure that the entire file could not be held in the RAID cache or system RAM (4GB of I/O per file for the measurements shown).

Odmblast sequences through a series of tests. These tests are sequential read, sequential write, random read, random write, and random read/write mix. Each test reads or writes an amount of data equal to the file size (i.e., with two 4GB files, 4GB of I/O requests are issued to each file for a total of 8GB of I/O).

For the sequential read and write tests, I/Os are issued with increasing offset and fixed block size, alternating between the two files. Therefore, the first set of I/Os uses offsets *file1@0*, *file2@0*, *file1@1*blocksize*, *file2@1*blocksize*,..., *file1@7*blocksize*, *file2@7*blocksize*; the second set of requests uses offsets *file1@8*blocksize*, *file2@8*blocksize*,..., *file2@15*blocksize*. For random read and write tests, *odmblast* still alternates between file1 and file2, but we used a uniform random number generator to create offsets. For the random read/write mix test, the offset was chosen randomly and the choice between read and write was also decided by the random number generator. Both random and sequential offsets are limited to full disk sectors (512 bytes) to allow the use of unbuffered I/O.

Figure 2 shows read performance across a range of block sizes. The DAFS results highlight the limitations of our ServerNet II network. Local disk sequential reads run over twice as fast as sequential reads over DAFS. Random reads do not perform quite as well as sequential, especially for small block sizes. This difference between sequential and random reads was evident for both DAFS and local I/O, and is probably due to the ineffectiveness of the read-ahead RAID cache (i.e., read-aheads often went unused due to the random access pattern) and the 64KB RAID block size. For both sequential and random I/O, DAFS has a bandwidth ceiling at about 90 MBytes/sec. The largest factor in this bandwidth limit is ServerNet II. As mentioned previously, the maximum ServerNet II bandwidth for our system was 110 MBytes/sec. While we might expect DAFS reads to reach the full ServerNet II bandwidth, there are several factors preventing this. These include some protocol inefficiency, imperfect request pipelining, as well as competition for PCI bus resources (at

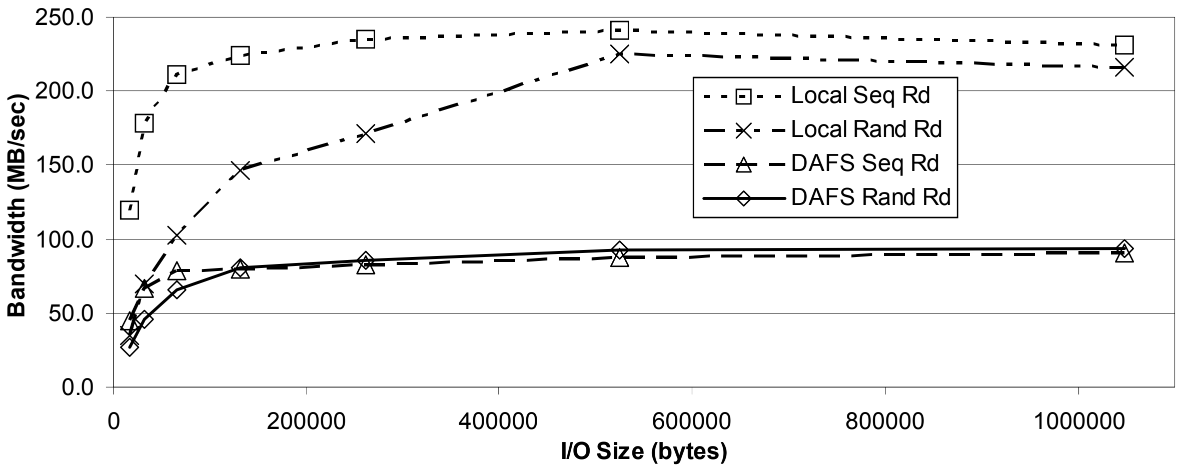

Figure 2. Odmblast read bandwidth vs. I/O block size

least one of the RAID controllers shares a PCI bus with the ServerNet II NIC).

Figure 3 shows write performance across a range of block sizes. Write performance for both local disk and DAFS was much lower than read. The lower local disk write performance was mostly due to the expense incurred through mirroring. This expense was somewhat offset by the RAID cache, but the *odmblast* test wrote enough data that the system had to eventually wait for data to flush to disk. What is interesting is that small random writes exceeded the performance of small sequential writes. When random writes increase in size they become more like sequential writes, so the two performance curves converge. The differences between local disk and DAFS were harder to explain. For small random writes, DAFS performance is actually higher than local disk. We do not have a good explanation for this anomalous behavior. For large block sizes, performance of DAFS is 25%-35% worse than local I/O. We can not blame this behavior on bandwidth limitations, since it never exceeds the achieved DAFS read performance, so it must be due to inefficiencies in our DAFS/ODM code.

In Figure 4, we show performance for a random 50%/50% read/write mix at random offsets. Performance is better than random writing alone, which is not surprising since half of the operations are reads. For local I/O the performance falls in the expected mid-ground between read and write performance. What is interesting is that the DAFS random read/write bandwidth actually exceeds both the DAFS random read and DAFS random write performance demonstrated previously. DAFS performance is ultimately limited to <110 MBytes/sec by the ServerNet II bandwidth. However, this indicates that our DAFS/ODM code pipelines requests better, and more efficiently uses its network bandwidth, when the workload is a mix of reads and writes.

5.2 Odmlat

Odmlat was designed to measure the latency of I/O requests. Our prototype DAFS/ODM was designed to optimize throughput at the expense of latency, so we did not expect the latency to be extremely low. *Odmlat* simply issues a single I/O with a fixed size, waits for it to complete, then repeats this with increasing offsets within the file. This continues for a specified number of iterations, enough to get an accurate

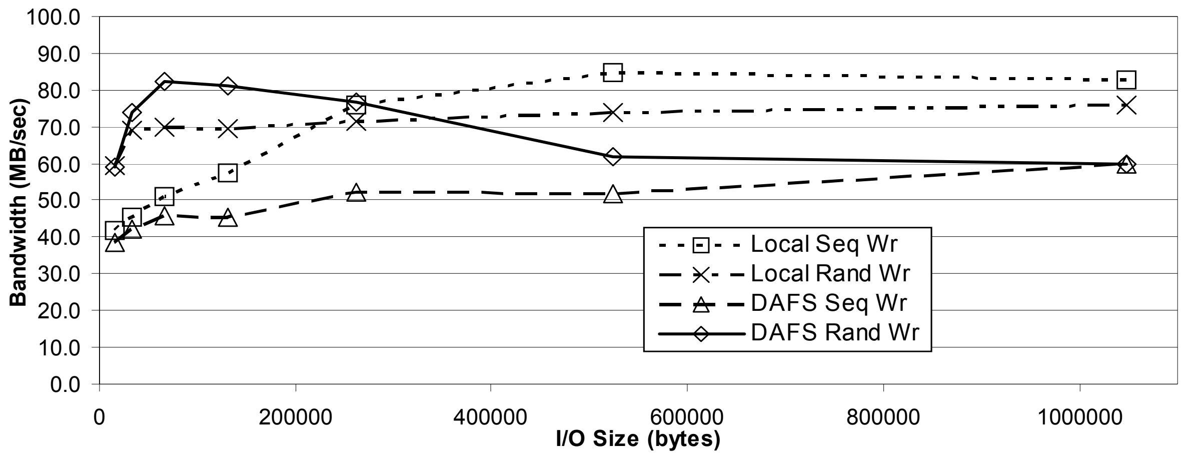

Figure 3. Odmblast write bandwidth vs. I/O block size

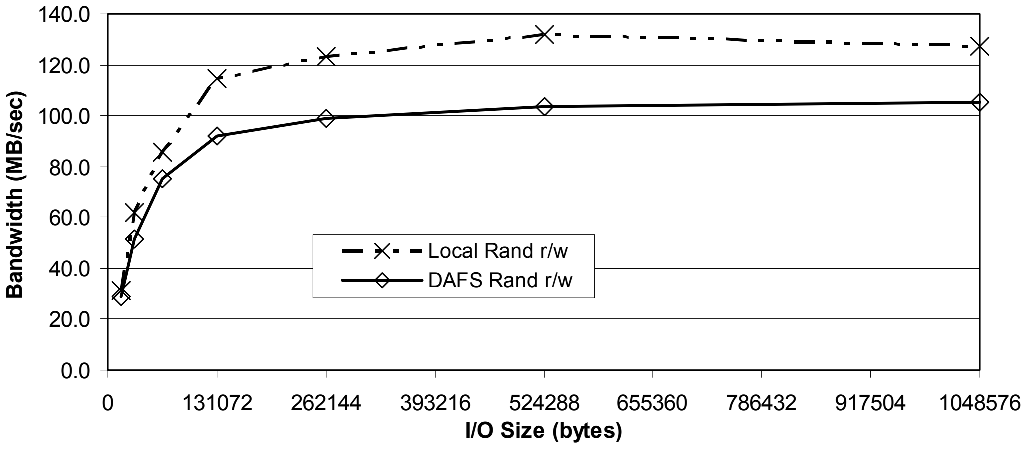

Figure 4. Odmblast random read/write mix bandwidth vs. I/O block size

timing. The total time is divided by the number of operations to get the time per operation.

Our goal was to use *odmlat* to determine the components that affect DAFS/ODM response time. Therefore, we used sequential I/O to normalize the effects of the RAID cache. In other words, we wanted the RAID cache to affect all I/O equivalently, rather than randomly. We only ran these tests over DAFS/ODM (not on local disk) and we performed a linear regression of the data to separate out the response time components.

The components that we expected to see from a DAFS/ODM I/O were the following.

T_{req}: The time required to perform a zero-byte request/response. This is essentially the fixed amount of time it takes to send/receive a request, process the request, and send/receive a response message.

T_{disk}: The time required to perform a disk I/O on the server. This fixed time is essentially the amount of time it takes for the server to perform a zero-byte I/O to disk, including system call overhead (recall our DAFS server runs in user-space).

T_{rdma}: The time required to perform a server initiated zero-byte RDMA operation. This fixed value includes the time required to initiate an RDMA operation and process its completion. This time will also include the fixed portion of the amount of time required for registering a user buffer.

R_{inline}: The overall data rate (measured MBytes/sec) to disk for a single inline I/O. This is an aggregate measure of how fast data moves in an inline operation. It is affected by the speed of memory copies (recall that for inline I/Os we copy the users I/O buffer into a pre-registered message buffer), data transfer for a send/receive network operation as well as the disk transfer rate. It is not the I/O bandwidth, since I/Os must include some of the previously described fixed components as well.

R_{direct}: The overall data rate (MBytes/sec) to disk for a single direct I/O. This is an aggregate measure of how fast data moves in an direct operation. It is affected by the speed of memory registration (recall that we register user I/O buffers every time we perform a direct I/O), RDMA data transfer rate, as well as the disk transfer rate. It is not the I/O bandwidth, since I/Os must include some of the previously described fixed components as well,

From these components we can create performance models for inline and direct I/Os. Inline I/Os take the following time:

$T_{inline\ I/O} = T_{req} + T_{disk} + Size/R_{inline}$.

Therefore, they consist of a request/response, a disk I/O, and data transfer. Direct I/Os take the following time:

$T_{rdma\ I/O} = T_{req} + T_{rdma} + T_{disk} + Size/R_{direct}$.

Therefore, direct I/O consists of a request/response, an RDMA operation, a disk I/O, and data transfer. Note that because direct I/O data transfer is different than inline I/O data transfer, we use a different data transfer rate parameter.

The basic measurements are shown in Figure 5. This graph shows that write latencies are slightly higher than read latencies. This probably reflects the benefit of the RAID controller's read-ahead buffering, especially since the I/O was sequential. Zero-byte read/write operations took about 90 microseconds. The DAFS server does not issue an I/O for the zero-byte case, so the zero-byte latency (90 microseconds) should be T_{req}. As I/O size increases, the latency increases due to the fact that a disk I/O must be issued on the server side, and also due to the disk and network bandwidth. We performed a linear regression and determined that the fixed portion of this increase (T_{disk}) was 535 microseconds for write and 519 microseconds for read. At an I/O size of 16 KBytes (16384 bytes), ODM switches from inline to direct I/O. The graph shows a continuous line between these points, however, in reality the line should have a step between 16383 bytes and 16384 bytes. The added latency (T_{rdma}) was determined to be 446 microseconds for write and 372 microseconds for read. This was due to the additional RDMA operation and the memory registration overhead. What is interesting is that the slope of the graph for I/Os <16 KBytes (inline) is essentially the same as the slope for >16 KByte I/Os (direct). From these slopes we calculated the rates shown in Table 1. The difference between inline and direct I/O transfer rates was <3%. For both inline and direct I/O, reads were about 10% faster than writes.

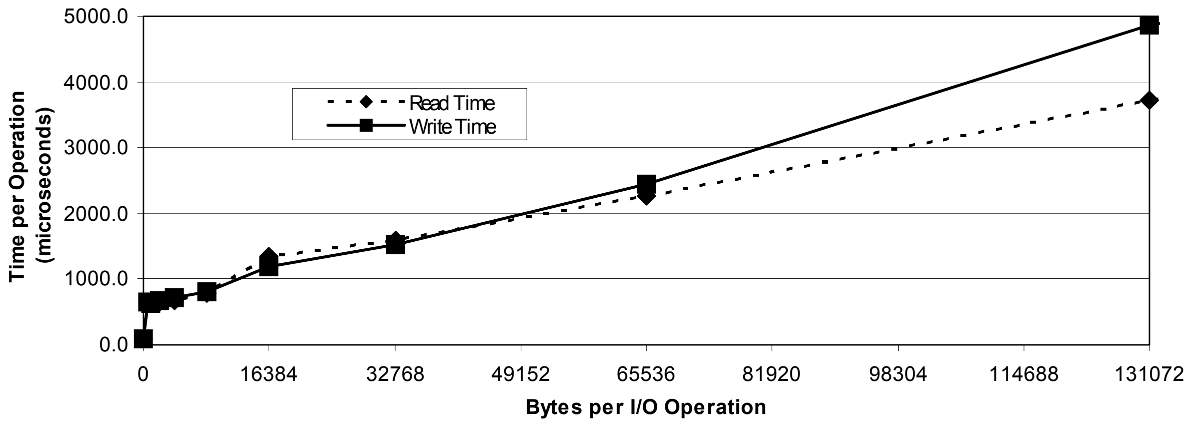

Figure 5. Odmlat I/O latency vs. block size

The difference between read and write performance reflects those seen with *odmblast* in Section 5.1.

Transfer Rate Type	Read (MBytes/sec)	Write (MBytes/sec)
R_{inline}	47.4	44.4
R_{direct}	48.8	43.5

Table 1: Data transfer rates for DAFS I/O requests

This data might naively indicate that we should always use inline I/O since the direct I/O transfer rate is not higher and direct I/O must incur additional latency (i.e., $T_{direct\ I/O} = T_{inline\ I/O} + T_{rdma}$). However, this only works for I/Os that can fit in a single message. For larger messages: $T_{inline\ I/O} = \text{ceiling}(Size_{total}/Size_{inline\ max})T_{req} + Size_{total}/R_{inline}$ (where ceiling is a function that rounds to the next higher integer). Because our ODM library was limiting the size of inline requests to 16383 bytes (our server's actual inline request size limit was set to 32K and could have been bigger at the expense of system RAM), we did not see any inline I/Os requiring multiple messages.

In addition, the real cost of inline I/O is the additional CPU overhead due to buffer copying. This cost can be large, especially when (unlike *odmlat*) multiple I/Os are in flight simultaneously, as is the normal case for database I/O. In addition, we are being penalized by the total time it takes to register memory, even though memory registrations are parallelized across the ODM threads. In an ideal world, we would require Oracle to pre-register its buffers, but this would require changes to both Oracle and the ODM specification.

6. ORACLE BASED TPC-H RESULTS

The previously described tests, while useful in characterizing DAFS/ODM performance, do not represent a realistic usage model. ODM is an Oracle I/O interface, so it was critical that we make measurements using the Oracle database. We chose to utilize TPC-H for our Oracle tests. TPC-H is a decision support benchmark created by the Transaction Processing Performance Council [18].

6.1 TPC-H Experiments

TPC-H is designed to emulate database systems that examine large volumes of data, execute complex queries, and give answers to business questions. The queries give answers to real-world business questions, simulate ad-hoc queries, and are far more complex than OLTP queries. The database must be continuously open to queries from multiple users, so we simultaneously ran multiple query "streams." In addition, the database was continuously updated throughout the test by an "update thread," and these updates could not be allowed to corrupt running queries.

The TPC-H test was run on Oracle 9i Enterprise Edition (Release 1) for Windows. We set the Oracle block size to 16 KBytes, which meant that Oracle would never issue I/Os of less than 16 KBytes. The large block size enabled more efficient access to both local and remote disk, and it was reasonable given the large queries in TPC-H. In addition, we set the ODM physical block size to 512 bytes to ensure that all I/Os would be 512-byte (sector) aligned, which was needed for Windows unbuffered I/O.

We used two different disk configurations, as described in Section 4. For the "local disk" test, we ran Oracle on the "local disk" system and placed the files on the attached disks. The database files were stored on NTFS, not on a "raw" filesystem. The database's tablespace files were spread across the two RAID filesystems to improve performance. For the "DAFS/ODM" test we ran Oracle on the "diskless" system and ran our DAFS server software on the "local disk" system. The ODM ".dll" file was installed on the "diskless" system and Oracle built its database files using ODM's file APIs. ODM files were stored on the "local disk" system, and were accessed remotely using the ODM library, which communicated with the DAFS server software over ServerNet II. The database's files were spread across the same two RAID filesystems used for "local disk" Oracle, with file locations matching in both cases.

The goal of our tests was to measure the relative performance of DAFS/ODM vs. local disk. We were not attempting to create a fully auditable TPC-H implementation that could be directly compared with other published results. Our tests used the 30GB database size from the TPC-H specification, which was sufficiently large so that it would not fit in RAM or disk

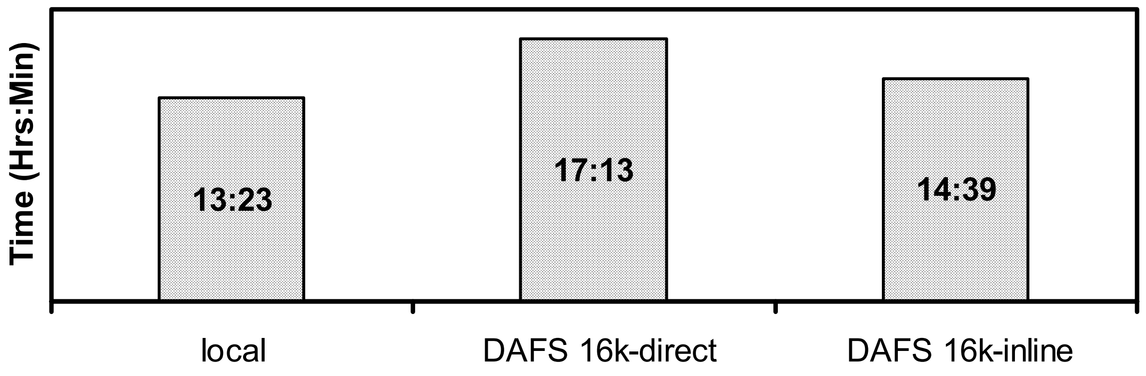

Figure 6. Average query stream runtime

cache. Per the TPC-H spec, we ran a single update thread while the queries were executing.

Unfortunately, given our system configuration, the full benchmark took an unreasonable amount of time. Therefore we chose to run only three simultaneous query streams instead of the 4 required by the benchmark rules. While this change may have somewhat reduced our peak concurrency, it did significantly reduce execution time. Therefore, there was some bottleneck (most likely I/O rate, i.e., number of I/O operations per second our disks could process) preventing the 4th query's additional work from being overlapped effectively. We also left out query 13, which we were unable to run successfully. This was due to incompatibilities that we were unable to resolve between the SQL code provided by TPC and Oracle's SQL.

In addition to varying whether the tests were run on local disk or DAFS, we also varied the "inline cutoff" size where ODM switched from "inline" to "direct" I/O. We wanted to use inline transport for small I/Os since it had lower latency. However, as I/Os grew, the latency became less relevant and we want to take advantage of the benefits of direct (RDMA) I/O. Because 16 KBytes was the basic block size of our Oracle database, and would therefore be used frequently, we decided to compare TPC-H performance with two different settings of the "inline cutoff" size. One of these settings would perform 16 KByte I/Os inline, and the other would preform 16 KByte I/Os direct.

We chose not to calculate the "TPC-H power" metric, instead we measured the runtimes for each of the query streams as well as the update thread. Runtimes of the streams and update thread varied significantly between runs. In order to make some sense of the measurements, Figure 6 shows the average runtime of the query streams across each run category. From this graph, the first observation you should make is that TPC-H with local I/O was faster than DAFS. In second place is DAFS with 16K-inline I/Os, and in last place is DAFS with 16K-direct I/Os. Its is not really surprising that local I/O is faster than DAFS since the DAFS server issues essentially the same file commands that the local disk version uses and doesn't need to send data through network. In addition, local disk I/O through Ultra3 SCSI was faster than ServerNet II, as was shown with *odmblast* in Section 5.1. The only advantage the DAFS case has over local I/O was that the overhead of accessing the NTFS filesystem was off-loaded to the server. However, Windows 2000 unbuffered I/O running over SCSI was extremely efficient, and the overhead was actually the same or less than the ServerNet II message processing overhead (although both were quite low).

To demonstrate the difference in overhead, we measured CPU utilization for the three cases. Figures 7 and 8 compare the CPU utilization for TPC-H over DAFS with the local disk case. We have smoothed these graphs (the line represents a running average of CPU utilization over a 20 minute period) in order to make them more readable. The first observation one can make is that the CPUs were not saturated in any of these cases, so there is likely additional performance tuning that could have been performed on the benchmark.

Referring to the local I/O and DAFS client curves, TPC-H CPU utilization reached a peak of about 80%, and was mostly between 40% and 70%. On average, CPU utilization was higher for the DAFS clients than for local I/O. This reflects the low CPU utilization of Windows I/O when running in "unbuffered" mode. The ODM client also uses relatively little CPU, but it does add some protocol processing overhead, which turned out to be greater than the Windows I/O overhead.

Comparing the 16K-inline vs. 16K-direct results, we see very similar client CPU utilization. In order to determine the relative impact of 16K I/Os, we enabled tracing in our DAFS server (the trace code was removed for all performance

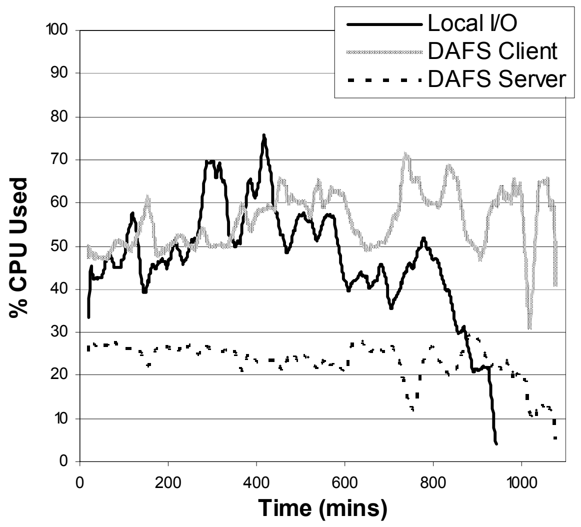

Figure 7. Comparison of TPC-H CPU Utilization (16K-direct I/Os)

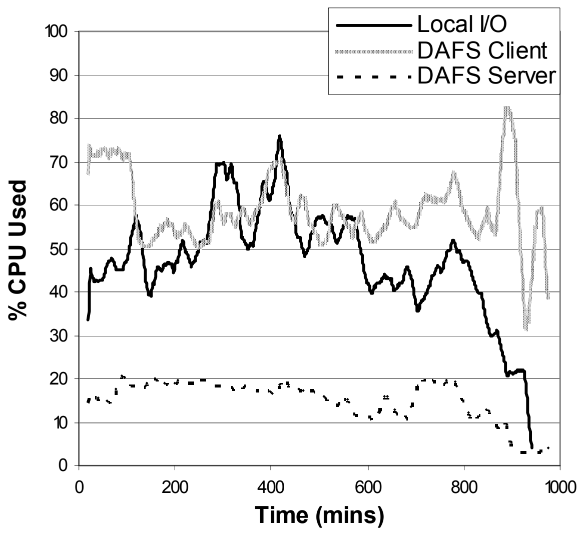

Figure 8. Comparison of TPC-H CPU Utilization (16K-inline I/Os)

measurements). From these traces we were able to measure the relative frequency of different I/O operations. Figure 9 shows the distribution of I/O types and sizes in TPC-H as seen by the DAFS server. As you can see, the vast majority of TPC-H's I/Os were 16 KByte reads. Therefore, if the client CPU utilization for both cases is similar, the additional overhead for client memory registration is similar to the overhead associated with copying data to pre-registered buffers (for 16 KByte reads).

Referring again to Figures 7 and 8, server-side direct I/O CPU utilization is on average about 5%-10% higher than for inline I/O. Recall from Section 5.2 that direct I/Os incur an additional overhead (T_{rdma}) due to the extra RDMA step necessary on the server side. The server-side RDMA overhead includes both network latency as well as CPU overhead for creating, executing, and completing the RDMA in the server (it does not include the portion of T_{rdma} resulting from memory registration on the client). These results indicate that the additional server-side CPU overhead associated with 16K-direct reads is actually greater than the server-side buffer copying overhead associated with 16K-inline reads.

The additional CPU utilization associated with small direct reads not only increased server CPU utilization, but it also

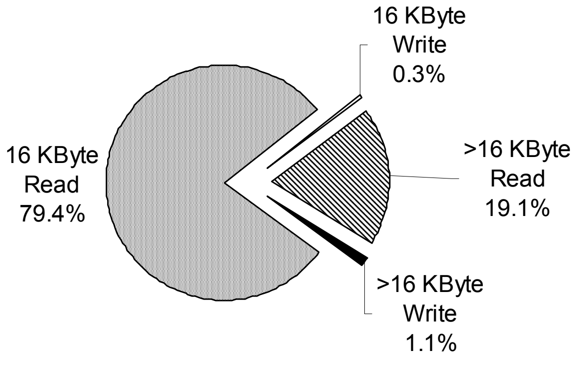

Figure 9. TPC-H I/O Operation Frequency

decreased overall throughput. This is due to the imperfect pipelining of I/O requests, which did not allow all of the latency to be overlapped. The lower throughput of 16K-direct I/Os, combined with the distribution of I/O operations (16 KByte reads were the dominant operation over all other operations by a factor of about 4 to 1), resulted in the 16K-inline case having better overall TPC-H performance.

6.2 TPC-H Summary

The Oracle TPC-H benchmark tests demonstrated that, for this system configuration, local I/O was still faster. This was due to several factors including inefficiencies in our DAFS/ODM code, limited ServerNet II bandwidth, and the efficiency of Windows unbuffered I/O. Performance with DAFS, however, was not much worse. Run-times with inline 16 KByte transfers were <10% worse than local disk, and with direct 16 KByte I/Os performance was <30% worse than local disk. These results are similar to the 22% TPC-C performance degradation Zhou demonstrated with wDSA [19] (wDSA is similarly designed transparent user-space DLL for performing I/O over VIA).

While the DAFS server results were worse than local disk, they do not reflect many of the advantages of DAFS over local disk. DAFS provides shared access for Oracle Real Application Clusters (RAC), and it provides system management flexibility because it virtualizes the disks. While there are other network filesystems like NFS or CIFS, none of these directly support Oracle. Oracle can also support cluster database access with shared Fiber Channel or iSCSI attached storage, but only with raw partitions that are harder to manage than DAFS. Also, while we did not have the opportunity to measure it for this work, Fiber Channel I/O typically performs slightly worse than direct attached Ultra3 SCSI due to its higher adapter to disk latency.

Finally, both this work and [19] indicate that there is a deficiency in user-level APIs that do not expose memory registration. Neither ODM or the win32 API (used for wDSA) can correctly cache memory registrations. This problem needs to be addressed either by improved operating system support or better user-space APIs. Until this occurs, user space I/O will not be able to reach its full potential.

7. CONCLUSIONS

In this paper we have evaluated the performance of a user-space DAFS server and ODM client. We have shown that our server and client achieve performance close to the limits of our ServerNet II system area network. Local SCSI I/O, however, was able to outperform DAFS due to the higher bandwidth of the local I/O buses in our test configuration. In addition, we have shown that for a non-trivial database application DAFS can get performance within 10% of local disk, while providing the advantages of a network-attached file system.

While our performance was good, there are several factors that limited our results. First, we observed that local disk performance that exceeded ServerNet II bandwidth, especially for reads. Unfortunately, our ServerNet II drivers did not support multiple NICs, so we had no way to address this (without switching to another network technology). Another limiting factor was the fact that Oracle would not pre-register memory for ODM. Therefore ODM had to register memory as part of every direct I/O operation. We would have also benefitted from a kernel-based DAFS server, since it would have eliminated the server-side system call overhead. Finally, we needed to tune our TPC-H benchmark better, either by adding more disks or changing some other configuration parameters, so that CPU utilization would be closer to 100% (i.e., so that the CPU would be the bottleneck, not I/O). With a different or better tuned benchmark, we might have observed some of the performance benefits provided by RDMA.

8. ACKNOWLEDGEMENTS

The authors would like to thank the members of HP's Industry Standard Servers (ISS) Storage group for their assistance in performing this work. In addition, we would like to thank Kostas Magoutis for his assistance in preparing the final version of this paper.

9. REFERENCES

[1] Bailey, S., *The Remote Direct Memory Access Protocol (iWarp)*, Internet Draft draft-bailey-roi-rdma-00, Internet Engineering Task Force, February 2002.

[2] Callaghan, B., *NFS Illustrated*, Addison Wesley Longman, Inc., 2000.

[3] Compaq, Intel, Microsoft, *Virtual Interface Architecture Specification*, December 1997, `http://www.intel.com/design/servers/vi/the_spec/specification.htm`.

[4] DAFS Collaborative, *DAFS: Direct Access File System Protocol, Version 1.00*, September 2001, `http://www.dafscollaborative.org`.

[5] DAFS Collaborative, *Direct Access File System Application Programming Interface (DAFS API), Version 1.00*, November 2001, `http://www.dafscollaborative.org`.

[6] DeBergalis, M., et. al., "The Direct Access File System," *FAST '03: 2nd USENIX Conference on File and Storage Technologies*, pp. 175-188, April 2003.

[7] Garcia, D., and Watson, W., "Servernet II," *2nd Parallel Computer Routing and Communication Workshop*, June 1997.

[8] Heirich, A., Garcia, D., Knowles, M., Horst, R., *ServerNet II: a Reliable Interconnect for Scalable High Performance Cluster Computing*, Tandem Labs Technical Report, September, 1998.

[9] Intel Corporation, *Intel Virtual Interface Architecture Developers Guide, version 1.0.*, September 1998, `http://developer.intel.com/design/servers/vi/developer/ ia_imp_guide.htm`.

[10] Infiniband Trade Association, *InfiniBand^{TM} Architecture Specification, Version 1.0*, October 2000.

[11] Magoutis, K., "Design And Implementation of a Direct Access File System (DAFS) Kernel Server for FreeBSD," *USENIX BSDCon 2002 Conference*, February 2002.

[12] Magoutis, K., et. al., "Structure and Performance of the Direct Access File System," *USENIX Annual Technical Conference,* Monterey, CA, June 2002.

[13] Message Passing Interface Forum, *MPI-2: Extensions to the Message-Passing Interface*, July 1997, `http://www.mpi-forum.org`.

[14] Oracle Corporation, *Oracle Disk Manager Interface and Functions*, March 2001.

[15] Oracle Corporation, *Oracle Disk Manager: an Oracle White Paper*, April 2001.

[16] Shepler, S., et. al. *NFS version 4 Protocol*, Internet Engineering Task Force RFC3010, December 2000

[17] Storage Networking Industry Association, *Common Internet File System (CIFS) Technical Reference, Revision 1.0*, SNIA Technical Proposal, March 2002, `http://www.snia.org/English/Collaterals/Work_Group_Docs/NAS/CIFS/CIFS_Technical_Reference.pdf`.

[18] Transaction Processing Performance Council (TPC), *TPC Benchmark H (Decision Support) Standard Specification Revision 1.4.0*, April 2002, `http://www.tpc.org/tpch/spec/tpch140.pdf`.

[19] Zhou, Y., et. al., "Experiences with VI Communication for Database Storage," *29th Annual International Symposium on Computer Architecture*, pp. 257-268, May 2002.