Multimedia Systems 9: 56–67 (2003)
Digital Object Identifier (DOI) 10.1007/s00530-003-0078-3

Multimedia Systems
© Springer-Verlag 2003

Improving the I/O performance of intermediate multimedia storage nodes

Pål Halvorsen, Thomas Plagemann, Vera Goebel

Department of Informatics, University of Oslo, Norway; e-mail: {paalh, plageman, goebel}@ifi.uio.no

Abstract. The data retrieval operations in servers and proxies for Media-on-Demand applications represent a severe bottleneck, because a potentially (very) high number of users concurrently retrieve data with high data rates. In the Intermediate Storage Node Concept (INSTANCE) project, we have developed a new architecture for Media-on-Demand storage nodes that maximizes the number of concurrent clients a single node can support. We avoid the traditional bottlenecks, like copy operations, multiple copies of the same data element in main memory, and checksum calculation in communication protocols, by designing, implementing, and tightly integrating three orthogonal techniques: a zero-copy-one-copy memory architecture, network level framing, and integrated error management. In this paper, we describe the INSTANCE storage node, and present an evaluation of our mechanisms. Our experimental performance results show that the integration of these three techniques in NetBSD at least doubles the number of concurrent clients that a single storage node can serve in our testbed.

Key words: I/O performance – Multimedia storage nodes – Operating system enhancements

1 Introduction

In the last few years, there has been a tremendous growth in the use of Internet services. In particular, the world-wide web and audio- and video streaming applications like News- and Video-on-Demand have become very popular. Thus, the number of users, as well as the amount of data each user downloads from servers in the Internet, is rapidly increasing. The use of multimedia data to represent information in a user-friendly way is one important reason for these two developments. Today, contemporary mid-price personal computers are capable of handling the load that such multimedia applications impose on the client system. However, the potentially (very) high number of concurrent users that download data from Media-on-Demand (MoD) servers represents a generic problem for this kind of client-server applications. To solve this problem, proxies are used to cache data closer to the client and offload MoD servers by serving clients directly from the proxy with the cached data. However, proxies are confronted with the same scalability problem as MoD servers: a successful caching strategy can lead to the situation that a high number of concurrent clients has to be served by a single proxy. Servers and proxies do not only share the scalability problem, they also use exactly the same operating system mechanisms to retrieve stored multimedia data and transmit it to receivers. Since servers and proxies are placed somewhere in the distribution chain between information provider and information consumer, we call them *intermediate storage nodes*, or simply *storage nodes*.

The scalability of storage nodes can be improved by three orthogonal approaches: (1) improve the I/O performance to maximize the number of concurrent clients a single storage node can support; (2) combine multiple single storage nodes in a server farm or cluster; and (3) use multiple single storage nodes in the distribution infrastructure as proxies. In the Intermediate Storage Node Concept (INSTANCE) project, we concentrate on the first approach developing a new architecture for single storage nodes. This architecture is optimized to support concurrent users in MoD-like applications retrieving multimedia data from the storage node. Thus, the task of reading data from disk and transmitting it through the network to remote clients with minimal overhead is our challenge and aim. In this context, we are currently interested in the maximal possible I/O improvements for concurrent retrieval. Optimized protocols for loading data from servers to proxies and caches are beyond the scope of this paper, and subject to ongoing research.

It was already recognized many years ago (1990) that *operating systems are not getting faster as fast as hardware* [1]. Until today, the gap in speedup of operating systems and hardware has not been reduced. Therefore, commodity operating systems represent a major performance bottleneck in MoD storage nodes. The crucial issues that contribute to this situation include copy operations, context switches, multiple copies of the same data element in main memory, and checksum calculation in communication protocols [2]. The key idea of INSTANCE is to avoid the above mentioned bottlenecks and improve the storage node performance by designing and tightly integrating the following three orthogonal techniques in a new operating system architecture:

- *Zero-copy-one-copy memory architecture*: memory copy operations have been identified as a bottleneck in high data rate systems. Several zero-copy architectures have been designed removing physical data copy operations to optimize resource usage and performance using shared memory, page remapping, or a combination of both [2]. These approaches reduce resource consumption of individual clients, but concurrent clients requesting the same data require each its own set of resources. Traditional broadcast or multicast is an easy way of dealing with per client resource allocation, but start-up delay might be a problem. To minimize the start-up delay, a couple of broadcast partitioning schemes have been proposed [2]. Still, zero-copy and delay-minimized broadcasting only reduce the per-data or per-client resource usage, respectively. To optimize both, we integrate both mechanisms to have no physical in-memory copy operations and to have only one shared copy of each data element in memory.
- *Network level framing (NLF) mechanism*: each time a client retrieves data from a storage node, the data is processed through the communication system protocols executing the same operations on the same data element several times, i.e., once for each client. To reduce this workload, we regard the server or the proxy as an *intermediate node* in the network, where only the lower layers of the protocol stack are processed. When new data is sent to the storage node to be stored on disk, only the two lowest protocol layers in the TCP/IP protocol suite are handled, and the resulting transport protocol packet is stored on disk. When data is requested by remote clients, the transport level packet is retrieved from disk, the destination IP address and port numbers are filled in, and the checksum is updated by adding the checksum value of the new addresses. Thus, the end-to-end protocols that perform the most costly operations in the communication system, especially the transport level checksum, are almost completely eliminated.
- *Integrated error management scheme*: when transferring data from disk in a storage node through the network to remote clients, the correctness of the transferred data is checked multiple times, i.e., there is redundant error management functionality. In INSTANCE, we integrate the error management in a disk array and a forward error correction (FEC) scheme in the communication system. Contrary to the traditional data read from a redundant array of inexpensive disks (RAID) system, where the parity information is only read when a disk error occurs, we retrieve also the redundant error recovery data (also called parity data). All data is passed over to the communication system, where the parity information from the RAID system is reused as FEC information over the original data. Thus, by using the same parity information in both subsystems, the FEC encoder can be removed from the storage node communication system, and both memory and CPU resources are made available for other tasks.

Zero-copy solutions have been studied before, and there is some work related to our NLF idea (see Sect. 3.5). However, our work presents several unique contributions: (1) the integrated error management concept or similar ideas have not been reported by others; (2) there are no publications that describe the integration of zero-copy and NLF, or of all three mechanisms; and (3) there are no research results reported that come close to the server performance improvement we have demonstrated in the INSTANCE project.

In earlier publications [3,4], we have individually described design considerations, as well as some early performance evaluations of the three mechanisms. It is the goal of this paper to present design, implementation, and an in-depth performance evaluation of the integrated system. The rest of the paper is organized as follows. In Sect. 2, we describe the design and implementation of our system. Section 3 presents the results of our performance experiments. We summarize and conclude the paper in Sect. 4.

2 Design, implementation, and integration

The basic idea of the INSTANCE project is to create a fast in-kernel data path for the common case operation in a storage node, i.e., retrieval of data from the storage system and sending it to receiver(s). Consequently, we remove as many instructions as possible per read and send operation. In this section, we describe the design, implementation, and integration of our mechanisms tuning the I/O performance in a storage node.

2.1 Zero-copy-one-copy memory architecture

The term *zero-copy-one-copy* refers to a combination of a zero-copy implementation of an in-kernel data path with a broadcasting scheme that serves multiple clients with one in-memory copy. This section describes our memory architecture.

2.1.1 In-kernel data path

The basic idea of our zero-copy data path is shown in Fig. 1. The application process is removed from the data path, and in-kernel copy operations between different sub-systems are eliminated by sharing a memory region for data. The file system sets the `b_data` pointer in the `buf` structure to the shared memory region. This structure is sent to the disk driver, and the data is written into the `buf.b_data` memory area. Furthermore, the communication system also sets the `m_data` pointer in the `mbuf` structure to the shared memory region, and when the `mbuf` is transferred to the network driver, the driver copies the data from the `mbuf.m_data` address to the transmit ring of the network card.

To implement this design, we have analyzed several proposed zero-copy data path designs [2]. In INSTANCE, we have a specialized system and do not need to support all general operating system operations. Therefore, we have used a mechanism that removes the application process from the data path, and transfers data between kernel subsystems without copying data (for example, see work from Buddhikot [5] and Fall and Pasquale [6]). As a starting point, we have chosen to use the MMBUF mechanism [5], because of its clean design and reported performance. We have made the following modifications in order to support our requirements and system components, including a later version of NetBSD, and

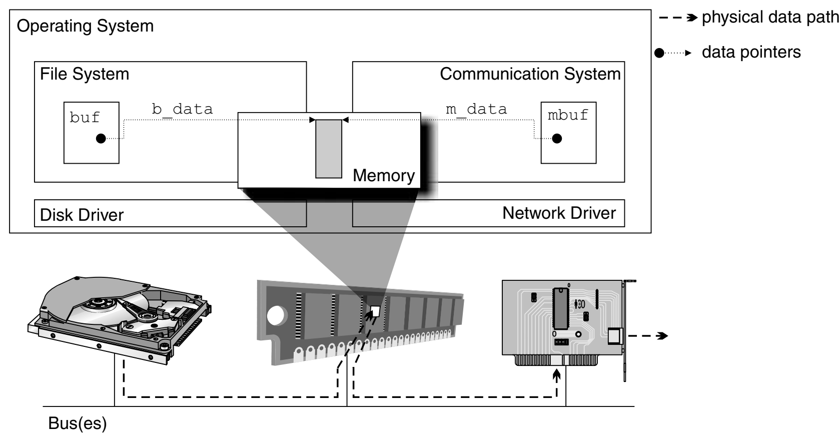


Fig. 1. Basic idea

to further increase the performance: First, the allocation and deallocation of MMBUFs have been changed to use the pool mechanism in NetBSD to reduce the time used to allocate and free a buffer item. On a PentiumIII 933 MHz machine, `malloc()` and `free()` (which is used in the original design) used 5.80 µs and 6.48 µs, respectively. Using the pool mechanism, these times are reduced to 0.15 µs for both allocating and deallocating the MMBUF. Second, the network send routine is changed to allow UDP/IP processing, because the native MMBUF mechanism used ATM. Third, the native stream operation blocks if data from previous operations is not yet transmitted onto the network before issuing a new operation of the current chain. Instead, a new memory block is allocated for the chain regardless whether the data is transmitted or not. Therefore, we do not issue a wait on the chain, and the possible blocking delay is removed.

2.1.2 Broadcasting schemes

The zero-copy data path eliminates the in-memory copy operations in our storage node, but each client that is served by the storage node still requires its own set of resources in the storage node. For each client request, the data for the respective client has to be stored in the virtual address space of the serving process. This means that physical memory is allocated and mapped into the virtual address space of the process according to available resources and a global or local allocation scheme. This approach is also called user-centered allocation, because each client has its own share of the resources. However, traditional memory allocation on a per-client basis suffers from a linear increase of required memory with the number of clients.

To better use the available memory, several systems use so-called data-centered allocation, where memory is allocated to data objects rather than to a single process. In our storage node we use a periodic service like pyramid broadcasting [7], or derivatives [8,9,10]. The data is split into partitions of growing size, because the consumption rate of one partition is assumed to be lower than the downloading rate of the subsequent partition. Each partition is then broadcast in short intervals on separate channels. A client does not send a request to the storage node, but instead it tunes into the channel transmitting the required data. The data is cached on the receiver side, and during the playout of a partition, the next partition is downloaded. Performance evaluations show that the data-centered allocation schemes scale much better with higher numbers of users, compared to user-centered allocation [7]. The total buffer space required is reduced, and the average response time is minimized by using a small partition size at the beginning of a movie.

2.1.3 Integration of data path and broadcasting scheme

The integration of the zero-copy data path and the delay-minimized broadcasting scheme is depicted in Fig. 2. The data file is split into partitions of growing size. Each of these partitions are transmitted continuously in rounds using the zero-copy data path (see Fig. 1), where data is sent to the network using UDP and IP multicast. Thus, our memory architecture removes all in-memory copy operations, making a zero-copy data path from disk to network, i.e., the resource requirement per-data element is minimized. Additionally, this mechanism reduces the per-client memory usage by applying a delay-minimized broadcasting scheme running on top of the in-kernel data path.

2.2 Network level framing

Each time data is retrieved from a storage node, it is processed through the communication system protocols executing the same operations on the same data element several times (i.e., an identical sequence of packets, differing only in the destination IP address and the port number fields, is created each time). Most of the time consumed is due to reformatting data into network packets and calculating the checksum. Since the common operation in a storage node is to retrieve data from the storage system and send it to receivers, we regard the transport layer packet processing as unnecessary overhead and process only the lower layers of the protocol stack like in an *intermediate network node*. When new data is uploaded to a storage node for disk storage, only the lowest two protocol layers of the TCP/IP protocol suite are handled, and the resulting transport protocol packets are stored on disk. When data is requested by remote clients, the transport level packets are retrieved from disk, the destination port number and IP address are filled

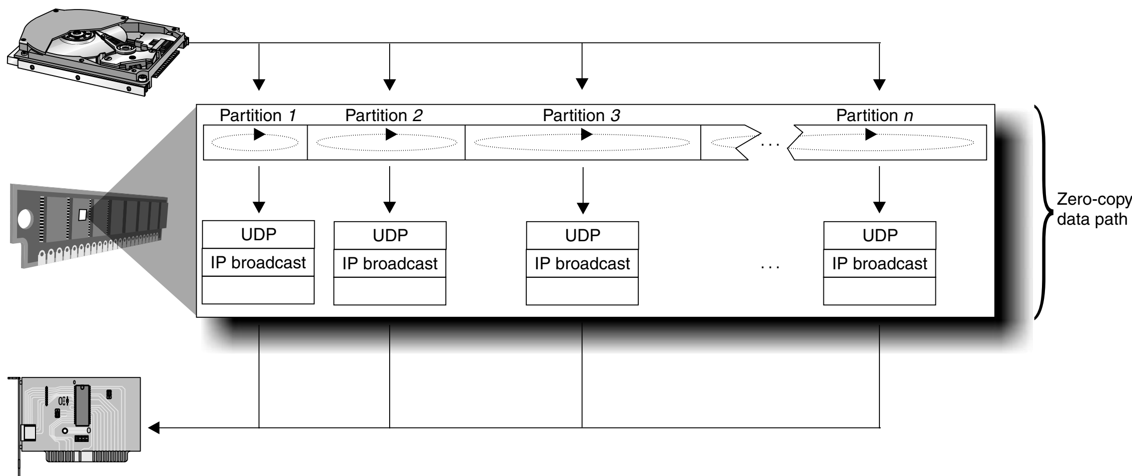


Fig. 2. The Zero-copy-one-copy memory architecture

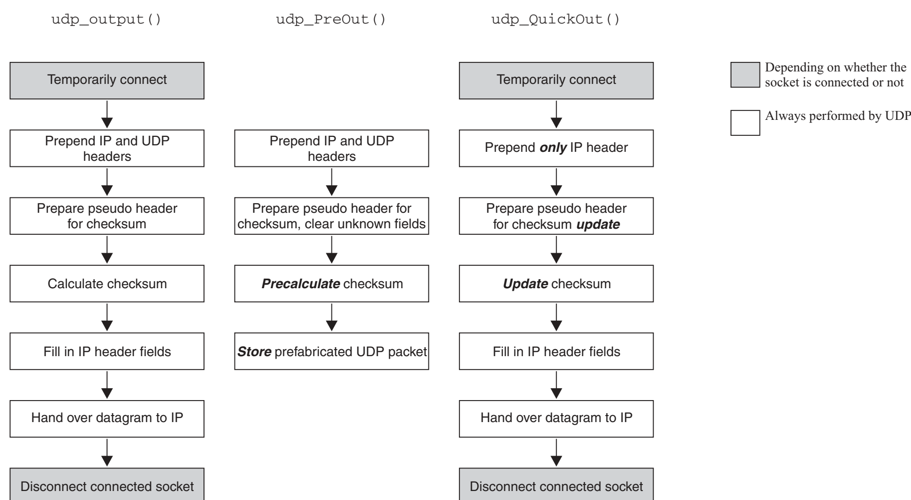


Fig. 3. Splitting the UDP protocol

in, and the checksum is updated over the new header destination fields. Thus, the overhead of the end-to-end protocol which performs the most costly operations in the communication system, especially the transport level checksum, is almost completely eliminated.

In the current implementation of our NLF prototype, we use UDP as transport protocol, because it is suited for multicast and real-time environments [11], i.e., it is fast and does not suffer from large delay variations due to retransmissions. However, as incoming packets will have invalid headers, meaning that the port numbers and IP addresses are incorrect, we cannot put these packets directly on disk. Furthermore, to assure that the storage node receives all the data, a reliable protocol like TCP or hybrid ARQ [12] must be used to upload data to a storage node. In order to prefabricate UDP packets and to store them on disk, we split the protocol, which is implemented by the NetBSD `udp_output()` procedure, in two parts (Fig. 3). The first part, called `udp_PreOut()`, is executed "off-line" during the data upload operation. It calculates the checksum over the packet payload and stores the checksum in a meta-data file. The packet header itself is not generated or stored in the current prototype. The second part of our UDP protocol, named `udp_QuickOut()`, is performed at transmission time and generates the packet header, adds the checksum over the header to the stored checksum, which is prefetched into memory at stream initialization time, and transmits the packet via IP to the network (please refer to [3] for more details).

2.3 Integrated error management

In INSTANCE, we look at a storage node using a RAID system for data storage, and to move the CPU intensive data recovery operations in the communication system to the clients, we use FEC for network error correction. Thus, when transferring data from disk in a storage node through the communication system to remote clients, the correctness of the information is normally checked multiple times, i.e., when data is retrieved from the disk array and when data arrives at the client side. The FEC encoding operation is a CPU intensive operation and represents a severe bottleneck in the storage node when transmitting data to a large amount of concurrent users. To remove this bottleneck, we integrate the error management in the storage system and the communication system. This is achieved

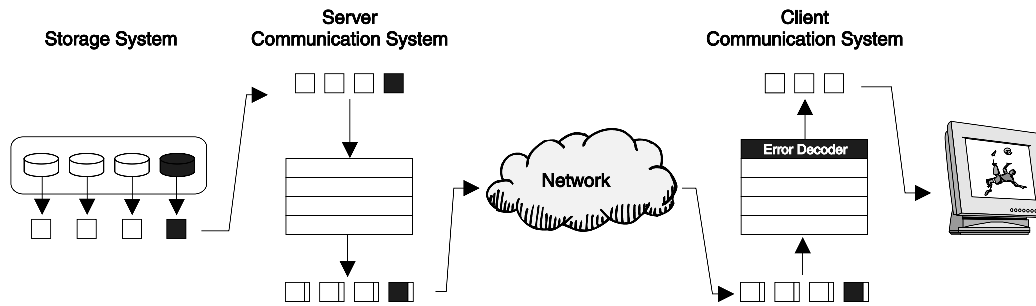


Fig. 4. Integrated error management

by reusing the storage system parity data as FEC recovery data in the communication system, as shown in Fig. 4[1]. When uploading data to the storage node, we write data to the disk array in a manner similar to RAID level 4, i.e., having one parity disk, but replace the traditional RAID XOR parity calculation by a correcting code that can be used by both storage and communication system. A read operation retrieves both application level data and the new RAID parity data. All this data is transferred to the communication system where the parity data is used in the FEC decoder at the client side.

However, the storage system and the communication system have different error models, and it is a challenge to find a code that recovers from all kinds of errors and at the same time is efficient enough to enable high data rate multimedia streams. In the storage system, a disk block may be damaged, and in the network, bit errors occur due to noise, and packets are lost because of congestion. To create an error model for the network, we have studied several loss statistics from measurements done by others [4].

We have evaluated different codes to see if an integrated error management scheme could be designed. We performed several tests measuring coding performance using different code configurations. Our analysis shows that the Cauchy-based Reed Solomon Erasure code [13], which is able to correct known errors (erasures), has the necessary properties and can achieve adequate performance. Therefore, the storage system in our prototype stores parity data that is generated with this code. When transmitting data to a remote client, the storage system parity data is reused for FEC parity data reconstructing lost packets. Bit errors, for example introduced by noise in the network, are detected at the client side with the help of the transport level (UDP) checksum. If a packet is corrupted, i.e., one or more bits are damaged, it is detected by the checksum algorithm, and the packet is marked as corrupted. Afterwards, the FEC scheme is applied to reconstruct both lost and damaged packets using the storage system parity data.

We decided to configure our prototype such that it is able to correct maximum 12.5% lost or corrupted data packets. For an eight disk RAID system, we tailor the size of the codeword, which is the amount of data subject to reconstruction, to contain 256 symbols, i.e., 224 symbols for application data and 32 symbols for parity data. A symbol is a single data element within the codeword. The symbol size is a configurable parameter, and each symbol is transmitted as a single packet, i.e., packet size equals symbol size (for details, see Halvorsen et al. [4]).

[1] The idea of saving CPU cycles this way does not require reuse of storage system parity data. FEC computations can be done offline and stored as briefly suggested by Rizzo [14], but the integrated error management scheme additionally saves disk bandwidth and storage space as the disk array already provides parity data.

2.4 Integration

Our three mechanisms can be used separately or tightly integrated. In order to prove our assumption, that the total system performance gain integrating the three techniques is approximately the sum of each individual gain, we have designed and implemented an integrated version. However, a tight integration requires that the three mechanisms are adapted to each other. The purpose of this subsection is to outline these necessary adaptions.

For the integrated error management scheme, we must find a symbol size that is appropriate for the disk blocks to be able to recover from disk errors. At the same time, a symbol is transmitted as a single packet in the communication system, and the size should be suitable for the used network. As described later, we have chosen and tested symbol sizes of 1 KB, 2 KB, 4 KB, and 8 KB.

In our integrated server design, the data stored on our storage node is processed through a “prefabrication filter” before put on disk. This generates FEC information for the integrated error management scheme and packet payload checksums for NLF. Thus, these operations are not performed on-the-fly during transmission. An important integration issue is the order of the operations, i.e., parity packets should also have precalculated checksums, and must be generated before processing the data through the NLF filter.

At transmission time, the server uses a periodic service like the cautious harmonic broadcasting protocol [10], reducing the number of streams transmitted for the most popular (prefabricated) data elements. The in-kernel zero-copy data path, integrated with the NLF “lightweight” version of the UDP protocol, is used during data retrieval to minimize the overhead of moving both application level data and FEC data from the disk to the network card. Finally, any lost or corrupted data is restored at the client side using the FEC information in an error correcting layer in the client application.

3 Performance evaluation

We have performed several tests to monitor the performance of different data transmissions using a *Dell Precision WorkStation 620* with an Intel 840i chipset [15] with two PCI

busses (32 bit, 33 MHz and 64 bit, 66 MHz), a PentiumIII 933 MHz processor, 32 KB first-level cache, 256 KB second-level cache, and 256 MB RDRAM with a theoretical bandwidth (measured bandwidth is reported in detail in the thesis from Halvorsen [16]) of 25.6 gigabit-per-second (Gbps), i.e., dual 12.8 Gbps memory controller interface and 25.6 Gbps system bus. This machine has only one 9.1 GB, 10,000 rounds-per-minute SCSI hard disk. Therefore, we simulated a RAID system by storing both application level data and parity data on the same disk. We connected this machine to another PC using an isolated, point-to-point gigabit Ethernet network. To measure time in kernel and user space, we have implemented a software probe using the Intel `RDTSC` instruction. This instruction reads the processor cycle count giving a nano-second granularity [17]. To avoid issues affecting the measured cycle count like out-of-order execution, we used the `CPUID` instruction in the software probe forcing every preceding instruction in the code to complete before allowing the program to continue. The overhead of executing this software probe comprises *206 cycles* (0.22 µs) running inside the kernel and *273 cycles* (0.29 µs) in user space. In the performance results presented in the following subsections, this overhead is already subtracted.

3.1 Memory management

The broadcasting scheme enables several clients to share a single data element in memory, i.e., we are able to support a lot of users, but still only a limited amount of concurrent streams. To determine the maximum amount of concurrent streams that a single storage node can support, we have analyzed the performance of the zero-copy data path by measuring the time transmitting a 1 GB file from storage system to the network. Our first observation is that a single top-end disk in our test system is too slow compared to the rest of the system to see the benefits of the zero-copy data path. With this disk we can only achieve a maximum throughput of 205 megabits-per-second (Mbps), regardless of whether copy operations are used. Therefore, we used a memory file system to perform the same measurements, which means the data retrieval times are reduced from a disk access time to the time to perform a memory copy operation. Since we cannot store a 1 GB file in main memory on our test machine, we read instead a 28662512 B file 38 times (1.015 GB) in a tight loop. The average results are plotted in Fig. 5. If there is no load on the system (Fig. 5a), the time to transmit the data through the storage node is reduced by 45.81% (1 KB packets), 45.04% (2 KB packets), 49.33% (4 KB packets), and 52.70% (8 KB packets). Furthermore, if we have a highly loaded[2] machine (Fig. 5b), the respective times are reduced by 70.73% (1 KB packets), 71.49% (2 KB packets), 71.33% (4 KB packets), and 73.43% (8 KB packets). This shows that a zero-copy data path also scales much better than the traditional mechanisms, because less resources, like memory and CPU, are used. The performance of the zero-copy mechanism degrades from low system load to high system load by factors 6.30 (1 KB packets), 5.90 (2 KB packets), 6.20 (4 KB packets), and 6.08 (8 KB packets) whereas the performance of the traditional storage node is reduced by factors 11.67 (1 KB packets), 11.37 (2 KB packets), 10.96 (4 KB packets), and 10.83 (8 KB packets).

Since some of these values indicate a throughput close to 1 Gbps when reading data from the memory file system, we checked if the network card could be a bottleneck. Our tests show that packets sent to the network card's transmit ring are occasionally put back to the driver queue meaning that the 1 Gbps network card is a bottleneck in our implementation.

The amount of time spent in the kernel is significantly reduced when removing the copy operations. The total time is reduced in average by 34.27% (1 KB packets), 36.95% (2 KB packets), 39.88% (4 KB packets), and 36.10% (8 KB packets).

Additionally, we have looked at the performance when transmitting several concurrent streams using a packet size of 2 KB[3]. Each stream transmitted 1 GB data from a memory file system[4] through the communication system to the network, and we measured the transmission completion time for each individual stream. During these tests the memory file system consumed approximately 35 % of the CPU resources for all tests. Our results show that our zero-copy mechanism performs better than the native NetBSD mechanisms. Figure 6a presents the difference in time to transmit the data per stream, and this time increases faster using traditional system calls compared to our stream mechanism. In the *Zero-Copy 1* measurement, we have used separate *stream* read and send system calls. The test *Zero-Copy 2* represents a scenario where the system call overhead is reduced by merging the read and send operations into one single system call. Figure 6b shows the per client throughput, and the *Zero-Copy 2* test indicates a throughput improvement per stream of at least a factor two (about 2.5). The number of concurrent clients may therefore be increased similarly[5]. Compared to the single-user scenario (Fig. 5), the total server performance decreases slightly due to context switch overhead.

3.2 Network level framing

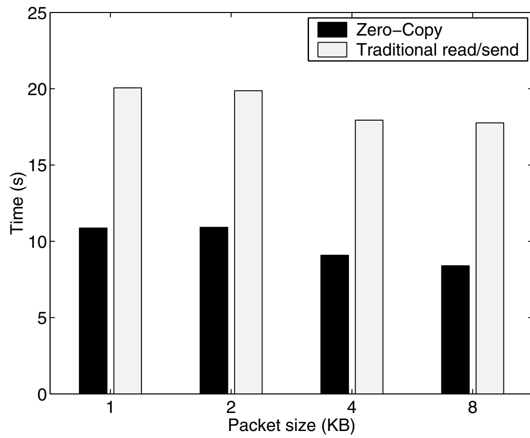
To see the performance gain of using NLF, we have measured the time spent in the traditional UDP protocol and the respective time spent in our protocol which used prefabricated checksum over the packet payload. We transmitted a large data file of 225 MB via the zero-copy data path with packet sizes of 1 KB, 2 KB, 4 KB, and 8 KB. The results are shown in Fig. 7. The first observation is that small packets require more time to transmit the whole file, because more packets have to be generated and especially more header checksums have

[2] A workload of 10 CPU intensive processes, performing float and memory copy operations, were added, i.e., the CPU was available for about 9% (time for context switches not included) of the time to serve client requests.

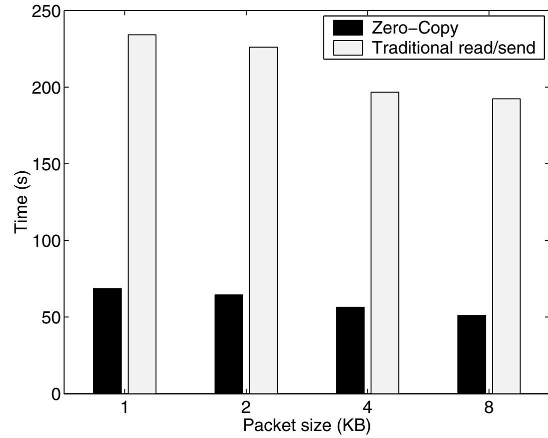
[3] Due to time restrictions, we could only test one packet size, and 2 KB is randomly chosen from the supported packet sizes.

[4] As memory is a scarce resource, several streams had to read the same file, i.e., streams using traditional system calls could have a caching effect on the buffer cache which will not be present in our storage node broadcasting data. To have equal conditions for all tests, we removed this caching effect by reducing the buffer cache size from 13,168 KB to 200 KB.

[5] Figure 6b indicates an increase of almost a factor three, e.g., about 12 (6) Mbps can be transmitted to 25 (50) concurrent clients using native NetBSD mechanisms whereas *Zero-Copy 2* can manage about 75 (150) clients.

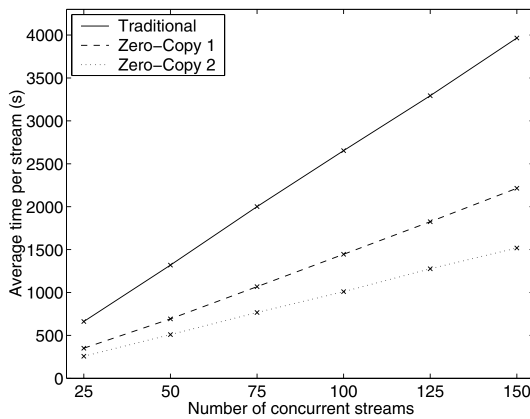


a No background load.

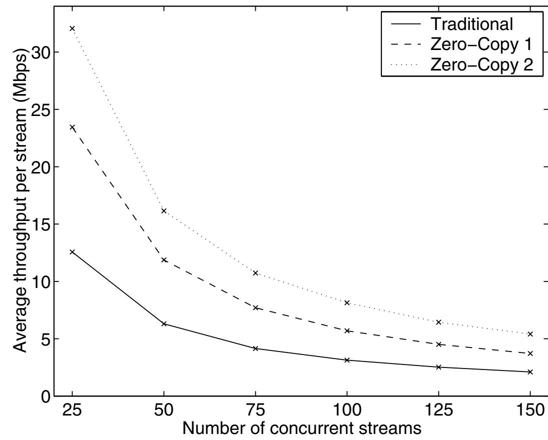


b High CPU background load.

Fig. 5. Average time to transmit a 1 GB file from a memory file system

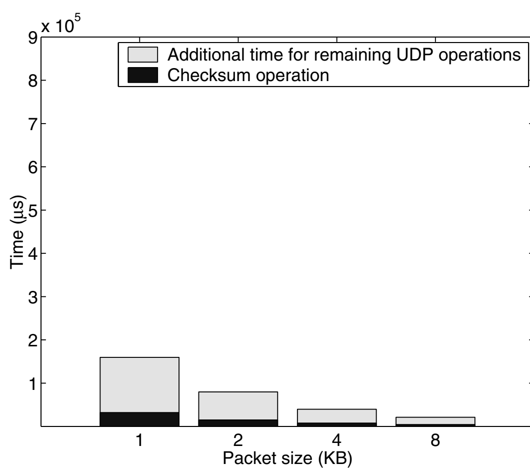


a Time.

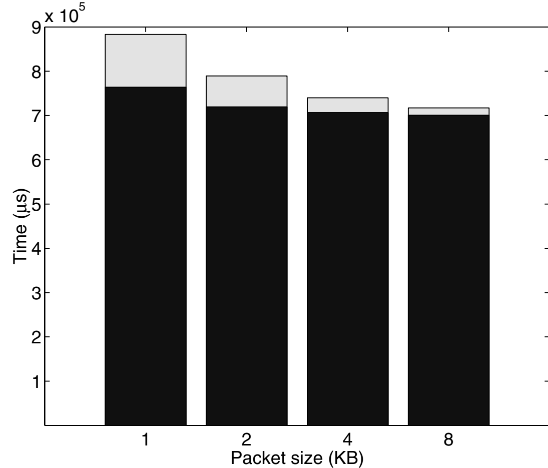


b Throughput.

Fig. 6. Average per stream performance transmitting a 1 GB file from a memory file system



a UDP with NLF.



b Traditional UDP.

Fig. 7. Accumulated UDP protocol execution time versus total time spent on checksum operation

to be calculated. Moreover, as we can see from the figures, our version of the UDP protocol is faster than the traditional protocol. In average, the time to execute the checksum procedure is reduced by 95.98% (1 KB packets), 97.87% (2 KB packets), 98.91% (4 KB packets), and 99.45% (8 KB packets) when using NLF. Furthermore, as we only compute the checksum over the packet headers, the overhead of processing the packet through the UDP protocol is nearly constant using

NLF. Using the traditional protocol, we see that this overhead depends on the packet payload and varies more, because it is more vulnerable to context switches. Finally, the amount of effective time used in the operating system kernel by the storage node is also greatly reduced using NLF compared to using the traditional UDP protocol, i.e., in average by 51.32% (1 KB packets), 54.07% (2 KB packets), 61.16% (4 KB packets), and 61.66% (8 KB packets).

3.3 Integrated error management

The codeword symbol size (which is equal to the packet size) has impact on the decoding performance. Decoding throughput and additional introduced start-up delay due to decoding depends on the codeword configuration, the symbol size, and the available CPU resources. To find suitable symbol size values, we tested the Cauchy-based Reed Solomon Erasure code with symbol sizes ranging from 32 B to 64 KB using the codeword configuration described in Sect. 2.3 in a worst case loss scenario. This scenario is defined by the maximum loss that can be repaired, i.e., losing 12.5% of the packets, and a 167 MHz Sun UltraSparc1 machine simulating a low performance client.

Figure 8a shows that the throughput increases with larger symbol sizes. If we use symbol sizes equal or larger than 1 KB, the client system will be able to decode streams with data rates of 6 Mbps and more on this system. However, increasing throughput by using larger symbol sizes also increases the worst case start up delay (see Fig. 8b), because there is more data to decode per codeword. Therefore, to reduce the experienced start-up delay, we do not use symbol sizes larger than 8 KB. This results in a (worst-case) decoding delay of two seconds or less on the Sun UltraSparc 1.

To further analyze the performance of our prototype, we have designed a simple benchmark application in which we transfer a 225 MB file between the server process and the client process. We performed a worst-case decoding performance test introducing a maximum amount of errors within the code's correcting limit on several different machines using symbol (packet) sizes of 1 KB, 2 KB, 4KB, and 8 KB.

3.3.1 Client side

By reconstructing the lost data at the client side instead of waiting for a retransmission, the clients experience a better data presentation at the cost of increased memory and CPU usage. Our measurements show that our code is able to decode the data in time for a high data rate presentation. The average decoding performance on our different machines is displayed in Table 1. Most of our experiments show that a standard MPEG-2 Digital Versatile Disk (DVD) video data stream (average bit rate of 3.5 Mbps and maximum bit rate of 9.8 Mbps [30]) with a maximum data loss (of 12.5%) can be recovered in time. Furthermore, there are no large performance differences varying the symbol size between 1 KB and 8 KB, and all these configurations are adequate regarding throughput. However, our tests indicate some minimal hardware requirement, depending on the supported data rate, because the Tech Pentium (166 MHz) machine is unable to support a higher data rate than about 3 Mbps.

Table 1 also shows the experienced start-up decoding delays (time to decode the first codeword) in our experiment. Due to the decoding cost, the client might experience an increased start-up delay ranging from about 0.1–4.5 seconds (if a maximum amount of repairable errors occur), depending on the processor speed and the used block size. The delay increases with the size of the symbol, and since there are no large differences in throughput for the evaluated schemes on the same machine, a symbol size of 1 KB or 2 KB is appropriate.

Finally, as there usually is some variance in the accessibility of the processor (unless some kind of reservation-based scheduling is provided), and thus in the decoding throughput, some client-side buffering should be provided. Nevertheless, despite all the overhead introduced on the client side, the recovery of a 3.5 Mbps video stream can be made in time, with exception of the Intel Pentium (166 MHz) machine, assuming client hardware similar to the machines we used in our experiments.

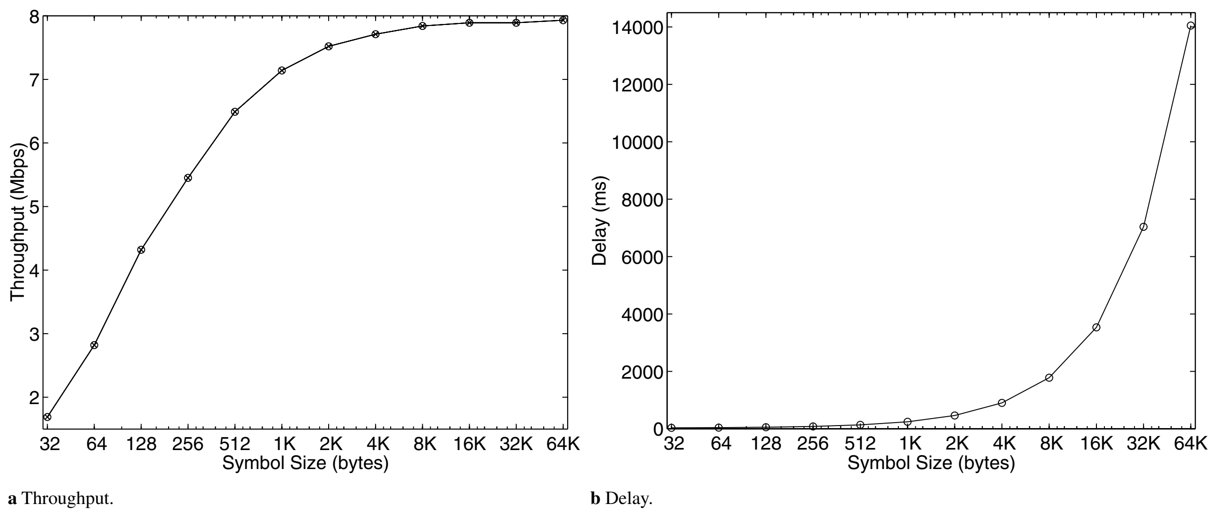
3.3.2 Server side

The server side performance gain, integrating the error management mechanisms and reading the parity data from disk, is substantial. Storing and retrieving parity data from disk requires no extra storage space compared to traditional RAID systems, because one disk is already allocated for parity data. Furthermore, it requires no extra time for data retrieval, because the recovery disk is read in parallel with the original application data. We have no overhead managing retransmissions, and the usage of the parity data from the RAID system as FEC recovery data offloads the storage node from the encoding operation resulting in severe performance improvements. A PC with a PentiumIII 933 MHz CPU is capable of encoding data at a maximum throughput of 24.34 Mbps (1 KB packets), 22.07 Mbps (2 KB packets), 22.67 Mbps (4 KB packets), 22.94 Mbps (8 KB packets) using the Cauchy-based Reed Solomon Erasure code. Without this encoding operation, the same system can achieve a throughput of 1 Gbps (see Sect. 3.1). The only overhead on the storage node side in our approach is 12.5% of increased buffer space and bandwidth requirement to hold the redundant data in memory and transmit it over the buses and the network. Thus, using our integrated error management mechanism, the storage node workload is greatly reduced compared to traditional FEC schemes, and the clients experience a smoother data playout compared to ARQ based schemes, because no latency is introduced due to retransmissions.

3.4 Integrated server performance

The INSTANCE enhancements are orthogonal, and as shown in the previous sections, each mechanism increases performance in a particular part of the system. In this section, we show that our assumption in Sect. 2.4 holds quite well, i.e., the performance gains of a tightly integrated server increase approximately equal to the sum of each individual gain.

The integrated error management removes the 24 Mbps (1 KB packets) encoding bottleneck on our test machine at the cost of 12.5% extra required storage space and bandwidth



a Throughput. **b** Delay.

Fig. 8. Cauchy-based Reed Solomon erasure decoding

Table 1. Average decoding throughput (Mbps) and start-up delay (seconds) varying the symbol (packet) size

	Throughput (Mbps)				Delay (s)			
Machine	**1 KB**	**2 KB**	**4 KB**	**8 KB**	**1 KB**	**2 KB**	**4 KB**	**8 KB**
Sun UltraSparc 1, 167 MHz, Solaris 2.6	6.02	6.51	6.36	6.20	0.30	0.56	1.16	2.45
Sun Ultra 4, 300 MHz, Solaris 2.6	9.99	10.71	10.55	10.49	0.18	0.33	0.69	1.47
Dell PowerEdge 6300 500 MHz, Red Hat Linux 6.1	8.16	7.43	6.52	6.47	0.14	0.49	1.11	2.25
Dell Inspiron 7000, 400 MHz, Red Hat Linux 6.1	10.03	9.27	9.48	9.59	0.18	0.40	0.82	1.54
Cinet PPI-600, 350 MHz, Red Hat Linux 6.1	8.96	9.22	9.05	8.89	0.20	0.39	0.85	1.63
Tech Pentium, 166 MHz, Red Hat Linux 6.1	3.18	3.32	3.34	3.25	0.58	1.11	2.16	4.50
Tech AMD K7, 700MHz, Red Hat Linux 6.1	17.10	17.40	16.67	16.53	0.11	0.21	0.44	0.95
Cinet PPI-600, 350 MHz, NetBSD 1.4.1	10.35	10.71	10.37	10.12	0.18	0.34	0.71	1.45
Dell Inspiron 7000, 400 MHz, NetBSD 1.4.2	10.62	10.05	10.52	10.66	0.17	0.36	0.70	1.38
Cinet PPI-600, 350 MHz, OpenBSD 2.6	14.87	16.28	14.85	14.22	0.12	0.22	0.49	1.04
Dell Precision 620, 933 MHz, NetBSD 1.5ALPHA2	23.03	20.67	20.80	20.75	0.08	0.18	0.35	0.71

using the current configuration. The encoding bottleneck and the retransmission overhead are removed, which means that an effective throughput of about 875 Mbps could be achieved as the zero-copy measurements show a total throughput close to 1 Gbps.

Figure 9 shows the gain in used CPU time in the kernel (Fig. 5 shows corresponding completion time) running a server without FEC. We see that using the zero-copy data path together with NLF reduces the used CPU time by 66.18% (1 KB packets), 70.37% (2 KB packets), 75.95% (4 KB packets), and 75.25% (8 KB packets) compared to the traditional data path. In total, a traditional server used approximately 10 seconds on the processor to transmit the whole data file. If the server should also perform FEC encoding, an additional overhead of 346.92 seconds (1 KB packets), 388.54 seconds (2 KB packets), 380.50 seconds (4 KB packets), and 375.45 seconds (8 KB packets) must be added using the traditional data path. Additionally, the total time used by the system includes the time spent executing instructions in the server application, but as these times are below 0.5 seconds, regardless of which server we use, we have not added this overhead into the figure.

A broadcast protocol for periodic services also improves the efficiency of our MoD server. Such schemes reduce the server bandwidth requirement at the cost of increased client workload and resource requirements. For example, using a cautious harmonic broadcasting protocol [10], the total server bandwidth requirement in a News-on-Demand (NoD) scenario, i.e., transmitting 3.5 Mbps video streams to n concurrent clients (a bandwidth requirement of $n \times 3.5$ Mbps for each video clip using unicast), can be reduced to 16.26 Mbps [16] per video clip, regardless of the number of concurrent clients

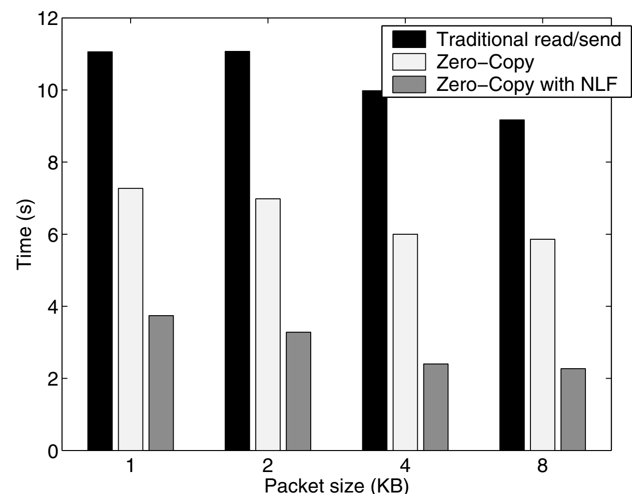


Fig. 9. Histogram over used CPU time in the kernel

assuming a maximum startup latency of five seconds and a video-clip length of three minutes. However, such a broadcasting scheme increases the client reception rate in that the client receives data from several segments of the data file concurrently (but not the consumption rate). If we perform error correction when the data arrives, the broadcasting scheme influences the integrated error management scheme, i.e., error correction must be performed on several file segments at the same time. If loss recovery is performed as the data is played out, the scheme does not influence the integrated error management mechanism except that additional storage space is needed for parity data. Independent of when the data is decoded, all machines with reasonable performance are capable of decoding multiple streams from the broadcast protocol, e.g., our NoD example requires a 16.26 Mbps reception rate and some of the tests indicate that even a 350 MHz machine can decode (error correct) these streams.

Conclusively, the gain of our bottleneck removals in our integrated server implementation approximately adds up as assumed in Sect. 2.4. The integrated error management and the periodic broadcasting scheme do not influence the performance of other mechanisms, but remove the FEC encoding operation and share data between concurrent users, respectively. The gain of removing copy operations is a about 40% (see Sect. 3.1 and [16]), and the removal of checksum operations should further decrease processing cost of the communication system, which comprises most of our remaining overhead, by about 60% [18]. Our measurements (see Fig. 9) indicate similar numbers and thus verifies the total gain assumption.

3.5 Comparison with related work

The research area of improving system performance for MoD applications, especially memory management and communication protocol processing, has received much attention, and there are a lot of existing research results. However, our approach differs from other works in that we integrate several mechanisms and look at the total server performance gain.

We have presented in [16,2] several copy-free data paths and compared several different broadcasting schemes. Nevertheless, for our kind of application scenarios, there are no other mechanisms that perform better (maybe equally good), because we have designed a tailored streaming system and have optimized the code to meet our needs. For example, our system performs better compared to more general-purpose mechanisms using page remapping or transfers of access rights. Furthermore, to the authors' knowledge, there are no other mechanisms trying to combine mechanisms to both reduce in-memory copy operations and reduce per-client data elements.

Figure 7 shows that, by using NLF, packet header generation is the most time consuming operation in our modified UDP protocol. Thus, by either including the packet header in the NLF implementation (as in the basic design), or additionally using the idea of pregenerating header templates on connection setup [19,20], the overhead of filling in header fields can be reduced. In this context, the headers will always be identical, with the exception of the checksum field, so it will probably be preferable to only pregenerate a header template at stream initialization time, instead of retrieving all this information from disk. The checksum for the header template could also be precalculated during stream initialization, minimizing the checksum operation without storing the entire packet including header. If we precalculate the header template checksum, the template can, after performing the checksum operation (due to using the pseudo-header in this operation), also include most of the IP protocol header fields, and the IP protocol processing will thereby also be minimized. However, such an implementation of NLF and a further analysis of this topic is ongoing work.

Another approach to reduce communication system overhead is to use special hardware on the network adapters which have become available in the last few years. These cards come with on-board firmware that may calculate the checksum(s) on the network adapter. Furthermore, there are several proposed solutions of how to move the whole protocol processing on-board the network adapter [21,22,23], where performance issues like copying, checksumming, context switching, policing offloading, etc., are addressed. These new network adapters may make the NLF mechanism unnecessary in some cases, but yet this hardware is not available for all platforms, and they have limited CPU power and amount of memory. This hardware can replace NLF, but can also be used together with NLF. Since the number of CPU cycles is limited, there might be resource shortage if several streams perform checksum operations, encryption, and FEC. Doing NLF checksumming offline and providing header templates will also save CPU cycles and memory references on the network adapter and reduce the end-to-end latency. We are about to start investigating whether such a design can be implemented, and whether it will be cost-effective with respect to protocol complexity and performance gain.

Several approaches try to optimize error correction, e.g., trying to find optimal codes or making hybrid solutions for specific application scenarios [16]. However, to our best knowledge there are no solutions that take the server workload into consideration. Our approach minimizes the server overhead to optimize performance, and is able to recover a corrupted data stream on the client side.

There are several important results in the area of zero-copy implementation and in the area of pyramid broadcasting schemes. Some works report the usage of prefabricated packets [24,25]. However, none of them has reported a combination of these three techniques or any two of them or a corresponding performance improvement.

3.6 Discussion

Experience with communication protocol processing shows that bigger packets are better due to less costs for transmitting the data [26]. The packet sizes are so far determined only by the recovery code's coding performance, and based on the experiments evaluating these schemes, a packet size of 1 KB or 2 KB is appropriate with respect to start-up latency and decoding throughput. However, the packet size also affects performance in other components like the network itself and the routers. Using large packets, the reduced costs associated with the network are less data to send (less packet headers), fewer routing decisions, and reduced protocol processing and device interrupt handling overhead. Smaller packets give less packet fragmentation and reduce latency in the intermediate

nodes. Thus, the optimal packet size is determined by several factors which vary for each link in a heterogeneous environment, and more research is required in this area [26].

In our experiments, we have a source of errors in the measurements, because we experienced some congestion in the Ethernet queue. This means that not all low level (ethernet) protocol code is executed for all the packets processed through UDP/IP. The loss-experiments show that the error margin is about 2–3% [16] in single-stream scenarios transmitting 1 GB in about 10 seconds (about 10^{-7}%, i.e., insignificant, in the multi-stream scenarios). However, if we add the execution time of these functions for the lost packets (approximately 200 ms for the 1 GB transmission), we still have a large improvement compared to the traditional data path using about 20 seconds transmitting the same amount of data. Thus, even though we have some server side loss, the results give a good indication of the improvements using our mechanisms. However, there is a need for either (1) some rate control mechanism which is not provided by UDP or a mechanism which waits if the queue is overloaded so that we do not lose packets in the server end-system, or (2) some higher performance network cards than our 1 Gbps network card (available today).

If we look at the used CPU time and assume that the disks and the network card are not a bottleneck, the used CPU times to process data through the system indicate a throughput of 1.49 Gbps and 3.83 Gbps using the zero-copy data path without and with NLF, respectively. Thus, if the operating system processing is the only bottleneck, we are able to achieve data transmissions at these speeds. This means that using the mechanisms described in this paper, the operating system is no longer a bottleneck, because data can be processed through the system faster than our hardware components can manage.

Our performance experiments show that a single disk storage system, a 1 Gbps network card, and a 64 bit, 66 MHz PCI bus are severe bottlenecks in a Gbps MoD environment. However, there are several hardware components for a similar testbed that is better than ours. The storage system bottleneck can be solved using parallel off-the-shelf disks. For example, the Seagate Cheetah X15 [27] achieves a minimum data rate of 299 Mbps. Connecting several such disks to the new SCSI fiber channel interface, which offers bus data rates of up to 3.2 Gbps for a dual loop, may solve the Gbps storage system bottleneck. Likewise, a 10 Gbps network card, like the Intel IXP 2800 [28], and a (coming) 64 bit, 533 MHz PCI-X I/O bus (34.1 Gbps) [29] may solve the network card and I/O bus limitations, respectively. However, there also exist more powerful processors able to process data faster through the operating system reintroducing the same bottlenecks. Thus, removing the bottlenecks addressed in this paper is a step towards the goal of invalidating the phrase *operating systems are not getting faster as fast as hardware* [1] – at least in the context of our special read-only MoD scenario.

In summary, our mechanisms perform as expected, and the total performance gain in our server is approximately equal to the sum of each mechanism's individual gain. Storing parity data on disk removes the FEC encoding bottleneck. The zero-copy data path and NLF reduce the time to process data from disk to network interface, and the broadcasting protocol enables data sharing between concurrent clients by broadcasting one set of data to all clients viewing the same file.

4 Conclusions

The overall goal of INSTANCE is to improve the I/O performance of intermediate storage nodes by avoiding the major bottlenecks in the common case operation of storage nodes, i.e., retrieving data from disk and sending it to remote clients. We have achieved this goal by tightly integrating three orthogonal techniques: zero-copy-one-copy memory architecture, NLF, and integrated error management.

The integrated error management frees the storage node from all resource intensive error management tasks by using precomputed parity data. The described scheme allows to recover from one disk failure. Reusing the storage system parity data allows us to correct correspondingly 12.5% of packet loss or corruption in the network. Furthermore, the decoding time at the client is relative to the amount of network errors and introduces in the worst case only a delay of 100–700 ms depending on the packet size on a client with a 933 MHz CPU. The removal of copy operations reduces the amount of needed memory and CPU resources, and minimizes the time to transmit data. We have shown that the zero-copy data path increases the number of concurrent clients that can be supported by at least 100%. The NLF mechanism further reduces the resource requirement per stream, and combined with our in-kernel data path, the kernel time is reduced by 66.18% (1 KB packets), 70.37% (2 KB packets), 75.95% (4 KB packets), and 75.25% (8 KB packets). In summary, the total number of concurrent streams is increased by a factor of two, the kernel processing cost is reduced to a quarter, we eliminate per-client data elements, and we remove the FEC encoding bottleneck.

In this paper, we show that the INSTANCE approach and its potential to improve the I/O performance of a single storage node by a factor of at least two is to the best of our knowledge unique. There are several important results in the area of optimizing performance for multimedia applications. However, none of them have reported a combination of these three techniques or any two of them, or a corresponding performance improvement.

Ongoing work in this area includes extensions to NLF to cover the whole communication system, and the integration of NLF and on-board processing. We are also working on the design and implementation of optimized upload protocols. Future extensions of the described work include the study of the impact of using standardized protocols like RTP and RTCP, and a TCP-friendly congestion control combined with UDP. Furthermore, we are currently working on the usage of hybrid FEC/ARQ schemes in INSTANCE.

Acknowledgements. This research is sponsored by the Norwegian Research Council in the DITS program under contract number 119403/431.

References

1. J. K. Ousterhout (1990) Why aren't operating systems getting faster as fast as hardware? Proceedings USENIX Summer Conference, Anaheim, CA, pp 247–256
2. T. Plagemann, V. Goebel, P. Halvorsen, O. Anshus (2000) Operating system support for multimedia systems Computer Communications 23(3):267–289

3. P. Halvorsen, T. Plagemann, V. Goebel (2000) Network level framing in INSTANCE. Proceedings 6th International Workshop on Multimedia Information Systems (MIS 2000), Chicago, IL, pp 82–91
4. P. Halvorsen, T. Plagemann, V. Goebel (2001) Integrated error management for media-on-demand services. Proceedings 20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2001), Anchorage, AK, pp 621–630
5. M. M. Buddhikot (1998) Project MARS: scalable, high performance, web based multimedia-on-demand (MOD) services and servers. PhD Thesis, Sever Institute of Technology, Department of Computer Science, Washington University, St. Louis, MO
6. K. Fall, J. Pasquale (1993) Exploiting in-kernel data paths to improve I/O throughput and CPU availability. Proceedings USENIX Winter Technical Conference, San Diego, CA, pp 327–333
7. S. Viswanathan, T. Imielinski (1996) Metropolitan area video-on-demand service using pyramid broadcasting. Multimedia Systems 4(4):197–208
8. L. Gao, J. Kurose, D. Towsley (1998) Efficient schemes for broadcasting popular videos. Proceedings 8th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'98), Cambridge, UK
9. K. A. Hua, S. Sheu (1997) Skyscraper broadcasting: a new broadcasting scheme for metropolitan video-on-demand system. Proceedings ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'97), Cannes, France, pp 89–100
10. J.-F. Paris, S. W. Carter, D. D. E. Long (1998) Efficient broadcasting protocols for video on demand. Proceedings 6th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '98), Montreal, Canada, pp 127–132
11. W. R. Stevens (1998) UNIX network programming, volume 1, networking APIs: sockets and XTI. 2nd edn. Prentice Hall
12. D. Haccoun, S. Pierre (1997) Automatic repeat request. In: J. D. Gibson (ed) The Communications Handbook. CRC Press
13. J. Blömer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, D. Zuckerman (1995) An XOR-based erasure-resilient coding scheme. Technical Report TR-95-048, International Computer Science Institute (ICSI), The University of California at Berkeley, CA
14. L. Rizzo (1997) Effective erasure codes for reliable computer communication protocols. ACM Computer Communication Review 27(2): 24–36
15. Intel Corporation (2000) Intel 840 chipset – product overview. http://developer.intel.com/design/chipsets/840/
16. P. Halvorsen (2001) Improving I/O Performance of multimedia servers. PhD Thesis, Department of Informatics, University of Oslo, Norway
17. Intel Corporation (1998) Using the RDTSC instruction for performance monitoring. ftp://download.intel.com/software/idap/-media/pdf/rdtscpm1.pdf
18. J. Kay, J. Pasquale (1993) The importance of non-data touching processing overheads in TCP/IP. Proceedings ACM Conference on Communications Architectures, Protocols and Applications (SIGCOMM'93), San Francisco, CA, pp 259–268
19. D. D. Clark, V. Jacobson, J. Romkey, H. Salwen (1989) An analysis of TCP processing overheads. IEEE Communication Magazine 27(2): 23–29
20. J. H. Saltzer, D. D. Clark, J. L. Romkey, W. C. Gramlich (1985) The desktop computer as a network participant. IEEE Journal on Selected Areas in Communications 3(3):468–478
21. C. Beauduy, R. Bettati (1999) Protocols aboard network interface cards. Proceedings IASTED International Conference on Parallel and Distributed Computing and Systems, Cambridge, MA
22. U. Dannowski, H. Haertig (2000) Policing offloaded. Proceedings 6th IEEE Real-Time Technology and Applications Symposium (RTAS 2000), Washington, DC, pp 218–227
23. P. Shivam, P. Wyckoff, D. Panda (2001) EMP: zero-copy OS-bypass NIC-driven gigabit Ethernet message passing. Proceedings Supercomputing Conference (SC2001), Denver, CO
24. M. Kumar (1998) Video-server designs for supporting very large numbers of concurrent users. IBM Journal of Research and Development 42(2):219–232
25. N. J. P. Race, D. G. Waddington, D. Sheperd (2000) A dynamic RAM cache for high quality distributed video. Proceedings 7th International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services (IDMS'00), Enschede, The Netherlands, pp 26–39
26. W. R. Stevens (2000) TCP/IP illustrated, volume 1 – the protocols. Addison-Wesley Longman
27. Seagate (2001) Disk products by product number. http://www.seagate.com/cda/products/discsales/index
28. Intel Corporation (2002) Intel IXP2800 network processor. http://www.intel.com/design/network/products/npfamily/-ixp2800.htm
29. PCI SIG (2002) PCI-X 2.0: The next generation of backward-compatible PCI. http://www.pcisig.com/specifications/-pci_x_20/
30. MPEG.org: DVD Technical Notes – Bitstream breakdown, http://mpeg.org./MPEG/DVD, March 2000