A Scalable and High Performance Software iSCSI Implementation

Abhijeet Joglekar, Michael E. Kounavis, and Frank L. Berry
Intel Research and Development, Hillsboro, OR, 97124, USA

Abstract

In this paper we present two novel techniques for improving the performance of the Internet Small Computer Systems Interface (iSCSI) protocol, which is the basis for IP-based networked block storage today. We demonstrate that by making a few modifications to an existing iSCSI implementation, it is possible to increase the iSCSI protocol processing throughput from 1.4 Gbps to 3.6 Gbps. Our solution scales with the CPU clock speed and can be easily implemented in software using any general purpose processor without requiring specialized iSCSI protocol processing hardware.

To gain an in-depth understanding of the processing costs associated with an iSCSI protocol implementation, we built an iSCSI fast path in a user-level sandbox environment. We discovered that the generation of Cyclic Redundancy Codes (CRCs) which is required for data integrity, and the data copy operations which are required for the interaction between iSCSI and TCP represent the main bottlenecks in iSCSI protocol processing. We propose two optimizations to iSCSI implementations to address these bottlenecks. Our first optimization is on the way CRCs are being calculated. We replace the industry standard algorithm proposed by Prof. Dilip Sarwate with 'Slicing-by-8' (SB8), a new algorithm capable of ideally reading arbitrarily large amounts of data at a time while keeping its memory requirement at reasonable level. Our second optimization is on the way iSCSI interacts with the TCP layer. We interleave the compute-intensive data integrity checks with the memory access-intensive data copy operations to benefit from cache effects and hardware pipeline parallelism.

1. Introduction

Networked block storage technologies are likely to play a major role in the development of next generation data centers. In this paper we address the problem of efficiently implementing networked block storage systems focusing on systems that operate on top of the TCP/IP protocol stack. We analyze the performance of the iSCSI protocol [30], which is the basis for IP-based networked block storage today and suggest ways to improve its performance. There are two primary reasons why we believe IP-based networked block storage is important. First, such type of storage enables efficient remote backup and recovery operations on top of large-scale and geographically distributed networks. Second, using the same IP-based technology in both storage and regular communication networks makes network management easier and less expensive since there is only one type of network to manage. More elaborate discussions on networked storage are presented in [28, 33].

Commercial iSCSI solutions have been designed thus far using TCP/IP offload engines (TOEs) or iSCSI host bus adapters (HBAs). These systems offload either the TCP/IP protocol stack or both the TCP/IP and the iSCSI protocols into specialized hardware units. In this paper we follow an alternative approach to offloading by focusing on a software-only iSCSI implementation. The reason why we focus on a software iSCSI implementation is because such implementation scales better with the CPU clock speed and the number of processing units available and can be easily realized using general purpose processors without specialized iSCSI protocol processing hardware. Our work is also motivated by earlier studies that have demonstrated that accessing protocol offload engines may become a bottleneck for some protocol processing workloads. For example, Sarkar et al [27] compare a software iSCSI stack with two industry standard solutions, a TOE and an HBA, operating at 1 Gbps. Their paper shows that while current generation hardware solutions do achieve better throughput-utilization efficiency as compared to software for large block sizes, accessing the hardware offload engines becomes a bottleneck for small block sizes.

The contributions of this paper can be summarized as follows: First, through measurements and simulations performed on a sandbox implementation of iSCSI, we quantify the processing costs of each of the protocol components including data structure manipulation, CRC generation, and data copies. We identify the CRC generation and data copies as the primary bottleneck in iSCSI processing. Second we replace the industry-standard CRC generation algorithm developed by Prof. Dilip Sarwate [29] with a new 'Slicing-by-8' (SB8) algorithm, capable of ideally reading arbitrarily large amounts of data at a time while keeping its memory requirement at reasonable level. A third contribution of our paper is a novel way to implement the interaction between the iSCSI and TCP layers. We interleave the compute-intensive data integrity checks with the

memory access-intensive data copy operations to benefit from cache effects and hardware pipeline parallelism. This optimization was inspired by the idea of integrated copy-checksum as first suggested by Clark et al [6]. We demonstrate that these two novel implementation techniques can increase the processing throughput of our implementation from 1.4 Gbps to 3.6 Gbps. These optimizations correspond to a small number of changes in the source code of a software iSCSI implementation. Our work relies on the acceleration of TCP on the CPU which is a well researched problem [4, 5, 13, 26].

The paper is organized as follows. In Section 2 we provide an overview of the iSCSI protocol, and describe typical receive and transmit fast paths in an iSCSI initiator stack. The information presented in this section is essential so as the reader can understand our optimizations. For more information on iSCSI, the reader can look at [30]. In section 3, we propose two optimizations that address the two primary bottlenecks in an iSCSI implementation - the CRC generation process and the data copies. In Section 4, we describe our sandbox iSCSI implementation, and our measurement and simulation methodology. In Section 5 we evaluate our approach and discuss our results. In Section 6 we present related work in the area and, finally, in Section 7 we provide some concluding remarks.

2. Overview of iSCSI processing

2.1 The Protocol

The iSCSI protocol maps the SCSI client-server protocol onto a TCP/IP interconnect. Initiators (clients) on a SCSI interface issue commands to a SCSI target (server) in order to request the transfer of data to or from I/O devices. The iSCSI protocol encapsulates these SCSI commands and the corresponding data into iSCSI Protocol Data Units (PDUs) and transmits them over a TCP connection. An iSCSI PDU includes a Basic Header Segment (BHS), followed by one or more Additional Header Segments (AHS). Additional header segments are followed by a data segment. Headers and data are protected separately by a digest based on the CRC32c standard [30].

An iSCSI session has two phases. It starts with a 'login' phase during which the initiator and target negotiate the parameters for the rest of the session. Then, a 'full feature' phase is used for sending SCSI commands and data. Based on the parameters negotiated during the login phase, an iSCSI session can use multiple TCP connections multiplexed over one or more physical interfaces, enable data integrity checks over PDUs, and even incorporate different levels of error recovery. iSCSI sessions are typically long-lived. The login phase represents only a small part of the overall protocol processing load. Because of this reason we have decided to investigate optimizations on the 'full feature' phase of the protocol only. Figure 1 depicts a typical layered protocol stack on an initiator.

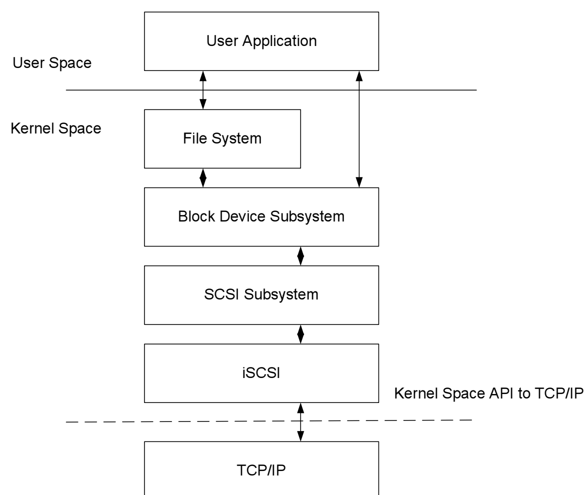

Figure 1: Typical initiator protocol stack

User-level applications issue 'read' and 'write' system calls that are serviced by the SCSI subsystem through the file system and block device layers. A SCSI 'upper' layer creates a SCSI Command Descriptor Block (CDB) and passes it to a SCSI 'lower' layer. The SCSI lower layer is the iSCSI driver for an IP interconnect. This iSCSI driver uses the kernel interface to the TCP/IP stack to transmit SCSI commands and data to a target. Each SCSI read command is transmitted as an iSCSI command PDU to a target. A target then services the read command by encapsulating SCSI data into one or multiple iSCSI data PDUs and by sending them to the initiator. The data PDUs are eventually followed by a status PDU from the target signaling the completion of the read operation. SCSI write commands are similarly implemented by first sending a SCSI write command to a target followed by a pre-negotiated number of unsolicited bytes. The target paces the flow of data from the initiator by issuing flow-control messages based on the availability of target buffers. As in the case of the read command, the end of the data transfer is indicated by the transmission of a status PDU from the target to the initiator.

2.2 iSCSI Read Processing

Figure 2 shows the processing of an incoming data PDU (also called 'data-in' PDU) in an initiator stack performing SCSI read operations.

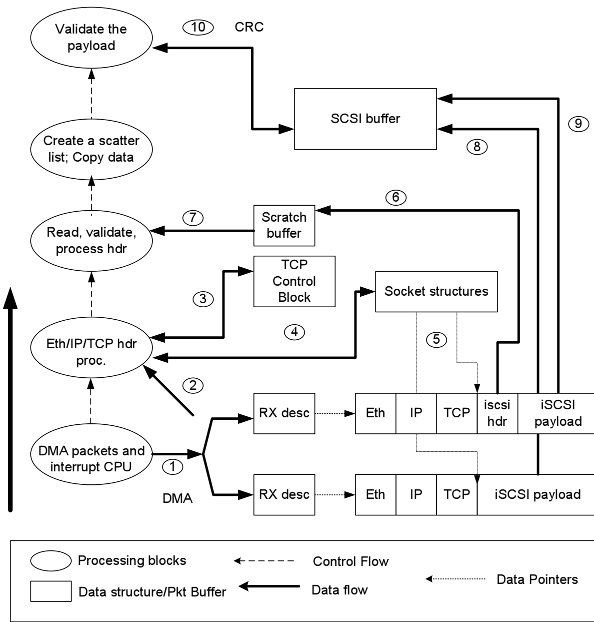

Figure 2: Incoming data PDU processing for reads

An iSCSI PDU, which has a default size of 8KB, can span multiple TCP segments. As the Network Interface Card (NIC) receives these segments, it places them using the Direct Memory Access (DMA) technique into a number of NIC buffers and interrupts the CPU (step 1 in the figure). The device driver and stack then use the Rx descriptors and the TCP Control block in order to perform TCP/IP processing and to strip off the Eth/IP/TCP headers (steps 2 and 3). The segment payloads are then queued into socket descriptor structures (steps 4 and 5). So far, steps 1-5 describe processing associated at the TCP/IP layer and below.

Steps 6-10 describe processing associated with the iSCSI layer. The iSCSI layer first reads the iSCSI header from the socket layer into an internal scratch buffer (step 6). The header consists of a fixed 48 byte basic header, a header CRC, and in some cases additional header bytes. It then computes a CRC over the header and validates this CRC value by comparing it with the CRC attached to the header. If the CRC value is valid, the iSCSI layer processes the header and identifies the incoming iSCSI PDU as a data PDU (step 7). A tag value included in the header is used for identifying the SCSI buffer where the data PDU should be placed. Based on the length of the PDU and its associated offset, both of which are indicated in the iSCSI header, the iSCSI layer creates a scatter-list pointing to the SCSI buffer. Then, the iSCSI layer passes the scatter list to the socket layer which copies the iSCSI PDU payload from the TCP segments into the SCSI buffer (steps 8 and 9). Finally, the data CRC is computed and validated over the entire PDU payload (step 10). This is the last step in the processing of an incoming PDU.

2.3 iSCSI Write Processing

Figure 3 shows the handling of an outgoing data PDU (also called 'data-out' PDU) in an initiator stack performing SCSI writes to the target.

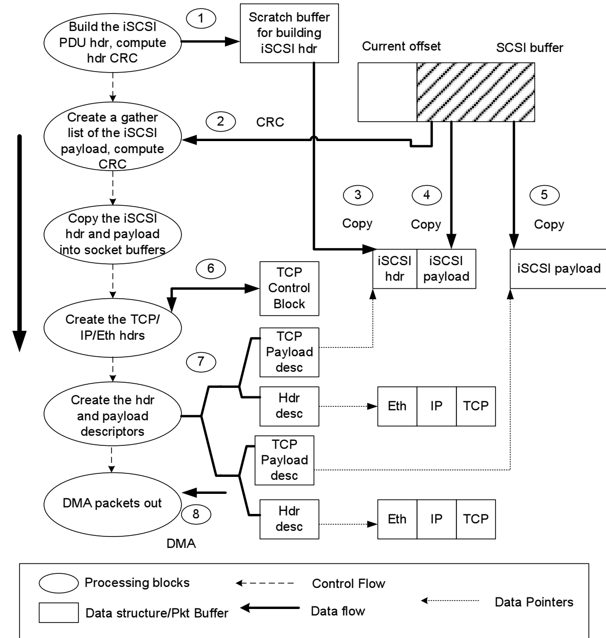

Figure 3: Outgoing data PDU processing for writes

The iSCSI protocol layer at the initiator maintains a pointer into the SCSI buffer for the next data PDU to be sent to the target. In reply to flow control (i.e., R2T) PDUs that are received from the target, the iSCSI layer transmits the data PDU. It first constructs the iSCSI header that describes the data PDU, computes a CRC value on the header, and attaches this CRC value to the header (step 1). The iSCSI layer then builds a gather list that describes the header, the header CRC, and the payload of the iSCSI PDU. It computes a data CRC on the payload and attaches it to the gather list (step 2). Finally, it uses the kernel interface to the TCP/IP stack to send the PDU to the target.

Based on the TCP/IP stack implementation, the data is either directly transmitted from the SCSI buffer to the NIC or undergoes a copy to temporary socket buffers, as shown in steps 3, 4, and 5. The TCP/IP stack then creates the transport, network, and link headers associated with each of the TCP segments that hold a portion of the iSCSI PDU. It also creates header and payload descriptors that point to the packet headers and the TCP payload respectively (steps 6 and 7). Finally,

the stack signals the NIC to send the packets out onto the wire via DMA (step 8).

3. Software Optimizations

One can expect that the CRC generation process and data copies are the most time consuming parts of iSCSI processing. CRC generation is costly because it requires several logical operations to be performed on a byte-by-byte basis for each byte of data. Data copies are costly because accessing off-chip memory units typically requires several hundreds of clock cycles to complete. In sections 4 and 5, we describe our sandbox implementation and processing profile of the iSCSI stack. The performance of our sandbox implementation was measured on a 1.7 GHz Intel® Pentium® M processor with a 400 MHz Front Side Bus (FSB), and a single channel DDR-266 memory subsystem. In summary, we found that the generation of CRC32c codes indeed represents the most time consuming component of the stack operating at a rate of 6.5 cycles per byte, followed by data copy operations that operate at a rate of 2.2 cycles per byte. In this section, we describe how we address these bottlenecks using a new CRC generation algorithm and a technique to interleave data copies with the CRC generation process.

3.1 The CRC generation process

Cyclic redundancy codes (CRC) are used for detecting the corruption of digital content during its production, transmission, processing or storage. CRC algorithms treat each bit stream as a binary polynomial $B(x)$ and calculate the remainder $R(x)$ from the division of $B(x)$ with a standard 'generator' polynomial $G(x)$. The binary words corresponding to $R(x)$ are transmitted together with the bit stream associated with $B(x)$. The length of $R(x)$ in bits is equal to the length of $G(x)$ minus one. At the receiver side, CRC algorithms verify that $R(x)$ is the correct remainder. Long division is performed using modulo-2 arithmetic. Additions and subtractions in module-2 arithmetic are 'carry-less' as illustrated in Table 1. In this way additions and subtractions are equal to the exclusive OR (XOR) logical operation.

0+0 = 0-0 = 0
0+1 = 0-1 = 1
1+0 = 1-0 = 1
1+1 = 1-1 = 0

Table 1: Modulo-2 arithmetic

Figure 4 illustrates a long division example. In the example, the divisor is equal to '11011' whereas the dividend is equal to '1000111011000'. The long division process begins by placing the 5 bits of the divisor below the 5 most significant bits of the dividend. The next step in the long division process is to find how many times the divisor '11011' 'goes' into the 5 most significant bits of the dividend '10001'. In ordinary arithmetic 11011 goes zero times into 10001 because the second number is smaller than the first. In modulo-2 arithmetic, however, the number 11011 goes exactly one time into 10001. To decide how many times a binary number goes into another in modulo-2 arithmetic, a check is being made on the most significant bits of the two numbers. If both are equal to '1' and the numbers have the same length, then the first number goes exactly one time into the second number, otherwise zero times. Next, the divisor 11011 is subtracted from the most significant bits of the dividend 10001 by performing an XOR logical operation. The next bit of the dividend, which is '1', is then marked and appended to the remainder '1010'. The process is repeated until all the bits of the dividend are marked. The remainder that results from such long division process is the CRC value.

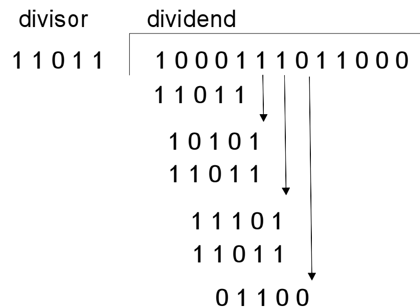

Figure 4: Long division using modulo-2 arithmetic

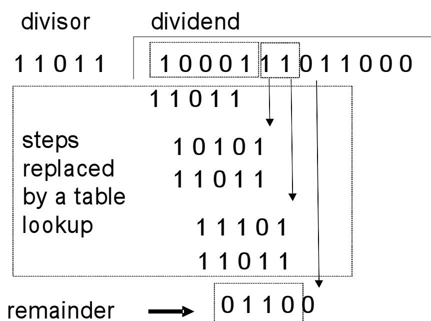

Figure 5: Accelerating the long division using table lookups

The long division process is a compute-intensive operation because it requires in the worst case one shift operation and one XOR logical operation for every bit of a bit stream. Most software-based CRC generation algorithms, however, perform the long division quicker than the bit-by-bit marking process described above.

One commonly used technique for accelerating the long division process is to pre-compute the current remainder that results from a group of bits and place the result in a table. Before the beginning of the long division process all possible remainders which result from groups of bits are pre-computed and placed into a lookup table. In this way, several long division steps can be replaced by a single table lookup step.

The main idea behind this technique is shown in Figure 5. In the example of Figure 5, the remainder '0110', which is formed in the third step of the long division process is a function of the five most significant bits of the dividend '10001' and the next two bits '11'. Since these bits are known, the remainder 0110 can be calculated in advance. As a result, 3 long division steps can be replaced by a single table lookup. Additional table lookups can further replace subsequent long division steps. To avoid using large tables, table-driven CRC acceleration algorithms typically read no more than 8 bits at a time.

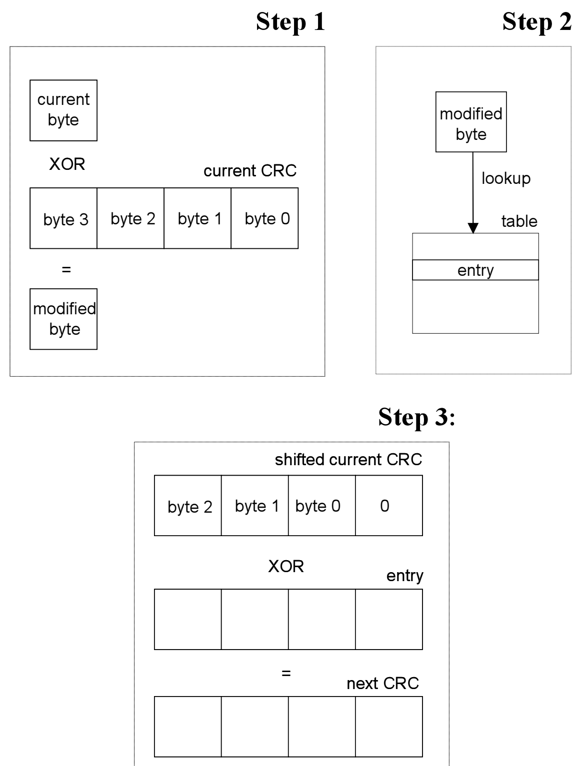

Figure 6: The Sarwate algorithm

The most representative table-driven CRC generation algorithm used today is the algorithm proposed by Dilip V. Sarwate, shown in Figure 6. The length of the CRC value generated by the Sarwate algorithm is 32 bits. The Sarwate algorithm is more complicated than the straightforward lookup process of Figure 5 because the amount of bits read at a time (8 bits) is smaller than the degree of the generator polynomial.

Initially, the CRC value is set to a given number which depends on the standard implemented (e.g., this number is 0xFFFFFFFF for CRC32c). For every byte of an input stream the algorithm performs the following steps: First, the algorithm performs an XOR operation between the most significant byte of the current CRC value and the byte from the stream which is read (Step 1). The 8-bit number which is produced by this XOR operation is used as an index for accessing a 256 entry table (Step 2). The lookup table used by the Sarwate algorithm stores the remainders from the division of all possible 8-bit numbers shifted by 32 bits to the left with the generator polynomial. The value returned from the table lookup is then XOR-ed with the 24 least significant bits of the current CRC value, shifted by 8 bit positions to the left (Step 3). The result from this last XOR operation is the CRC value used in the next iteration of the algorithm's main loop. The iteration stops when all bits of the input stream have been taken into account. Detailed justification and proof of correctness of the Sarwate algorithm is beyond the scope of this paper. The reader can learn more about the Sarwate algorithm in [29].

3.2 Optimizing the CRC generation process

The Sarwate algorithm was designed at a time when most computer architectures supported XOR operations between 8 bit quantities. Since then, computer architecture technology has progressed to the point where arithmetic operations can be performed efficiently between 32 or 64 bit quantities. In addition modern computer architectures comprise large on-chip cache memory units which can be accessed in a few clock cycle time. We believe that such advances call for re-examination of the mathematical principles behind software-based CRC generation.

The main disadvantage of existing table-driven CRC generation algorithms is their memory space requirement when reading a large number of bits at a time. For example, to achieve acceleration by reading 32 bits at a time, table-driven algorithms require storing pre-computed remainders in a table of 2^{32} = 4G entries. To solve this problem we propose a new algorithm that slices the CRC value produced in every iteration as well as the data bits read into small terms. These terms are used as indexes for performing lookups on different tables in parallel. The tables differ between each other and are constructed in a different manner than Sarwate as explained in detail below. In this way, our algorithm is capable of reading 64 bits at a time, as opposed to 8, while keeping its memory space requirement to 8KB.

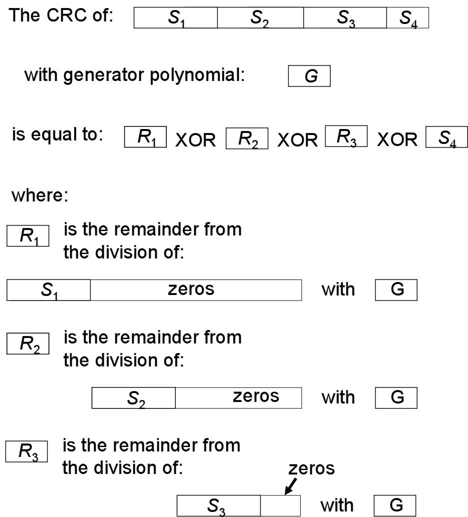

Figure 7: The Bit slicing principle

CRC is a linear code. This means that CRC(A+B) = CRC(A) + CRC(B). The linearity of CRC derives from the arithmetic used (modulo 2). The design of our algorithm is based on two principles associated with the linearity of CRC, namely a 'bit slicing' and a 'bit replacement' principle. The bit slicing principle is shown in Figure 7. According to this principle, if a binary number is sliced into two or more constituent terms the CRC value associated with the binary number can be calculated as a function of the CRC values of its constituent terms. As it is shown in the figure, the CRC of the number consisting of slices S_1, S_2, S_3, and S_4 is the result of an XOR operation between values R_1, R_2, R_3, and S_4. Value R_1 is the CRC of the original number if all slices but S_1 are replaced with zeros. Values R_2 and R_3 are defined in a similar manner. The bit slicing principle is important because it allows us to compute the CRC of a potentially large number as a function of the CRCs of smaller terms. Thus, the bit slicing principle can potentially solve the memory explosion problem associated with existing table-driven algorithms.

The bit replacement principle is shown in Figure 8. According to this principle, an arbitrarily long prefix of a bit stream can be replaced by an appropriately selected binary number, without changing the CRC value of the stream. The binary number used for replacing a prefix is the remainder from the division of the prefix with the CRC generator polynomial. In the example of Figure 8, the U_1 prefix of the binary number [U_1:U_2] can be replaced by the remainder R_1 from the division of U_1 with G. It is this bit replacement principle which we take advantage of in the design of our algorithm in order to read 64 bits at a time.

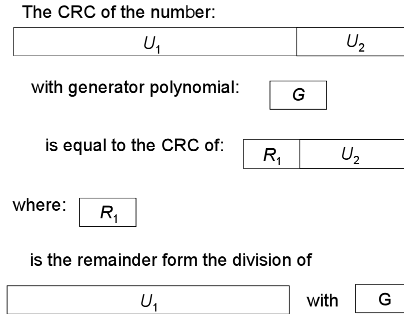

Figure 8: The Bit replacement principle

Our algorithm, called 'Slicing-by-8' (SB8), is illustrated in Figure 9. As in the case of the Sarwate algorithm, the initial CRC value of a stream is set to a given number which depends on the standard implemented (e.g., this number is 0xFFFFFFFF for CRC32c). For every 64-bit chunk of an input stream the algorithm performs the following steps: First, the algorithm performs an XOR operation between the 32 most significant bits of the chunk and the current CRC value (Step 1). The 64-bit value produced from this XOR operation is then sliced into 8 slices of equal length (Step 2). Each slice is used for accessing a separate lookup table (Step 3). The lookup tables used by the algorithm store the remainders from the division of all possible 8-bit numbers shifted by a variable number of bits to the left with the generator polynomial. The offset values used for calculating the table entries begin with 32 for Table 1 and increase by 8 for every table. The values returned form all table lookups are XOR-ed to one another producing the CRC value used in the next iteration of the algorithm's main loop.

The benefit from slicing comes from the fact that modern processor architectures comprise large cache units. These cache units are capable of storing moderate size tables (e.g., 8KB tables as required by the Slicing-by-8 algorithm) but not sufficient for storing tables associated with significantly larger strides (e.g., 16GB tables associated with 32-bit strides). If tables are stored in an external memory unit, the latency associated with accessing these tables may be significantly higher than when tables are stored in a cache unit. For example, a DRAM memory access requires several hundreds of clock cycles to complete by a Pentium® M processor, whereas an access to a first level cache memory unit requires less than five clock cycles to complete. The processing cost associated with slicing is typically insignificant when compared to the cost of accessing off-chip memory units.

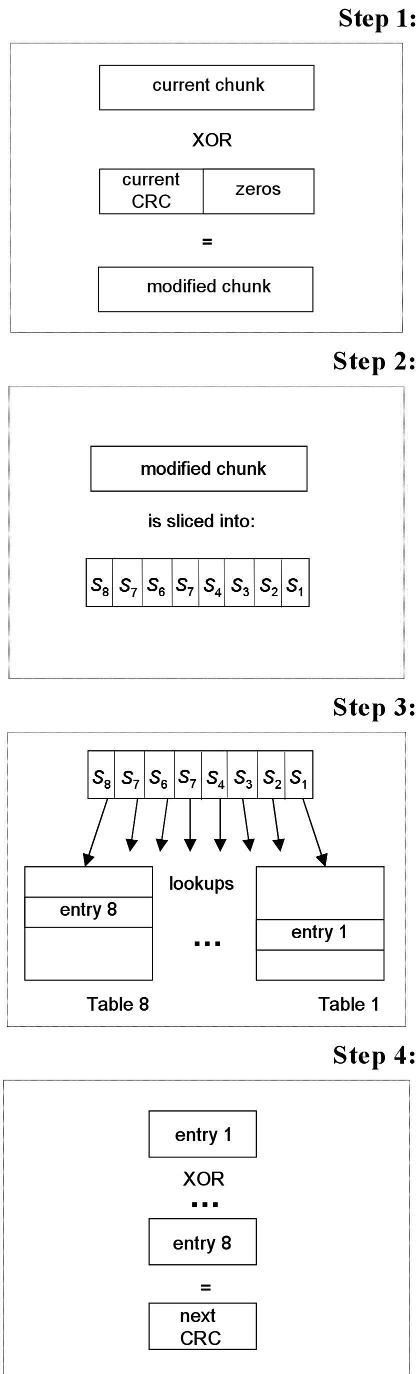

Figure 9: The Slicing-by-8 algorithm

Slicing is also important because it reduces the number of operations performed for each byte of an input stream when compared to Sarwate. For each byte of an input stream the Sarwate algorithm performs the following: (i) an XOR operation between a byte read and the most significant byte of the current CRC value; (ii) a table lookup; (iii) a shift operation on the current CRC value; and (iv) an XOR operation between the shifted CRC value and the word read from the table. In contrast, for every byte of an input stream the Slicing-by-8-algorithm performs only a table lookup and an XOR operation. This is the reason why the Sicing-by-8 algorithm is faster than the Sarwate algorithm. Detailed description and proof of correctness of Slicing-by-8 can be found in reference [16].

3.3 Interleaving data copies with the CRC generation process

Following CRC, the next big overhead in iSCSI processing is data copy. Data copy is a memory access-intensive operation, and as such its performance depends on the memory subsystem used. In this respect, data copy differs from the CRC generation process since the latter is compute-intensive and scales with the CPU clock. To further speed up data touching operations beyond CRC as discussed in Sections 3.1 and 3.2, we investigate how data copies and CRC generation take place in iSCSI.

On the inbound path, a packet which is part of an iSCSI PDU is copied from the network buffer to its appropriate offset in the SCSI buffer. In a typical software Linux iSCSI implementation, the placement of the data is controlled by the iSCSI layer, and is performed based on the information contained in the iSCSI PDU header. The copy operation by itself, however, is performed by the sockets/TCP layer. Once the entire PDU payload is placed in the SCSI buffer, the iSCSI layer computes and validates a CRC value over the entire payload. On the outbound path, the iSCSI layer calculates CRC over the entire PDU payload. It then uses the kernel sockets layer to send out this PDU as multiple TCP packets. Again, the CRC is implemented at the iSCSI layer, while the copy is a part of the kernel sockets layer.

Thus, if iSCSI is implemented in a strictly layered fashion over a kernel sockets interface to TCP/IP, copy and CRC are treated as two separate operations. However, if copy and CRC are interleaved, the combined operation results in better system performance because of two reasons:

- Parallelism among the CRC generation (compute-intensive) and data copy (memory access-intensive) operations.
- Warming of the cache memory since the data upon which CRC is computed is transferred to the cache as the copy operation runs ahead.

In essence, we apply the principle of Integrated Layer Processing (ILP) [6] in order to interleave the iSCSI CRC generation process with the data copy operations. In our approach, the sockets/TCP layer computes a CRC over the payload while copying the data between the sockets/TCP buffers and the SCSI buffers. On the

inbound path, the generated CRC value is pushed up to the iSCSI layer for validation. On the outbound path, it is inserted into the iSCSI stream. Figure 10 illustrates how the sequential and interleaved copy-CRC operations take place over time.

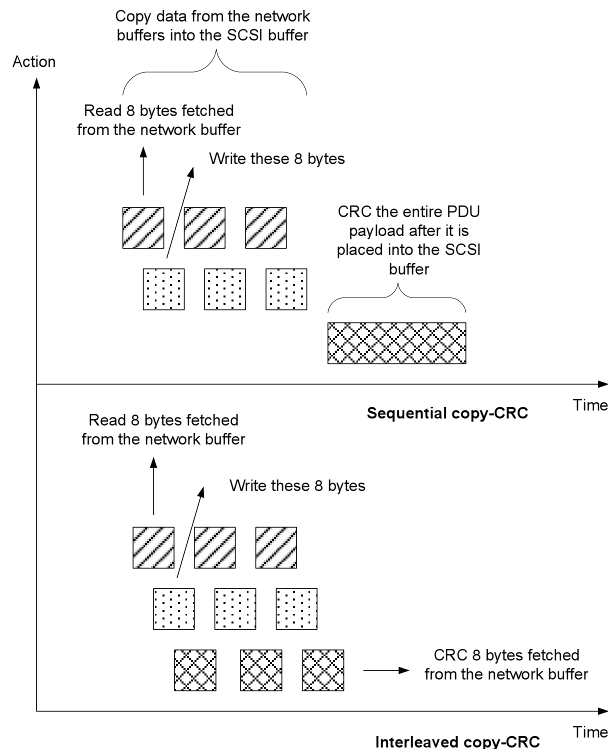

Figure 10: Sequential versus interleaved data copies and CRC

To enable our interleaved copy-CRC optimization, the interface between the iSCSI and TCP layers needs to be modified. This interface needs to allow the calculation of a CRC value on some specified set of bytes while performing a copy. If iSCSI markers are disabled [30], then the bytes that are copied are the same as the bytes on which the CRC is computed. However, when markers are enabled, the copy and CRC bytes are different, since iSCSI CRC does not cover markers. Solving the marker problem is fairly simple. As explained in Section 2, the iSCSI layer creates a scatter-gather list that describes the SCSI buffer elements where the PDU payload is to be copied from or copied to. If the payload contains markers, then the scatter-gather list also includes descriptors pointing to the scratch buffers that receive/source marker bytes. Each such list element can use a single bit 'C' in its descriptor to differentiate between marker versus non-marker bytes. If the bit C is equal to 1, this implies that the socket/TCP layer must validate the data pointed to by the descriptor. If C is equal to 0, this implies that only a copy operation should be performed on the data without CRC validation. With this scheme, skipping markers for CRC computation becomes easy. The iSCSI layer can simply set C equal to 0 for list elements that describe marker bytes and 1 for all other bytes.

Interleaving data copies and the CRC generation process does not typically modify the functional behavior of the iSCSI protocol stack. This is the case for most stack implementations, which use intermediate buffers for validating PDUs before copying these PDUs into their final buffers (e.g., application or file cache buffers). In our approach, the data copies into these intermediate buffers are interleaved with the CRC generation process. Some stack implementations, however, avoid the extra copy by directly copying the PDU to their application/file cache buffers from the socket/TCP buffers. If the PDU data is corrupted, then these stacks can end up polluting the application/file cache buffers. This can happen if the data is not validated before the copy. Interleaving copy and CRC cannot be used in such stacks if such pollution cannot be tolerated. However, for implementations where the application/file system does not make any assumptions about the buffer contents, the interleaving optimization can still be applied.

4. The iSCSI fast-path

4.1 User-level Sandbox Environment

There are several open-source prototype implementations of iSCSI (e.g. [22], [32]). Some of the earlier work in this area [14, 15, 27, 28] has used these implementations to analyze the iSCSI performance characteristics. These implementations, however, operate on top of standard, unmodified TCP/IP stacks. To evaluate iSCSI in an environment where TCP/IP optimizations like header splitting and interrupt coalescing [4, 26] are present, we took the alternative approach of implementing our own iSCSI fast-path in a user-level sandbox environment. This sandbox environment is closely coupled with an optimized TCP/IP implementation developed at Intel labs [26]. Our implementation is compliant with iSCSI RFC 3720 [30], and includes all protocol level checks in the fast-path as indicated by the specification of the protocol.

There are several benefits associated with a user-level sandbox implementation of a protocol. First, the time required for implementing and testing new ideas in a sandbox is typically much smaller than the time required for a kernel-level prototype. Second, it is easier to run the sandbox implementation on different processor and platform simulators in order to study the scaling of processing costs with architectural

improvements on processors. The main drawback of a sandbox implementation is that it cannot put bits on the wire and thus cannot interact with real world protocol stacks. However, we believe that such an implementation is a useful first step in analyzing complex protocols, especially when protocols are implemented in the kernel.

4.2 Implementation

Our sandbox implementation includes the iSCSI fast-path code for read and write commands, packet data structures, an interface to the SCSI layer, and an interface to the sockets/TCP layer. Our implementation is optimized to align key data structures along cache lines and uses pre-fetching of data structures wherever possible to avoid high memory access latencies. Similarly, SCSI buffers are page aligned and initialized so that the operating system can page-in the buffers. This emulates the effect of pre-pinned SCSI buffers in real implementations so that there are no page faults during the fast path. Before running the code of interest, the test application purges the fast-path data structures out of the cache memory. It then pre-fetches the packet headers or other data structures in order to warm the cache. For example, for inbound read processing operations, the application warms packet headers in order to emulate the effect of TCP header processing. Similarly, SCSI buffers are purged out of the cache or warmed based on whether the run is emulating cold data or recently created application warm data.

For studying the processing costs associated with the incoming (data-in) PDUs of read commands, our initialization code creates state at each layer (i.e., the SCSI, iSCSI and sockets/TCP layer) to emulate the outstanding read commands sent to a target. This includes creating SCSI command structures, the SCSI buffer, iSCSI session information, and command contexts. Incoming TCP segments that make up an iSCSI data PDU and a status PDU are created at the TCP/sockets layer. State is created at the TCP layer as if TCP processing is over and the TCP payload is queued into socket buffers. The cache memory is then purged and, if required, any warming of the cache is done as described earlier. The test application is now ready to execute and measure the fast path, as described in Section 2.1.

For studying the processing costs associated with the outgoing (data-out) PDUs of write commands, our initialization code creates state that emulates outstanding SCSI write commands. This includes creating the SCSI commands and inserting them into the iSCSI session queues based on a Logical Unit Number (LUN). The fast-path then measures the cost of sending out unsolicited data-out PDUs for the write commands. For solicited data-out PDUs, the initialization code also creates state as if R2T PDUs were received from the target soliciting specific portions of the SCSI write data. The fast path then measures the cost to send out solicited data-out PDUs to the target, as described in Section 2.2.

4.3 Measurement and Simulation techniques

We measure the processing cost of executing the iSCSI stack using two techniques. The first technique runs the stack on a real machine. We use the RDTSC and CPUID instructions [11] of the IA32 processor architecture to measure the cycles spent in the fast path code. To minimize operating system interruptions, our implementation ran at real-time priority and suspended itself before each performance run in order to keep the system stable. Using processor performance counters, we were also able to find out other interesting statistics like instructions retired per PDU, number of second level (L2) cache misses per PDU, and average cycles per instruction (CPI).

The second technique we used involves examining our iSCSI implementation on an instruction-by-instruction basis by running it on a cycle-accurate CPU simulator. The simulator allows us to take a closer look at the micro-architectural behavior of the protocol on a particular processor family. This helps us determine portions of the code that result in cache misses and optimize the code by issuing pre-fetches wherever possible. Simulator runs also help in projecting protocol performance on future processors and platforms. In this way we can determine how different optimizations scale with architectural or clock-speed improvements.

5. Evaluation

5.1 Analysis of iSCSI processing

We begin our evaluation by examining how existing iSCSI implementations perform (i.e., implementations with Sarwate CRC and no copy-CRC interleaving). As mentioned earlier iSCSI processing involves four main components – data structure manipulation, CRC generation, data copies, and marker processing. In this section we also refer to data structure manipulation as 'protocol processing' (even though CRC generation and copies are also part of the protocol processing). Our implementation supports all features except markers. Since we started our investigation on a pure software implementation that uses standard kernel sockets for interfacing iSCSI with TCP, we did not implement markers since marker benefits are tied to the close coupling between iSCSI and TCP.

For both the direct execution (i.e., the execution on a real machine) and the CPU simulation runs, we used a 1.7 GHz Intel® Pentium® M processor, with a 400 MHz FSB, and a single channel DDR-266 memory subsystem. The workload consisted of a single session with 8 connections, and 40 commands (read and write commands) issued on a round-robin fashion over the connections. Table 2 shows our parameters for read command processing.

Parameter	Value
Maximum receive data segment length	8KB
Max burst length	256KB
Data PDU in order	No
Data sequence in order	No
Header digest	CRC32c
Data digest	CRC32c

Table 2: Read command parameters

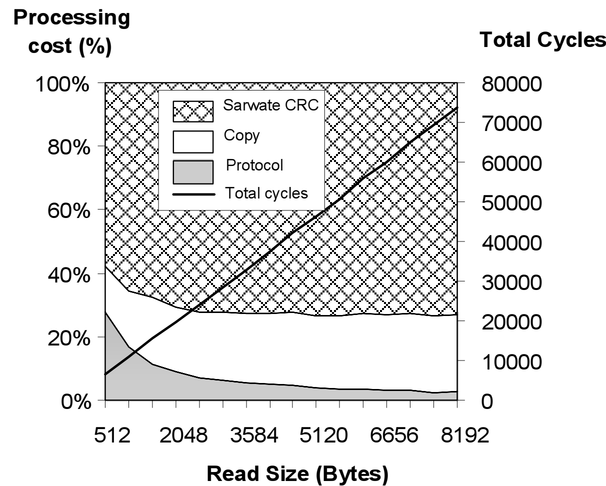

Figure 11: Data-in PDU processing cost (with Sarwate CRC)

Figure 11 shows the iSCSI processing cost for a single data-in (read) PDU at the initiator. The horizontal axis represents the size of the read I/O command issued by the initiator. The primary vertical axis (i.e., the axis on the left) represents the relative processing costs of the protocol, the CRC generation process, and the data copies. The secondary vertical axis (i.e., the axis on the right) represents the absolute number of cycles spent on iSCSI processing for a single data-in PDU. Since the read I/O workload size varies from 512B to 8KB, and the PDU size is set to 8KB, this implies that each iSCSI read command results in the reception of a single data-in PDU. In this experiment we also enabled the 'phase-collapse' feature of iSCSI. Thus each data-in PDU also carried a status response from the target.

For the smallest I/O size (i.e., 512B), the protocol cost is about 28% of the total cost, the copy cost is about 14% of the total cost, whereas CRC accounts for the remaining 58%. As the PDU size increases, the protocol cost remains the same on a per PDU basis, whereas the CRC and copy costs increase. For the largest I/O size (i.e., 8KB) the protocol cost is barely 3% of the total cost, whereas CRC is 73% of the total cost and copy accounts for 24%.

The protocol cost is paid once per PDU. It is about 1800 cycles and remains constant for each command with a slight bump at the page boundary (i.e., at 4KB), since crossing a page boundary involves adding another scatter element to the SCSI buffer. On the other hand, the CRC cost is paid on a per byte basis and increases linearly as the I/O size increases. For a workload of 8KB, the total CRC cost is about 54000 cycles, or about 6.5 cycles per byte, while the copy cost is about 2.15 cycles per byte. Thus for the 8KB PDU size which is a common PDU size, the CRC cost completely dominates the iSCSI protocol processing cost, and is significantly more than the copy cost.

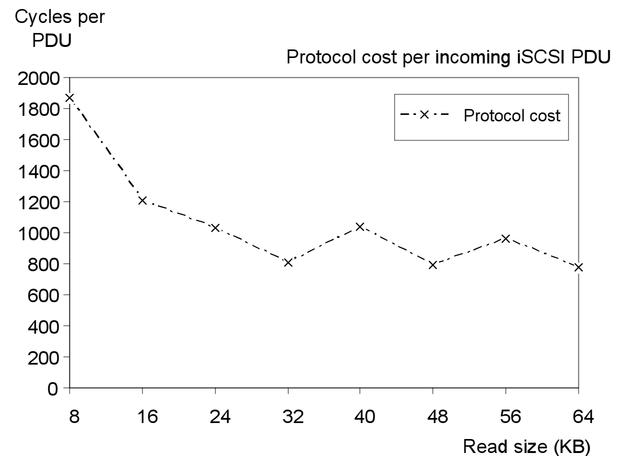

Figure 12: Data-in PDU Protocol processing cost

To get a deeper understanding of the cycles spent on protocol processing and the CRC generation process, we ran the same experiment on a cycle accurate simulator and measured the number of instructions executed on a per PDU basis and the number of L2 misses occurring per PDU. The protocol execution path for an 8KB PDU was about 438 instructions long, with about 6 L2 misses per PDU. These L2 misses can be attributed to accessing an iSCSI connection context, a command context, a score-boarding data structure for out-of-order PDUs, a SCSI buffer descriptor, and a SCSI context to store the response received from the target. The instruction path length for CRC is about 8 instructions per byte. We measured that it takes about 6.55 cycles per byte to compute a CRC value, with a

‘Cycles per Instruction’ (CPI) value of slightly less than 1. Next, we ran our simulator with a ‘perfect cache’ option enabled in order to simulate the ideal case in which all contexts and data are warm in the cache. In this way we were able to measure the best-case protocol processing cost that needs to be paid for processing a single data-in PDU with an embedded response. This cost was about 430 cycles which agrees with the instruction path length of 438.

In order to determine how close the protocol processing cost for a data-in PDU can approach the asymptote of 430 cycles, we ran the same experiment but with the read I/O workload size ranging from 8KB to 64 KB. Since the PDU size is fixed at 8KB, this means that each iSCSI command resulted in the reception of 1 to 8 data-in PDUs. For these experiments the last data-in PDU contained the embedded status.

Figure 12 shows the average protocol processing cost for an 8KB PDU. As seen in the graph, this cost drops down from about 1800 cycles to about 770 cycles. This drop can be attributed to the fact that some of the contexts like the SCSI descriptors are now warm in the cache. Thus, subsequent PDUs after the first PDU benefit from cache warming. The larger the read size is, the lower the per-PDU protocol cost becomes. On the other hand, the per byte cost of CRC remains the same whether the initiator reads 64 KB of data as a single 64KB read operation or 8 8KB read operations. Thus, while data structure manipulation becomes more efficient with larger read sizes, the CRC cost remains the same.

The protocol processing cost of write commands is smaller than the cost of read commands, while the CRC processing cost is the same since CRC generation incurs the same per byte cost independent of the direction of the data. In addition most stacks completely avoid or have at most one copy operation on the outbound path. Because of these reasons, CRC is a bigger bottleneck for write commands as compared to read commands.

5.2. Impact of Optimizations

To evaluate the performance benefits of the software optimizations we discussed in Section 3, we added the Slicing-by-8 (SB8) CRC implementation into our sandbox iSCSI stack. The test system and workload for these experiments were the same as described in Section 5.1. Figure 14 compares the performance of iSCSI read runs with two different CRC generation algorithms. The read I/O workload size ranges from 512B to 8KB. The horizontal axis represents the read I/O size, while the vertical axis shows the total cycles spent on computing a CRC over that I/O size.

As seen in the graph, the Slicing-by-8 algorithm requires lesser cycles than the Sarwate algorithm to compute the CRC at all data points. Specifically, for the 8KB data point, the Slicing-by-8 algorithm takes about 2.15 cycles per byte to generate a CRC value, while the Sarwate algorithm takes 6.55 cycles per byte. Thus, Slicing-by-8 accelerates the CRC generation process by a factor of 3, as compared to Sarwate. The Sarwate algorithm requires executing 35 IA32 instructions in order to validate 32 bits of data whereas the Slicing-by-8 algorithm requires 13 instructions only. This is the reason why the Slicing-by-8 algorithm is three times faster than the Sarwate algorithm.

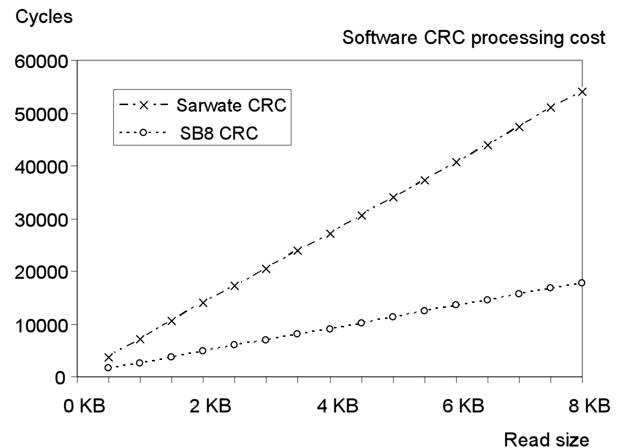

Figure 14: SB8 CRC versus Sarwate CRC

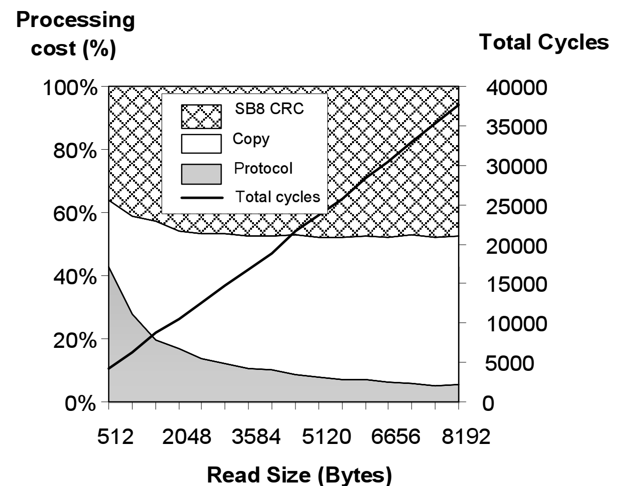

Figure 15: Data-in PDU processing cost (with SB8 CRC)

Figure 15 shows the performance profile characterizing the processing of a data-in PDU where the CRC generation algorithm is Slicing-by-8. Operating at 2.15 cycles per byte, the Slicing-by-8 algorithm demonstrates the same cost as the data

copies. For the 8KB PDU size, each of data copies and CRC generation represent about 47% of the total processing cost, while the protocol processing is about 5%. Comparing these numbers with the costs of Figure 11 (similar profile but with the Sarwate CRC algorithm), we can see that the total processing cost for an 8KB PDU has now decreased from 73761 cycles to about 37579 cycles, resulting in two times faster iSCSI processing.

We also performed a second group of experiments to evaluate the impact of interleaving data copies with the CRC generation process. We modified our sandbox implementation in order to support a new iSCSI/TCP interface and the interleaved copy-CRC operations as described in Section 3.3.

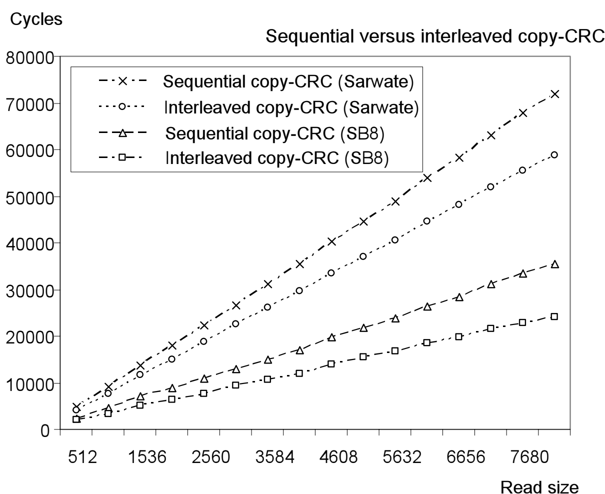

Figure 16: Interleaved Copy-CRC cost

Figure 16 shows the measured cost of sequential copy and CRC operations for both the Sarwate and Slicing-by-8 algorithms while executing iSCSI reads. It also shows the interleaved copy-CRC cost. As seen in the figure, for both CRC algorithms, interleaving copy with CRC reduces the overall cycle cost for the two operations. For the interleaved Sarwate CRC and copy, the total cycle reduction is about 13000 cycles or 18%. For the interleaved Slicing-by-8 CRC and copy, the cycle reduction is about 11570 cycles corresponding to a performance improvement of 32%.

We then extended our measurements and analysis to an entire storage stack consisting of our iSCSI implementation, and an optimized TCP/IP stack implementation. Our goal was to understand the impact of the optimizations (i.e., the Slicing-by-8 algorithm and the interleaved copy-CRC) on the performance of the entire stack. Figure 17 shows the overall throughput as seen at the iSCSI layer for different values of a read I/O workload size. The x-axis represents the size of the read command, while the y-axis shows the achieved iSCSI throughput in MB/s across the range of read sizes. The topmost line depicts the maximum iSCSI line rate that can be achieved for a 10Gbps Ethernet link. As seen in the figure, the iSCSI throughput improves from 175 MB per second (or 1.4 Gbps) to about 445 MB per second (or 3.6 Gbps) when both the optimizations are turned on.

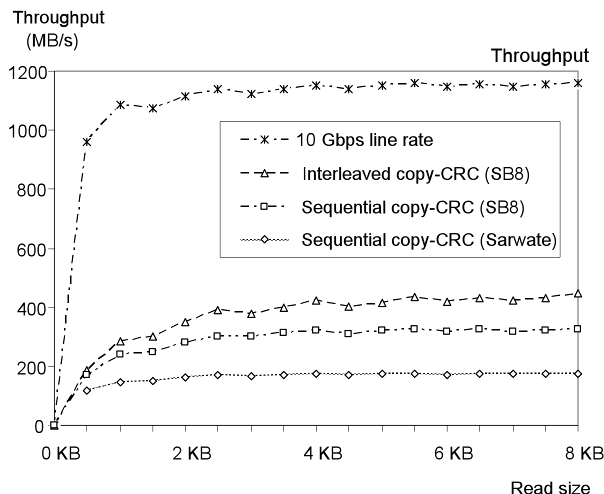

Figure 17: Projected software Read throughput

To further evaluate our Slicing-by-8 algorithm, we measured the impact of Slicing-by-8 on a real-world Linux iSCSI stack developed by UNH [32]. We modified this implementation by replacing the Sarwate algorithm with the Slicing-by-8 algorithm. Our experimental environment consisted of a 3 GHz single-threaded uni-processor Xeon server board with a 533 MHz FSB and a 64bit/133 MHz PCI bus for the initiator. The initiator was running Linux kernel 2.4.20. The iSCSI initiator stack was connected to 3 UNH iSCSI targets through gigabit NICs. We measured the normalized throughput (bits/Hz) of iSCSI with the two CRC algorithms and found that for an 8KB I/O workload, replacing the Sarwate algorithm with Slicing-by-8 results in an increase in the normalized throughput by 15%.

The reason why Slicing-by-8 does not result in a similar improvement like the one demonstrated in the sandbox environment is because the TCP/IP and SCSI overheads are significant in the Linux 2.4 kernel [15]. The 2.6 kernel, though, supports a more optimized implementation of the TCP/IP and SCSI protocols. Unfortunately at the time of writing we did not have access to a modified 2.6 Linux kernel with TCP/IP optimizations. We believe that, as TCP/IP stack implementations become more optimized in the future the CRC overhead in iSCSI will stand out, and the impact of Slicing-by-8 will be greater. As future work, we plan to test our optimizations using the 2.6 kernel.

6. Related Work

Several studies on the performance of iSCSI have been published [1, 14, 15, 17, 21, 24, 27, 28]. These studies have focused on comparing software with hardware implementations in LAN and WAN environments. Sarkar et al [27] compared the performance of three competing approaches to implementing iSCSI. Aiken et al [1] evaluated a commercial implementation of iSCSI and found it to be competitive with a Fiber Channel interface at gigabit speeds. Khosravi et al [15] studied the architectural characteristics of iSCSI processing including CPI, cache effects, and instruction path length. Our work builds on all the previous work described above. We implemented our own iSCSI fast path and demonstrated a performance improvement on the iSCSI processing throughput using two new optimizations.

Radkov et al [24], and Lu et al [17] have compared iSCSI and NFS/SMB-based storage. Magoutis et al [19] performed a thorough comparison and analysis of the DAFS and NFS protocols. Our work is narrower in scope in that it looks only at iSCSI-based storage, but takes a much deeper dive into the performance issues of iSCSI.

Efficient implementation of the CRC generation process has been the subject of substantial amount of research [2, 3, 7-10, 12, 20, 25, 29, 31, 34]. Software-based CRC generation has been investigated in [8-10, 12, 25, 29, 34]. Among these algorithms the most commonly used today is the one proposed by Sarwate [29]. Feldmeier [8] motivated by the fact that table-driven solutions are subject to cache pollution presented an alternative software technique that avoids the use of lookup tables. Our algorithm is distinguished from [8, 9, 25, 29, 34] by the fact it can ideally read arbitrarily large amounts of data at a time.

The concept of parallel table lookups which we use in our algorithm also appears in early CRC5 implementations [10] and in the work done by Braun and Waldvogel [3] on performing incremental CRC updates for IP over ATM networks. Our work is distinguished from [3, 10] in that our algorithms reuse the same lookup tables in each iteration, thus keeping the memory requirement of CRC generation at reasonable level. On the other hand, if the contribution of each slice to the final CRC value is computed using the square and multiply technique as in the work by Doering and Waldvogel [7], the processing cost may be too high in software.

Our algorithm also bears some resemblance with a recent scheme published by Joshi, Dubey and Kaplan [12]. Like our algorithm the Joshi-Dubey-Kaplan scheme calculates the remainders from multiple slices of a stream in parallel. The Joshi-Dubey-Kaplan scheme has been designed to take advantage of the 128-bit instruction set extensions to IBM's PowerPC architecture. In our contrast our algorithm does not make any assumptions about the instruction set used.

7. Conclusions and Future Work

In this paper, we report on an in-depth analysis of the performance of IP-based networked block storage systems based on an implementation of the iSCSI protocol. Our data shows that CRC is by far the biggest bottleneck in iSCSI processing, and its impact will increase even further as TCP/IP stacks become more optimized in the future. We demonstrate significant performance improvement through a new software CRC algorithm that is 3 times faster than current industry standard algorithm and show that enhancements to the iSCSI/TCP interface can result in significant performance gains.

We expect that in the future, iSCSI performance will demonstrate near linear scaling with the number of CPU cores available in a system and will support data rates greater than what a single 10 Gigabit Ethernet interface will provide. Three factors lead us to this conclusion: (i) the increasing commercial availability of dual-core and multiple-core CPUs; (ii) evolving operating system technologies such as receive-side scaling that allow distribution of network processing across multiple CPUs; and (iii) the use of multiple iSCSI connections between an initiator and one or more storage targets as supported in many iSCSI implementations today. In the future, we would like to extend our analysis to study the performance and scalability of iSCSI across multiple CPU cores, as well as the application-level performance of iSCSI storage stacks for both transaction-oriented applications as well as backup and recovery applications.

References

[1] S. Aiken, D. Grunwald, and A. Pleszkun, "A performance analysis of the iSCSI protocol", *Proceedings of the Twentieth IEEE/NASA International Conference on Mass Storage Systems and Technologies (MSST 2003),* San Diego, CA, April, 2003.

[2] G. Albertengo, and R. Sisto, "Parallel CRC Generation", *IEEE Micro*, Vol. 10, No. 5., pg. 63-71, 1990.

[3] F. Braun, and M. Waldvogel, "Fast Incremental CRC Updates for IP over ATM Networks", *Proceedings of the IEEE Workshop on High Performance Switching and Routing (HPSR 2001),* Dallas, TX, May, 2001.

[4] J. Chase, A. Gallatin, and K. Yocum, "End-System Optimizations for High-Speed TCP", *IEEE Communications Magazine*, Vol. 39, No.4, pg. 68-74, 2001.

[5] D. Clark, V. Jacobson, J. Romkey, and H. Salwen, "An Analysis of TCP Processing Overhead", *IEEE Communications Magazine*, Vol. 27, No.6, pg. 23-29, 1989.

[6] D. Clark, and D. Tennenhouse, "Architectural Considerations for a new generation of protocols", *Proceedings of the ACM symposium on Communications Architectures and Protocols*, Pennsylvania, PA, September 1990.

[7] A. Doering, and M. Waldvogel, "Fast and flexible CRC calculation", *Electronics Letters*, Vol. 40, No.1, pg. 10-11, 2004.

[8] D. Feldmeier, "Fast Software Implementation of Error Correcting Codes", *IEEE/ACM Transactions on Networking*, Vol. 6, No.3, pg. 640-651, 1995.

[9] G. Griffiths, and G. C. Stones, "The Tea-leaf Reader Algorithm: An Efficient Implementation of CRC-16 and CRC-32", *Communications of the ACM*, Vol. 30, No. 7, pg. 617-620, July 1987.

[10] C. M. Heard, "AAL2 CPS-PH HEC calculations using table lookups", *Public Domain Source Code*, available at ftp://ftp.vvnet.com/aal2_hec/crc5.c

[11] "IA-32 Intel Architecture Software Developer's Manual, Volumes 2A and 2B: Instruction Set Reference", ftp://download.intel.com/design/Pentium4/manuals/

[12] S. M. Joshi, P. K. Dubey, and M. A. Kaplan, "A New Parallel Algorithm for CRC Generation", *Proceedings of the International Conference on Communications*, New Orleans, LA, June, 2000.

[13] H. Keng and J. Chu, "Zero-copy TCP in Solaris", *Proceedings of the USENIX 1996 Annual Technical Conference*, San Diego, CA, January, 1996

[14] H. Khosravi, and A. Foong, "Performance Analysis of iSCSI And Effect of CRC Computation", *Proceedings of the First Workshop on Building Block Engine Architectures for Computers and Networks (BEACON 2004)*, Boston, MA, October, 2004.

[15] H. Khosravi, A. Joglekar, and R. Iyer, "Performance Characterization of iSCSI Processing in a Server Platform", *Proceedings of the Twenty Fourth IEEE International Performance Computing and Communications Conference (IPCCC 2005),* Phoenix, AZ, April 2005.

[16] M. E. Kounavis and F. Berry, "A Systematic Approach to Building High Performance, Software-based, CRC Generators", *Proceedings of the Tenth IEEE International Symposium on Computers and Communications (ISCC 2005),* Cartagena, Spain, June, 2005.

[17] Y.Lu, and D. Du, "Performance Study of iSCSI-Based Storage Subsystems", *IEEE Communications Magazine*, Vol. 41, No. 8, pg. 76-82, 2003.

[18] K. Magoutis, S. Addetia, A. Fedorova, and M. Seltzer, "Making the Most out of Direct-Access Network Attached Storage", *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST 2003)*, San Francisco, CA, 2003.

[19] K. Meth, and J. Satran, "Features of the iSCSI Protocol", *IEEE Communications Magazine*, Vol. 41, No. 8, pg. 72-75, 2003.

[20] M. C. Nielson, "Method for High Speed CRC computation", *IBM Technical Disclosure Bulletin*, Vol. 27, No. 6, pg. 3572-3576, 1984.

[21] W. T. Ng, H. Sun, B. Hillyer, E. Shriver, E. Gabber, and B. Ozden, "Obtaining High Performance for Storage Outsourcing", *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST 2002),* Monterey, CA, 2002.

[22] "Open iSCSI project", *Public Domain Source Code*, available at http://www.open-iscsi.org/

[23] A. Perez, "Byte-wise CRC Calculations", *IEEE Micro*, Vol. 3, No. 3, pg. 40-50, 1983.

[24] P. Radkov, L. Yin, P. Goyal, P. Sarkar, and P. Shenoy , "A Performance Comparison of NFS and iSCSI for IP-Networked Storage", *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, San Francisco, CA, March 2004.

[25] T. V. Ramabadran, and S. V. Gaitonde, "A Tutorial on CRC Computations", *IEEE Micro*, Vol. 8, No. 4, pg. 62-75, 1988.

[26] G. Regnier, S. Makineni, R. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, and D. Newell, "TCP Onloading for Datacenter Server: Perspectives and Challenges", *IEEE Computer Magazine*, Vol. 37, No. 11, pg. 48-58, November 2004.

[27] P. Sarkar, S. Uttamchandani, K. Voruganti, "Storage over IP: When Does Hardware Support Help?", *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST 2003)*, San Francisco, CA, March 2003.

[28] P. Sarkar, and K. Voruganti, "IP Storage: The Challenge Ahead", *Proceedings of the Nineteenth IEEE/NASA International Conference on Mass Storage Systems and Technologies (MSST 2002),* College Park , MD, April, 2002.

[29] D. V. Sarwate, "Computation of Cyclic Redundancy Checks via Table Lookup", *Communications of the ACM*, Vol. 31, No 8, pg.1008-1013, 1988.

[30] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner, "Internet Small Computer Systems Interface (iSCSI)", *RFC 3720*, April 2004.

[31] M. D. Shieh, M. H. Sheu, C. H. Chen and H. F. Lo, "A systematic Approach for Parallel CRC Computations", *Journal of Information Science and Engineering*, Vol. 17, pg. 445-461, 2001

[32] UNH iSCSI project, *Public Domain Source Code*, available http://unh-iscsi.sourceforge.net/

[33] K. Voruganti, and P. Sarkar, "An Analysis of Three Gigabit Storage Networking Protocols", *Proceedings of the Twentieth IEEE International Performance Computing and Communications Conference (IPCCC 2001),* Phoenix, AZ, April 2001.

[34] R. Williams, "A painless guide to CRC Error Detection Algorithms", *Technical Report*, available at: ftp://ftp.rocksoft.com/papers/crc_v3.txt, 1993.

Intel and Pentium are trademarks or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries. Copyright © 2005, Intel Corporation.