An Architecture for Software-Based iSCSI: Experiences and Analyses

Annie Foong, Gary McAlpine, Dave Minturn,
Greg Regnier, and Vikram Saletore

Intel Corporation, 2111 NE 25th Ave, Hillsboro, OR 97124
{annie.foong, gary.l.mcalpine, dave.b.minturn,
greg.j.regnier, vikram.a.saletore}@intel.com

Abstract. Supporting multi-gigabit/s of iSCSI over TCP can quickly saturate the processing abilities of a SMP server today. Legacy OS designs and APIs are not designed for the multi-gigabit IO speeds. Most of industry's efforts had been focused on offloading the extra processing and memory load to the network adapter (NIC). As an alternative, this paper shows a software implementation of iSCSI on generic OSes and processors. We discuss an asymmetric multiprocessing (AMP) architecture, where one of the processors is dedicated to serve as a TCP engine. The original purpose of our prototype was to leverage the flexibility and tools available in generic systems for extensive analyses of iSCSI. As work proceeded, we quickly realized the viability of generic processors to meet iSCSI requirements. Looking ahead to chip-multiprocessing, where multiple cores reside on each processor, understanding partitioning of work and scaling to cores will be important in future server platforms.

Keywords: iSCSI, Asymmetric Multiprocessing, TCP optimization.

1 Motivation

The concept of Internet Protocol (IP)-everywhere is appealing – the same infrastructure and expertise can be deployed across all networks. Additionally, the economies of scale that is available through commodity IP networking infrastructure (GbE adapters, switches, cables) makes deployment affordable. While IP network file-based protocols (e.g. NFS, CIFS) had been around for years, many enterprise workloads, such as database systems and high definition streaming video, are optimized for direct access to block storage. The recently defined iSCSI (SCSI over IP) standard is an alternative to currently deployed FC-based storage area networks (SAN) and offers the potential of an IP-converged SAN. Although iSCSI is transport-agnostic, a current workable implementation is most likely to be deployed over TCP. To achieve IP-converged cluster deployments, the performance and scalability of iSCSI must approach that of FC SANs. We recognized (and shall show in this paper), that the major overhead in iSCSI is not iSCSI itself, but TCP. In the case of FC, the bulk of protocol processing is offloaded to hardware on the host-bus adapters (HBAs).

As such, implementers in the IP space have envisioned that a reasonable solution would be a similar hardware-assisted approach, and offload TCP and/or iSCSI to TCP offload engines (TOEs) and iSCSI HBAs respectively. However, hardware implementation is difficult and fraught with errors [17]. Interfaces between host and engines are crucial, but are typically not well understood [12]. Finally, TCP has a far more complex state machine than other transports. Unlike FC which is designed specifically for hardware implementation from ground up, TCP began life as a software stack. Corner cases abound that are not so easily addressed if the solutions are hardwired. Researchers [12, 17] also reasoned that the complex NIC chips often lag behind the performance of generic processors that tend to ride Moore's law.

We, therefore, chose to focus on an architecture for iSCSI in *software*. Three major trends motivated our direction: (1) Commercially viable chip multiprocessors (CMP), where increased processing power is achieved through multiple cores per CPU, rather than clock speed ramp [8]; (2) Integration of the memory controller on the CPU die will effectively scale memory bandwidth with processing power; (3) Huge strides in bus bandwidth improvement. (2) & (3) can potentially remove the chipset and bus as system bottlenecks. The availability of many cores in CMP can potentially breach the proverbial cpu-memory gap if software is optimally re-architected for thread-level parallelism. To effectively leverage many cores for networking becomes the key to designing future CMP servers.

2 Approach

Network protocol stacks of general-purpose monolithic operating systems, are known for their inability to scale well in SMPs [11, 22]. We take an asymmetric multiprocessing (AMP) approach, where one of the processors is dedicated to serve as a TCP engine. By separating the protocol processing from the rest of the operating system (OS), we hope to provide a clearly-defined sandbox whereby protocol processing can progress independent of limitations incurred by generic OSes. Additionally, we incorporated well-known network optimization techniques, including zero copy and asynchronous interfaces. We recognize the impact of effective application interfaces on network performance. In particular, existing iSCSI implementations [21] use BSD-style sockets, and force SCSI (which is inherently asynchronous) to be built upon synchronous interfaces. We therefore took a full iSCSI stack implementation and re-architected it from ground up. We modified a reference implementation of iSCSI to use an asynchronous sockets-like API that we built. This allowed us to perform an experimental-based evaluation of our APIs and overall architecture as compared to current practices. Research questions of interest to us in this study are: *1) What is the processing requirements of iSCSI in SMP mode ? 2) What parts of iSCSI processing can benefit from the AMP model ? 3) What constitutes an optimal architecture for software-based iSCSI ?*

We give a brief overview of our software-based architecture in Section 3. In Section 4, we discuss our experimental work and how iSCSI uses asynchronous interfaces. In section 5, we present in-depth performance analyses of iSCSI processing requirements on SMP Linux and our AMP prototype. By separating the stack into functional bins, we are able to pinpoint exactly where and why the AMP model makes a difference. Wherever possible, we share the hands-on experiences and lessons learnt from this work. We conclude with related and future research.

3 Overview of Architecture

Full architectural details of our AMP prototype were described in [16]. In that work, our prototype was functional to a point where limited runs of one-way bulk data transfers was possible. The work presented here is work done well beyond that phase. Fig 1(a) shows a typical iSCSI implementation over generic (2P) SMP Linux. The image footprint is duplicated symmetrically across the 2 processors. Our iSCSI/AMP architecture takes a hard affinity, asymmetric viewpoint whereby application and network processing are partitioned along very deliberate lines (Fig 1b). We refer to the 'host' as the processor where the generic OS and applications reside. The 'packet processing engine' (PPE) is where network protocol processing (i.e. TCP/IP) is performed. The PPE in our prototype is a loadable Linux module consisting of only the TCP/IP stack. Once inserted, the PPE goes into a poll loop and never yields the processor. Here lies the crux of our AMP design: The hard partitioning allows the PPE to continuously poll for work from the NICs or the host. *The PPE does not get interrupted by devices, nor scheduled by the host OS.* The PPE runs without interference from the OS and devices. The poll is performed on shared memory. The PPE can poll NIC descriptors for synchronization, without causing memory bus traffic, until the cache-lines of the associated shared memory is modified. Finally, since the PPE determines its own path of execution, it can predict memory usage and pre-fetch accurately. The host to PPE interface is implemented as a set of asynchronous queues [4] in cache-coherent, shared host memory. The doorbell queue is emulated in software and is the mechanism for the host to inform the PPE that work has been posted on the work queues.

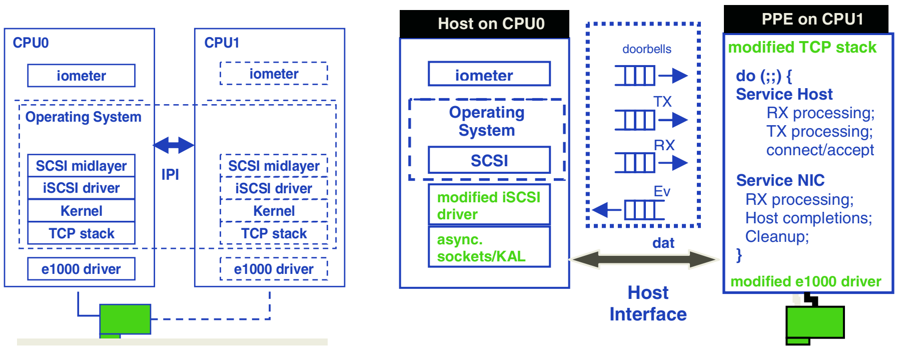

Fig. 1. (a) iSCSI in SMP mode (Linux) (b) iSCSI in AMP mode

Conversely, the event queue is a means for the PPE to inform the host that work has been completed. Furthermore, we have specifically designed our host-engine interface to remove locking. Our queues are one-way (e.g Host only posts to work queues; PPE reads from work queues). This explicit producer-consumer model completely eliminates locking between PPEs and hosts. In the general case with multiple hosts and PPEs, locking will only be required between host to host, and PPE to PPE. Looking ahead to Receive-side scaling enabled NIC implementations [22], which have the ability to parse and direct flows to specific processors/cores, there will be minimal need for locking among PPEs. The programming interface is an asynchronous, sockets-like interface, over which we have built over iSCSI implementation. A Kernel Adaptation Layer (KAL) provides the necessary shim that hides the innards of work and event queues manipulation from the host application. This interface also enables pre-posting, and is essential to the implementation of zero-copy. There is true zero-copy on the transmit path (loosely based on *tcp_sendpage()* used by in-kernel applications in Linux); and zero-copy from the host's point of view on the receive path. The PPE does the copy on behalf of host.

4 Implementation Details

4.1 Lessons in Experimental Setup

Our initiator is a system with 2P Intel® Xeon™ 2.4GHz processors on a 400 MHz front-side bus. The OS is the Linux-2.4.18 kernel. 2 standard Intel Pro1000 GbE controllers, on a 64bit/133MHz PCI bus, are used to match the performance of 2Gbps FC. The base iSCSI initiator/target code is from SourceForge [21]. We used IOmeter [19] to exercise read/write transactions on the SCSI layer. We modified dynamo (IOmeter's driver) to use raw IO in order to turn off buffer caches on the initiator. Operating in this mode forces a serialization of requests in the SCSI midlayer. Because of various serializations throughout a complex system typical of any iSCSI setup, we went through great lengths to ensure that bottlenecks, other than iSCSI processing, are removed. We ran 2 instances of dynamo to parallelize block requests and get around Linux's SCSI serialization. We configured IOmeter and our iSCSI initiator driver to issue multiple outstanding IO commands. This functionality is necessary to circumvent the roundtrip latencies of TCP. However, the iSCSI targets we used did not have such support. We emulated such support through over-provisioning socket buffers (set to 128KB) on our targets. This workaround worked for our particular one-initiator (one connection) to on-target setup. Finally, we used RAM disks on the targets to remove dependencies on disk IO speeds.

4.2 iSCSI's Usage of Asynchronous APIs

iSCSI encapsulates SCSI command data blocks into TCP/IP packets that can be sent over IP networks [21]. The iSCSI protocol is an end-to-end transaction-based protocol, very much like SCSI. It goes through phases of authentication and discovery before entering the full feature phase where data is transferred. iSCSI is typically implemented as a low-level driver, and may be called in interrupt context. Polling for events would be inefficient, and explicit blocking is not an option. Since processors must not block waiting for disks, the SCSI midlayer is asynchronous. However, most iSCSI implementations are built over BSD synchronous sockets, and require additional threads to emulate a non-blocking iSCSI driver **Fig. 2**(a). At least 2 threads are required per connection. Substantial synchronization among threads will become an issue as the number of connections increases. Asynchronous APIs (**Fig. 2**(b)) simplifies the picture. Transmits are posted asynchronously without the need for a token thread. We have used a receive thread (RX) in this prototype version. The final version, using an event callback mechanism, removes the need for a RX thread altogether. An asynchronous interface allows our iSCSI implementation two ways of waiting for an event: (1) by *polling* the event queue or (2) by *explicit blocking* (sleeping) if there are no pending events. In the latter case, an inter-processor interrupt (IPI) is generated only if it is necessary to wake the sleeping process. The ability to either poll or sleep is a powerful tool that allows the user to control its own response to events.

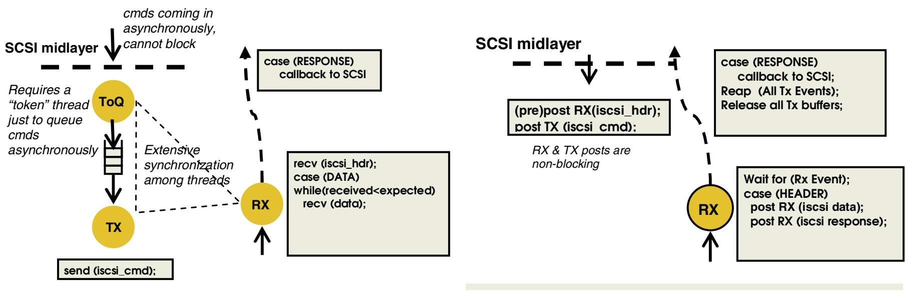
cmds coming in asynchronously, cannot block
SCSI midlayer
Requires a "token" thread just to queue cmds asynchronously
ToQ
Extensive synchronization among threads
TX
RX
send (iscsi_cmd);
case (RESPONSE) callback to SCSI
recv (iscsi_hdr); case (DATA) while(received<expected) recv (data);
SCSI midlayer
(pre)post RX(iscsi_hdr); post TX (iscsi_cmd);
RX & TX posts are non-blocking
case (RESPONSE) callback to SCSI; Reap (All Tx Events); Release all Tx buffers;
Wait for (Rx Event); case (HEADER) post RX (iscsi data); post RX (iscsi response);
RX

Fig. 2 (a). iSCSI reads over synchronous APIs (b) asynchronous APIs

While iSCSI HBA implementation deals with entire requests, our software-based iSCSI has an inherent byte-stream nature to its user interface (both synchronous and asynchronous versions of sockets). The non-existent message boundary makes it difficult to determine the granularity of events (i.e. when work is considered as done). Along similar lines, we noted in our work the awkwardness of pre-posting for reads without a direct data placement [23] awareness in the PPE. During an iSCSI read, we can at best pre-post for the reading of the iSCSI header (typically 48B). It is on parsing the header contents that we know how much more to receive. If direct data placement is available, the initiator would have pre-posted enough buffers to handle the headers, data payloads, and command statuses. On correct placement of all packets, the PPE would issue 1 event that signals the completion of an entire request.

In our particular implementation, we emulated this behavior by limiting our initiator to communicate with only one target per connection. We depended on the socket buffers as the command and data pipeline, so that throughput does not suffer. In this way, we can pre-post buffers, for both the header and data, at command issue time. In our case, we know that whatever comes back is for the current outstanding command. This is not generally applicable since responses (i.e. associated data) to commands can, and will come out of order. Even with this optimization, there is not an appreciable improvement. The *wait()* semantics that our prototype implemented only allowed for the reaping of one event. Every posting of work must be accompanied by a corresponding *wait()* (or *poll()*). In the worst case, this may translate to an interrupt from the PPE for every event. Working with iSCSI exposed the need for a *wait_N()* functionality (i.e. wait for N events before waking the host). With *wait_N(),* an application can better coalesce events according to what was previously posted. E.g. *Wait_2()* would wake the host only on the arrival of both headers and data. The earlier issue of byte-stream semantics still haunts us in the PPE, and we are unable address it unless a framing protocol [23] is added to TCP.

5 Performance Analysis

In this section, we present a series of results comparing the performance of iSCSI over generic SMP Linux and our AMP prototype for sequential iSCSI reads/writes of 8KB and 64KB requests. We also noted frequency scaling characteristics to project to future processors. Finally, we used Oprofile (a profiler based on event sampling) [20] to determine the breakdown of processing hotspots and perform in-depth analyses to fully understand performance impact and provide reasons behind them.

5.1 Throughput and Utilization

Fig 3(a) shows improved bandwidth and reduced CPU utilization of AMP over SMP. Worthy of note is the AMP prototype's ability to push throughput at line rate (~200MB/s). A more illuminating view is to look at the cost metric of GHz/Gbps (i.e. cycles per bit transferred) in Fig 3(b). The AMP model is 30-40% more efficient than SMP. The ability (or inability) to scale across processor frequencies exposes dependencies on non-processing components (Fig 4).

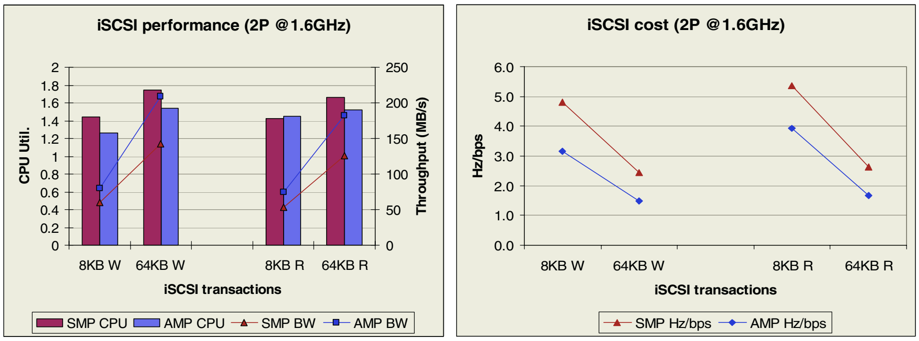

Fig. 3. (a) iSCSI throughput & CPU utilization (2P full utilization is 200%) (b) iSCSI cost

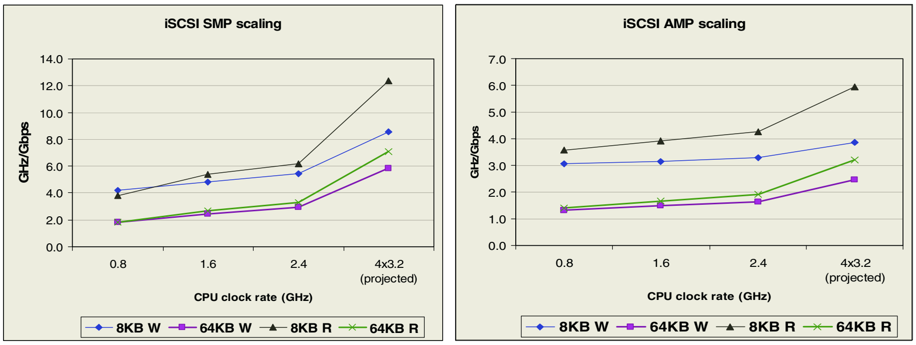

Fig. 4. SMP versus AMP scaling (*the flatter than lines, the better the scaling*)

5.2 Hotspot Profiles

The SMP model implies that the image footprint is duplicated exactly across processors. However, it does not imply exact load balancing. In Windows NT and Linux default configurations, interrupts go only to CPU0 [2, 22]. Scheduling is dependent not only on running processes, but also on interrupts coming in from devices. (**Fig 5, Fig 6**) give a visual representation of iSCSI processing distribution for the 2 models. Exact percentages will be given in the next section. We have carefully binned Linux functions into logical layers and abstracted processing to a level where analysis gave useful insights. As much as we can, we have separated the compute-intensive parts of TCP protocol processing (*TCP*), i.e. the cranking of the state machine, from the kernel support, memory-intensive parts of TCP processing (*kernel*). Kernel support includes memory/buffer management routines, the manipulation of TCP contexts, timers, etc. A full implementation of the sockets interface includes not only the obvious BSD sockets API, but also system calls, and schedule-related routines. This is how an application causes a socket action to be executed from the user level all the way down to the TCP stack. We put all these functions into the *schedule* bin. *Interrupts* refer to NIC interrupt processing, which logically also belongs to interfaces. We have separated them here to showcase that our PPE does not take device interrupts. For AMP, *q-processing* roughly takes the place of interrupts and system calls. *Driver* refers to the NIC driver routines. *Others* refer to miscellaneous user routines including runtime libraries and Oprofile. Finally, *copies* are of movement of payload data. This allows us to contrast the one-copy path of standard OS with our prototype's zero-copy implementation on writes. We have established that the major hotspots of iSCSI processing are in *SCSI*, *TCP* engine, *kernel* support and *copies*. iSCSI (without data integrity computation) takes less than 5% of processing. Because of Linux 2.4 use of a single io_request_lock spinlock for the entire block device system, the SCSI midlayer makes up a much larger overhead.

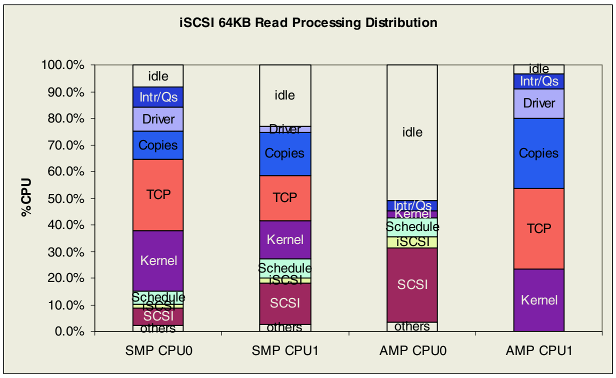

Fig. 5. Profile comparison of iSCSI 64KB reads

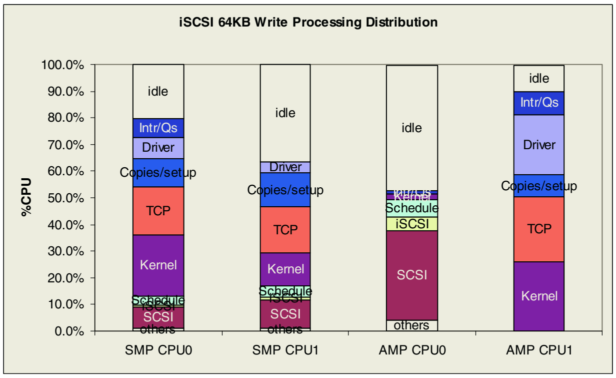

Fig. 6. Profile comparison of iSCSI 64KB writes

5.3 In-depth Analyses

Intuitively, the most notable benefit of hard partitioning is improved cache locality. In this section, we will quantify that statement, and pinpoint exactly where and why improvements occur as we go from the SMP to AMP mode. Also, we have combined interrupts, system calls, queue processing and scheduling appropriately into the *interface* bin. In addition to counting cycles (time), we also noted cache misses and machine clears, as they tend to have the largest performance impact [7].

We begin by first getting a per-CPU view of iSCSI processing (Table 1). All counters have been normalized to work done. There is a reduction of cycles per instruction (CPI) in all bins, between SMP and AMP modes, showcasing the overall higher efficiency in AMP processing. As expected, the *others* and *SCSI* layers, which are untouched, show similar counts in both modes. We believe it is useful to call out that the %distributions by themselves cannot reveal this insight, an evaluation of absolute counts is necessary. Due to space limitation, we will call out analyses using writes as an example (reads can be similarly interpreted)). *TCP* protocol processing has a CPI of 3.7-4.5 for SMP, and 3 on the PPE for AMP. *Kernel* takes 22.5% on CPU0 and 12.3% on CPU1 for SMP. The different loading on CPU is due to the glaringly large difference in the number of machine clears. NIC interrupts go to CPU0, in addition to IPIs from CPU1. *Kernel* support overheads are primarily on the PPE for AMP, once again with much improved CPI. iSCSI "writes" incurs no copies, but do require setup for zero-copy. Zero-copy had essentially transformed *copies* from a "memory-intensive" bin (note cache misses in SMP W) to a "compute-intensive" bin (note number of instructions in AMP W). We should point out that copies on read (under Linux-2.4.x) are implemented via `rep movl` (repeat string moves) which explains the large CPIs (18.7-21.9). Copy for writes is a carefully crafted rolled-out loop that moves data efficiently based on alignment that is known beforehand. Reads were implemented assuming arbitrary arrival of bytes. An optimized version of copy on read had since appeared in Linux-2.6 [18]. The largest improvement in CPI is seen in *interface* (SMP: 10.5-15.1, AMP:1.3-5.2). Once again, the large number of machine clears on CPU0 is notable. Interestingly, *interfaces* do not contribute to large processing hotspots in themselves. But, they have an indirect impact on pipeline and caching behaviors. These overheads are the price a software network stack pays for existing in a generic OS environment. They represent *intrusions* points into the processing path other than themselves, and are tricky to characterize. We found machine clears to be a reasonable counter to quantify such impact. Machine clears (i.e. instruction pipeline flushes) are caused by context switches due to interrupts (e.g. from devices, IPIs, page faults, etc) and scheduling. Characterization of machine clears, therefore, allows us to get a handle on these elusive intrusion points. Evidence of their impact surfaces wherever large number of machine clears occur.

Finally, to provide a *quantifiable value* to our discussions, we combined total processing requirements on both processors and perform a speedup analysis (Table 2). For example, to derive % improvement in the number of machine clears (or other counters) in the *TCP* engine, when going from the SMP to AMP mode is obtained as follows:

$$\%\ Improvement = (clears\text{-}TCP_{SMP} / clears\text{-}total_{SMP}) \times (1 - clears\text{-}TCP_{AMP} / clears\text{-}TCP_{SMP})$$

$clears\text{-}TCP_{SMP}$ = Number of machine clears (per work done) observed in *TCP* on SMP
$clears\text{-}TCP_{AMP}$ = Number of machine clears (per work done) observed in *TCP* on AMP
$clears\text{-}total_{SMP}$ = Total number of machine clears (per work done) observed on SMP

Wherever we see an improvement in cycles, we also see corresponding improvements in cache misses and machine clears on AMP. There is a 11% (*out of a total of 34.6%*) improvement in cycles in *TCP*. Zero copy accounts for 9.4%, showcasing the importance of this optimization. Another 10.2% comes from improvement in *kernel* support. The routines used in *kernel* and *TCP* are essentially the same in either SMP or AMP modes. The improvements come from fewer cache misses in AMP mode. Despite our best efforts, *TCP* still contain memory-related routines (e.g. reading of TCP contexts when calculating window size). The elimination of interrupts and scheduling on the PPE resulted in much fewer machine clears (14.4% fewer in *kernel*, 15.9% fewer in *TCP*, out of a total of 46.8%). Since host and PPE processing are confined to the same processor on the AMP, the number of IPIs needed for synchronization is also reduced. The reason for the improvement seen in *copy* for reads is worth pointing out. In SMP, the bottom and top halves of the TCP stack can potentially be executed on any processor. Copies can incur cache misses on both source and destinations. In the AMP case, sources are always on the PPE, and destinations on the host. Looking back at the per-CPU breakdown in Table 1, we confirmed that the reduction in cache misses is primarily on the host in AMP. Functional partitioning had enabled destinations of *copy* to remain warm in cache.

Table 1. Per CPU characterization of SMP and AMP (normalized)

	CPU0						CPU1					
SMP W	**cycles**	**instr**	**clears**	**L2 misses**	**%CPU**	**CPI**	**cycles**	**instr**	**clears**	**L2 misses**	**%CPU**	**CPI**
idle	8104615	506010	6424	114	20.4%		14580192	1024038	10114	481	36.8%	
others	359423	185096	0	1202	0.9%	1.9	545192	216346	319	956	1.4%	2.5
SCSI	3130769	1086538	3462	2332	7.9%	2.9	3987500	1614183	4345	2374	10.1%	2.5
iSCSI	310385	64904	126	1022	0.8%	4.8	356731	108173	204	913	0.9%	3.3
Interface	4338654	287260	30349	4381	10.9%	15.1	1696154	161058	4537	3840	4.3%	10.5
Kernel	8938462	1676683	24351	11563	22.5%	5.3	4874615	1080529	8395	9531	12.3%	4.5
TCP	7068077	1914663	22260	8347	17.8%	3.7	6935000	1540865	43882	12332	17.5%	4.5
Copies	4142885	1304087	6911	16352	10.4%	3.2	5012115	1657452	7530	19159	12.6%	3.0
Driver	3299423	638221	6683	9099	8.3%	5.2	1660769	371394	4724	2813	4.2%	4.5
Total Non-idle	31588077	7157452	94141	54297			25068077	6750000	73936	51917		

	CPU0						CPU1					
AMP W	**cycles**	**instr**	**clears**	**L2 misses**	**%CPU**	**CPI**	**cycles**	**instr**	**clears**	**L2 misses**	**%CPU**	**CPI**
idle	9109619	681548	9458	122	47.0%		2120190	732738	2036	33	10.1%	
others	786476	290476	738	268	4.1%	2.7	0	0	0	0	0.0%	
SCSI	6494857	2534524	7854	991	33.5%	2.6	0	0	0	0	0.0%	
iSCSI	988857	339286	878	446	5.1%	2.9	0	0	0	0	0.0%	
Interface	1587333	305357	3911	1402	8.2%	5.2	1878762	1429762	1967	991	8.9%	1.3
Kernel	217714	45238	167	229	1.1%	4.8	5537905	1716667	5560	2220	26.3%	3.2
TCP	198667	63095	280	33	1.0%	3.1	5065619	1670238	6152	4875	24.0%	3.0
Copies	0	0	0	0	0.0%		1712667	757143	2616	286	8.1%	2.3
Driver	0	0	0	0	0.0%		4773810	1747619	4768	4970	22.6%	2.7
Total Non-idle	10273905	3577976	13827	3435			18968762	7321429	21063	13342		

	CPU0						CPU1					
SMP R	**cycles**	**instr**	**clears**	**L2 misses**	**%CPU**	**CPI**	**cycles**	**instr**	**clears**	**L2 misses**	**%CPU**	**CPI**
idle	2560159	197421	2103	0	7.9%		7073492	543651	4851	233	21.9%	
others	708254	138889	714	2078	2.2%	5.1	824762	255952	908	1736	2.6%	3.2
SCSI	2165079	657738	2307	1781	6.6%	3.3	5011429	1983135	6156	3775	15.5%	2.5
iSCSI	519524	158730	352	957	1.6%	3.3	647302	230159	521	1161	2.0%	2.8
Interface	4152222	262897	32480	4067	12.7%	15.8	2271905	263889	6161	5427	7.0%	8.6
Kernel	7323333	1277778	18948	10665	22.5%	5.7	4677302	671627	13661	8348	14.5%	7.0
TCP	8747937	1897817	30303	9742	26.9%	4.6	5586667	1093254	34157	10342	17.3%	5.1
Copies	3507619	187500	4866	19286	10.8%	18.7	5331905	243056	7004	28482	16.5%	21.9
Driver	2886032	575397	9792	8373	8.9%	5.0	821270	173611	3249	1920	2.5%	4.7
Total Non-idle	30010000	5156746	99762	56949			25172540	4914683	71815	61190		

	CPU0						CPU1					
AMP R	**cycles**	**instr**	**clears**	**L2 misses**	**%CPU**	**CPI**	**cycles**	**instr**	**clears**	**L2 misses**	**%CPU**	**CPI**
idle	9872667	701190	10208	170	51.1%		737333	268452	673	0	3.5%	
others	660762	289881	551	229	3.4%	2.3	0	0	0	0	0.0%	
SCSI	5420762	2247024	6446	976	28.1%	2.4	0	0	0	0	0.0%	
iSCSI	779048	260119	771	473	4.0%	3.0	0	0	0	0	0.0%	
Interface	2100667	346429	7649	2033	10.9%	6.1	1151143	493452	1235	1321	5.5%	2.3
Kernel	293714	50000	321	369	1.5%	5.9	4927143	1413095	3551	4783	23.5%	3.5
TCP	178571	55952	414	12	0.9%	3.2	6304286	1966667	5345	5485	30.0%	3.2
Copies	0	0	0	60	0.0%		5525524	487500	6313	32057	26.3%	11.3
Driver	0	0	0	0	0.0%		2335810	711905	2378	5521	11.1%	3.3
Total Non-idle	9433524	3249405	16152	4152			20243905	5072619	18821	49167		

Table 2. Comparative analysis of AMP over SMP

64KB W	Improvements in		
	Cycles	L2 Misses	Clears
idle			
others	0.1%	0.8%	-1.5%
SCSI	0.8%	7.9%	-0.1%
iSCSI	-0.4%	0.7%	-1.4%
Interface	3.2%	5.9%	6.3%
Kernel	10.2%	12.3%	14.4%
TCP	11.0%	15.6%	15.9%
Copies	9.4%	8.8%	9.4%
Driver	0.2%	6.2%	3.6%
TOTAL	**34.6%**	**58.2%**	**46.8%**

64KB R	Improvements in		
	Cycles	L2 Misses	Clears
idle			
others	1.3%	0.6%	1.6%
SCSI	2.7%	10.4%	2.6%
iSCSI	0.6%	1.5%	0.2%
Interface	4.9%	7.7%	7.6%
Kernel	10.5%	12.0%	16.3%
TCP	12.1%	16.1%	20.1%
Copies	5.1%	9.9%	6.4%
Driver	2.1%	1.9%	4.7%
TOTAL	**39.3%**	**60.1%**	**59.6%**

6 Related Work

The major overheads of a software TCP stack are well documented [5, 6, 10]. Much of these overheads lie in memory touching operations that do not scale well with processor speeds. More commonly overlooked are the overheads incurred by scheduling and interrupting [10]. To improve cache locality, static affinity [7] and improved scheduling affinity in Linux-2.6 [18] are efforts to direct interrupts and processes to the most appropriate processors. While this works for relatively homogenous applications, dynamically changing applications still pose challenges. As such, other researchers have built prototypes based on functional partitioning [13, 14,15]. An interesting use of functional partitioning [3] has shielded CPUs service only user-defined high priority tasks and interrupts, to deliver real time response using standard Linux. Early works on dedicating a CPU to do specific functions were done primarily on proprietary OS on supercomputers. Processors are dedicated to perform message-passing among nodes [9, 14]. More recently, the AsyMOS project [13] logically attaches general purpose CPUs for the purpose of adding intelligence to devices. They made use of a lightweight device kernel on their device CPU, in place of a full OS. The TCP Servers project [15] defines a generic architecture to offload TCP processing to either processors or nodes. Unlike our focus on removing OS and device intrusions, their focus is the use of a kernel thread dispatcher to dynamically schedule TCP functions onto dedicated processing nodes. Communication between CPUs in [13, 15] is based on inter-processor and/or remote procedure calls, whereas we have implemented our interfaces with queues in cache-coherent shared memory.

7 Conclusion

The ability to process iSCSI at line rate is the key to enabling IP-based storage area networks. Our software implementation, not only achieved line rate performance, but also had the same efficiency as HBAs discussed in [1, 17]. We found that successful, software-based iSCSI architecture must include the following: Zero copy, or exploit thread-level parallelism to hide copy costs; Re-architect applications to leverage well-designed interfaces; and reduce/eliminate device interrupts and OS intrusions.

Working with iSCSI allowed us to gain insights to real usage models to drive API requirements. Due to the clean separation of the TCP engine, lessons learnt here are also equally applicable to offload adapters. We believe that the hard affinity approach gives the upper bound on network performance possible on existing processors and platform architectures. Because of partitioning, we were able to fully control the scheduling of protocol processing. This however does introduce resource balancing issues, where the question of PPEs to host processors ratio must be addressed. The next challenge is to architect an approach that allows for the existence of multiple partitioned PPEs, and be able to optimally balance these resources within the confines of mainstream operating systems. Looking ahead to chip-multiprocessing, where multiple cores reside on each processors, asymmetric partitioning of work to cores provides an effective and viable OS model to multiprocessing.

Acknowledgement

We would like to thank Don Cameron for being the biggest fan of our iSCSI over ETA work; and Jason Fung for his help in consolidating megabytes of data, and carefully abstracting them to a point where sane analysis can proceed.

References

1. S. Aiken, D. Grunwald and P. Andrew. A performance analysis of the iSCSI protocol. In *Proceedings of the 20th IEEE Conf. on Mass Storage Systems and Technologies*, 2003.
2. V. Anand and B. Hartner. TCPIP Network Stack Performance in Linux Kernel 2.4 and 2.5. In *Proc. of the Ottawa Linux Symposium,* Ottawa, 2002
3. S. Brosky and S. Rotolo. “Shielded Processors: Guaranteeing Sub-millisecond Response in Standard Linux”, 4th Real-Time Linux Workshop, Boston, Dec 2002.
4. D. Cameron and G. Regnier. *The Virtual Interface Architecture*. Intel Press, 2002.
5. J. Chase, G. Gallatin, K. Yocum. End system optimizations J. for high-speed TCP. *IEEE Comms, Special Issue on high-speed TCP*, 2001, 39(4).
6. Foong, T. Huff, H. Hum, J. Patwardhan and G. Regnier. TCP performance re-visited. In *Proc of the IEEE ISPASS,* Austin, Mar 2003.
7. Foong, J. Fung, D. Newell, P. Irelan, A. Lopez-Estrada, S. Abraham. Architectural characterization of the impact of Processor Affinity on Network Processing, *to appear IEEE ISPASS*, Mar 2005.
8. L. Hammond, B. Nayfeh and K. Olukotun. A single-chip multiprocessor. *IEEE Computer*, Sept 1997.
9. J. Hsu and P. Banerjee,. A message passing coprocessor for distributed memory multicomputers, In *Supercomputing*, 720 – 729, 1990.
10. J. Kay and J. Pasquale. The Importance of Non-Data Touching Processing Overheads in TCP/IP. In *Proc of ACM SIGCOMM*, 1993.
11. P. Leroux. Building Scalable Networking Equipment Using SMP. *Dedicated Systems Magazine*, 2001.
12. J. Mogul. TCP offload is a dumb idea whose time has come. In *Proc of HotOS IX*, Lihue, May, 2003.
13. S. Muir and J. Smith. AsyMOS: An asymmetric multiprocessor OS. In *Proceedings of OPENARCH '98*, April 1998.

14. P. Pierce and G. Regnier. The Paragon implementation of the NX message passing interface. In *Proc of SHPCC 94*, 1994.
15. M. Rangarajan, A. Bohra, K. Banerjee, E. Carrera, and R. Bianchini. TCP servers: Offloading TCP processing in internet servers. Technical report, Rutgers University, 2002.
16. G. Regnier, D. Minturn, G. McAlpine, V. Saletore and A. Foong. ETA: Experience with an Intel Xeon Processor as a packet processing engine. *IEEE Micro*, Jan 2004.
17. P. Sarkar, S. Uttamchandani, and K. Voruganti. Storage over IP: When does hardware support help ? In *Proc of the 2nd USENIX Conf on File and Storage Technologies*, 2002.
18. M. Meredith, D. Vianney. Linux 2.6 Performance in the Data Center. Linux World Expo, Jan 2004.
19. *Iometer performance Analysis Tool*. http://www.iometer.org.
20. Oprofile: A system-wide profiling tool for Linux. http://oprofile.sourceforge.net.
21. *UNH-iSCSI, Intel iSCSI reference stacks.* http://sourceforge.net/.
22. Scalable Networking: Eliminating the Receive Processing Bottleneck—Introducing RSS. http://www.microsoft.com/whdc/
23. Remote direct data placement protocol. http://www.ietf.org/html.charters/rddp-charter.html