An Efficient Networking Transmit Mechanism for Solaris: Multidata Transmit (MDT)*

Adi Masputra
Internet Engineering
adi.masputra@Sun.COM

Frank Dimambro
Networking & Security
frank.dimambro@Sun.COM

Kacheong Poon
Internet Engineering
kacheong.poon@Sun.COM

ABSTRACT

This paper describes an efficient transmit mechanism in the Solaris networking stack and driver, which reduces per-packet transmit processing overhead. By reducing the costs throughout the layers associated with the transmission path, significant performance gain is achieved on most hardware platforms. We begin by revealing the per-packet transmission costs. We then compare different transmit optimization schemes, and explain our precedence for Multidata Transmit (MDT) technique. Finally, we explain our implementation and its performance.

1. INTRODUCTION

Solaris has often been critized for its moderate bulk data transfer networking performance. In fact, for many years there has been a common conception of formulating the network throughput to directly correlate to the host CPU speed, i.e. 1 megabit per second (Mbps) of network throughput per 1 megahertz (MHz) of CPU speed. Although such paradigm may be sufficient in the past for low-bandwidth network environment, it may not be adequate enough for today's high-speed network mediums, where bandwidth specified in units of gigabit per second (Gbps) is becoming increasingly common.

Networking software overhead can be classified into per-byte and per-packet costs. Previous examination of per-byte data movement costs in Solaris networking stack shows that data copy and checksum overhead dominate host CPU processing time [3]. Our more recent analysis of the per-packet costs reveals that the overhead associated with *stream head*, cache misses, as well as SPARC I/O Memory Management Unit (IOMMU) are as significant as the per-byte costs.

By applying our MDT scheme with minor changes to the networking stack and driver, we are able to reduce the overhead. Significant throughput improvement in TCP bulk data transfer, combined with reduction in the host CPU utilization, are witnessed. On certain hardware platforms and network parameters, we are able to achieve over 2 Mbps of network throughput per 1 MHz host CPU speed, doubling the current rate of transmission.

Although we only implement MDT for TCP/IP, we believe that the design is flexible enough for it to be used to optimize the performance of other networking protocols. We also use Sun's PCI *Gigaswift Ethernet*[1] Network Interface Card (NIC) – which provides a 1 Gbps Ethernet interface – and implement the MDT capability in the driver. We choose to use *Gigaswift Ethernet* since it's the latest gigabit Ethernet card offering that Sun has at the time of writing. We strongly emphasize, however, that nothing in the MDT design requires any hardware-specific features from the NIC, and therefore most of Sun's network drivers (e.g. *hme*, *qfe*, *eri*, *ge*) may be easily modified to support it.

In §2, we explain the per-packet transmit overhead ranging from the stream head[2] down to the driver. In §3 we compare different cost-reduction schemes, and explain why we choose the MDT technique. §4 covers in details the general concepts on MDT and shows how we implement it throughout Solaris. In §5 we describe the software and hardware parameters used for our measurements, and present the results in §6. Finally, we provide possible future enhancements in §7, before providing our conclusions in §8.

2. PER-PACKET TRANSMIT OVERHEAD

This section presents the per-packet costs associated with network data transmission in Solaris.

2.1 Stream Head Associated Costs

The stream head is the end of the stream nearest the user process. All system calls made by user-level applications on a stream are processed by the stream head. In the general networking case, a socket's STREAMS *plumbing* consists of the stream head, the transport module (e.g. TCP or UDP), the network module (e.g. IP or IP6) and the network device driver. Application data residing in the kernel buffers are sent down through each module's queue via the STREAMS framework, by calling the `putnext()` function. The framework determines the destination queue for the message, hence providing a sense of abstraction between the modules.

Stream head copies the application data from user buffer into kernel buffer, and during the *copyin* process, it may divide the data into smaller fragments, based on the `sd_qn_maxpsz`[3] and `sd_maxblk`[4] values. *Stream head* may also reserve some

*Copyright 2002, Sun Microsystems

[1]Internally known as *Cassini*.
[2]Data may originate from places other then the stream head, such as in the case of NFS; we choose to use stream head to illustrate the costs associated with the data source module.
[3]`sd_qn_maxpsz` defaults to `INFPSZ`-1.
[4]`sd_maxblk` defaults to `INFPSZ`.

extra space in front of each allocated kernel buffer depending on the sd_wroff value[5].

Currently, TCP utilizes these parameters in hope of optimizing the transmit dynamics and reducing allocation cost for the TCP/IP and link-layer headers. By setting sd_wroff to a size large enough to hold the headers while setting sd_qn_maxpsz to 2 TCP Maximum Segment Size (MSS)[6] and sd_maxblk to 1 MSS, TCP effectively instructs the stream head to divide the application data into 2 kernel buffers (each sd_wroff+MSS bytes), for every putnext() to the TCP module.

For applications performing bulk data transfer, it's not uncommon to see buffer sizes in the range of 32 KB, 64 KB, or larger [10]. Applications typically informs the TCP/IP stack of these sizes – in order for it to configure and possibly optimize the transmit characteristics – by configuring the send buffer size (SO_SNDBUF) socket option[7].

Ironically for TCP, this strategy has no effect in optimizing the stream head behavior, due to the fact that user buffer is broken up into MSS fragment. For example, a 1 MB user buffer *written* to the socket causes over 700 kernel buffer allocations in the typical 1460-bytes MSS case, regardless of the SO_SNDBUF size. This method is quite inefficient not only because of the costs incurred per allocation, the application data *written* to the socket is no longer kept in larger contiguous fragments.

One way to reduce the number of CPU context switches involved in a write() or send() socket system call is to increase the *copyin* size. Several measurements taken to assess the effects of changing stream head's sd_qn_maxpsz and sd_maxblk for a TCP connection reveal some interesting facts. As shown in Table 1, instructing the stream head to allocate and *copyin* the entire[8] user buffer per write(), by specifying INFPSZ size, introduces a reasonable reduction (around 60%) in the number of CPU context switches (CSW).

However, the measurement also tells us that the number of CPU cross-calls (XCALLS) is significantly increased when the stream head is instructed to *copyin* the user buffer up to maximum allowed size. It turns out that by doing so, we exceed the maximum dblk_t cache size[9], and force the STREAMS message allocator to revert to allocating the kernel buffers using kmem_alloc() instead of obtaining them from the dblk_t cache. When this happens, the memory is allocated from kmem's *oversize arena*, which has no per-CPU cache associated with it and is further protected by a single lock. This may have an impact on the system's scalability factor.

By increasing the maximum dblk_t cache size as well as the sd_qn_maxpsz and sd_maxblk values, TCP's transmission efficiency is increased due to larger kernel buffers without the extra CPU cross-calls. Doing so also reduces the number of calls between the stream head and TCP.

Nevertheless, further improvements can be made by making the maximum dblk_t cache size either a dynamic [5] or a tunable system parameter.

Table 1: CPU Overhead of Blocking write()†

sd_qn_maxpsz/sd_maxblk	Write size (KB)	XCALLS (per sec.)	CSW (per sec.)
2×MSS/MSS	48	324	14573
2×MSS/MSS	64	320	15579
INFPSZ/INFPSZ	48	38796	5596
INFPSZ/INFPSZ	64	36850	6456

†Using *statit* on S10 with *netperf*, between two E-450s (4x400 MHz US-II each).

2.2 Cache Misses

Under most conditions, the networking stack may be able to transmit more than one packets at a time (e.g. TCP bulk data transfer). However, the transmit mechanism throughout the stack and driver does not allow for multiple packet processing in one call.

This approach is suboptimal and does not lend itself to maintaining high instruction execution rate and data locality[10]. Each packet transmission results in traversing the transmit path across different modules, and this process is repeated until all of the packets are sent.

A quick measurement taken during a *netperf* micro benchmark on a Sun Blade 1000 (2x600MHz UltraSPARC-III) reveals a Cycles Per Instruction (CPI) ratio of 2.45[11]. This is far from optimal, considering that the UltraSPARC processor is capable of launching up to four instructions per clock cycle. Although the networking stack is only partly responsible for this inefficiency, improvements made to the way it handles multiple packets will benefit the entire system by reducing the CPU utilization. Our MDT optimization shows that by allowing batches of packets to be processed at a time, 20% reduction in the CPI ratio for the same benchmark is achieved. This directly translates to a higher percentage of CPU idle time that may be used by the system to handle other tasks.

2.3 IOMMU Overhead

SPARC-based platforms supports Direct Virtual Memory Access (DVMA), where the memory management unit is capable of translating a contiguous virtual image (that may be mapped to discontiguous physical pages) into the proper physical addresses. On *sun4d* and *sun4u* architectures, the type of DVMA mapping (STREAMING or CONSISTENT) tells the hardware what conditions must be met, and allows the I/O controller to optimize performance for those conditions.

[5] sd_wroff defaults to 0.

[6] TCP NDD tunable tcp_maxpsz_multiplier defaults to 2 in the currently-developed S10.

[7] SO_SNDBUF in S10 defaults to around 48 KB.

[8] Limited by the strmsgsz tunable value, which defaults to 64 KB in S10.

[9] DBLK_MAX_CACHE size in S10 is 32 KB.

[10] Previous measurements done on the receiving path reveal that the receive mechanism suffers similarly [6].

[11] Using *cpustat*, the UltraSPARC CPU performance counters are used to monitor Cycle_cnt and Instr_cnt events for both user and system.

The STREAMING mapping type is used to optimize for maximum throughput, and specifically exempts the mapped memory from cache coherency requirements (i.e. the I/O controller is allowed to cache and buffer STREAMING data as needed to achieve maximum throughput). Since the data is not kept cache coherent, it is the responsibility of the device driver to flush stale data out of the I/O controller using `ddi_dma_sync()` whenever data in memory is modified.

The CONSISTENT mapping type requires cache coherency, and is appropriate for items which both the hardware and software will access (e.g. descriptor rings). When a DVMA device fetches data from a buffer mapped CONSISTENT the I/O controller goes all the way to main memory. This is slower but ensures the data read is always cache coherent. The driver is still required to call `ddi_dma_sync()` but it will return quickly as there will be no work required [4].

Since the overhead associated with the mapping and flushing vary across different packet sizes, Sun network drivers use different methods of transferring data between the host memory and the hardware as part of their transmit optimization techniques. A mixture of `bcopy()` along with various DVMA transfer methods are used to increase the transmit efficiency [4]. For example, the *Gigaswift Ethernet* driver uses `bcopy()` for tiny packets whose size span is `(0, 256]`. It uses `ddi_dma_addr_bind_handle()` in CONSISTENT and STREAMING mapping types for small-sizes `(256, 512]` and medium-sizes `(512, 1024]`, respectively. For large-sizes `(1024, 1500]`, it uses `dvma_kaddr_load()` which implicitly indicates STREAMING mode. In addition, it calls `ddi_dma_sync` to synchronize the device's view of the memory object (FORDEV) prior to each DMA transfer.

These IOMMU overhead also vary across different types of hardware platforms. Tables 2 presents the DVMA mapping and flushing costs for 1 page of data[12], as well as the derived maximum theoretical link transmit throughput measured across several platforms. While we can't completely eliminate IOMMU overhead altogether, we can reduce the number of IOMMU-related operations down to the minimum. By processing packets whose headers and payloads reside in separate blocks of memory (where each block is virtually contiguous), the number of IOMMU operations can be significantly decreased, because they are now done twice per packet batch (as opposed to one set of operations per individual packet).

3. RELATED WORK

Several attempts have been made to reduce the above per-packet processing costs on both transmit and receive paths of the Solaris networking stack. Some of these efforts concentrate mostly on lowering the inter-module message passing costs associated with `put()` and `putnext()`, while others are related to utilizing specialized firmware on the NIC which performs TCP segmentation upon transmit.

3.1 STREAMS-based Packet Chaining

Packet chaining with STREAMS is one in which multiple packets (each represented by a `mblk_t`) are chained altogether using the existing `b_prev` and `b_next` fields defined in the `mblk_t` structure. Such scheme has several limitations -

- **Complex** – Things can get quite complicated when modifying the STREAMS framework to properly handle and preserve the message chains. Currently, the framework expects a module to *nullify* the two fields prior to its passing the `mblk_t` from one queue to another. This is done mainly because they are used internally by STREAMS to place messages in the synchronization queue (*syncq*). Making STREAMS be aware of the chain concept would require some fundamental changes to the way *syncq* is managed, since chain elements from a given `put()` or `putnext()` operation need to stay intact when the *syncq* is drained.
- **Overloaded** – Numerous modules overload them for their own internal use[13]. For example, several networking modules (e.g. TCP, IP, ARP) utilize these fields to store items for various internal purposes, such as those related to handling urgent messages, maintaining the reassembly, etc.

Because it works on smaller, non-contiguous payload blocks, this scheme does not lend itself well for possible IOMMU optimizations at the driver. Given these constraints and the small performance gain achieved (around 10% [8]) for the amount of changes involved throughout the kernel, such strategy is unlikely to be accepted as efficient.

3.2 Hardware Large Send Offload (LSO)

This is a hardware feature, and is offered by an external Ethernet NIC vendor through specialized firmware. A prototype supporting this feature works by *virtualizing* the link Maximum Transmission Unit (MTU) (typically up to 64 KB) from the networking stack's point of view. This enables the TCP/IP stack to reduce the per-packet transmission costs by the increased virtual packet size. Upon receiving the jumbo packet from the networking stack, the driver instructs the NIC firmware to divide the TCP payload into smaller segments whose sizes are based on the real TCP MSS (typically 1460 bytes). Each of this segment is then transmitted along with the TCP/IP header created by the firmware, based on the TCP/IP header of the jumbo packet [2].

Results from ttcp and SPECweb99 benchmarks using the prototype show that this scheme dramatically reduces the per-packet transmission costs. It provides around 35% CPU utilization reduction in ttcp tests and a 50% improvement in SPECweb99 benchmark. Although the performance gain is substantial, we believe that this scheme does not provide a practical solution, due to the following -

- **It is exclusively tailored for TCP** – This method relies on the firmware's ability to correctly parse and generate the TCP/IP headers (including IP and TCP options). In addition, due to the virtualization of the packet size, many protocols and/or technologies which operate on the *real* headers and payload (e.g. IPsec ESP/AH) will cease to function.
- **It breaks TCP algorithms** – TCP is fooled into

[12] Page size is 8 KB

[13] Adding fields to the `mblk_t` structure will solve this problem, but the impact of doing this is unknown.

Table 2: IOMMU-Related Overhead Measurements

	E-450		Ultra-30	
	CONSISTENT	STREAMING	CONSISTENT	STREAMING
DDI bind/unbind‡	3.76 μsec	5.29 μsec	3.75 μsec	5.11 μsec
DVMA load/unload‡	-	2.37 μsec	-	2.20 μsec
DDI sync (FORDEV)‡‡	0.06 μsec	1.73 μsec	0.05 μsec	1.53 μsec
DVMA sync (FORDEV)‡‡	-	1.6 μsec	-	1.4 μsec
Theoretical Max Tx Bandwidth (DDI)¶	3.17 Gbps	1.73 Gbps	3.18 Gbps	1.83 Gbps
Theoretical Max Tx Bandwidth (DVMA)¶	-	3.05 Gbps	-	3.36 Gbps

‡Mapping values for 1 page [4]. ‡‡Values for 1500 bytes synchronization [4]. ¶For 1514 bytes packets, calculated using only IOMMU overhead.

using a MTU larger than the actual link MTU. Since the two connection endpoints have different notions of the TCP MSS, it inadvertently brings harm to TCP congestion control algorithms.

Even though the above problems are solvable, the solutions involved are somewhat cumbersome and impractrical. Supporting protocols other than TCP over plain IPv4 would require changes made to the firmware, which in itself is already complicated and poses a challenge for rapid software development/test cycle. Furthermore, full conformance to the TCP protocol demands for some fundamental changes to be made to the Solaris networking stack, in order to support the concept of *virtual* and *real* link MTUs, among others.

4. MULTIDATA TRANSMIT (MDT)

The above schemes are expensive, inextensible and address only specific areas of the per-packet transmission overhead. We therefore propose a new generic optimization scheme which overcomes the limitations and costs associated with the aforementioned efforts, in addition to reducing most of per-packet transmission overhead.

As we have shown in §2, the overhead associated with per-packet processing play a big role in determining the network and host CPU utilizations. The approach we have with MDT enables us to achieve a reduction of these costs without requiring any significant design changes throughout the OS, therefore reducing the overall risks. We design it such that the device driver is able to amortize the IOMMU-related costs across a number of packets. With some assistance from the networking stack, the driver needs to only perform the necessary IOMMU operations on two virtually contiguous memory blocks – which represents the headers (e.g. TCP/IP headers) and the payloads of multiple packets – during each transmit.

In order for this to work, the data source module (e.g. the stream head) needs to provide large enough buffers to the transport layer (e.g. TCP). This is done by having TCP instruct the stream head to *copyin* larger portions of the application data into kernel buffers. Because of the larger payload memory blocks (as opposed to the smaller blocks mentioned in §2), TCP can easily take advantage of this since depending upon the *usable* send window size, many segments in these continuous payload blocks may be transmitted in one `putnext()` call. TCP can simply create the TCP/IP headers of these segments – along with the information describing them – in separate memory blocks, and send these three blocks (metadata, headers and payloads) altogether to IP and further down to the device driver.

Our scheme is packet-oriented and protocol-independent, and it places no restrictions as to how each packet is composed throughout the blocks. This decision is left to the modules utilizing MDT. In the following, we describe our design goals and constraints. We then briefly elaborate on our implementation of MDT in the Solaris TCP/IP stack.

4.1 Design Goals

Our design goals for MDT are -

- Avoid having any dependencies on the NIC firmware and/or hardware.
- Avoid making fundamental changes to the STREAMS framework to minimize potential impact on system stability and performance.
- Current applications should benefit from the increase in network throughput and reduced CPU utilization.
- The interface is flexible and protocol-independent.

4.2 Constraints

The constraints involved in achieving our goals are -

- Minimal and localized changes throughout the kernel.
- Compliant with the networking protocol standards.
- Backwards compatible with non-MDT device drivers and other protocols or technologies, e.g. IPsec, IPQoS, CGTP.

4.3 Implementation

During the interface *plumbing* phase, IP probes the data-link layer driver for its link parameters and capabilities [1]. Including MDT as part of the capability queries enables the networking stack to be backwards-compatible with non-MDT drivers.

When the driver supports MDT, IP notifies[14] TCP about this so that it may instruct the stream head to *copyin* larger portions of the application data. TCP takes into account the `SO_SNDBUF` size (rounded up to the nearest TCP MSS) when specifying the `sd_qn_maxpsz` parameter, as well as setting the `sd_maxblk` to INFPSZ, so that the data would not be broken into small blocks. For example, by using a 64 KB `SO_SNDBUF` size, a 64 KB application data *written* to the

[14]This notification happens as part of the *bind* process between IP and TCP for both inbound and outbound connections.

socket will result in the stream head allocating and copying the data into a single 64 KB[15] kernel buffer. In contrast, this would result in over 40 distinct MSS-sized buffers in the non-MDT case.

Since TCP has full knowledge of the number of segments that it can send, it allocates two buffers large enough to hold the metadata and the TCP/IP headers. The metadata contains various properties of the Multidata, such as the packet descriptors, their count, layout, and any optional attributes [9].

This metadata, the header and the payload are then sent down for transmission to IP and the device driver. In some cases, however, IP would need to fall back to the legacy transmission path, which would happen if the packets are routed over a different interface incapable of MDT. In such case, the packets contained within the MDT blocks will be separated into individual units before being fed through the legacy transmission paths. IP would then notify TCP that MDT should be disabled for the current and subsequent connections going over the newly-assigned interface.

Upon receiving the MDT blocks, the driver performs the necessary IOMMU-related operations for the TCP/IP headers portion of the header block as well as the entire payload block. This effectively reduces the number of IOMMU overhead normally associated with single packet transmission, since it now only involves two DVMA mapping and flushing for a given number of packets.

4.4 Operation Steps

The major operations involved in our implementation of MDT with TCP/IP and the *Gigaswift Ethernet* device driver are described as follows.

4.4.1 MDT Capability Probing

Following the DL_CAPABILITY_REQ probing done by IP after the interface goes up, in `ill_capability_ack()` -

1. If a DL_CAPAB_MDT sub-capability response exists, call `ill_capability_mdt_ack`.
2. Process other sub-capabilities.

In `ill_capability_mdt_ack()` -

1. Check for supported MDT interface version.
2. Enable MDT for the interface by marking the appropriate fields in the interface's *ILL* structure.

4.4.2 IP & TCP TPI Binding

Upon receving a T_BIND_REQ in `ip_bind_connected()` -

1. Check if `ip_multidata_outbound` is 1.
2. Check if there's an *IRE* cache for the packet's destination address.
3. If the above conditions are true, call `ip_md_info_append()` to append MDT information to the T_BIND_ACK for TCP to consume.

[15] Using the default `strmsgsz` value (64 KB).

In `ip_md_info_append()` -

1. Check if the *IRE* has a corresponding *ILL*, and that the *ILL* is MDT-enabled.
2. Check if MDT can be enabled for the corresponding *IPC* and *IRE*, based on the following conditions:
   (a) IPsec outbound policy is not present.
   (b) Socket option which causes non-fastpath transmission is not set.
   (c) *IPC*'s Upper Layer Protocol is TCP.
   (d) Outbound IPQoS is not enabled.
   (e) CGTP is not enabled.
   (f) Not a loopback connection[16].
3. Allocate a MDT enable notification message and link it at the end of the T_BIND_ACK message.

When TCP gets the T_BIND_ACK in `tcp_rput_other()` -

1. Look for a MDT notification following the T_BIND_ACK message.
2. If found, call `tcp_mdt_update`.

In `tcp_mdt_update()` -

1. Enable or disable MDT for the TCP connection based on the notification message.
2. If enabled, configure TCP MDT parameters based on those provided by IP.

4.4.3 TCP Transmission

When `tcp_wput_data()` is called -

1. Check if the TCP connection is MDT-capable, and if so, check if it's worthwhile to pursue the MDT transmission path based on the following payload criterias:
   (a) MDT is enabled for this TCP connection.
   (b) IPsec per-connection policy is not persent.
   (c) TCP connection is in established state.
   (d) Usable TCP send window size is at least 1 MSS.
   (e) Contiguous unsent data is at least 1 MSS.
   (f) IP options not present.
2. If the above conditions are met, call `tcp_multisend()`. Otherwise, send the packets using the legacy transmission path by calling `tcp_send()`.

When `tcp_multisend()` is called -

1. Allocate the metadata (M_MULTIDATA) and the TCP/IP header (M_DATA) blocks based on the number of TCP segments that can be transmitted.
2. For each TCP segment, fill in the corresponding TCP/IP headers in the header block.
3. Send them down to IP.

[16] In theory, loopback Multidata can be done, but the we chose not to handle this at the moment.

4.4.4 IP Transmission

When `ip_wput()` is called -

1. Check if the message type is `M_MULTIDATA`. If not, go through the legacy IP transmission path.
2. Check if the *IRE* for the packet's destination address exists, and is associated with an *ILL* that is not MDT-capable. If the IRE doesn't exist, extract and run one packet through `ip_newroute()`. If the ILL isn't MDT-capable (interface has changed), notify TCP to disable MDT for the current and subsequent connection going over this new interface, and run all packets through the legacy `ip_wput()`.
3. Call `IRE_REFHOLD` on the *IRE*.
4. Calculate and fill in the checksum information of each packet if necessary.
5. Fill in the link-layer header (using either `M_DATA` fast-path or DLPI style) for each packet.
6. Send down the blocks to the driver.
7. Call `IRE_REFRELE` on the *IRE*.

4.4.5 Gigaswift Ethernet Driver Transmission

When `ce_wput()` is called -

1. Check if the message type is `M_MULTIDATA`. If not, go through one of the legacy transmission paths.
2. Call `ce_mtx_msg()`.

In `ce_mtx_msg()` -

1. Get number of elements in both the header and payload blocks. The sum of the elements is the maximum number of TX descriptors needed, and is used to obtain those descriptors by calling `ce_reclaim()`.
2. Call `ddi_dma_addr_bind_handle()` on the TCP/IP header portion of the header block to obtain the *header handle*.
3. Call `ddi_dma_sync()` on the *header handle*.
4. Call `ddi_dma_addr_bind_handle()` on the entire payload block to obtain the *payload handle*.
5. Call `ddi_dma_sync()` on the *payload handle*.
6. Place each packet into its corresponding TX descriptors.
7. Call `ddi_dma_sync()` for the TX descriptors.
8. Instruct the hardware to start the DMA transfers.

5. MEASUREMENT PARAMETERS

In the following, we describe our software and hardware setup used throughout our performance measurements.

5.1 OS Environment

We implement MDT with TCP/IP on an early version of Solaris 10. All OS-related tunable parameters are left to their default values. As part of our optimization effort, we make some additional enhancements and fixes to the OS -

- STREAM's `dblk_t` cache pool is enhanced to support larger allocation sizes, by adding larger values to the statically-defined cache sizes[17]. This is needed to support large kernel buffer allocations made at the stream head without incurring potential scalability issues related to CPU cross-calls.
- Several minor fixes related to the TCP receiving path (TCP timer mechanism and TCP *push queue* threshold) are incorporated.
- The transmit-side TCP *synchronous stream* interface is removed, and direct *putnext()* from upper layer is used in all cases.

At the time of writing, the larger `dblk_t` cache sizes is the only feature that's not part of the Solaris 10 code distribution. For fairness, we use the same Solaris kernel (and the *Gigaswift Ethernet* driver binaries) on the test machines[18] to obtain the results of the benchmark programs running with and without MDT. This is made possible due to the boolean switch `ip_multidata_outbound` in the IP module, which enables or disables MDT.

5.2 Network Environment

In all of our measurements, all networking-related tunable parameters (including those related to the *Gigaswift Ethernet* driver) are left to their default values. For clarification, some parameters are explained -

- **Socket sizes** – The default `SO_SNDBUF` and `SO_RCVBUF` sizes are used for benchmarks other than *netperf* and *knetperf*. For these two, various sizes ranging from the default value (48 KB) to 1 MB are used in order to see the effects they have in conjunction with the Bandwidth Delay Product (BDP)[19].
- **TCP MSS** – The default MTU size for IP over Ethernet is 1500, and typically TCP will negotiate a MSS option of 1460 or 1448 bytes – the latter is due to the presence of TCP timestamp option (12 bytes) that's present by default in every TCP segment for window sizes larger than 64 KB.

In addition, the *Gigaswift Ethernet* driver implements a *worker thread* model which distributes the received packets across multiple threads, therefore reducing the time spent in the driver's interrupt thread. This optimization is enabled on machines with 4 CPUs or more.

5.3 Benchmark Programs

Several programs are used for performance evaluation -

- **netperf** – This is a popular sockets-based micro benchmark program for measuring TCP and or UDP throughput. The revision used for our measurement is 2.1pl3, and the test type used is TCP_STREAM.
- **knetperf** – This is another micro benchmark program for measuring *in-kernel* TCP throughput that is developed by Sun [7]. It consists of user applications used

[17]A more elegant solution is the dynamic `dblk_t` allocator[5].

[18]The client machines used in SPECweb99 benchmark are installed with stock S10 kernel.

[19]The *delay* factor also includes the host processing at the two endpoints, in addition to network latency.

for connection setup, and a kernel module pushed on top of TCP (on both ends of the connection) which acts as the data source and sink. A kernel thread is used on the sending side to allocate and transmit the payloads to TCP; the receiving side simply frees them.

- **SPECweb99** – This is a software benchmark product developed by the Standard Performance Evaluation Corporation (SPEC), and is designed to measure a system's ability to act as a web server for static and dynamic pages. The benchmark runs a multi-threaded HTTP load generator on a number of driving "client" systems that will do static and dynamic GETs of a variety of pages from, and also do POSTs to, the SUT (System Under Test). In our measurement, we use Zeus Technology's *Zeus Web Server*.

5.4 Test Machines

UltraSPARC-based machines (*sun4u* architecture) are used throughout the experiments. We use three groups of machines for our benchmark measurements: one for micro benchmark tests[20], one for the NFS tests, and another for SPECweb99.

The machines in Table 3 are installed with our MDT kernel and MDT-enabled network device driver. Each machine provides a gigabit Ethernet interface through the *Gigaswift Ethernet* NIC. We use them for our micro benchmark measurements (those other than SPECweb99) -

Table 3: Test Machines

Machine Name	Desription
$\mathbf{M}_1$	*Sun Ultra 60 UPA/PCI* 2x450MHz UltraSPARC-II, 512 MB RAM, OBP 3.31.0, POST 2.0.3.
$\mathbf{M}_2$	*Sun Ultra 80 UPA/PCI* 4x450MHz UltraSPARC-II, 2048 MB RAM, OBP 3.31.0, POST 1.2.8.
$\mathbf{M}_3$	*Sun Enterprise 450* 4x400MHz UltraSPARC-II, 2048 MB RAM, OBP 3.22.0, POST 6.1.0.
$\mathbf{M}_4$	*Sun Blade 1000* 2x600MHz UltraSPARC-III, 512 MB RAM, OBP 4.5.4, POST 4.5.4.

For the NFS measurements, a pair of SunFire 3800's are used. The client is a 4x900Mhz UltraSPARC-III+ with 16 GB of memory and the server is a 8x900 Mhz UltraSPARC-III+ with 64 GB of memory.

For the SPECweb99 benchmark, we use a Sun Enterprise 450 (4x400MHz UltraSPARC-II, 4096 MB RAM, OBP 3.22.0, POST 6.1.0) to host the web server. A couple of *Gigaswift Ethernet* NICs are installed, providing two gigabit Ethernet interfaces. The server is connected to a total of 10 clients – 8 machines similar to the M_1 (except for the 100 Mbps NICs) and 2 machines similar to M_4. They reside in two subnets (5 clients each) and connected to the server's *Gigaswift Ethernet* interfaces through a switch. Our MDT kernel is installed on the server machine, and early builds of Solaris 10 (stock kernel) are installed on the clients.

In the applicable cases, we place the *Gigaswift Ethernet* cards on the corresponding 64-bit/66MHz PCI slots of the machines.

[20] Due to the limited number of machines available at our disposal, not all of the machine permutations (see tables in §6) can be satisfied.

5.5 Performance Metric

We base our performance characteristics on the following aspects -

- **TCP throughput** – The total number of transferred data bits divided by the elapsed time, in Mbps unit.
- **Conforming Simultaneous Connections** – The metric reported by SPECweb99 for each iteration of the benchmark, which represents the number of connections made in a measurement period.
- **CPU utilization** – The percentage of CPU *system* time of the sending machine during the benchmark. A Sun internally-developed tool equivalent to *vmstat*, called *statit*, is used to obtain the CPU utilization.
- **Cycles per Instruction (CPI)** – The ratio of CPU clock cycles per instruction which reflects the amount of instruction cache hit or miss of the sending machine. We use the *cpustat* utility to sample the `Cycle_cnt` and `Instr_cnt` events for *system* and *user* using the UltraSPARC CPU performance counters.

6. RESULTS & ANALYSIS

As previously mentioned, the per-packet transmission costs vary among different platforms. Our measurement results taken with various types of Sun machines (see §5.4) agree with this. For reference, Table 4 presents the performance metric abbreviations (see §5.5 for details) used throughout the following tables and figures.

Table 4: Abbreviations

$\mathbf{M}_n\mathbf{s}$	Sender test machine.
$\mathbf{M}_n\mathbf{r}$	Receiver test machine.
$\mathbf{C}_s$	Sender CPU utilization (%).
$\mathbf{C}_m$	Sender CPU utilization with MDT (%).
$\mathbf{N}_s$	Maximum NFS throughput (Mbps).
$\mathbf{N}_m$	Maximum NFS throughput with MDT (Mbps).
$\mathbf{T}_s$	TCP throughput (Mbps).
$\mathbf{T}_m$	TCP throughput with MDT (Mbps).

6.1 netperf Test

The *netperf* micro benchmark involves sockets-based user applications on both ends of the connection. The sender implements a circular buffer mechanism used for transmission, and the receiver simply consumes the data upon reception. Although this benchmark doesn't represent any real-world application, it exposes the general sending and receiving networking dynamics (i.e. application scheduling, context switches, data copy) involved in networked applications.

We measure single TCP connection performance in this benchmark. Various sizes are used for the local and remote sockets (both `SO_SNDBUF` & `SO_RCVBUF`) as well as the `write()` and

read() sizes[21]; we use 48 KB (default), 64 KB and 1 MB sizes. Each test run consists of multiple iterations (each lasting 30 seconds) in order to achieve a ± 2.5% confidence interval at 99% confidence level. We present the results in Tables 6, 7, and 8, which are depicted throughout Figures 1, 2, 3, and 4.

Our measurements reveal several interesting facts -

- With MDT, the TCP throughput increases with larger send and receive buffer sizes, as one would expect. The opposite behavior is observed when MDT is disabled – the throughput stays the same – due to the transmission costs involved at the stream head, and those between the application and OS.
- The I/O subsystem (e.g. I/O controller), along with the number of CPUs in the *receiver* play a big role in determining the amount of data that can be handled, and in certain cases, the speed of the host CPUs on the receiver is less important. From the measurements, we can see that M_3 is able to receive more than than M_4, even though the latter has a much higher CPU speed. This shows that the current networking stack design is quite scalable.
- Regardless of MDT, the network throughput depends heavily on the ability of the receiver to handle incoming network traffics and distribute them to the applications. A slow receiver (such as M_1) is only capable of receiving around 500 Mbps regardless of the sender or the network parameters.

Results from Tables 6, 7 and 8 show the benefits that MDT offers in improving both the sending machine's CPU utilization as well as the network throughput, across different hardware platforms. In some cases, we can achieve over 45% reduction of CPU utilization, or over 100 % increase in network throughput.

Table 5: netperf User & System CPI Ratios#

Buffer Size	Standard	MDT
48 KB	2.357	2.046
64 KB	2.338	2.113
1 MB	2.367	1.883

From M_4 to M_2.

As shown in Table 5, MDT provides 10% to 20% CPI improvement due to better cache efficiency.

6.2 knetperf Test

The *knetperf* micro benchmark is similar yet different in many ways from *netperf*. As in the *netperf* case, it uses sockets-based applications to configure and establish the TCP connection. However, the data source and sink portions reside in a STREAMS module, which is pushed on top of TCP (on both ends) after the connection is established. Upon instruction from the application, the sending module allocates kernel buffers and passes them down to TCP for transmission[22], and the receiving module simply frees the data upon getting them.

[21] The actual kernel buffer sizes are limited by strmsgsz.

[22] It fully obeys STREAMS flow-control.

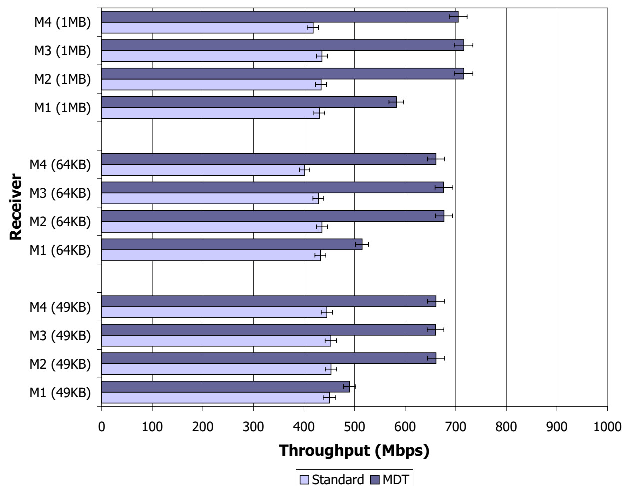

Figure 1: Single TCP netperf (Sender M_1)

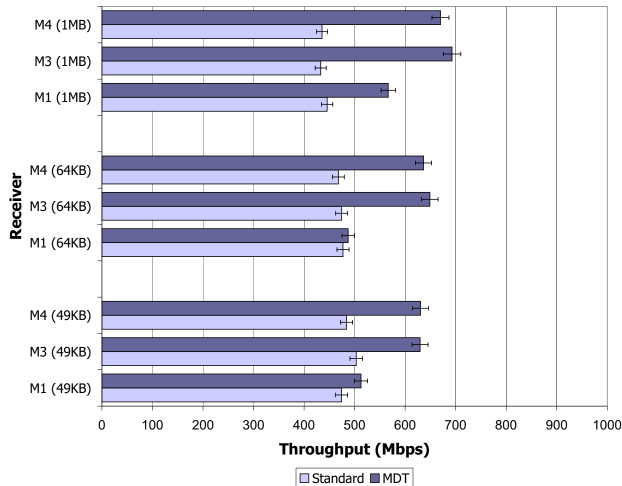

Figure 2: Single TCP netperf (Sender M_2)

While it does not represent any real-world *in-kernel* network applications, *knetperf* allows us to measure the raw performance of the networking stack, device driver, and the underlying hardware. It permits us to establish upper bound performance numbers (both CPU utilization and network throughput) for a given software and hardware parameter settings.

We measure single TCP connection performance using sizes similar to the *netperf* test for local socket, remote socket (both SO_SNDBUF & SO_RCVBUF), and write()[23]. Each test run consists of multiple iterations (each lasting 30 seconds) repeated in order to achieve at most 5% Standard Deviation of Mean. We present the results in Tables 9, 10 and 11, and depict them in Figures 5, 6, 7 and 8.

These measurements also reveal several interesting facts -

[23] Write size is the size of kernel buffer passed down to TCP; read size doesn't apply in this benchmark.

Table 6: Single TCP Connection netperf Measurements Data (48 KB)[#]

	M_1s				M_2s				M_3s				M_4s			
	T_s	C_s	T_m	C_m	T_s	C_s	T_m	C_m	T_s	C_s	T_m	C_m	T_s	C_s	T_m	C_m
M_1r	450.50	63.40	490.14	49.66	474.30	40.08	512.75	29.81	449.97	38.63	456.68	28.28	498.12	58.92	507.65	32.06
M_2r	453.35	60.52	660.91	56.42					451.17	38.49	686.14	33.77	603.81	64.98	832.02	48.87
M_3r	453.09	61.11	660.03	55.76	503.32	40.18	629.58	32.41	441.25	36.96	748.21	34.40	618.46	66.16	821.14	45.06
M_4r	445.18	56.48	660.89	57.46	484.01	39.60	630.49	34.07	437.13	36.72	651.55	36.20				

[#] SO_SNDBUF = 49152 bytes; SO_RCVBUF = 49312 bytes; write size = 49152 bytes; read size = 49152 bytes.

Table 7: Single TCP Connection netperf Measurements Data (64 KB)[##]

	M_1s				M_2s				M_3s				M_4s			
	T_s	C_s	T_m	C_m	T_s	C_s	T_m	C_m	T_s	C_s	T_m	C_m	T_s	C_s	T_m	C_m
M_1r	432.39	68.53	514.86	49.20	477.03	40.41	487.24	30.82	416.53	35.56	452.66	29.49	494.63	57.66	513.89	31.77
M_2r	435.46	59.47	676.74	57.89					413.72	35.82	709.19	35.12	553.84	60.87	836.85	47.01
M_3r	428.34	58.56	676.00	57.56	474.41	39.80	648.96	34.06	393.86	32.72	723.47	35.29	549.60	61.22	849.72	45.69
M_4r	401.40	59.37	660.88	58.52	467.91	39.52	636.37	33.97	400.08	34.04	683.32	35.87				

[##] SO_SNDBUF = 65536 bytes; SO_RCVBUF = 65928 bytes; write size = 65536 bytes; read size = 65536 bytes.

Table 8: Single TCP Connection netperf Measurements Data (1 MB)[###]

	M_1s				M_2s				M_3s				M_4s			
	T_s	C_s	T_m	C_m	T_s	C_s	T_m	C_m	T_s	C_s	T_m	C_m	T_s	C_s	T_m	C_m
M_1r	430.30	58.44	582.57	54.03	445.66	36.94	566.54	33.58	401.13	34.09	550.25	31.16	559.54	64.66	596.47	33.78
M_2r	433.80	59.23	716.03	88.18					396.82	33.84	762.64	48.56	566.44	61.72	870.61	56.53
M_3r	435.49	59.54	715.98	88.19	432.93	35.40	692.82	48.21	400.98	33.08	868.23	50.26	555.44	61.31	907.21	75.43
M_4r	418.05	63.70	704.87	89.36	435.53	35.01	669.97	48.08	391.89	32.59	720.89	50.69				

[###] SO_SNDBUF = 1048576 bytes; SO_RCVBUF = 1048952 bytes; write size = 1048576 bytes; read size = 1048576 bytes.

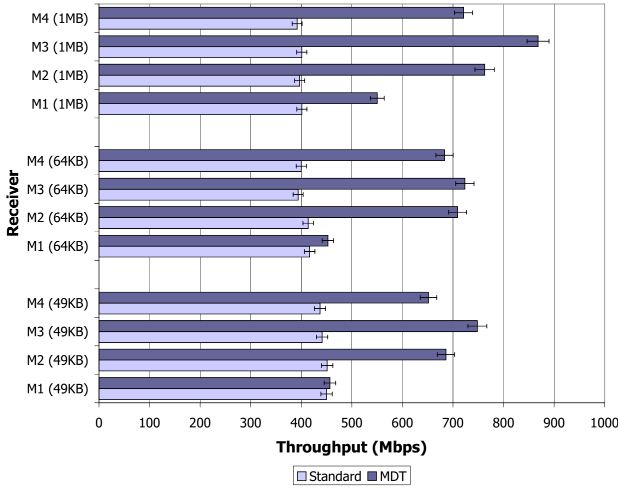

Figure 3: Single TCP netperf (Sender M_3)

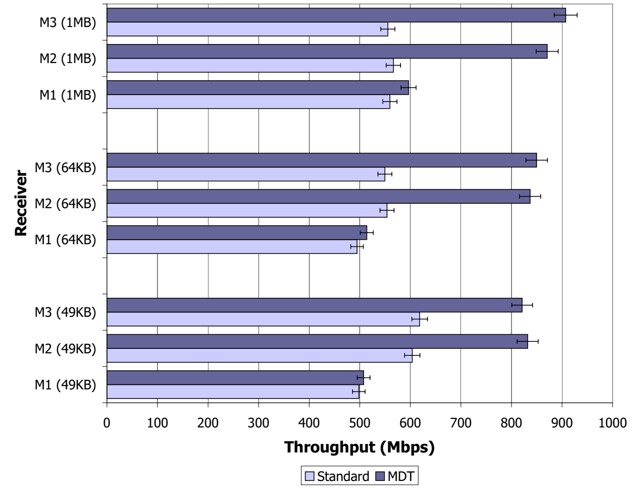

Figure 4: Single TCP netperf (Sender M_4)

- The *delay* factor of the BDP is substantially less due to the simplified sending and receiving model, and the default buffer sizes (48 KB) is sufficiently close enough to the BDP. This is the reason why increasing the buffer sizes does increase the network throughput, and it illustrates the effect that host processings have in affecting the delay factor.
- The current Solaris networking stack architecture can easily handle inbound high-bandwidth network traffics. As shown in Table 9, machines M_2 and M_3 can handle close to the *Gigaswift Ethernet* wire speed[24].

As we can see from Tables 9, 10 and 11, enabling MDT on the sender results in a 25% CPU reduction or close to 70% throughput increase.

6.3 SPECweb99 Test

[24]The actual TCP/IP line speed is around 950 Mbps.

Table 9: Single TCP Connection knetperf Measurements Data (48 KB)[$]

	M_1s				M_2s				M_3s				M_4s			
	T_s	C_s	T_m	C_m	T_s	C_s	T_m	C_m	T_s	C_s	T_m	C_m	T_s	C_s	T_m	C_m
M_1r	631.69	54.59	616.65	39.78	579.02	28.69	594.91	24.71	523.82	26.26	559.73	28.82	571.01	50.35	612.12	31.59
M_2r	632.58	54.83	807.27	41.54					523.81	30.05	782.73	27.17	730.71	53.07	936.47	40.54
M_3r	625.94	54.28	806.26	41.99	580.66	27.03	785.02	23.36	510.79	26.53	867.72	28.18	742.65	55.49	885.54	38.52
M_4r	633.45	54.46	808.22	42.28	581.33	28.63	782.35	25.69	510.32	28.61	783.53	28.26				

[$] SO_SNDBUF = 49152 bytes; SO_RCVBUF = 49152 bytes; write size = 49152 bytes.

Table 10: Single TCP Connection knetperf Measurements Data (64 KB)[$$]

	M_1s				M_2s				M_3s				M_4s			
	T_s	C_s	T_m	C_m	T_s	C_s	T_m	C_m	T_s	C_s	T_m	C_m	T_s	C_s	T_m	C_m
M_1r	613.67	55.04	613.11	37.29	568.85	29.19	593.78	24.95	504.01	27.67	565.73	27.71	603.933	51.33	621.85	31.06
M_2r	618.58	55.54	799.18	40.87					495.75	28.86	773.71	28.06	741.55	56.98	925.52	37.89
M_3r	614.14	55.48	799.82	41.12	578.82	29.45	778.86	24.95	514.86	29.44	862.95	29.50	747.43	56.85	867.38	37.23
M_4r	605.67	54.58	799.46	41.59	556.86	28.58	773.80	26.07	502.85	27.11	774.71	27.52				

[$$] SO_SNDBUF = 65536 bytes; SO_RCVBUF = 65928 bytes; write size = 65536 bytes.

Table 11: Single TCP Connection knetperf Measurements Data (1 MB)[$$$]

	M_1s				M_2s				M_3s				M_4s			
	T_s	C_s	T_m	C_m	T_s	C_s	T_m	C_m	T_s	C_s	T_m	C_m	T_s	C_s	T_m	C_m
M_1r	635.92	54.13	636.57	33.63	604.57	29.25	609.69	22.04	531.31	29.07	569.06	27.30	682.72	49.60	664.26	49.63
M_2r	626.71	54.15	815.55	65.37					547.34	29.03	804.88	37.48	756.95	57.29	938.15	63.41
M_3r	623.72	54.26	821.27	64.29	603.87	29.25	799.19	30.70	538.66	29.00	913.58	40.46	772.78	56.23	935.38	55.39
M_4r	621.36	54.76	777.77	71.55	609.31	30.45	760.05	37.94	519.49	28.46	770.12	40.38				

[$$$] SO_SNDBUF = 1048576 bytes; SO_RCVBUF = 1048952 bytes; write size = 65536 bytes.

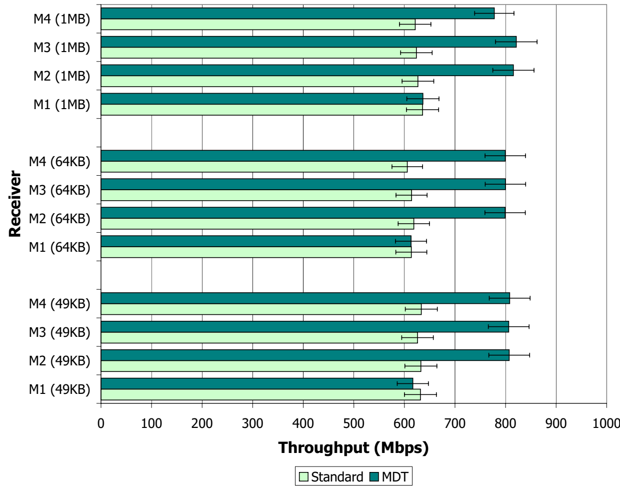

Figure 5: Single TCP knetperf (Sender M_1)

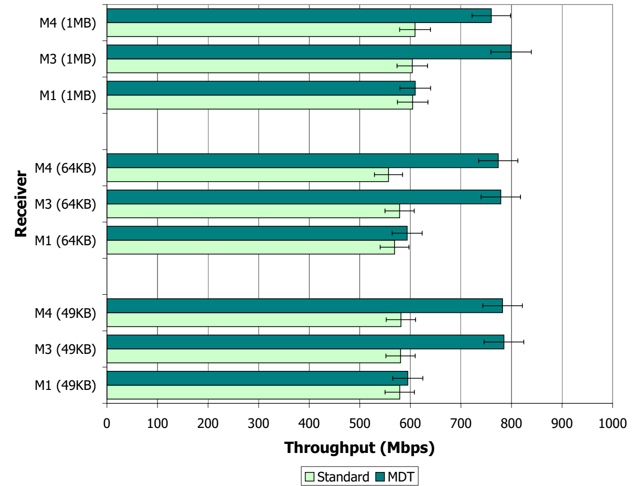

Figure 6: Single TCP knetperf (Sender M_2)

The SPECweb99 benchmark allows us to evaluate how MDT performs in a web server network environment. The conforming connections in Table 12 represent the numbers of simultaneous connections that the web server can support using the predefined workload. This benchmark reflects real-world usage by simulating accesses to a web service provider that offers dynamic operations, i.e. dynamic contents creation.

Throughout the measurements, roughly 60% of the transferred data is handled in the MDT transmission path. This is due to the small file sizes (35% of the files are less than 1KB) used by this benchmark – see 4.4.3 on the algorithm used by TCP to enable MDT. Therefore, MDT is only able to achieve a slightly moderate gain (15% in 1-CPU measurement) due to reduction in CPU utilization. This is attributed by its efficient handling of the transmitted data throughout the MDT path of the OS and driver, which further allows the CPU to be able to accept and handle more connections.

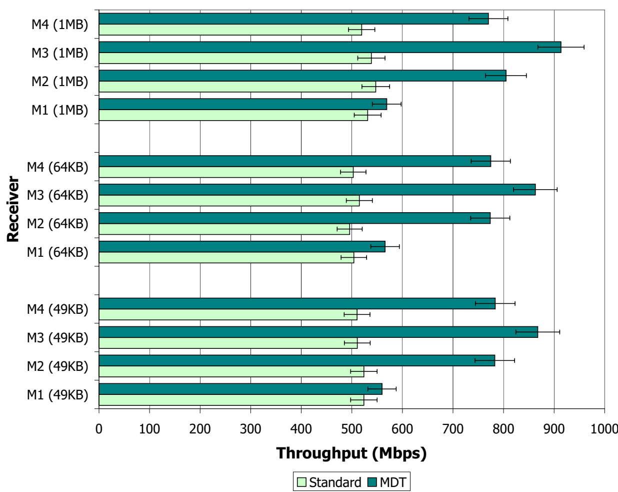

Figure 7: Single TCP knetperf (Sender M_3)

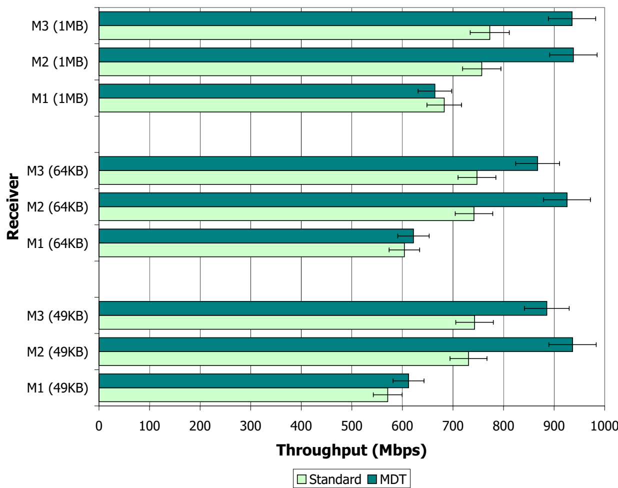

Figure 8: Single TCP knetperf (Sender M_4)

6.4 NFS Test

MDT may also be used to accelerate the NFS version 3 read operation. The default Solaris NFS read size is 32 KB which will benefit from the batch packet processing and reduced IOMMU costs. The NFS server does not directly deal with the stream head for data operations and instead uses *rpcmod* to directly *putnext* data into the TCP module.

In order to efficiently used MDT, the NFS read operation and *rpcmod* need some minor changes. The first change is to insure that the full NFS read response is presented to TCP as a single large `mblk_t`. For each response, the existing code generates two `putnext()` calls into TCP. The first, a 132 byte `mblk_t` containing the NFS and RPC headers followed by a second 32 KB `mblk_t` containing the file data. This behavior results in only every other `mblk_t` taking the optimized MDT code path. The NFS server read operation is modified to allocate extra space at the beginning of the 32 KB file data `mblk_t` to contain the required NFS and RPC headers. *rpcmod* is also modified to detect chains of `mblk_t` structures and to coelesce them into a single large `mblk_t` if

Table 12: SPECweb99 Measurements Data♮

Number of CPU	Conforming Connections Standard	MDT	Server CPU Idle Time
1	350	400	0%
2	600	660	3%
4	750	800	15-20%

♮ Due to bugs in the Zeus web server, 4 CPU results are not finalized.

space is available. As a result, every NFS read reply is now eligible to use the MDT code path.

In order to observe the benefits that MDT brings in increasing the NFS read performance, several file-transfer benchmarks measurements are taken between the client and server machines. Each machine runs the MDT kernel with the above changes as well as some minor enhancements to the NFS flow control processing. The test client is a custom multi-threaded user level program designed to prototype a high performance NFS client. The client rapidly sends multiple NFS read requests over one or more TCP connections and sinks the responses with a minimal amount of processing. The large amount of memory ensures that the test measures memory to memory performance and not disk effects.

The baseline peak NFS file throughput performance on a single TCP connection is measured to be 676.22 Mbps. The MDT peak file throughput on a single connection is measured to be 885.28 Mbps (see Table 13). The theoretical maximum NFS file throughput on a gigabit network is 893.6 Mbps. NFS using MDT is able to utilize 99% of the available bandwidth [11].

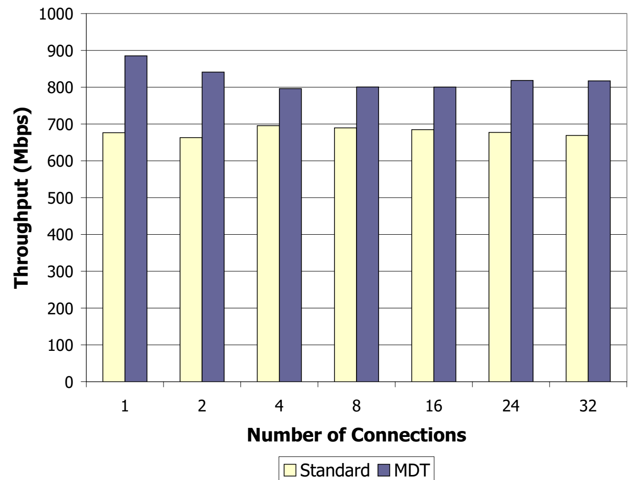

Figure 9: NFS 3 Read Measurements

Table 13: NFS 3 Read Measurements Data♯

	Number of TCP connections 1	2	4	8	16	24	32
N_s	676.22	663.03	695.44	689.56	684.58	677.16	668.96
N_m	885.28	841.00	796.18	800.75	800.43	818.46	817.18

♯ Results provided by David Robinson [11].

7. FUTURE WORK

MDT fits well with NFS as we have shown in §6.4, due to its bulk data-transfer nature. Further improvements can be made to the NFS and RPC subsystems in optimizing their data delivery methods to and from the networking stack. By not breaking the data into small pieces and improving the server flow control algorithms [11], significant performance gain can easily be achieved with MDT.

The *Gigaswift Ethernet* adapter implements *header-splitting* feature, which allows the header and payload portions of the packet to be placed in separate buffers upon reception. In addition, it also embeds *flow-reassembly* feature that permits multiple segments of the same TCP conection to be aggregated by the hardware. By combining these features, a concept similar to MDT can be applied to the receiving path[25], with some changes made to the driver and the networking stack.

Results from our *netperf* and *knetperf* measurements tell us that the overhead involved in copying the application data (*copyin* or *copyout*) is also quite significant. This presents an opportunity for improvements, such as implementing transmit and receive *zero-copy* for *sun4u* architecture[26]. We believe our MDT design will fit nicely with future *zero-copy* efforts.

8. CONCLUSIONS

The MDT scheme is a simple, and efficient networking transmit mechanism. The packet batching concept upon which it is based reduces the per-packet costs associated with transmission, which includes the IOMMU-related operations that are proven to be costly. Sending batches of packets through the networking stack also increases the system's overall cache utilization, and is cheaper than traversing the entire transmission path for the same number of packets from the perpective of host CPUs and memory subsystems. As reflected by the benchmark numbers, MDT provides measurably better system and network performance.

9. ACKNOWLEDGMENTS

Mohan Parthasarathy first proposed the idea. Jerry Chu initiated the *large send* engineering effort and served as our technical lead. David Robinson provided the NFS measurement results and analysis, as well as valuable critics. Alexander Kolbasov assisted us with the dynamic STREAMS data block cache. Thanks to Sandeep Sharma and Chris Schmechel for the exhaustive SPECweb99 measurements and their help throughout the project. We would also like to thank Shimon Muller for his hardware-related assistance.

Most of all, we thank our families for putting up with us during this project.

10. REFERENCES

[1] Jack Bochsler. PSARC/2001/070: DL_CAPABILITY_REQ/DL_CONTROL_REQ extensible interface for detecting, enabling and controlling DLS provider capabilities. March 2001. http://sac.eng/PSARC/2001/070/.

[2] Hsiao-Keng Jerry Chu. TCP LSO (large segment offload) - an update. http://arachnid.eng/inet/InternetPerf/multidata/ref/lso.PDF.

[3] Hsiao-Keng Jerry Chu. Zero-copy TCP in solaris. *Usenix Annual Technical Conference*, pages 253–264, 1996.

[4] Denny Gentry. Driver manifesto. http://npweb.ebay.sun.com/asics/nw_asics/drvopt/.

[5] Alexander Kolbasov. Dynamic dblock caches. August 2002. http://streams.eng/alloc/dblk_alloc.pdf.

[6] Mark Mason. Increasing cache efficiency in solaris IP or go fast with a little help from your cache. February 2001.

[7] Adi Masputra. knetperf: in-kernel netperf. http://arachnid.eng/inet/InternetPerf/autopet/files/knp.tar.gz.

[8] Adi Masputra. Packet chaining with STREAMS. http://arachnid.eng/inet/cassini/pclass/test_results/ipv4_rx.

[9] Adi Masputra. Multidata interface design specification. August 2002. http://arachnid.eng/inet/InternetPerf/multidata/mmd.pdf.

[10] Mohan Parthasarathy. DVMA optimization for network. http://arachnid.eng/inet/InternetPerf/multidata/ref/mohan_dvma2.0.pdf.

[11] David Robinson. Gigabit ethernet NFS performance. http://jurassic.eng/ robinson/projects/nfs-client/NFS_Throughput.pdf, May 2002.

[25] Multidata Receive (MDR).

[26] The current networking *zero-copy* in Solaris only applies to older architectures.