Multidata V.2: VA-Disjoint Packet Extents Framework Interface Design Specification*

Revision 1.0 – August 1, 2004

Adi Masputra
Solaris Network Performance Group - Data Movement
adi.masputra@Sun.COM

ABSTRACT

The Solaris Multidata mechanism [3, 4] allows for more than one packet to be sent from one module to another in a given call, thereby reducing the per-packet processing costs. This translates to improving the host CPU utilization and/or network throughput[1].

The original interface specification [3] (henceforth referred to as V.1) limits the number of payload buffer to at most one per M_MULTIDATA message. While this is adequate for packets whose payload areas are contained within the corresponding payload buffers, it does not work well for those which span across disjoint ones (e.g. an IP packet containing a TCP segment whose areas extend beyond a payload buffer into the next, both of which are disjointed apart at the virtual address spaces; we dub this condition as *split packet*.)

This paper describes minor extensions to the V.1 interface that allow for multiple payload buffers to be associated with a M_MULTIDATA message, thus solving the *split packet* problem. In addition, they provide a mechanism for other transport protocols to utilize the Multidata interface when multiplexing several independent streams on a particular session or connection, e.g. per stream payload buffer for chunks multiplexed over a SCTP connection.

We refer to this new interface as Multidata V.2 (henceforth referred to as V.2).

1. INTRODUCTION

We encourage the reader to familiarize himself with Multidata [3, 4] before proceeding further with this document.

In the following sections, we examine the shortcomings of the V.1 interface and explain how we address them in V.2. We then elaborate on the modified and the newly-introduced kernel library function calls (APIs) which allow for a module to easily deal with handling and accessing multiple payload buffers. Finally, we provide performance results comparison between the two versions.

*Sun Proprietary/Confidential: Internal Only.

[1] At the time of writing, the Solaris MDT capability based on the Multidata framework is supported on both *ce* (Sun Gigaswift Ethernet) and *ibd* (IP over InfiniBand) device drivers on both SPARC and x86 platforms.

2. PAST LIMITATIONS

The original specification of Multidata mandates that there be at most one payload buffer – in addition to one header buffer – per M_MULTIDATA message. Although this is sufficient for the majority of the cases, it does not support those which involve packets spanning over several payload buffers (VA-disjoint blocks).

Such restriction bears a measurable cost, as not only avoiding the use of Multidata in those cases reverts the handling to the less-efficient legacy (non-Multidata) processing path, it affects the Solaris TCP transmission characteristics which may lead to suboptimal performance.

2.1 Split Packets

The most obvious example that prevented the efficient use of Multidata is the TCP *split packet* scenario. Data generated by a module arrives in TCP in the form of message blocks, or *mblks*. Each *mblk* points to a VA-contiguous data block that contains the data to be processed.

In the sending TCP case, the data-generating module (e.g. sockfs, rpcmod) either copies over the user data into kernel-allocated buffers, or borrows the data over from the VM subsystem (e.g. *segmap* faulted). In any case, the size of each data block passed onto TCP varies, and each of them is VA-disjointed from one another. If the data comes from kernel-allocated buffers allocated at the stream head, the size is limited by the `sd_qn_maxpsz` and/or `sd_qn_maxblk` variables [4]. When it is loaned by the VM subsystem, it will be at most a system *pagesize* in length [1].

The reverse is also true for the receiving TCP case, where the network interface card is capable of *header-splitting* and *payload-reassembly* features [2]. Reassembling the payload data typically involves handing off physically-contiguous *pagesize* buffers to the hardware in order for it to place the reassembled TCP data stream. Once a buffer has been filled, the hardware uses subsequent host-supplied buffers to store additional data for the same connection.

A data stream is divided by TCP into units of segments – whose sizes may vary and may begin at any offset within a data block – prior to passing it onto the next layer for processing. In particular, the size of a data block may not be fully divisible by the effective Send Maximum Segment Size (SMSS), thereby creating a case where a segment may lie across multiple data blocks, i.e. *splitted* between VA-

disjoint buffers (see Figure 1). This phenomenon is evident in both sending and receiving scenarios.

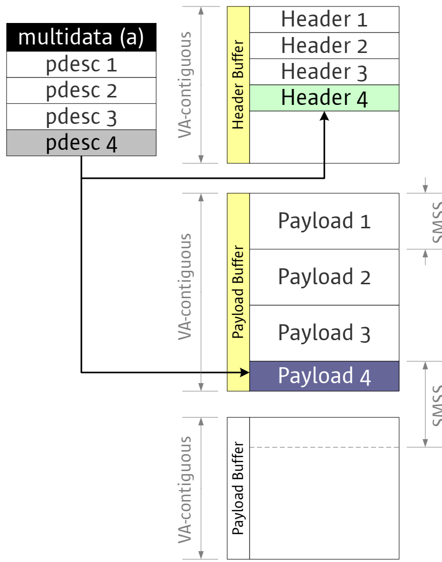

Figure 1: V.1 Packet Descriptor Layout

In either Multidata Transmit (MDT) or Multidata Receive (MDR) case, the inability of the framework to handle a *split packet* forces the packet-generating module to abandon further Multidata processing on the current data block, and either fallback to using M_DATA to represent the *split packet*, or generate additional M_MULTIDATA messages due to the lack of multiple payload buffers support. This is rather sub-optimal, and in most cases introduces performance degradations due to higher processing costs involved in frequently alternating between the two schemes, as well as to the increased number of packets generated on the network.

The impact is more apparent in the sending TCP case as documented in bug 4801229. The inability to deal with *split packets* results in the Solaris network stack generating partially-filled TCP segments, i.e. segments in the middle of a TCP burst which are less than SMSS (see Figure 2). This increases the number of packets used to transfer a given amount of user data, and therefore adds to the amount of incoming TCP ACKs – these are common phenomena since the data-generating modules seldom produce data blocks whose sizes are multiples of SMSS. Not only that it decreases the system performance due to the added per-packet processing costs, it negatively impacts the network by increasing its load with these unnecessary packets.

2.2 Disjoint Extents

Another disadvantage of the V.1 interface is that it restricts the packet payload to span over one *continuum* extent, therefore limiting its usages to cases such as TCP and fragmented UDP[2]. Other scenarios which may involve more than one extent per packet, such as SCTP, are not applicable for V.1.

[2]Neither the *stream head* nor UDP module performs any fragmentation, hence IP module possesses the unfragmented data block.

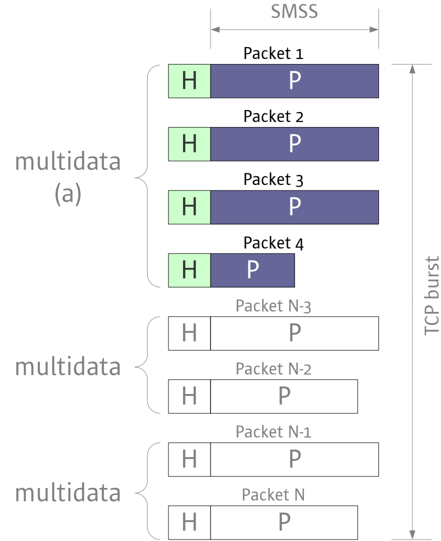

Figure 2: TCP/IP Packet Characteristics of V.1

SCTP defines a stream as a data flow in units of chunks. It employs a mechanism to multiplex several streams into one SCTP connection. Because each stream performs independent buffer management (separate data blocks per stream) the SCTP packets carrying such multiplexed chunks are made up of many extents scattered across the data blocks.

The Solaris SCTP implementation currently handles this by using M_DATA messages formed as a chain[3] representing a SCTP packet, where each message corresponds to a SCTP chunk. Although this fulfills the required functionality, the overall system performance can be further improved by leveraging the use of Multidata to carry the chunks.

3. INTERFACE CHANGES

To address the V.1 limitations, we extend Multidata to allow for more than one payload buffers per M_MULTIDATA message. We also enhance the packet descriptor structure to describe payload extents located throughout the buffers. In addition, we make some minor modifications to the existing interfaces to further refine their usage model.

3.1 Payload Buffers

In V.1, payload buffer association is made at the time of Multidata allocation, using the `mmd_alloc()` interface, and is limited to one per Multidata. Once allocated, no further association can be made against the Multidata and this condition holds true throughout its lifetime.

We modified this concept in V.2, by requiring payload buffers (up to the maximum allowed) to be associated post allocation, using the `mmd_addpldbuf()` interface – in essence, only header buffer association can be made through `mmd_alloc()`. To make the interface and implementation simple, we choose to limit the amount of payload buffer associations to a reasonable value[4] (16). As in V.1, Multidata buffer association

[3]Using `b_cont`.

[4]This value is chosen to accomodate at least 64KB of total payload data for *pagesize* buffers (4KB on i86pc, 8KB on sun4u; the latter gives 128KB), per Multidata.

is non-reversible, and therefore buffers can't be dissociated once they are associated with a Multidata.

In order to identify one payload buffer from another, each associated payload buffer is assigned an index. The assignment is done at the time of association, through `mmd_addpldbuf()`. This index can be used by a Multidata packet descriptor to provide for a mapping between a packet payload extent and a payload buffer (see §3.2). It also serves as an index to the Multidata payload buffer array in the `mbufinfo_t` data structure obtainable by calling `mmd_getregions()`.

This concept can be summarized as:

- A Multidata gets allocated with no payload buffer association, and may later be associated with one or more payload buffers up to a total of 16.
- Each associated payload buffer is assigned an index for identification purposes.
- The changes affect only the payload buffer scheme; the header buffer strategy is unchanged from that of V.1. *Id est*, there can be at most one header buffer per Multidata, whose association can only be made at allocation time.

3.1.1 Adding Payload Buffer

A new interface has been provided to allow for additional payload buffer association. `mmd_addpldbuf()` may be used to attach additional payload buffers to a particular Multidata.

- **Example.** To associate an additional payload buffer with Multidata handle `mmd`, we say:

```
int pmp_idx;
pmp_idx = mmd_addpldbuf(mmd, pmp);
```

Unique payload buffers may be associated up to the maximum allowed throughout the lifetime of a Multidata.

3.1.2 Obtaining Buffer Regions

Properties of the header and payload buffers are obtainable through the `mmd_getregions()` interface. The `mbufinfo_t` data structure has been modified to contain an assortment of information related to the payload buffers. They are defined as follows -

```
typedef struct mbufinfo_s {
    uchar_t *hbuf_rptr;  /* start address */
    uchar_t *hbuf_wptr;  /* end address */
    uint_t  pbuf_cnt;    /* # of pld buffers */
    struct pbuf_ary_s {
        uchar_t *pbuf_rptr;  /* start address */
        uchar_t *pbuf_wptr;  /* end address */
    } pbuf_ary[16];
} mbufinfo_t;
```

The total number of currently associated payload buffers is represented by `pbuf_cnt`. Indices obtained from calls to `mmd_addpldbuf()` may be used to access properties of the corresponding payload buffers contained in `pbuf_ary[]`.

3.2 Packet Descriptor

Figure 3 depicts how the V.2 packet descriptor layout accomodates a *split packet* in a Multidata. In contrast with the V.1 layout (see Figure 1), our new scheme allows for a packet's payload to span across different payload buffers. This permits the sending TCP to perform more optimally, by utilizing the mechanism for including the rest of the *split packet* into one SMSS packet. As shown in Figure 4, the number of packets used to send the user data would be reduced due to the more compact packet characteristics.

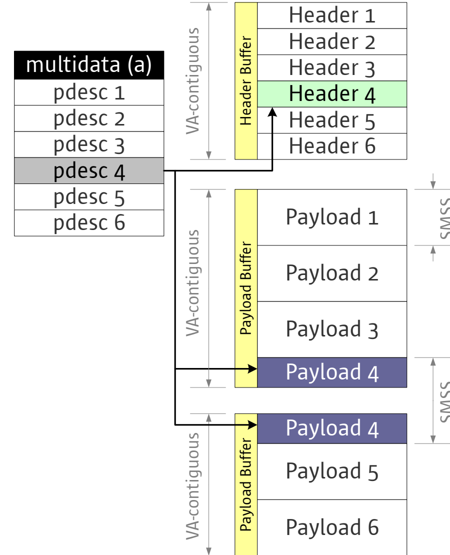

Figure 3: V.2 Packet Descriptor Layout

The packet descriptor data structure has been modified to include information related to the actual buffer used by each payload extent/fragment (henceforth referred to as span), as represented by the following -

```
typedef struct pdescinfo_s {
    uint_t  flags;       /* PDESC_{HBUF,PBUF}_REF */
    uchar_t *hdr_base;   /* start of hdr entry */
    uchar_t *hdr_rptr;   /* start of hdr span */
    uchar_t *hdr_wptr;   /* end of hdr span */
    uchar_t *hdr_lim;    /* end of hdr entry */
    uint_t  pld_cnt;     /* # of pld spans */
    struct pld_ary_s {
        int pld_pbuf_idx;    /* pld buffer index */
        uchar_t *pld_rptr;   /* start of pld span */
        uchar_t *pld_wptr;   /* end of pld span */
    } pld_ary[16];
} pdescinfo_t;
```

`pld_cnt` represents the total number of payload spans for a particular packet. Each payload span in `pld_ary[]` maps to a specific region of a payload buffer, which is identified by the buffer index value in `pld_pbuf_idx`.

Note that as in V.1, the framework does not impose any restrictions on how the Multidata user describes the spans that make up a packet, except that the regions described by the spans must fall within the VA boundaries of the header

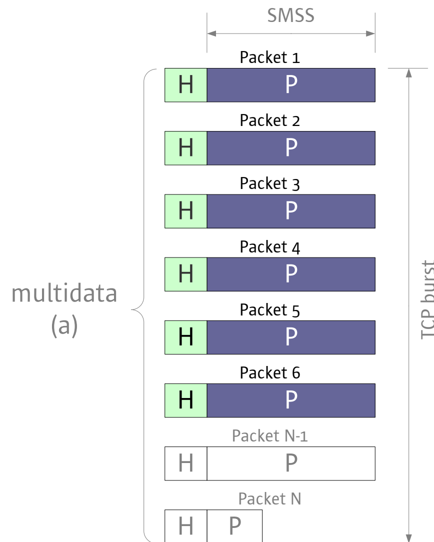

Figure 4: TCP/IP Packet Characteristics of V.2

and/or payload buffers. This means that the spans may overlap one another, be empty (zero-length), refer to the same payload buffer, or refer to different buffers in non-consecutive order. Therefore, it is entirely up to the user to define how the packet is constructed.

For simplicity, the number of allowable payload spans per descriptor is limited to the number of payload buffers (16). This is to accomodate the case where a particular packet's payload spans across all of the payload buffers.

3.2.1 Adding Packet Descriptor

Adding a descriptor is done by calling `mmd_addpdesc()` similar to V.1. However, the packet descriptor information now contains additional properties to describe multiple payload spans.

- **Example.** To create a packet descriptor with a header span plus two payload spans, we say:

```
pdescinfo_t pi = {
    PDESC_HBUF_REF | PDESC_PBUF_REF,
    0x10000, 0x10020, 0x10048, 0x10068, 2, {
      { 0, 0x20000, 0x201CC },
      { 1, 0x30000, 0x303E8 } } };

int err;
pdesc_t *pd;
pd = mmd_addpdesc(mmd, &pi, &err, KM_NOSLEEP);
```

  In this example, we specify `0x10000` as the base address of the header area, and that we reserve 32-bytes of head space before the header span – which is 40-bytes in length – followed by 32-bytes tail space. The packet's payload (1460-bytes) is contained in 2 parts: the first belongs to buffer 0 which begins at `0x20000` with a length of 460-bytes, while the second belongs to buffer 1 at `0x30000` for 1000-bytes. The handle of the newly-created descriptor is returned upon success. (Note: in the typical usage, the addresses would not be hard-coded.)

3.3 Miscellaneous

Several auxiliary interfaces are introduced for better statistics gathering, performance improvements, and Multidata attribute protection.

3.3.1 Obtaining Multidata Size

The `mmd_getsize()` routine may be called to retrieve the total size of the associated buffers, as well as the actual number of bytes referenced by the packet descriptors. This routine is somewhat analogous to the `msgdsize(9F)` interface for M_DATA messages.

3.3.2 Fast Descriptor Conversion

The `mmd_transmform_link()` provides for a faster way of converting a packet desciptor into a M_DATA message. Instead of returning a newly-allocated message containing the consolidated spans of the packet, it returns one or more M_DATA messages linked via the `b_cont` field. Each message block represents an individual span defined in the descriptor. While the original `mmd_transform()` exerts similar behavior as `msgpullup(9F)`, the newer `mmd_transform_link()` behaves more like `dupmsg(9F)`.

Both interfaces are provided by the framework in order to satisfy different user requirements.

3.3.3 Persistent Attribute

The `mmd_addpattr()` interface accepts an additional parameter which identifies whether or not the attribute is to be marked as persistent, i.e. it can not be removed through `mmd_rempattr()` and will stay with the Multidata throughout its lifetime.

This allows for the attribute association to be kept intact, since certain attributes (e.g. checksum information) may be used to describe crucial properties of the Multidata contents and therefore its integrity.

4. MULTIDATA INTERFACE SUMMARY

```
multidata_t *mmd_alloc(
    mblk_t *hdr_mp,       /* header buffer */
    mblk_t **pmmd_mp,     /* M_MULTIDATA */
    int kmflags);         /* kmem flags */
```

`mmd_alloc()` creates a Multidata message and accepts an optional header buffer message block. It performs the necessary work to allocate and initialize the Multidata message, and returns the handle of the newly-created Multidata upon success. The caller can also retrieve the M_MULTIDATA representative message block through `pmmd_mp`. This routine returns a NULL handle upon failure. (Note: Upon success, the caller must not free the associated header buffer message block.)

```
int mmd_addpldbuf(
    multidata_t *mmd,     /* multidata handle */
    mblk_t *pld_mp);      /* payload buffer */
```

`mmd_addpldbuf()` associates a payload buffer to the Multidata, and returns the buffer index (a positive integer) upon success. Up to a total of 16 distinct payload buffers may be

associated to a Multidata. It returns -1 when the buffer has been previously associated, or if the maximum assocation limit (16) has been reached. (Note: Upon sucess, the caller must not free the associated payload header buffer message block.)

```
multidata_t *mmd_getmultidata(
    mblk_t *bp);          /* message block */
```

`mmd_getmultidata()` returns the associated Multidata handle of the M_MULTIDATA representative message block. It returns `NULL` if `bp` does not point to a M_MULTIDATA.

```
void mmd_getregions(
    multidata_t *mmd,     /* multidata handle */
    mbufinfo_t *info);    /* pointer to info */
```

`mmd_getregions()` returns the start and end VAs including other properties of the Multidata associated buffers into the `mbufinfo_t` structure.

```
uint_t mmd_getcnt(
    multidata_t *mmd,     /* multidata handle */
    uint_t *hbuf_ref      /* header buffer ref. */
    uint_t *pbuf_ref);    /* payload buffer ref. */
```

`mmd_getcnt()` returns the number of packet descriptors in a Multidata. If the optional `hbuf_ref` or `pbuf_ref` argument is specified by the caller, the total reference count to the corresponding buffer made by the packet descriptors will be returned through the argument.

```
void mmd_getsize(
    multidata_t *mmd,     /* multidata handle */
    uint_t *tot_sz,       /* total bytes buffer */
    uint_t *inuse_sz);    /* total bytes in-use */
```

`mmd_getsize()` returns the total number of bytes for the entire buffers as well as the regions of associated buffers referenced (active/in-use) by the packet descriptors. If the optional `tot_sz` or `inuse_sz` argument is specified by the caller, the corresponding information will be returned through the argument.

```
pdesc_t *mmd_addpdesc(
    multidata_t *mmd,     /* multidata handle */
    pdescinfo_t *info,    /* packet layout */
    int *err,             /* pointer to errval */
    int kmflags);         /* kmem flags */
```

`mmd_addpdesc()` adds a packet whose layout across the associated buffer(s) is described in the `pdescinfo_t` structure. Upon success, the newly-created descriptor handle is returned. A NULL handle value indicates a failure, and the caller may check the error code returned through `err`; `EINVAL` if the packet layout is invalid, or `ENOMEM` if the allocation fails. (Note: Upon succes, the new descriptor is added to the end of the descriptor list.)

```
void mmd_rempdesc(
    pdesc_t *pd);        /* descriptor handle */
```

`mmd_rempdesc()` removes a packet descriptor from the descriptor list in the Multidata. This function will also free any local attribute associated with corresponding descriptor.

```
pdesc_t *mmd_getfirstpdesc(
    multidata_t *mmd,   /* multidata handle */
    pdescinfo_t *info); /* packet layout */
```

```
pdesc_t *mmd_getlastpdesc(
    multidata_t *mmd,   /* multidata handle */
    pdescinfo_t *info); /* packet layout */
```

`mmd_getfirstpdesc()` and `mmd_getlastpdesc()` return the first and last packet descriptor from the descriptor list in the Multidata. The packet layout is returned through the optional `pdescinfo_t` argument. The functions return `NULL` if the descriptor list is empty.

```
pdesc_t *mmd_getprevpdesc(
    pdesc_t *pd,        /* descriptor handle */
    pdescinfo_t *info); /* packet layout */
```

```
pdesc_t *mmd_getnextpdesc(
    pdesc_t *pd,        /* descriptor handle */
    pdescinfo_t *info); /* packet layout */
```

`mmd_getprevpdesc()` and `mmd_getnextpdesc()` return the packet descriptor before and after `pd` descriptor. The packet layout is returned through the optional `pdescinfo_t` argument. The functions return `NULL` if the end of the descriptor list has been reached.

```
pdesc_t *mmd_adjpdesc(
    pdesc_t *pd,        /* descriptor handle */
    pdescinfo_t *info); /* packet layout */
```

`mmd_adjpdesc()` allows for the layout of the corresponding packet descriptor to be modified. If returns the handle of the modified packet descriptor upon success, or NULL otherwise.

```
mblk_t *mmd_transform(
    pdesc_t *pd);       /* descriptor handle */
```

`mmd_transform()` allocates a M_DATA message and copies over the spans associated with the descriptor into a newly-created area. It returns the newly-created M_DATA message upon success, or NULL upon an allocation failure.

```
mblk_t *mmd_transform_link(
    pdesc_t *pd);       /* descriptor handle */
```

`mmd_transform_link()` returns one or more M_DATA messages, where each message corresponds to a span defined in the Multidata packet descriptor. This routine differs from `mmd_transform()` in that it does not allocate a new M_DATA message for the packet, but rather returns one or more M_DATA messages linked through `b_cont`. The routine returns NULL upon an allocation failure.

Table 1: ttcp results showing the effects of V.2 on TCP segments & ACKs

Size (bytes)	Out Segs		In ACKs		Throughput (KB/sec)		Sender CPU$_{total}$ (sec)		Receiver CPU$_{total}$ (sec)	
	V.1	V.2	V.1	V.2	V.1	V.2	V.1	V.2	V.1	V.2
1460	10000033	7541517	2500128	1507770	73584.69	92763.41	124.80	103.83	155.94	122.89
1536	10000035	7653244	2500050	1541634	69909.82	93634.23	133.22	105.16	169.53	128.09
2048	10000054	9410409	2500053	2136297	90395.28	92414.40	137.68	129.86	179.20	169.30
2496	10009128	9205969	2504589	1565119	106775.75	110352.52	141.53	109.96	185.36	153.97
4096	15000091	14823395	2576963	2048236	114502.07	114575.34	165.86	147.56	218.30	225.35
6144	25001032	21857892	3655882	2742214	113729.59	114694.62	204.93	170.56	313.70	310.18
8192	30016214	29191530	3998198	3652644	114513.63	114682.70	217.62	209.43	390.85	408.52

```
int mmd_dupbufs(
    multidata_t *mmd,   /* multidata handle */
    mblk_t **phmp,      /* duplicated hdr mblk */
    mblk_t **ppmp);     /* duplicated pld mblks */
```

`mmd_dupbufs()` returns duplicate message blocks of the corresponding buffer specified by the `phmp` or `ppmp` argument. (When there are multiple associated payload buffers, `ppmp` will contains a chain of message blocks – each corresponding to a payload buffer – linked through `b_cont`.) It returns 0 on success, or -1 otherwise.

```
int mmd_getpdescinfo(
    pdesc_t *pd,        /* descriptor handle */
    pdescinfo_t *info); /* packet layout */
```

`mmd_getpdescinfo()` returns the packet layout through the `pdescinfo_t` argument. It returns 0 on success, or -1 otherwise.

```
pattr_t *mmd_addpattr(
    multidata_t *mmd,   /* multidata handle */
    pdesc_t *pd,        /* descriptor handle */
    pattrinfo_t *info,  /* attribute parameters */
    boolean_t persist,  /* persistent switch */
    int kmflags);       /* kmem flags */
```

`mmd_addpattr()` allocates an attribute whose parameters are described by the `pattrinfo_t`. The attribute may be associated globally by specifying `NULL` as the descriptor handle, or locally by supplying a valid packet descriptor. It returns the attribute handle upon success, or NULL otherwise.

```
void mmd_rempattr(
    pattr_t *pa);       /* attrribute handle */
```

`mmd_rempattr()` removes and frees an attribute (and its allocated resources) described by the attribute handle. If the handle describes a persistent attribute, this routine will not perform the attribute removal.

```
pattr_t *mmd_getpattr(
    multidata_t *mmd,   /* multidata handle */
    pdesc_t *pd,        /* descriptor handle */
    pattrinfo_t *info); /* attribute parameters */
```

`mmd_getpattr()` returns the handle of an attribute whose type is described in the `pattrinfo_t` argument. Passing in `NULL` as the packet descriptor handle implies that a search on the global attribute list is requested. Otherwise, the search is to be done on the local list associated with packet descriptor `pd`. Upon success, the function returns the attribute handle, along with the length and address of the buffer containing the attribute data through the `pattrinfo_t` argument. It returns `NULL` if a match is not found.

5. RESULTS & ANALYSIS

As previously stated in §2.1, one of the main limitations in V.1 is its inability to deal with *split packets*. To show that we addressed this problem in V.2, we provide results from our experiments on various system workloads, ranging from simple *ttcp* and zero-copy tests to the more complex SPECweb99 benchmark.

We measured the total CPU time (system + user + interrupt) on the machines by comparing the `cpu_stat` and `intrstat` kernel statistics before and after each run. We modified *ttcp* such that it is capable of calculating such statistics internally.

All of the results shown contain the averages of at least 3 benchmark runs.

5.1 ttcp

Our *ttcp* benchmark involved performing various small-size sends and receives between two hosts, in order to demonstrate the presence and impact of *split packet* phenomena in V.1, and how they are reduced (or eliminated) in V.2.

Each test was composed of transferring user data ranging from 1460 to 8192 bytes for 5M times across the machines, using 256KB send and receive socket buffer sizes (1448 bytes effective SMSS[5]). In both V.1 and V.2 cases, the stream head never broke down the data into chunks and instead copied the entire buffer at once during each write [4].

We used a pair of identical SunFire V240 machines; each equipped with 2 UltraSPARC-IIIi 1.3GHz CPUs, 2GB RAM and copper-based *Cassini* gigabit Ethernet cards (`rev_id` 48).

We can see from Table 1 that the occurence of *split packet* phenomena decreases with larger buffer sizes. In addition, due to the reduction in (or elimination of) *split packets*, our V.2 scheme provides for up to 25% reduction in the generated TCP segments with up to 40% less received ACKs. We can also witness up to 30% increased in throughput, as well

[5]32 bytes of TCP header due to the timestamp option.

as up to 20% and 17% reduction in sender and receiver total CPU time, respectively.

5.2 sendfile

The Solaris 10 `sosendfile64()` routine[6] has been modified to perform transmit-side zero-copy, by locking down the VM pages containing the file data and getting a kernel map of the file pages [1]. Each VA-contiguous data block that gets passed down to the networking stack is at most a `pagesize` in length.

We modified *ttcp* to utilize the `sendfile(3EXT)` interface when a file is provided as the data source. In this experiment, we provided a large file (slightly over 28MB) containing random data that *ttcp* used for repetitive transfers (80 times) over the same TCP connection, which resulted in over 2GB[7] of TCP payload data exchanged during each run. Both send and receive socket sizes were kept to their default values (48KB).

Table 2: Zero-copy sendfile on x86

	Throughput (Mb/sec)	TX CPU$_{total}$ (sec)
No MDT	901.39	15.49
MDT V.1	898.91	14.90
MDT V.2	903.85	12.66

Table 2 reflects the results from the experiments done between a pair of identical Sun LX-50 machines; each equipped with 2 Pentium-III 1.4GHz CPUs, 3GB RAM and fibre-based *Cassini* gigabit Ethernet cards (`rev_id` 17). Here we witness that V.2 provides for an additional 15% reduction in total CPU time compared to V.1.

Table 3: Zero-copy sendfile on SPARC

	Throughput (Mb/sec)	TX CPU$_{total}$ (sec)	DVMA TLB Misses
No MDT	900.56	13.29	1.88M
MDT V.1	897.85	12.41	680K
MDT V.2	903.71	10.04	635K

Table 3 reflects the results from similar experiments between a pair of identical SunFire V240 machines previously mentioned in §5.1. We see bigger improvement compared to x86, as V.2 is able to provide for an additional 19% reduction in total CPU time compared to V.1.

On SPARC platforms, we were able to collect the number of TLB misses that occur at the IOMMU controller using the `busstat(1M)` utility, in order to elaborate on the savings that MDT provides to the DVMA-related operations [4].

Unfortunately, we were not able to retrieve similar information on the x86 platforms, due to the differences in the x86 PCI nexus driver implementation and hardware architecture.

[6]An application needs to be compiled with large file support in order for `sendfile(3EXT)` or `sendfilev(3EXT)` to utilize zero-copy transmit on supported network interfaces.
[7]2369085440 bytes to be exact.

5.3 SPECweb99

The SPECweb99 benchmark allows us to evaluate how V.2 performs in a web server network environment. The conforming connections in Table 4 represent the numbers of simultaneous connections that the web server can support using the predefined workload. The frequency distribution for files used by this benchmark is 50% for $\leq$ 10KB, 35% for $\leq$ 1KB, 14% for $\leq$ 100K and 1% for $\leq$1MB.

We used a SunFire E6900 with 4 UltraSPARC-IV 1.2GHz CPUs, 32GB RAM, and 2 fibre-based *Cassini* gigabit Ethernet cards (`rev_id` 17).

Table 4: Impact of V.2 on SPECweb99

	Conforming Simultaneous Connections	Throughput ops/sec
MDT V.1†	2436	6840
MDT V.2	2536	7134

†Minor differences betweeen MDT turned off and V.1.

From Table 4 we witness over 4% improvements on both the number of connections and throughput as reported by the SPECweb99 benchmark.

6. CONCLUSIONS

Multidata V.2 is a minor extension to the original specification which allows for VA-disjoint portions of a packet to be handled in a more efficient manner, thus permitting for better system and network performance. Migrating from a V.1 to V.2 interface involves minor code changes on the driver, yet it provides for noticable performance improvements as reflected by the benchmark results.

7. ACKNOWLEDGMENTS

Many thanks to Frank DiMambro and Kanoj Sarcar for the hours spent in perfecting the MDT implementations on *ce* and *ibd* device drivers (both SPARC and x86). Also thanks to James D. Carlson for various feedbacks which led to the final design of the V.2 interface. Other people from NPT, IPSG, NSPG and PAE have also contributed their resources to various aspects of this effort, including (but not limited to) tools, testing, reviews, and many valuable inputs.

8. REFERENCES

[1] Jerry Chu. Zero-copy safe transmit. February 2004. http://sac.sfbay/PSARC/2004/174/materials/zero-copy.pdf.

[2] Sun Microsystems Inc. *Cassini ASIC Specification Rev. 1.0*, February 2001. http://npweb.sfbay/asics/nw_asics/cassini/cas.pdf.

[3] Adi Masputra. Multidata interface design specification. September 2002. http://sac.sfbay/PSARC/2002/276/materials/mmd.pdf.

[4] Adi Masputra, Frank Dimambro, and Kacheong Poon. An efficient networking transmit mechanism for solaris: Multi-data transmit (MDT). May 2002. http://sac.sfbay/PSARC/2002/276/materials/mdt.pdf.