```
**********************************************************
  120777 Wed Apr  3 12:38:54 2013
new/usr/src/uts/common/io/mac/mac_sched.c
918 Need better IP fanout (esp. with VLANs present)
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright 2011 Joyent, Inc.  All rights reserved.
 * Copyright 2012 Nexenta Systems, Inc. All rights reserved.
 */

#include <sys/types.h>
#include <sys/callb.h>
#include <sys/sdt.h>
#include <sys/strsubr.h>
#include <sys/strsun.h>
#include <sys/vlan.h>
#include <sys/stack.h>
#include <sys/archsystm.h>
#include <inet/ipsec_impl.h>
#include <inet/ip_impl.h>
#include <inet/sadb.h>
#include <inet/ipsecesp.h>
#include <inet/ipsecah.h>
#include <inet/ip6.h>

#include <sys/mac_impl.h>
#include <sys/mac_client_impl.h>
#include <sys/mac_client_priv.h>
#include <sys/mac_soft_ring.h>
#include <sys/mac_flow_impl.h>

static mac_tx_cookie_t mac_tx_single_ring_mode(mac_soft_ring_set_t *, mblk_t *,
    uintptr_t, uint16_t, mblk_t **);
static mac_tx_cookie_t mac_tx_serializer_mode(mac_soft_ring_set_t *, mblk_t *,
    uintptr_t, uint16_t, mblk_t **);
static mac_tx_cookie_t mac_tx_fanout_mode(mac_soft_ring_set_t *, mblk_t *,
    uintptr_t, uint16_t, mblk_t **);
static mac_tx_cookie_t mac_tx_bw_mode(mac_soft_ring_set_t *, mblk_t *,
    uintptr_t, uint16_t, mblk_t **);
static mac_tx_cookie_t mac_tx_aggr_mode(mac_soft_ring_set_t *, mblk_t *,
    uintptr_t, uint16_t, mblk_t **);
```



```
typedef struct mac_tx_mode_s {
        mac_tx_srs_mode_t       mac_tx_mode;
        mac_tx_func_t           mac_tx_func;
} mac_tx_mode_t;
_____unchanged_portion_omitted_
```

```
/*
 * In general we do port based hashing to spread traffic over different
 * softrings. The below tunables allow to override that behavior. Setting one
 * (depending on IPv6 or IPv4) to B_TRUE allows a fanout based on src
 * IPv6 or IPv4 address. This behavior is also applicable to IPv6 packets
 * carrying multiple optional headers and other uncommon packet types.
 * softrings. The below tunable allows to override that behavior. Setting it
 * to B_TRUE allows to do a fanout based on src ipv6 address. This behavior
 * is also the applicable to ipv6 packets carrying multiple optional headers
 * and other uncommon packet types.
 */
boolean_t mac_src_ipv6_fanout = B_FALSE;
boolean_t mac_src_ipv4_fanout = B_FALSE;

/*
 * Pair of local and remote ports in the transport header
 */
#define PORTS_SIZE 4

/*
 * mac_rx_srs_proto_fanout
 *
 * This routine delivers packets destined to an SRS into one of the
 * protocol soft rings.
 *
 * Given a chain of packets we need to split it up into multiple sub chains
 * destined into TCP, UDP or OTH soft ring. Instead of entering
 * the soft ring one packet at a time, we want to enter it in the form of a
 * chain otherwise we get this start/stop behaviour where the worker thread
 * goes to sleep and then next packets comes in forcing it to wake up etc.
 */
static void
mac_rx_srs_proto_fanout(mac_soft_ring_set_t *mac_srs, mblk_t *head)
{
        struct ether_header             *ehp;
        struct ether_vlan_header        *evhp;
        uint32_t                        sap;
        ipha_t                          *ipha;
        uint8_t                         *dstaddr;
        size_t                          hdrsize;
        mblk_t                          *mp;
        mblk_t                          *headmp[MAX_SR_TYPES];
        mblk_t                          *tailmp[MAX_SR_TYPES];
        int                             cnt[MAX_SR_TYPES];
        size_t                          sz[MAX_SR_TYPES];
        size_t                          sz1;
        boolean_t                       bw_ctl;
        boolean_t                       hw_classified;
        boolean_t                       dls_bypass;
        boolean_t                       is_ether;
        boolean_t                       is_unicast;
        enum pkt_type                   type;
        mac_client_impl_t               *mcip = mac_srs->srs_mcip;

        is_ether = (mcip->mci_mip->mi_info.mi_nativemedia == DL_ETHER);
        bw_ctl = ((mac_srs->srs_type & SRST_BW_CONTROL) != 0);

        /*
         * If we don't have a Rx ring, S/W classification would have done
         * its job and its a packet meant for us. If we were polling on
```

```
586          * the default ring (i.e. there was a ring assigned to this SRS),
587          * then we need to make sure that the mac address really belongs
588          * to us.
589          */
590         hw_classified = mac_srs->srs_ring != NULL &&
591             mac_srs->srs_ring->mr_classify_type == MAC_HW_CLASSIFIER;

593         /*
594          * Special clients (eg. VLAN, non ether, etc) need DLS
595          * processing in the Rx path. SRST_DLS_BYPASS will be clear for
596          * such SRSs. Another way of disabling bypass is to set the
597          * MCIS_RX_BYPASS_DISABLE flag.
598          */
599         dls_bypass = ((mac_srs->srs_type & SRST_DLS_BYPASS) != 0) &&
600             ((mcip->mci_state_flags & MCIS_RX_BYPASS_DISABLE) == 0);

602         bzero(headmp, MAX_SR_TYPES * sizeof (mblk_t *));
603         bzero(tailmp, MAX_SR_TYPES * sizeof (mblk_t *));
604         bzero(cnt, MAX_SR_TYPES * sizeof (int));
605         bzero(sz, MAX_SR_TYPES * sizeof (size_t));

607         /*
608          * We got a chain from SRS that we need to send to the soft rings.
609          * Since squeues for TCP & IPv4 sap poll their soft rings (for
610          * performance reasons), we need to separate out v4_tcp, v4_udp
611          * and the rest goes in other.
612          */
613         while (head != NULL) {
614                 mp = head;
615                 head = head->b_next;
616                 mp->b_next = NULL;

618                 type = OTH;
619                 sz1 = (mp->b_cont == NULL) ? MBLKL(mp) : msgdsize(mp);

621                 if (is_ether) {
622                         /*
623                          * At this point we can be sure the packet at least
624                          * has an ether header.
625                          */
626                         if (sz1 < sizeof (struct ether_header)) {
627                                 mac_rx_drop_pkt(mac_srs, mp);
628                                 continue;
629                         }
630                         ehp = (struct ether_header *)mp->b_rptr;

632                         /*
633                          * Determine if this is a VLAN or non-VLAN packet.
634                          */
635                         if ((sap = ntohs(ehp->ether_type)) == VLAN_TPID) {
636                                 evhp = (struct ether_vlan_header *)mp->b_rptr;
637                                 sap = ntohs(evhp->ether_type);
638                                 hdrsize = sizeof (struct ether_vlan_header);
639                                 /*
640                                  * Check if the VID of the packet, if any,
641                                  * belongs to this client.
642                                  */
643                                 if (!mac_client_check_flow_vid(mcip,
644                                     VLAN_ID(ntohs(evhp->ether_tci)))) {
645                                         mac_rx_drop_pkt(mac_srs, mp);
646                                         continue;
647                                 }
648                         } else {
649                                 hdrsize = sizeof (struct ether_header);
650                         }
651                         is_unicast =
```



```
652                             ((((uint8_t *)&ehp->ether_dhost)[0] & 0x01) == 0);
653                         dstaddr = (uint8_t *)&ehp->ether_dhost;
654                 } else {
655                         mac_header_info_t               mhi;

657                         if (mac_header_info((mac_handle_t)mcip->mci_mip,
658                             mp, &mhi) != 0) {
659                                 mac_rx_drop_pkt(mac_srs, mp);
660                                 continue;
661                         }
662                         hdrsize = mhi.mhi_hdrsize;
663                         sap = mhi.mhi_bindsap;
664                         is_unicast = (mhi.mhi_dsttype == MAC_ADDRTYPE_UNICAST);
665                         dstaddr = (uint8_t *)mhi.mhi_daddr;
666                 }

668                 if (!dls_bypass) {
669                         FANOUT_ENQUEUE_MP(headmp[type], tailmp[type],
670                             cnt[type], bw_ctl, sz[type], sz1, mp);
671                         continue;
672                 }

674                 if (sap == ETHERTYPE_IP) {
675                         /*
676                          * If we are H/W classified, but we have promisc
677                          * on, then we need to check for the unicast address.
678                          */
679                         if (hw_classified && mcip->mci_promisc_list != NULL) {
680                                 mac_address_t           *map;

682                                 rw_enter(&mcip->mci_rw_lock, RW_READER);
683                                 map = mcip->mci_unicast;
684                                 if (bcmp(dstaddr, map->ma_addr,
685                                     map->ma_len) == 0)
686                                         type = UNDEF;
687                                 rw_exit(&mcip->mci_rw_lock);
688                         } else if (is_unicast) {
689                                 type = UNDEF;
690                         }
691                 }

693                 /*
694                  * This needs to become a contract with the driver for
695                  * the fast path.
696                  *
697                  * In the normal case the packet will have at least the L2
698                  * header and the IP + Transport header in the same mblk.
699                  * This is usually the case when the NIC driver sends up
700                  * the packet. This is also true when the stack generates
701                  * a packet that is looped back and when the stack uses the
702                  * fastpath mechanism. The normal case is optimized for
703                  * performance and may bypass DLS. All other cases go through
704                  * the 'OTH' type path without DLS bypass.
705                  */

707                 ipha = (ipha_t *)(mp->b_rptr + hdrsize);
708                 if ((type != OTH) && MBLK_RX_FANOUT_SLOWPATH(mp, ipha))
709                         type = OTH;

711                 if (type == OTH) {
712                         FANOUT_ENQUEUE_MP(headmp[type], tailmp[type],
713                             cnt[type], bw_ctl, sz[type], sz1, mp);
714                         continue;
715                 }

717                 ASSERT(type == UNDEF);
```

```
718                 /*
719                  * We look for at least 4 bytes past the IP header to get
720                  * the port information. If we get an IP fragment, we don't
721                  * have the port information, and we use just the protocol
722                  * information.
723                  */
724                 switch (ipha->ipha_protocol) {
725                 case IPPROTO_TCP:
726                         type = V4_TCP;
727                         mp->b_rptr += hdrsize;
728                         break;
729                 case IPPROTO_UDP:
730                         type = V4_UDP;
731                         mp->b_rptr += hdrsize;
732                         break;
733                 default:
734                         type = OTH;
735                         break;
736                 }
737
738                 FANOUT_ENQUEUE_MP(headmp[type], tailmp[type], cnt[type],
739                     bw_ctl, sz[type], sz1, mp);
740         }
741
742         for (type = V4_TCP; type < UNDEF; type++) {
743                 if (headmp[type] != NULL) {
744                         mac_soft_ring_t                 *softring;
745
746                         ASSERT(tailmp[type]->b_next == NULL);
747                         switch (type) {
748                         case V4_TCP:
749                                 softring = mac_srs->srs_tcp_soft_rings[0];
750                                 break;
751                         case V4_UDP:
752                                 softring = mac_srs->srs_udp_soft_rings[0];
753                                 break;
754                         case OTH:
755                                 softring = mac_srs->srs_oth_soft_rings[0];
756                         }
757                         mac_rx_soft_ring_process(mcip, softring,
758                             headmp[type], tailmp[type], cnt[type], sz[type]);
759                 }
760         }
761 }
763 int     fanout_unaligned = 0;
763 int     fanout_unalligned = 0;
765 /*
766  * mac_rx_srs_long_fanout
767  *
768  * The fanout routine for VLANs, and for anything else that isn't performing
769  * explicit dls bypass.  Returns -1 on an error (drop the packet due to a
770  * malformed packet), 0 on success, with values written in *indx and *type.
768  * The fanout routine for IPv6
771  */
772 static int
773 mac_rx_srs_long_fanout(mac_soft_ring_set_t *mac_srs, mblk_t *mp,
774     uint32_t sap, size_t hdrsize, enum pkt_type *type, uint_t *indx)
775 {
776         ip6_t           *ip6h;
777         ipha_t          *ipha;
778         uint8_t         *whereptr;
779         uint_t          hash;
780         uint16_t        remlen;
781         uint8_t         nexthdr;
```



```
782         uint16_t        hdr_len;
783         uint32_t        src_val;
781         if (sap == ETHERTYPE_IPV6) {
784         boolean_t       modifiable = B_TRUE;
785         boolean_t       v6;
787         ASSERT(MBLKL(mp) >= hdrsize);
789         if (sap == ETHERTYPE_IPV6) {
790                 v6 = B_TRUE;
791                 hdr_len = IPV6_HDR_LEN;
792         } else if (sap == ETHERTYPE_IP) {
793                 v6 = B_FALSE;
794                 hdr_len = IP_SIMPLE_HDR_LENGTH;
795         } else {
796                 *indx = 0;
797                 *type = OTH;
798                 return (0);
799         }
801         ip6h = (ip6_t *)(mp->b_rptr + hdrsize);
802         ipha = (ipha_t *)ip6h;
804         if ((uint8_t *)ip6h == mp->b_wptr) {
787                 if ((unsigned char *)ip6h == mp->b_wptr) {
805                 /*
806                  * The first mblk_t only includes the mac header.
807                  * Note that it is safe to change the mp pointer here,
808                  * as the subsequent operation does not assume mp
809                  * points to the start of the mac header.
810                  */
811                 mp = mp->b_cont;
813                 /*
814                  * Make sure the IP header points to an entire one.
797                          * Make sure ip6h holds the full ip6_t structure.
815                  */
816                 if (mp == NULL)
817                         return (-1);
819                 if (MBLKL(mp) < hdr_len) {
802                         if (MBLKL(mp) < IPV6_HDR_LEN) {
820                         modifiable = (DB_REF(mp) == 1);
822                         if (modifiable && !pullupmsg(mp, hdr_len))
805                                 if (modifiable &&
806                                     !pullupmsg(mp, IPV6_HDR_LEN)) {
823                                 return (-1);
824                 }
809                         }
826                 ip6h = (ip6_t *)mp->b_rptr;
827                 ipha = (ipha_t *)ip6h;
828         }
830         if (!modifiable || !(OK_32PTR((char *)ip6h)) ||
831             ((uint8_t *)ip6h + hdr_len > mp->b_wptr)) {
815                     ((unsigned char *)ip6h + IPV6_HDR_LEN > mp->b_wptr)) {
832                 /*
833                  * If either the IP header is not aligned, or it does not hold
834                  * the complete simple structure (a pullupmsg() is not an
835                  * option since it would result in an unaligned IP header),
836                  * fanout to the default ring.
837                  *
838                  * Note that this may cause packet reordering.
```

```
                        * If either ip6h is not alligned, or ip6h does not
                        * hold the complete ip6_t structure (a pullupmsg()
                        * is not an option since it would result in an
                        * unalligned ip6h), fanout to the default ring. Note
                        * that this may cause packets reordering.
                 */
                *indx = 0;
                *type = OTH;
                fanout_unaligned++;
                        fanout_unalligned++;
                return (0);
        }

        /*
         * Extract next-header, full header length, and source-hash value
         * using v4/v6 specific fields.
         */
        if (v6) {
                remlen = ntohs(ip6h->ip6_plen);
                nexthdr = ip6h->ip6_nxt;
                src_val = V4_PART_OF_V6(ip6h->ip6_src);

                if (remlen < MIN_EHDR_LEN)
                        return (-1);
                /*
                 * Do src based fanout if below tunable is set to B_TRUE or
                 * when mac_ip_hdr_length_v6() fails because of malformed
                 * packets or because mblks need to be concatenated using
                 * packets or because mblk's need to be concatenated using
                 * pullupmsg().
                 */
                if (mac_src_ipv6_fanout || !mac_ip_hdr_length_v6(ip6h,
                    mp->b_wptr, &hdr_len, &nexthdr, NULL)) {
                        goto src_based_fanout;
                }
        } else {
                hdr_len = IPH_HDR_LENGTH(ipha);
                remlen = ntohs(ipha->ipha_length) - hdr_len;
                nexthdr = ipha->ipha_protocol;
                src_val = (uint32_t)ipha->ipha_src;
                /*
                 * Catch IPv4 fragment case here.  IPv6 has nexthdr == FRAG
                 * for its equivalent case.
                 */
                if (mac_src_ipv4_fanout ||
                    (ntohs(ipha->ipha_fragment_offset_and_flags) &
                    (IPH_MF | IPH_OFFSET)) != 0) {
                        goto src_based_fanout;
                }
        }
        if (remlen < MIN_EHDR_LEN)
                return (-1);
        whereptr = (uint8_t *)ip6h + hdr_len;

        /* If the transport is one of below, we do port/SPI based fanout */
                /* If the transport is one of below, we do port based fanout */
        switch (nexthdr) {
        case IPPROTO_TCP:
        case IPPROTO_UDP:
        case IPPROTO_SCTP:
        case IPPROTO_ESP:
                /*
                 * If the ports or SPI in the transport header is not part of
                        * If the ports in the transport header is not part of
                 * the mblk, do src_based_fanout, instead of calling
                 * pullupmsg().
```



```
                 */
                if (mp->b_cont == NULL || whereptr + PORTS_SIZE <= mp->b_wptr)
                        break;  /* out of switch... */
                /* FALLTHRU */
        default:
                        if (mp->b_cont != NULL &&
                            whereptr + PORTS_SIZE > mp->b_wptr) {
                goto src_based_fanout;
        }
                        break;
                default:
                        break;
                }

        switch (nexthdr) {
        case IPPROTO_TCP:
                hash = HASH_ADDR(src_val, *(uint32_t *)whereptr);
                *indx = COMPUTE_INDEX(hash, mac_srs->srs_tcp_ring_count);
                        hash = HASH_ADDR(V4_PART_OF_V6(ip6h->ip6_src),
                            *(uint32_t *)whereptr);
                        *indx = COMPUTE_INDEX(hash,
                            mac_srs->srs_tcp_ring_count);
                *type = OTH;
                break;

        case IPPROTO_UDP:
        case IPPROTO_SCTP:
        case IPPROTO_ESP:
                if (mac_fanout_type == MAC_FANOUT_DEFAULT) {
                        hash = HASH_ADDR(src_val, *(uint32_t *)whereptr);
                                hash = HASH_ADDR(V4_PART_OF_V6(ip6h->ip6_src),
                                    *(uint32_t *)whereptr);
                        *indx = COMPUTE_INDEX(hash,
                            mac_srs->srs_udp_ring_count);
                } else {
                        *indx = mac_srs->srs_ind % mac_srs->srs_udp_ring_count;
                                *indx = mac_srs->srs_ind %
                                    mac_srs->srs_udp_ring_count;
                        mac_srs->srs_ind++;
                }
                *type = OTH;
                break;

                        /* For all other protocol, do source based fanout */
                default:
                        goto src_based_fanout;
        }
        } else {
                *indx = 0;
                *type = OTH;
        }
        return (0);

src_based_fanout:
        hash = HASH_ADDR(src_val, (uint32_t)0);
        hash = HASH_ADDR(V4_PART_OF_V6(ip6h->ip6_src), (uint32_t)0);
        *indx = COMPUTE_INDEX(hash, mac_srs->srs_oth_ring_count);
        *type = OTH;
        return (0);
}
```

_____***unchanged_portion_omitted_***