Sea of Memories (SOM) and packet generation

The scope of this document is limited to SOM layout, table placement, table programming and packet generation. It is assumed that the way the DRMT processors interact with SOMs is already known to the reader.

Overview:

The number of SOMs could be greater than or equal to the number of DRMT processors. Each SOM has a set of memory blocks (SRAMs and TCAMs) that can be ganged together to form wider or deeper logical tables. A high level view of how packet flows in DRMT and how the processors interact with SOM is as follows

**Proc. 1**

Packets in

**Proc. 2**

**Proc. N**

Memory cluster 1

Memory cluster 2

Memory cluster N

Packets out

Queues

..…

Pkt

**Distrib.**

Pkt

Pkt

Pkt

Crossbar for

search keys & results

..…

Each SOM typically has:

* + SRAMs
  + TCAMs
  + Key Segements (on which keys are received)
  + Data Segements (on which data from tables is driven)
  + Controllers (CAM, Read, Write, Hash – one controller for each table)

Each SOM is independent of the other. SOMs can be homogeneous or heterogeneous in terms of SRAMs or TCAMs availability.

There are two perspectives of looking at P4 tables, logical and physical. Logical tables correspond one-to-one with P4 tables in the P4 program where as Physical tables are physical SRAMs and TCAMs in a SOM.

Every P4 table has:

* + Exactly one search key, explicitly defined in the P4 program, containing 0 or more fields from the packet vector
    - There can be tables with 0 fields (and thus 0 bits) in the search key. These are effectively config registers that return one of several control-plane programmable actions.
  + One or more (compound) actions, each with:
    - 0 or more action parameters, where each parameter is a bit vector
    - 0 or more primitive actions
  + No. of entries in a table

Associative data for logical table T is determined by the actions defined for table T.

* The table action with the maximum number of bits in its action parameters determines the width of the associative data.
* In addition to the action parameters, associative data must contain an “action id” to distinguish among possible actions for each table entry.

Table placement:

To place a table into an SOM, the following conditions must be honoured

* Availability of key segments without any conflicts
* Availability of data segments without any conflicts
* Availability of controllers depending on table implementation
* Availability of SRAMs or TCAMs for storing keys
* Availability of SRAMs for storing associative data

Depending upon the key type of a table either SRAM or TCAM will be used. The commonly used key types are exact, ternary and lpm. For exact key type, SRAMs are used and for the other types TCAMs are used.

For exact key implementation direct SRAM will be used if the table size is small. Else hash (cuckoo hash) will be used. For ternary and lpm implementations, TCAM grids are used. TCAMs are arranged in 2D array. Though there are no restrictions on how bit-side packing and word-side packing are implemented usually bit side packing is done along the same strip (columns) and word side packing is done row wise. There are certain considerations on how bit side packing can be done as we have to honour priority encoding for TCAMs.

All this configuration has to be handled by software i.e. for a given table, the compiler has to determine which SOM the table has to be placed and also configure the key segments and data segments. Along with these controllers, srams/tcams have to configured. This configuration is passed to hardware side.

Code coverage and packet generation:

The path of a packet depends on given P4 program (parser and control blocks), packet data and the entries programmed for each table. For complete code coverage we have to identify all possible paths in the parser block of P4 program and all possible paths in the control blocks which is called Control Flow Graph (CFG) and for each path in a CFG we have to identify the tables and all possible combinations of actions for each table along the path. In summary, full code coverage is a cross product of number of paths in a parser, number of paths in a CFG and all possible combinations of actions for each table along the path.

The input packet generation starts with a random bit vector of size 2048 bits. Depending upon the path considered in the parser block, appropriate fields in the bit vector are set. For e.g. if the path under consideration is ethernet -> ipv4 -> tcp, then ethertype field of ethernet header in the bit vector would be set to 0x800 and fragoffset, ihl and protocol fields of ipv4 header in the bit vector would be set to 0x0, 0x5 and 0x6 respectively. Once these fields are set all the conditionals and assignments along the current CFG path have to be gathered and solved using a solver (E.g. Z3) and the bit vector will be updated with all the solved values of the packet fields in the constraints. The bit vector now becomes the input packet. If solver could not solve the constraints then most likely the parser path selected (or packet type) is not suitable for the path in CFG and this case is ignored.

All the matches and actions are executed on the packet along the CFG path. For each table match, the key field defined in the table is extracted from the packet and used as key for that table in the table programming part to make sure that packet definitely finds a match in that table later on in the simulations. As the action for that table is also prefixed, all the instructions from that action block are applied on that packet. So the packet is evaluated continuously until the end of CFG. At this point the packet becomes an expected packet. So for each iteration, we will have input packet, table key and data and expected packet.

Example:

|  |
| --- |
| struct standard\_metadata\_t {  bit<8> ingress\_port;  bit<8> egress\_port;  }  struct fwd\_metadata\_t {  bit<24> l2ptr;  bit<24> out\_bd;  }  struct l3\_metadata\_t {  bit<16> lkp\_outer\_l4\_sport;  bit<16> lkp\_outer\_l4\_dport;  }  header ethernet\_t {  bit<48> dstAddr;  bit<48> srcAddr;  bit<16> etherType;  }  header ipv4\_t {  bit<4> version;  bit<4> ihl;  bit<8> diffserv;  bit<16> totalLen;  bit<16> identification;  bit<3> flags;  bit<13> fragOffset;  bit<8> ttl;  bit<8> protocol;  bit<16> hdrChecksum;  bit<32> srcAddr;  bit<32> dstAddr;  }  header ipv6\_t {  bit<4> version;  bit<8> trafficClass;  bit<20> flowLabel;  bit<16> payloadLen;  bit<8> nextHdr;  bit<8> hopLimit;  bit<128> srcAddr;  bit<128> dstAddr;  }  header tcp\_t {  bit<16> srcPort;  bit<16> dstPort;  bit<32> seqNo;  bit<32> ackNo;  bit<4> dataOffset;  bit<4> res;  bit<8> flags;  bit<16> window;  bit<16> checksum;  bit<16> urgentPtr;  }  header udp\_t {  bit<16> srcPort;  bit<16> dstPort;  bit<16> length;  bit<16> checksum;  }  struct metadata {  @name("fwd\_metadata")  fwd\_metadata\_t fwd\_metadata;  @name("l3\_metadata")  l3\_metadata\_t l3\_metadata;  }  struct headers {  @name("ethernet")  ethernet\_t ethernet;  @name("ipv4")  ipv4\_t ipv4;  @name("ipv6")  ipv6\_t ipv6;  @name("tcp")  tcp\_t tcp;  @name("udp")  udp\_t udp;  }  headers() hdr;  metadata() meta;  standard\_metadata\_t() standard\_metadata;  parser ParserImpl(packet\_in packet, out headers hdr, inout metadata meta, inout standard\_metadata\_t standard\_metadata) {  @name("parse\_ethernet") state parse\_ethernet {  packet.extract(hdr.ethernet);  transition select(hdr.ethernet.etherType) {  16w0x800: parse\_ipv4;  16w0x86dd: parse\_ipv6;  default: accept;  }  }  @name("parse\_ipv4") state parse\_ipv4 {  packet.extract(hdr.ipv4);  transition select(hdr.ipv4.fragOffset, hdr.ipv4.ihl, hdr.ipv4.protocol) {  (13w0x0, 4w0x5, 8w0x6): parse\_tcp;  (13w0x0, 4w0x5, 8w0x11): parse\_udp;  default: accept;  }  }  @name("parse\_ipv6") state parse\_ipv6 {  packet.extract(hdr.ipv6);  transition select(hdr.ipv6.nextHdr) {  8w0x6: parse\_tcp;  8w0x11: parse\_udp;  default: accept;  }  }  @name("parse\_tcp") state parse\_tcp {  packet.extract(hdr.tcp);  meta.l3\_metadata.lkp\_outer\_l4\_sport = hdr.tcp.srcPort;  meta.l3\_metadata.lkp\_outer\_l4\_dport = hdr.tcp.dstPort;  transition accept;  }  @name("parse\_udp") state parse\_udp {  packet.extract(hdr.udp);  meta.l3\_metadata.lkp\_outer\_l4\_sport = hdr.udp.srcPort;  meta.l3\_metadata.lkp\_outer\_l4\_dport = hdr.udp.dstPort;  transition accept;  }  @name("start") state start {  transition parse\_ethernet;  }  }  control egress(inout headers hdr, inout metadata meta, inout standard\_metadata\_t standard\_metadata) {  @name("\_nop") action \_nop() {  }  @name("NoAction") action NoAction() {  }  @name("rewrite\_mac") action rewrite\_mac(bit<48> smac) {  hdr.ethernet.srcAddr = smac;  }  @name("\_drop") action \_drop() {  mark\_to\_drop();  }  @name("send\_frame") table send\_frame {  actions = {  rewrite\_mac;  @default\_only NoAction;  }  key = {  hdr.ipv4.srcAddr: exact;  }  size = 256;  default\_action = NoAction();  }  apply {  send\_frame.apply();  }  }  control ingress(inout headers hdr, inout metadata meta, inout standard\_metadata\_t standard\_metadata) {  @name("\_nop") action \_nop() {  }  @name("NoAction") action NoAction() {  }  @name("set\_l2ptr") action set\_l2ptr(bit<24> l2ptr) {  meta.fwd\_metadata.l2ptr = l2ptr;  }  @name("set\_l2ptr") action dummy\_ac1(bit<24> out\_bd) {  meta.fwd\_metadata.out\_bd = out\_bd;  meta.fwd\_metadata.l2ptr = 24w0;  }  @name("\_drop") action \_drop() {  mark\_to\_drop();  }  @name("set\_bd\_dmac\_intf") action set\_bd\_dmac\_intf(bit<24> bd, bit<48> dmac, bit<2> intf) {  meta.fwd\_metadata.out\_bd = bd;  hdr.ethernet.dstAddr = dmac;  standard\_metadata.egress\_port = intf;  }  @name("ipv4\_da\_lpm") table ipv4\_da\_lpm {  actions = {  set\_l2ptr;  @default\_only NoAction;  }  key = {  hdr.ipv4.dstAddr: exact;  hdr.tcp.dstPort: exact;  }  default\_action = NoAction();  }  @name("mac\_da") table mac\_da {  actions = {  set\_bd\_dmac\_intf;  @default\_only NoAction;  }  key = {  hdr.ethernet.dstAddr: exact;  }  default\_action = NoAction();  }  @name("mac\_da") table dummy {  actions = {  dummy\_ac1;  @default\_only NoAction;  }  key = {  hdr.ethernet.dstAddr: exact;  }  default\_action = NoAction();  }    apply {  if( hdr.ethernet.srcAddr==123456 && hdr.ethernet.dstAddr > 2){  if (hdr.ethernet.srcAddr !=hdr.ethernet.dstAddr)  hdr.ethernet.dstAddr = hdr.ethernet.srcAddr; //P1  else if ((hdr.ipv4.isValid()) && !(hdr.ipv6.isValid()))  hdr.ethernet.dstAddr = hdr.ethernet.srcAddr + 2; //P2  else  hdr.ethernet.dstAddr = hdr.ethernet.srcAddr + 3; //P3  }  ipv4\_da\_lpm.apply();  }  }  control DeparserImpl(packet\_out packet, in headers hdr) {  apply {  packet.emit(hdr.ethernet);  packet.emit(hdr.ipv4);  packet.emit(hdr.tcp);  }  }  control verifyChecksum(in headers hdr, inout metadata meta) {  apply {  }  }  control computeChecksum(inout headers hdr, inout metadata meta) {  apply {  }  }  V1Switch(ParserImpl(), verifyChecksum(), ingress(), egress(), computeChecksum(), DeparserImpl()) main; |
|  |

In the above example, there are four possible paths in parser

ethernet, ipv4, tcp

ethernet, ipv4, udp

ethernet, ipv6, tcp

ethernet, ipv6, udp

From the control blocks there are four possible paths

if( hdr.ethernet.srcAddr==123456 && hdr.ethernet.dstAddr > 2){

if (hdr.ethernet.srcAddr !=hdr.ethernet.dstAddr)

hdr.ethernet.dstAddr = hdr.ethernet.srcAddr; //P1

}

if( hdr.ethernet.srcAddr==123456 && hdr.ethernet.dstAddr > 2){

if (!(hdr.ethernet.srcAddr !=hdr.ethernet.dstAddr)) && ((hdr.ipv4.isValid()) && !(hdr.ipv6.isValid()))

hdr.ethernet.dstAddr = hdr.ethernet.srcAddr + 2; //P2

}

if( hdr.ethernet.srcAddr==123456 && hdr.ethernet.dstAddr > 2){

if (!(hdr.ethernet.srcAddr !=hdr.ethernet.dstAddr)) && !((hdr.ipv4.isValid()) && !(hdr.ipv6.isValid()))

hdr.ethernet.dstAddr = hdr.ethernet.srcAddr + 3; //P3

}

if !(hdr.ethernet.srcAddr==123456 && hdr.ethernet.dstAddr > 2){

}

Two tables are used in the full CFG, ipv4\_da\_lpm and send\_frame. The list of possible actions for ipv4\_da\_lpm are set\_l2ptr and NoAction. For table send\_frame, possible actions are rewrite\_mac and NoAction. So total combinations of actions are 2x2=4.

Combining all these together the total number of packets to generate for complete code coverage is 4x4x4=64.