Hardware-based IP forwarding

Adnan Aziz

The University of Texas

adnan@ece.utexas.edu

References:

- A. McAuley and P. Francis. Fast Routing Table Lookup Using CAMS. *IEEE Infocom*, 1993.
- P. Gupta, S. Lin, and N. McKeown. Routing Lookups in Hardware at Memory Access Speeds. *IEEE Infocom*, 1998.
- A. Prakash and A. Aziz. OC-3072 Packet Classification Using BDDs and Pipelined SRAMs. *Hot Interconnects*, 2001.

CAMs

CAM — use to find out if anything in memory matches certain key value

- cell contains XOR gate which compares cell value with value on bitline
- match ⇒ do nothing; no match ⇒ pull matchline low
- connect all bits in word to precharged matchline ⇒ match high iff value in memory matches key
  - can have key-value pairs: search, read
  - if value is very long, store index to location instead

Some applications — can do without addressing

- write to next free location, search, read, delete location matching key

Flat addresses

When dealing with flat addresses — Ethernet bridge, simple IP address filtering

- initialize: load CAM words with MACs, output port IDs
- incoming frame — search for dest MAC, read port

Cannot use for IP forwarding — only one match, no hatch

Hierarchical addresses

First try: use "mask register" in conjunction with CAM

- start with mask register set to maximum 1s, iteratively decrease

Need multiple cycles because (1.) variable fields, (2.) best match desired

Solution: use multiple CAMs

- each has own mask register (gain some flexibility)
- priority encoder selects best match

DRAM lookup

Why can't we perform IP forwarding using "direct addressing"?

- IP address $\equiv$ 32 bit integer $\Rightarrow$ store output port corresponding to destination IP x at DRAM memory location x
- store $2^{32} \sim 4000000000$ words, each 4–8 bits ($\lg$ number of ports)

What if no prefix was longer than 24 bits?

- store $2^{24} \sim 16000000$ words, each 4–8 bits
- quite doable

Problem: some prefixes are longer than 24 bits (much fewer

than 2^{15})

- need to treat a few of the 2^{24} entries as "special cases"
  - those which are prefixes of a prefix of length greater than 24 bits
  - mark using a flag bit

Detailed implementation:

- store 16 bits per 24-bit prefix, index based on first 24 bits of dest address
- first bit not set: not special case, remaining 15 bits are next hop
- first bit set: special case
  - multiply remaining 15 bits by 256, add to 8 bits remaining in dest address

Ternary CAMs

Logically, cell stores $0, 1, *$ (implementation: two-bit encoding), about twice size of ordinary CAM

- use implicit priority — first match wins
  - +fast, -slow updates
- add priority field — iterate priorities
  - -slower access, more complex; +fast updates
- multiple CAMs, each with own priority
  - +fast access and updates; +complexity

CAMs — summary

Advantages:

- fast: exploit parallelism
- generalize to (simple) classification on multiple fields

Disadvantages:

- expensive: large dies
- low density: CYNSE70128 — 0.13 micron CMOS, 128k 36-bit entries, 50Mpps
  - other configurations: 64k/72, 32k/144, 16k/72
  - synchronous pipelined operation
- lots of switching: very high power consumption

Updating DRAM

Core router: few hundred updates per second

1. use two memory banks
2. processor sends update messages to DRAM
3. DRAM supports range updates: "change n entries starting at a_0 to v"
   - CPU may still need to send a large number of update messages for a single prefix because longer prefixes contained within it should not change
   - smarter DRAM: can tell which entries are of a higher priority (exploiting prefix containment property)

Synthesis formulation

Logic synthesis — compute optimized gate-level implementation of Boolean logic functions

- Router with 2^k output ports $\Rightarrow$ forwarding table encodes Boolean function $\mathcal{F}_T : 2^{32} \mapsto 2^k$
  - run logic synthesis on equations for $\mathcal{F}_T$
  - float 32 bit destination address to resulting circuit

Differentiator: forwarding table encoded in circuit, not an input to circuit

- use this to address secondary table

Discussion

Advantages:

- Pipeline $\Rightarrow$ 20 Mpps (50ns DRAM), 33MBytes, almost any forwarding table, simple hardware

Main problem: updates complex

- other problems: inefficient use of memory, no IPv6, classification on multiple fields

Using BDDs to represent $\mathcal{F}_T$

Ex. $\mathcal{T} = \{(0*, 0), (1*, 1), (01*, 1), (011, 0), (10*, 0), (011, 0)\}$.

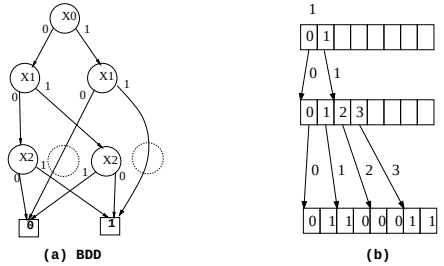


Figure 1: (a) BDD for $\mathcal{F}_T : \{0,1\}^3 \mapsto \{0,1\}$. (b) In-memory representation.

Architecture

Consider $\mathcal{F}_T \mapsto \{0,1\}$, i.e., 2 output ports.

Two ways to evaluate $\mathcal{F}_T$ from its BDD:

- Bottom up: map each BDD node to two muxes, output at top
  - nice implementation in pass-transistor logic
- Top down: map BDD for $\mathcal{F}_T$ to pipeline of 32 SRAMs
  - k-th SRAM holds BDD nodes for level k
  - {output of k-th level, $k+1$-th input bit} drives address lines of $k+1$-th SRAM

Technical issues

Table changes: need reprogrammability

- Natural candidate: FPGAs
  - Experimented with mapping forwarding table from CAIDA containing 57688 entries to Xilinx FPGAs
    - full-blown synthesis: ran for a day with no solution
  - place-and-route mux-based logic netlist for $\mathcal{F}_M$
    - 24 hours to complete LSB
    - delay of 85 ns — not competitive

BDDs

Binary Decision Diagrams: graph based representation of Boolean functions

- terminology: $f_x \triangleq f(\ldots, x = 1, \ldots)$, read as f cofactored wrt x
- BDD — recursive application of Shannon expansion: $f = x \cdot f_x \; + \; x' \cdot f_{x'}$
  - splitting variables occur in fixed order
  - nodes with equal children removed
  - isomorphic subtrees merged
- compact, easy to manipulate

Experiments – I

Forwarding table for MAE-WEST obtained from CAIDA:

- 57688 prefixes and 62 output ports
- maximum number of nodes at any level is 6803 (at level 20)
- build time is 20 seconds (P-III/500MHz/256Mbytes)

Observation: $\leq 2^c$ BDD nodes at any level $\Rightarrow$ suffices to store $2 \cdot 2^c$ c-bit words in each SRAM

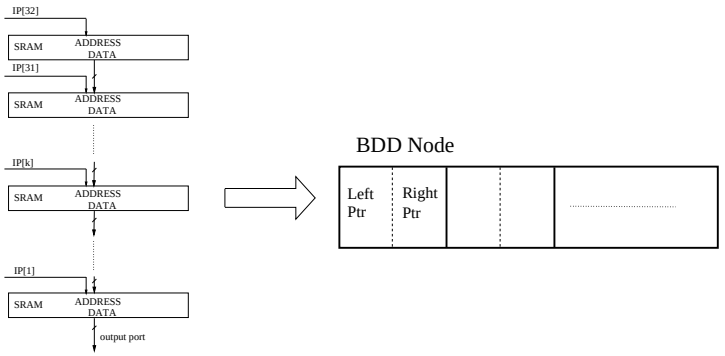
- Suffices to provision 16K nodes per SRAM
  - $2 \cdot 16K \cdot \log_2 16K <$ 64 Kbytes per SRAM

Experiments – II

Bit	No. of nodes	Bit	No. of nodes
14	2164	20	6803
15	2678	21	5207
16	3675	22	2745
17	5096	23	1001
18	6093	24	335
19	6745		

Table 1: |BDD nodes| vs. level

Architecture details



Extensions

- Allocate 16K nodes for each level $\Rightarrow$ can skip the first 14 levels
- Almost all prefixes are at most 24 bits long $\Rightarrow$ can skip the last 10 levels (need some auxiliary logic)
- Use mvBDDs to handle arbitrary number of output ports

Updates: compute new BDD offline

- one prefix — worst case need to change 32 BDD nodes
  - very little data, download via serial link
  - write all SRAMs in parallel

Tries

Can make a trie-like data structure and map to pipeline of SRAM banks

- Not exactly a trie — can't skip levels
- Additional logic to bypassing subsequent stages when longest prefix matched

Experimentally — for MAE-WEST forwarding table, need 183304 trie nodes

- compare to 45063 BDD nodes
- each trie node needs additional byte for port ID

Experiments – III

$\|P\|$	$\|BDD\|$	$\|BDD[14:24]\|$	$\|M\|$	*time*
11546	23308	18512	19128	7.2
23132	34065	28042	28918	9.3
28769	37847	31444	32445	11.4
34550	40855	34211	35264	13.3
40222	43777	36763	37926	15.2
46092	46370	39119	40363	17.0
51867	48189	40726	42014	18.7
57668	45063	40995	42188	20.6

Table 2: |BDD nodes| vs. number of prefixes

Hardware issues

Regular architecture, minimal control logic

- 64 Kbyte SRAM in 0.18 micron CMOS is 2500×1500 square microns, dissipates 0.25 watts

12 such banks ⇒ die size of 7.5mm × 6mm, power requirement of 1.5W

P-IV: single cycle access to L2 cache at 1.5 GHz (0.66 ns access time).

- budgeting 0.33 ns for interconnect delay, cycle time is 1 ns

Perform 1,000,000,000 matches per second

- 160 bit IP packets 160 Gbps = OC-3072 (or 16 OC-192 links)

Optimum height reduction

Class projects EE382M, Fall 2001 — Mandal & Kotla, Tong

Given BDD on N variables, want to reduce height to L

- $\binom{N-1}{L-1}$ ways of grouping variables to achieve this
- desirable to pick grouping which yields smallest DD

How to compute grouping which yields smallest DD, other than trying all $\binom{N-1}{L-1}$ combinations?

- think recursion: define $D(i,j,l)$ to be min memory when collapsing variables from index i to j (inclusive) to l levels

We want — $D(1,N,L)$

- $D(i,j,1)$ is easy — , know number of nodes at level i, each node at level i will have 2^{j-i} pointers to children, number children is number of nodes with index $j+1$ in original BDD
- $D(i,j,l) = \min_{i \le \kappa < j} \left(\min_{t<l} (D(i,\kappa,t) + D(\kappa+1,j,l-t)) \right)$

Cache intermediate computations to keep from becoming exponential

- space complexity $\Theta(N^2L)$, time complexity $\Theta(N^3L^2)$
- actual time complexity can be improved by special cases

Results: 32 $\rightarrow$ 11 levels minimal addition RAM, 32 $\rightarrow$ 6 levels 50% more RAM, compute in minutes

System design issues

- Feed 4 IP addresses in each cycle $\Rightarrow$ 128 input pins, pin bandwidth = 250 Mhz.
- Explore tradeoffs — 4 deep pipe, 125 Mpps

Height reduction

Reduce height by grouping variables:

- Ex. $\{x1, x2, x3\}, \{x4\}, \{x5, x6\}$ — first split is 8-way, second 2-way, third is 4-way
- results in DD on multivalued variables

No memory savings, so why reduce height?

- shallower pipe
- faster if performed in software