Hashing Round-down Prefixes for Rapid Packet Classification

Fong Pong and Nian-Feng Tzeng*

*Center for Advanced Computer Studies
University of Louisiana, Lafayette, USA

Outline

- Packet Classification
- Review of Existing Decision Tree and Hash Table-based Methods
- The HaRP (Hash Round-down Prefixes) Design
- Evaluation Results
- Conclusion

Packet Classification

- Perform action *A* on packets of type *T*, from *S* to *D, …*
  - Packet Filtering – Deny/Accept
  - Policy Routing – Send via designated network
  - Accounting & Billing – Precedence and accounting
  - QoS, Drop Precedence, Rate Limiting or Traffic Shaping
- Fields used can be widely varying
  - Source IP (prefix)
  - Destination IP (prefix)
  - Transport port numbers (Range)
  - Protocol number (Range)
  - VLAN, Flag, …
- Challenges
  - High speed/throughput
  - Low storage for growing number of rules
  - Incremental update for dynamic environments
  - Adaptive to changing rule specifications for different purposes

BROADCOM. Connecting everything

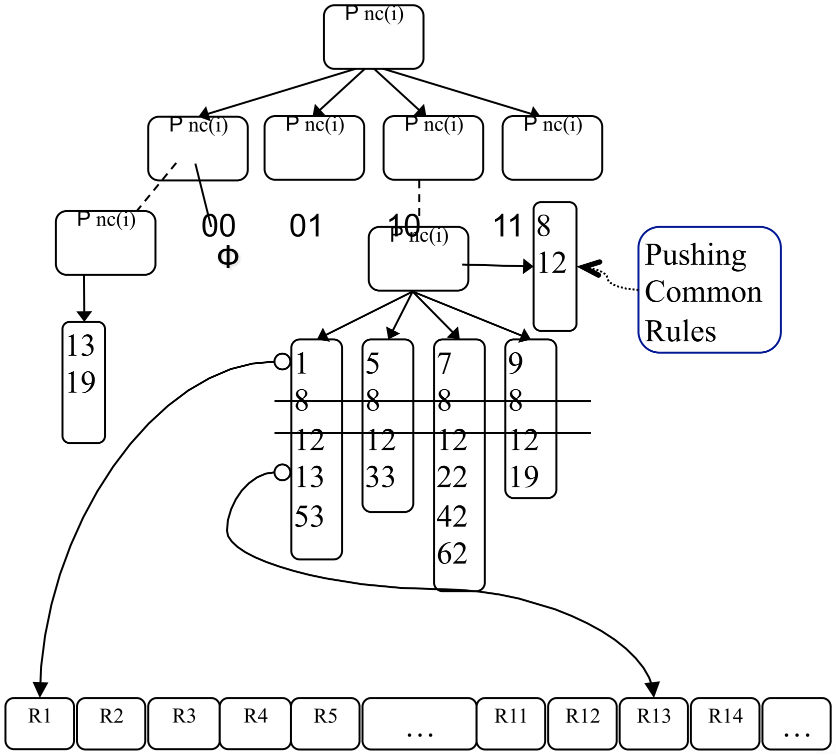
Prior Arts

Decision Tree-Based Methods (HyperCuts)

- An "*m*-ary" decision tree, at each node
  - max *m* children, $m = \prod_{i=1}^{D} nc(i)$
  - "cuts" made to multiple dimensions
- Challenges
  - Tree size explosion, sensitive to
    - selection of dimensions
    - number of cuts per dimension
    - wildcard fields (e.g. (SIP=*, DIP))
  - Difficulty in performing incremental updates
- Refinements
  - "Dead pointer" elimination; careful tuning of a space factor (SF),

$$\#splits \leq SF \times \sqrt{\#rules\ holding\ true\ at\ the\ node}$$

  - Use of "Extended Bit Map" to pack pointers in consecutive locations
  - Push Common Rules to intermediate nodes

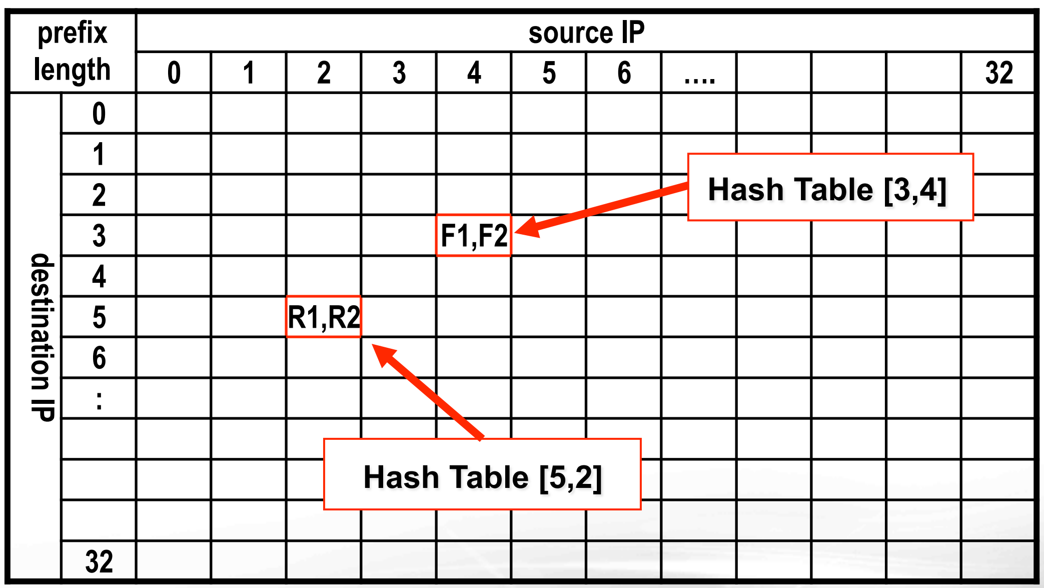


The real rules, stored in a consecutive array

BROADCOM. Connecting everything®

Hash Table-Based (Tuple Space)

- What is a tuple?
  - A vector of *k* integer elements, specifying the number of bits of fields used to form the hash key
  - For example, a 2-D filter tuple (3, 4) means destination IP DIP|3 and source IP SIP|4
- Each tuple is realized by a hash table



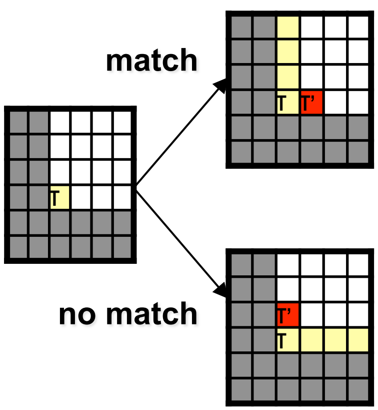
Rules

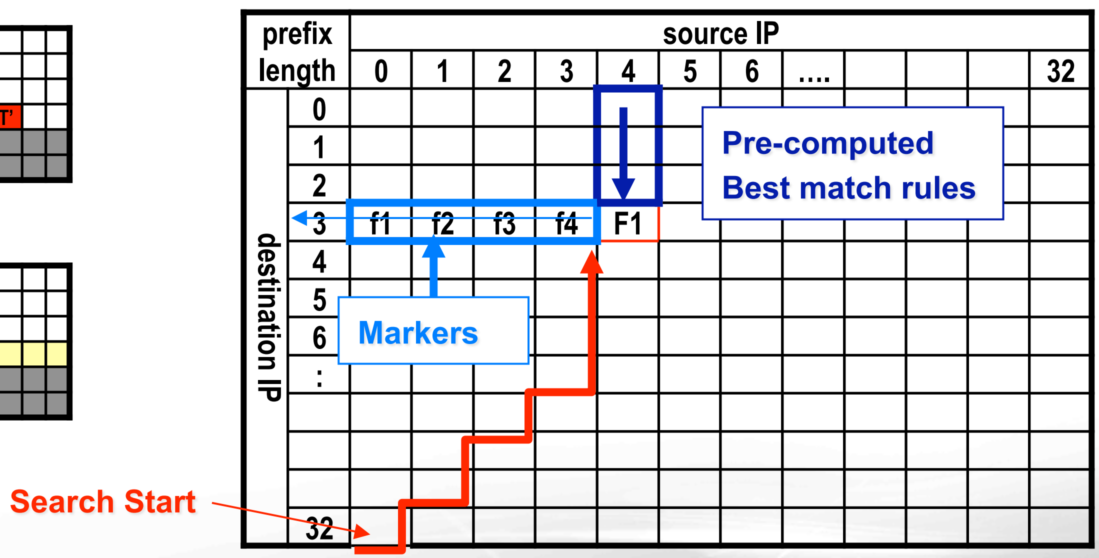
F1: 101*, 1110*
F2: 110*, 0101*

R1: 10111*, 10*
R2: 11010*, 01*

Challenges and Optimization

- Identify a tuple
  - e.g. (216.31.219.19, 69.147.114.16, 80, 2408, TCP), how many bits needed for hash keys?
- Reduce number of hash probes and keep small hash tables
- Optimization schemes include Tuple pruning, Rectangle search, Binary Search on Columns, Diagonal-based Search





Rules

F1: 101*, 1110*

Markers

f4: 101* 111*

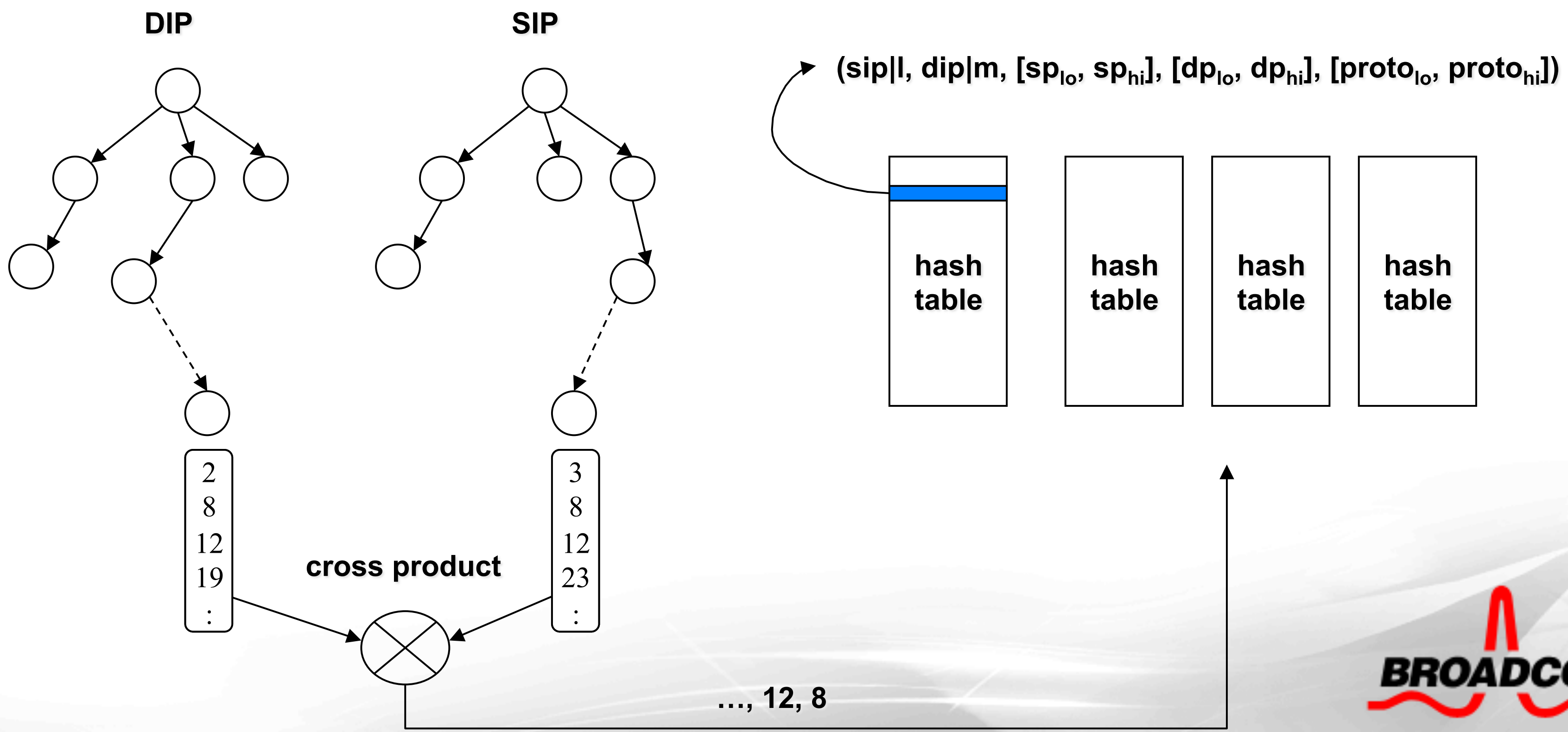
f3: 101* 11*

f2: 101* 1*

f1: 101* *

Practical Implementation

- Use two Decision Trees to perform Prefix Match
  - Produce two tuple lists
  - Cross product the two lists to reveal the hash tables for probing



Summary

- Decision tree
  - size explosion
  - difficult to do incremental updates
  - no good ways to tune for ideal configurations
- Tuple space
  - practical implementation uses tries, combined with hash tables
  - may suffer as decision trees
  - “many” hash tables to manage
  - markers and pre-computed results increase storage

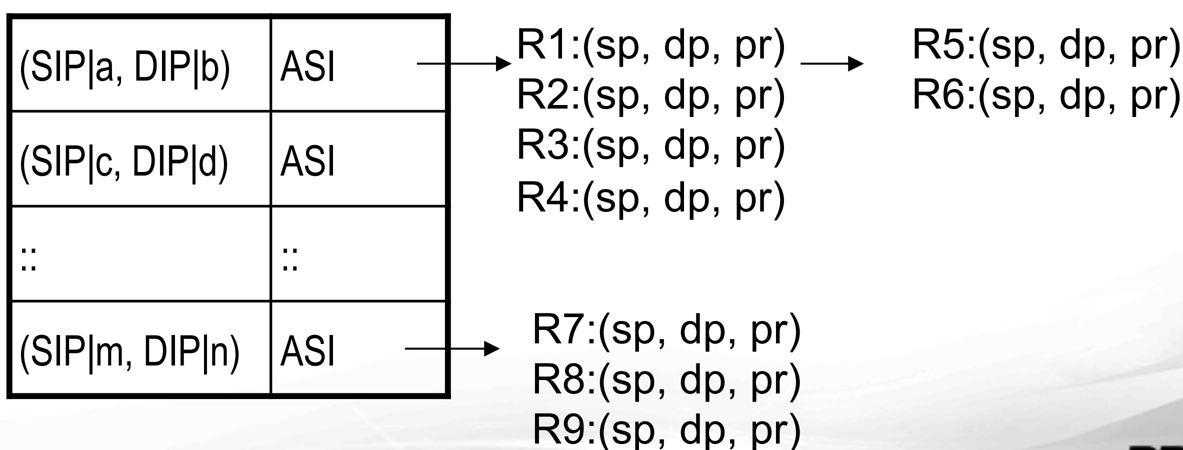
BROADCOM. Connecting everything®

HaRP (Hash Round-down Prefix)

- Simple method and data structures enable
  - parallel lookup for high performance
  - high memory efficiency and less storage
  - easy incremental updates

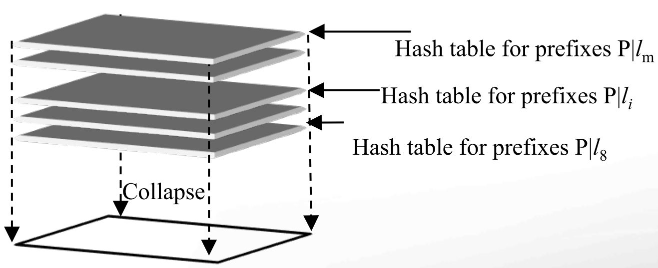
Two Stages

- Rules are broken into two parts: (SIP, DIP) + (SP, DP, Proto)
  - 1st stage percolate rules by prefix match on (SIP, DIP) via a simple hash table
  - 2nd stage inspects further on ASI (Application-Specific-Information); the rest of fields (SP, DP, Proto) via a simple linear search



Prefix Matches on (SIP, DIP)

- Choose <u>Designated Prefix Length (DPL)</u> $\{l_1, l_2, \ldots\ l_i, \ldots\ l_m\}$, for example, {32, 28, 24, 20, 16, 12, 8, 1}
- Round down prefix P|*w,* with $l_i \leq w < l_{i+1}$, to P|l_i, *e.g. 23→ 20*
- Each DPL tread *logically* defines a hash table, but …
- Achieve higher storage utilization by lumping all tables in one, and each bucket has *k* entries to mitigate hash collisions
- Storage efficiency (and less hash collisions) is further improved by <u>migrating</u> (SIP, DIP) among buckets



Total entries = B buckets * k entries per bucket

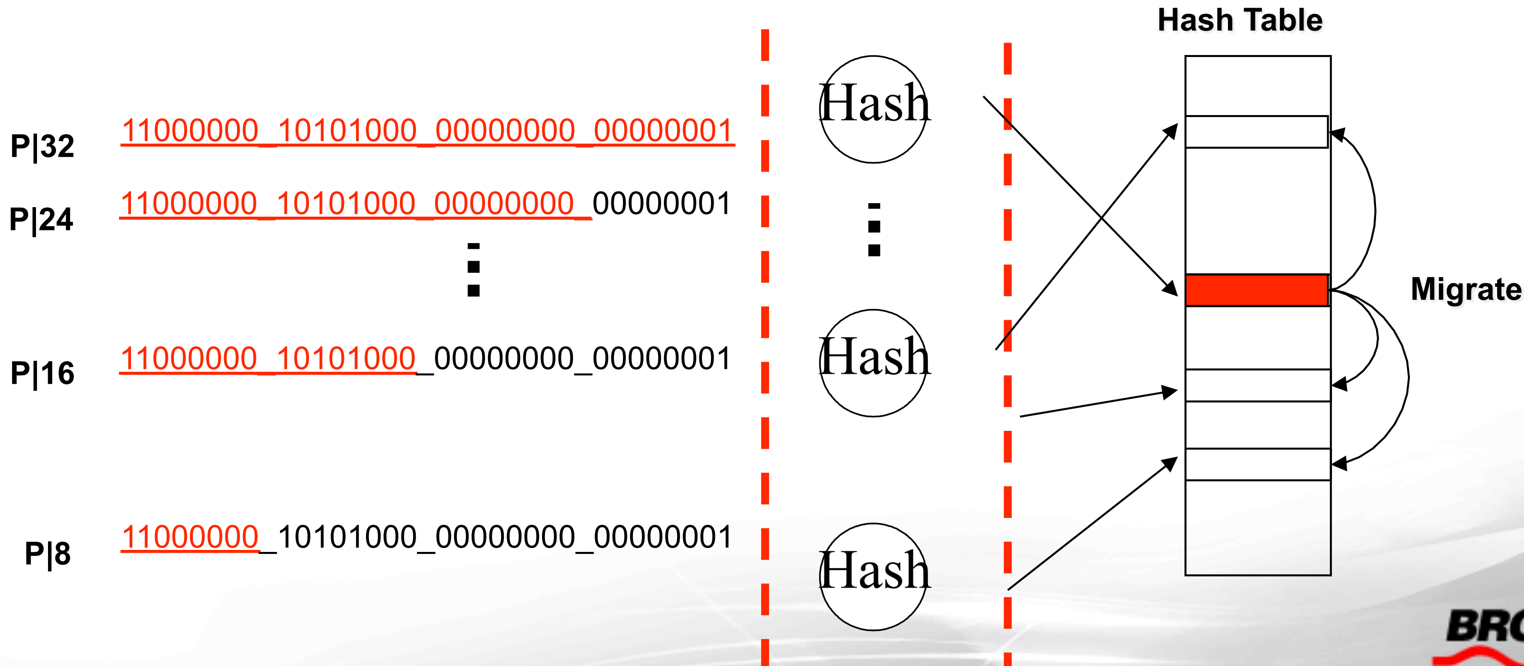
(SIP|a, DIP|b)	ASI
(SIP|c, DIP|d)	ASI
::	::
(SIP|m, DIP|n)	ASI

→ (sp, dp, pr) → (sp, dp, pr)
(sp, dp, pr) (sp, dp, pr)
(sp, dp, pr)
(sp, dp, pr)

BROADCOM.
Connecting everything

Re-balancing by Transitive Property

- Prefixes P1 >> P2 && P2 >> P3 → P1 >> P2 >> P3
- P3 can be installed in buckets identified by hash(P1), hash (P2) and hash (P3) so long we search all of them, <u>which we must do anyway</u>



Adding Rules

- Rule: (SIP|m, DIP|n, sp, dp, tcp)
  – Round DIP|m to next tread t1 in DPL
  – Round SIP|n to next tread t2 in DPL
- HaRP – basic algorithm installs (SIP, DIP) in
  – the bucket indexed by Hash(DIP|t1) or
  – the bucket indexed by Hash(SIP|t2)
  – effectively increase the bucket capacity to "2*k"
- HaRP* - enhanced algorithm installs (SIP, DIP) in (the "Host")
  – <u>any one</u> of the buckets indexed by Hash(DIP'), where DIP' >> DIP, or
  – <u>any one</u> of the buckets indexed by Hash(SIP'), where SIP' >> SIP
  – effectively increase the bucket capacity to "2*k* (i^s + i^d)"

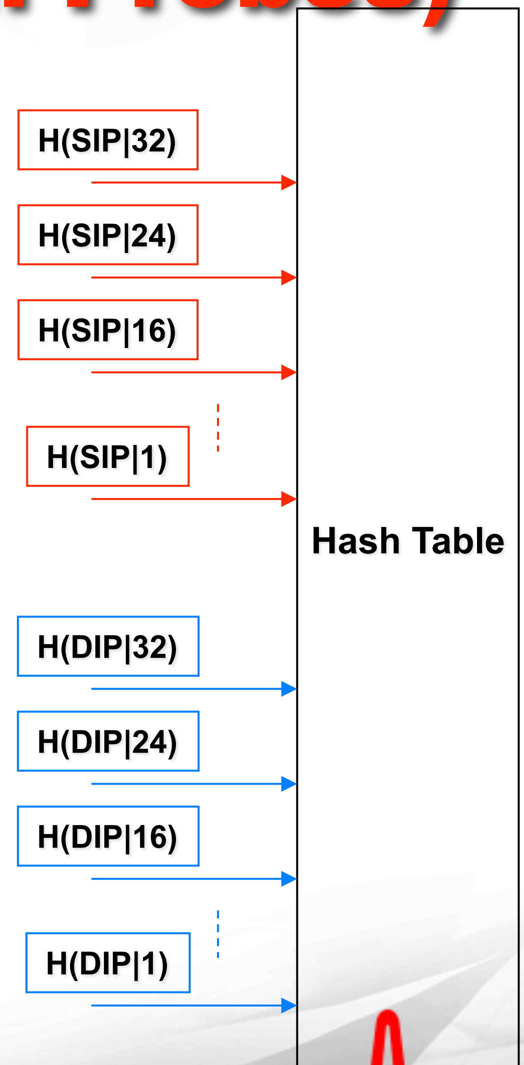
BROADCOM. Connecting everything

Lookup (Exact 2m Hash Probes)

<u>Input</u>: (SIP, DIP, SP, DP, Proto)

```
#define   mask(L)   ~((0x01 <<L) -1)
int   match_rule_id = n_rules;
Hash_Probe (key_select) ::
  key = (key_select == USE_DIP) ? dip : sip;
  for each tread t in DPL {  /* e.g. {32, 24, 20, ….} */
     h = hash_func(key&mask(t), t); /* round down prefix & hash */
     for each entry s in hash set LuHa[h] {
        if (PfxMatch((s.dip_prefix, dip),  s.dip_prefix_length) &&
             PfxMatch((s.sip_prefix, sip),   s.sip_prefix_length) {
             for each asi entry e in the chunk pointed by s.asi_pointer {
               if (e.sport_low <= sport <= e.sport_high &&
                    e.dport_low <= dport  <= e.dport_high &&
                    e.proto_low <= proto <= e.proto_high) {
                    /* Match! Choose rule with lower rule number */
                       if (match_rule_id >= e.ruleno)
                           match_rule_id = e.ruleno;
   }}}}}}

Hash_Probe(USE_DIP);
Hash_Probe(USE_SIP);
```

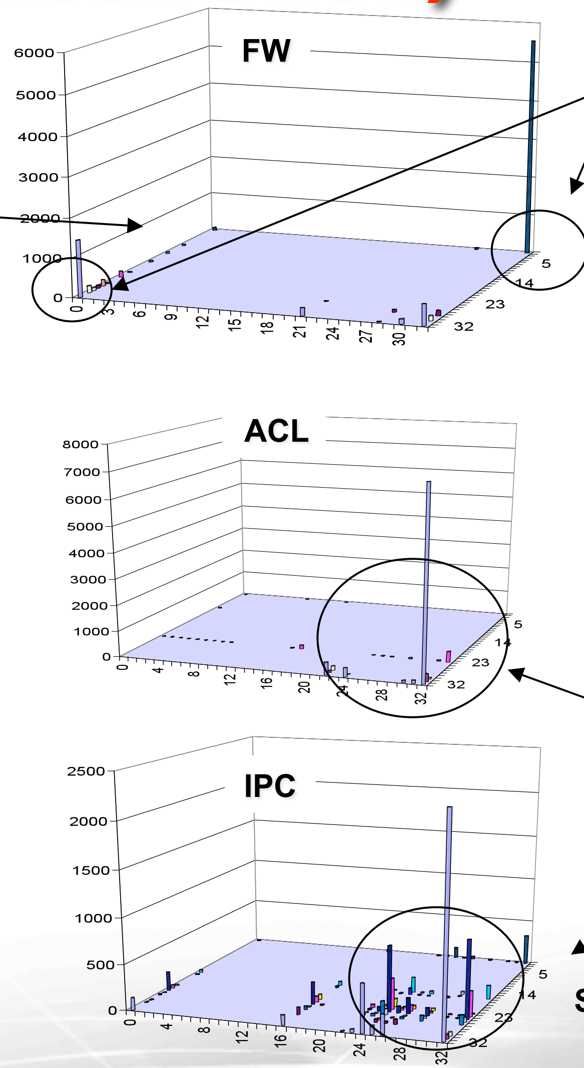


Evaluation Results

Rule Set Characteristics (ClassBench)

- **Short prefixes**
- **Weakness of HaRP*, (p1>>p2 means p2→→p1), if p2 is short, the chance for finding p1 dwindles**
- **Weakness can be easily overcome by**
  - **more DPL treads (smaller strides between treads)**
  - **multiple hashing**

Seed Filters (#filters)	Synthetic (#filters)
FW1 (269)	FW-10K (9311)
ACL1 (752)	ACL-10K (9603)
IPC1 (1550)	IPC-10K (9037)



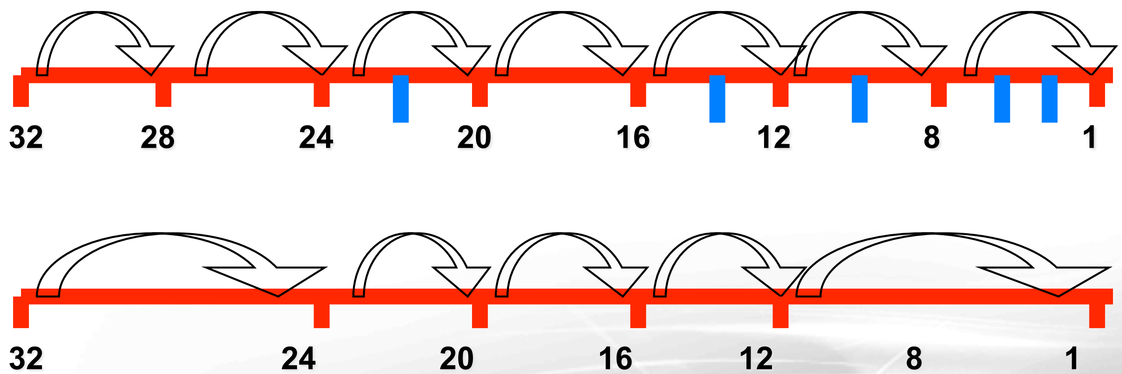
- **60% of prefix pairs have at least one wild-card address**
- **weakness of Trie-based methods**
- **Tree size explosion, difficult to be solved**
- **Majority comprises long and specific prefix pairs**

SIP Prefix Length

DIP Prefix Length

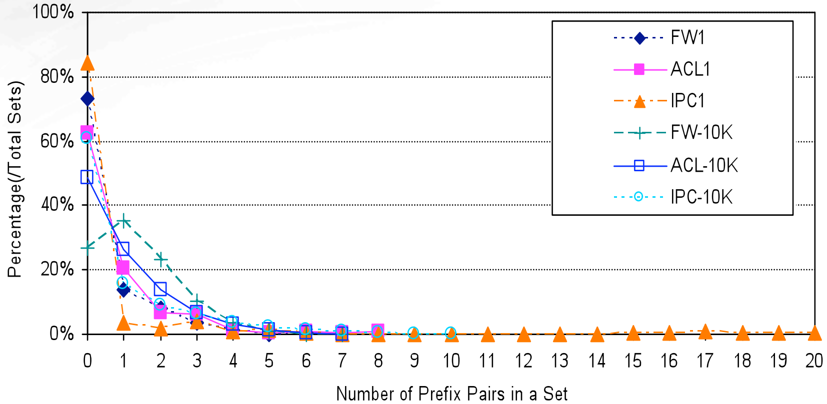
Tunable Parameters

- Dilation Factor ρ , table entry provision relative to the number of rules
  - In theory, a larger table has fewer overflows
- Number of DPL Treads, |DPL| = *m*
  - More treads gives better (SIP, DIP) load distributions at the cost of more hash probes (2**m*)
  - Fewer treads mean wider strides between treads, and more prefixes rounded down to the same tread, which lead to congestions and busy buckets (overflows)
- Different DPLs for SIP and DIP

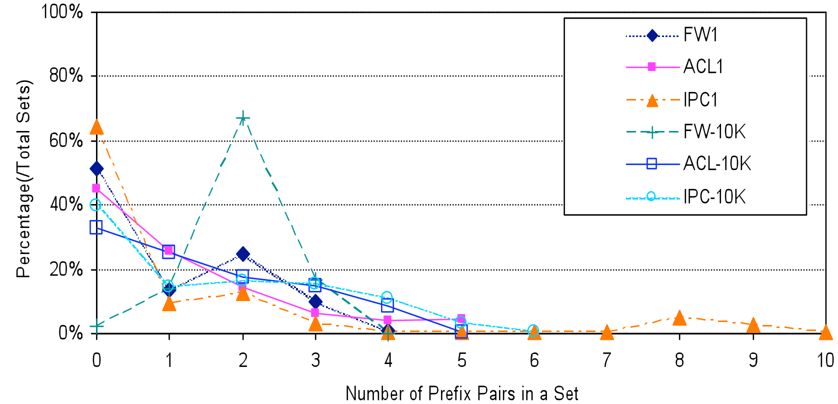


(SIP, DIP) Hash Distribution (Bucket Size k = 4)

HaRP[1], with dilation factor = 2 and DPL of 8 treads



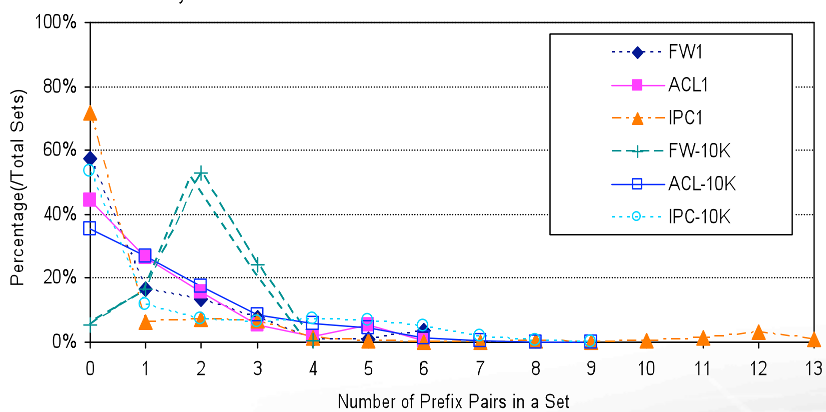
HaRP*, with dilation factor = 2 and DPL of 6 treads



Overflow occurs when more than 4 elements mapped to the same bucket. Basic HaRP with 8 treads show 4%-6% overflowing buckets.

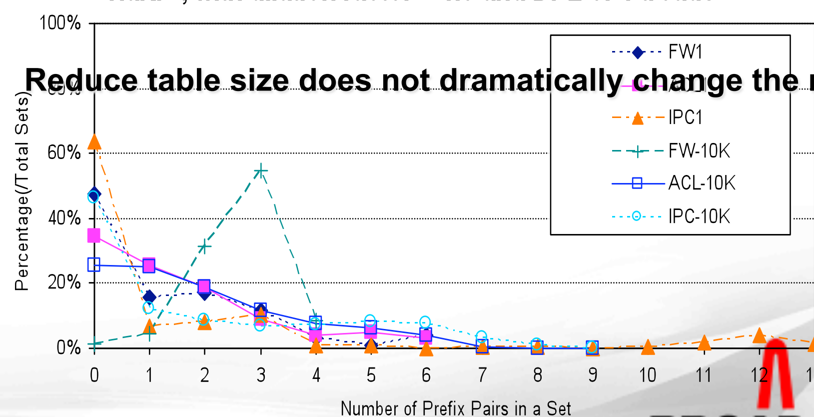
Reduce number of tread from 8 to 6 and use HaRP* to migrate elements. Reduce overflowing buckets to 2%.

HaRP*, with dilation factor = 1.5 and DPL of 6 treads



HaRP*, with dilation factor = 1.5 and DPL of 4 treads

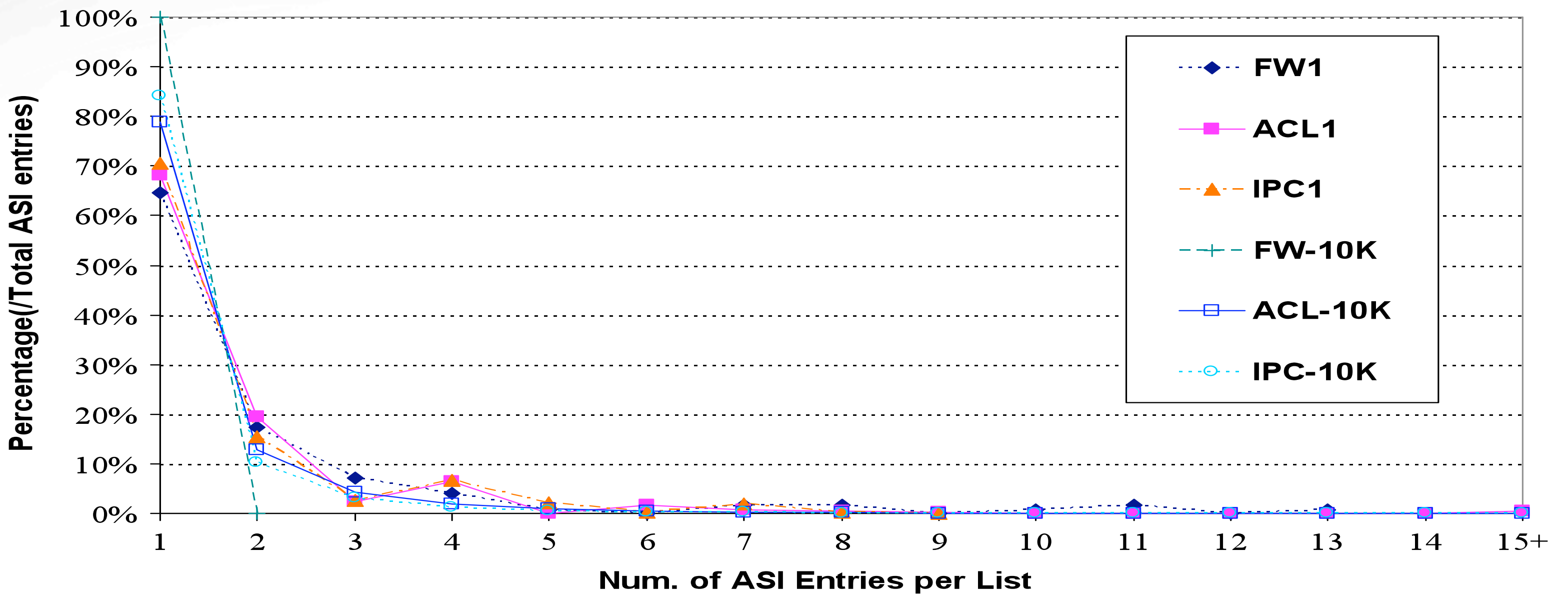
Reduce table size does not dramatically change the results.



Further reducing the number of treads causes more busy buckets, but overflows are still contained.

BROADCOM. Connecting everything®

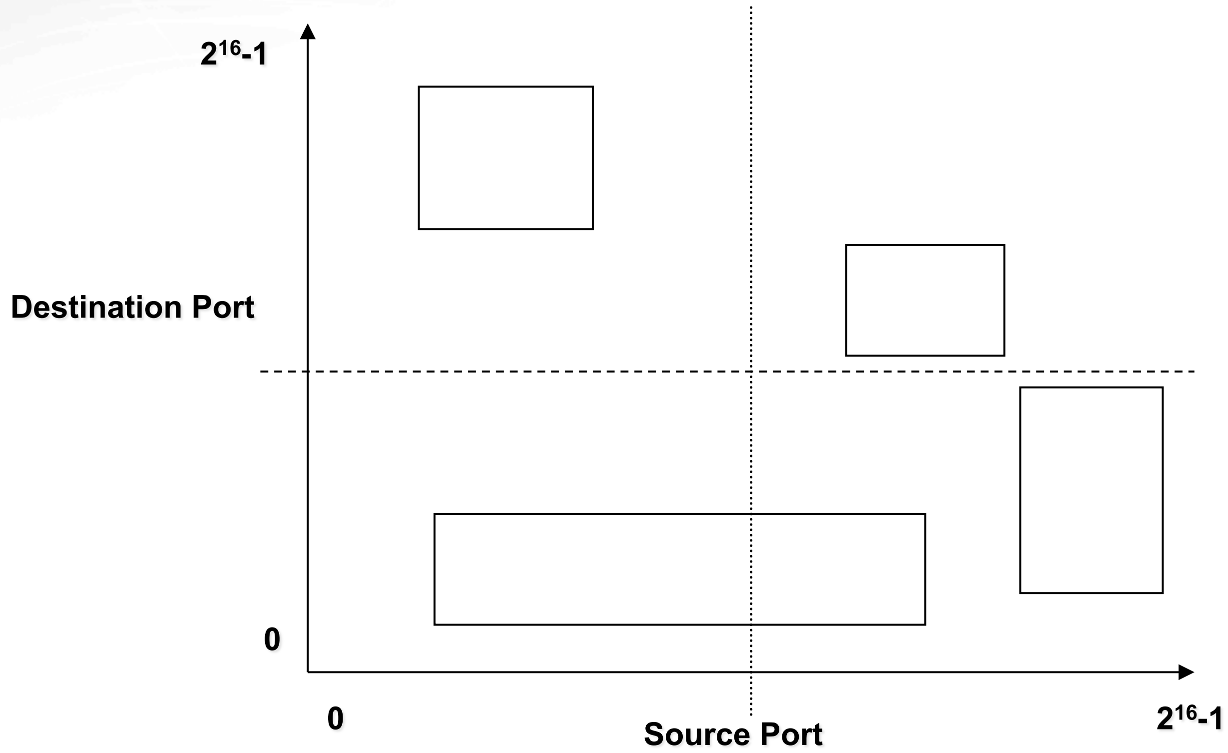
Search of the ASI Lists



- Most ASI lists are short (90% <=2, 95% <=5)
- Linear search is found to be adequate
- When long ASI lists do happen, they can be dealt with by simple methods

BROADCOM.
Connecting everything®

Deal with Long ASI Lists



- Divide a long ASI list to several short lists by selected yardsticks

Storage Requirement

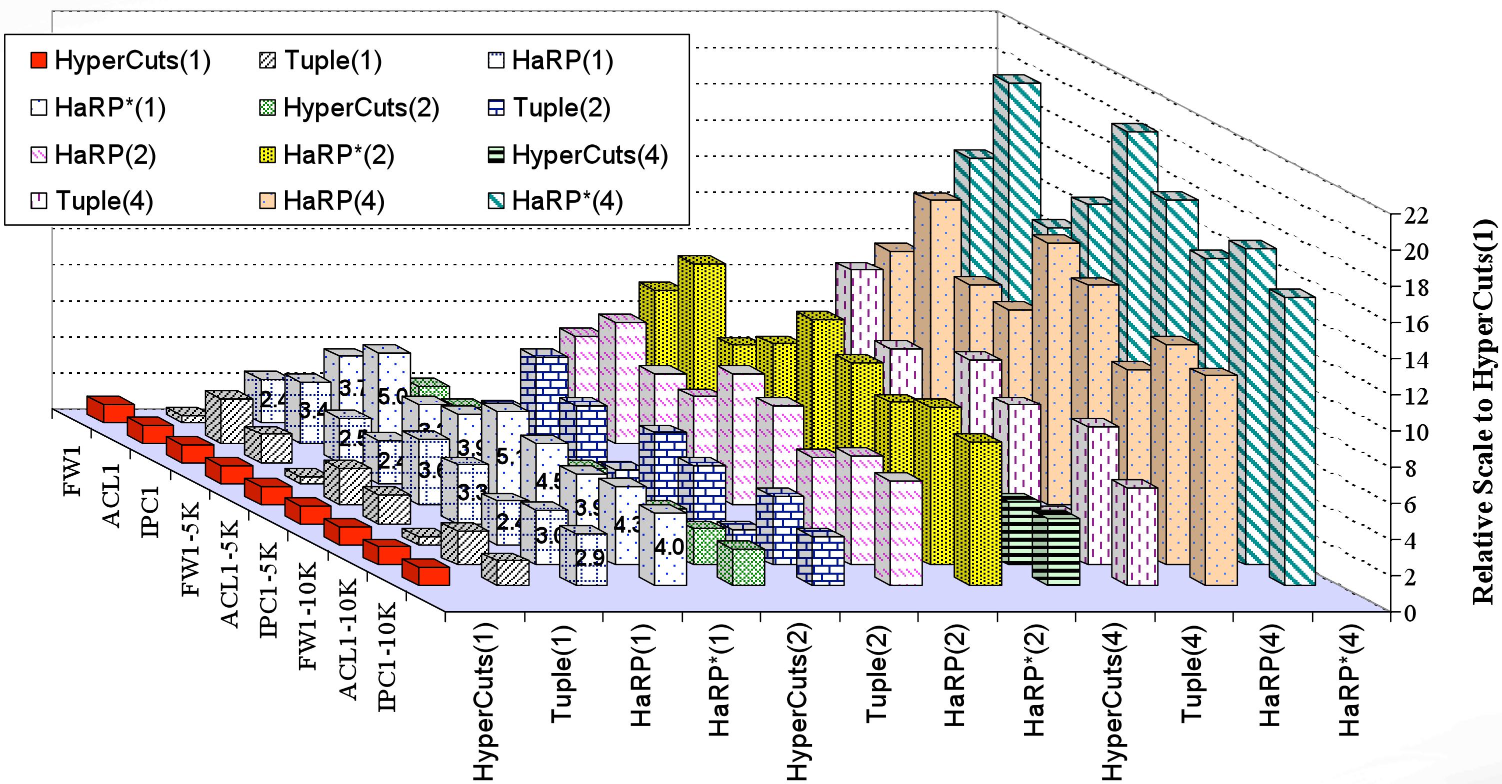
	Total Storage (in KB, or otherwise MB as specified)			Memory Efficiency		
	HaRP* (ρ=1.5)	Tuple Space	Hyper-Cuts (sf=2)	HaRP* (ρ=1.5)	Tuple Space	Hyper-Cuts (sf=2)
FW1	4.64	22.72	10.19	1.35	3.60	1.93
ACL1	13.79	44.19	20.24	1.31	2.51	1.38
IPC1	29.17	56.26	91.19	1.31	1.55	3.01
FW-5K	101.0	629.5	4.10M	1.32	5.77	46.21
ACL-5K	76.54	157.7	136.8	1.31	1.52	1.59
IPC-5K	90.56	199.4	332.6	1.31	1.91	3.82
FW-10K	217.3	1.68M	25.05M	1.31	7.88	141.0
ACL-10K	192.5	403.4	279.4	1.31	1.79	1.49
IPC-10K	187.5	449.8	649.5	1.37	2.12	3.68

Measured Lookup Performance

- Execute the program on Broadcom’s 4-way Multi-core SoC
  - 4 x 700MHz MIPS cores
  - Each core is a 4-way superscalar design
  - 32KB non-blocking L1 cache that allows 8 outstanding misses
  - 1MB shared L2 cache
- Same result trends are observed for more powerful systems
  - AMD Opteron @2.8GHz w/ 1MB Cache
  - Intel Xeon @3.16GHz w/ 6MB Cache

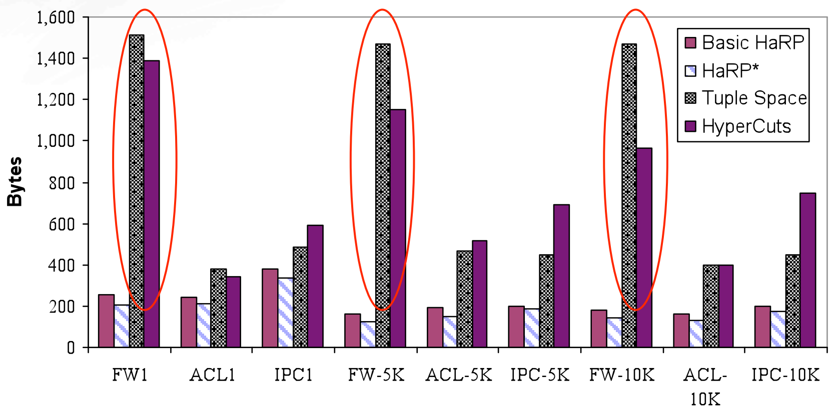
BROADCOM. Connecting everything®

Execution Performance

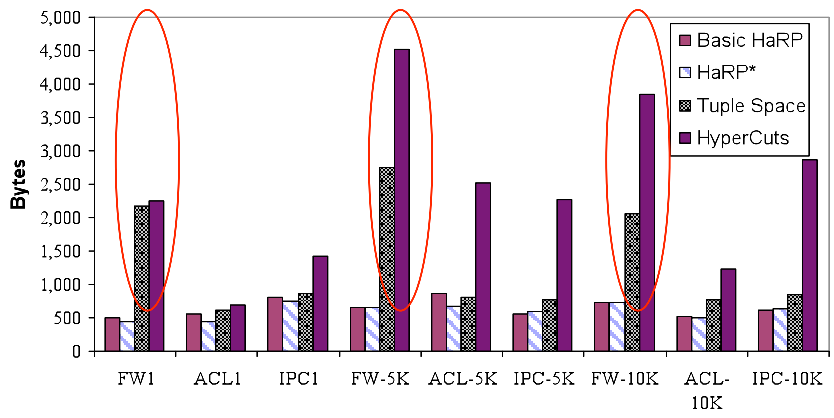


Connecting everything

Data Footprint



Average number of byte fetched per lookup

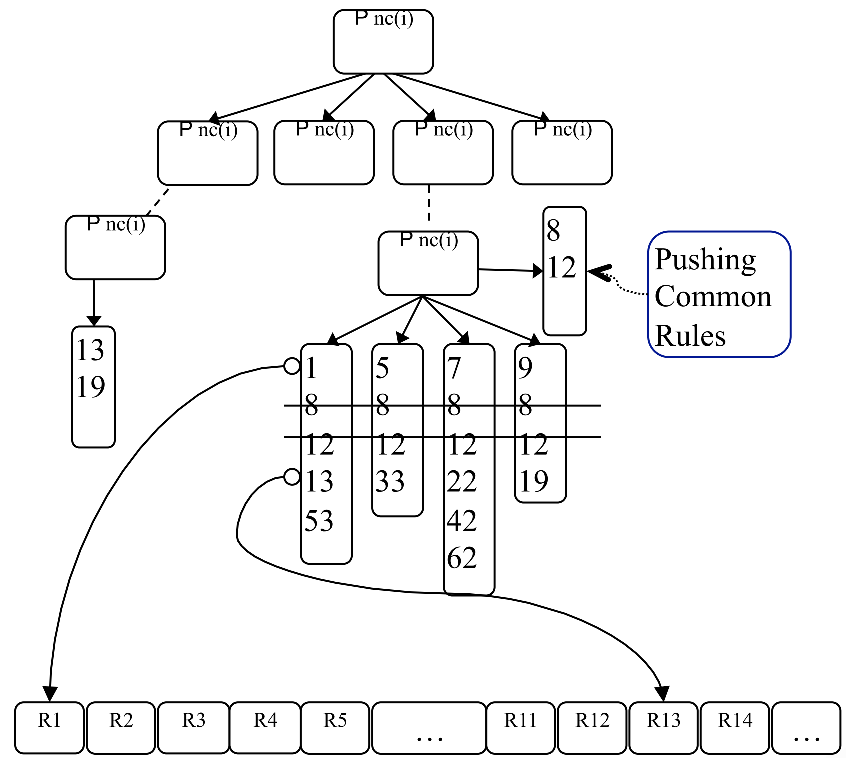


Worse case number of bytes accessed

- **Average Case: HC & Tuple >> HaRP**
- **Worst Case: HC >> Tuple >> HaRP**
- **FW data sets always show the worst results (due to wildcard addresses)**

HyperCuts

	SF	Tree Depth	Total Nodes	Total Stored Rules	Total Pushed Rules
FW-10K	2	5	820,294	6,476,700	121,177
ACL-10K	2	10	3,818	16,472	1,180
IPC-10K	2	13	21,075	73,597	5,769

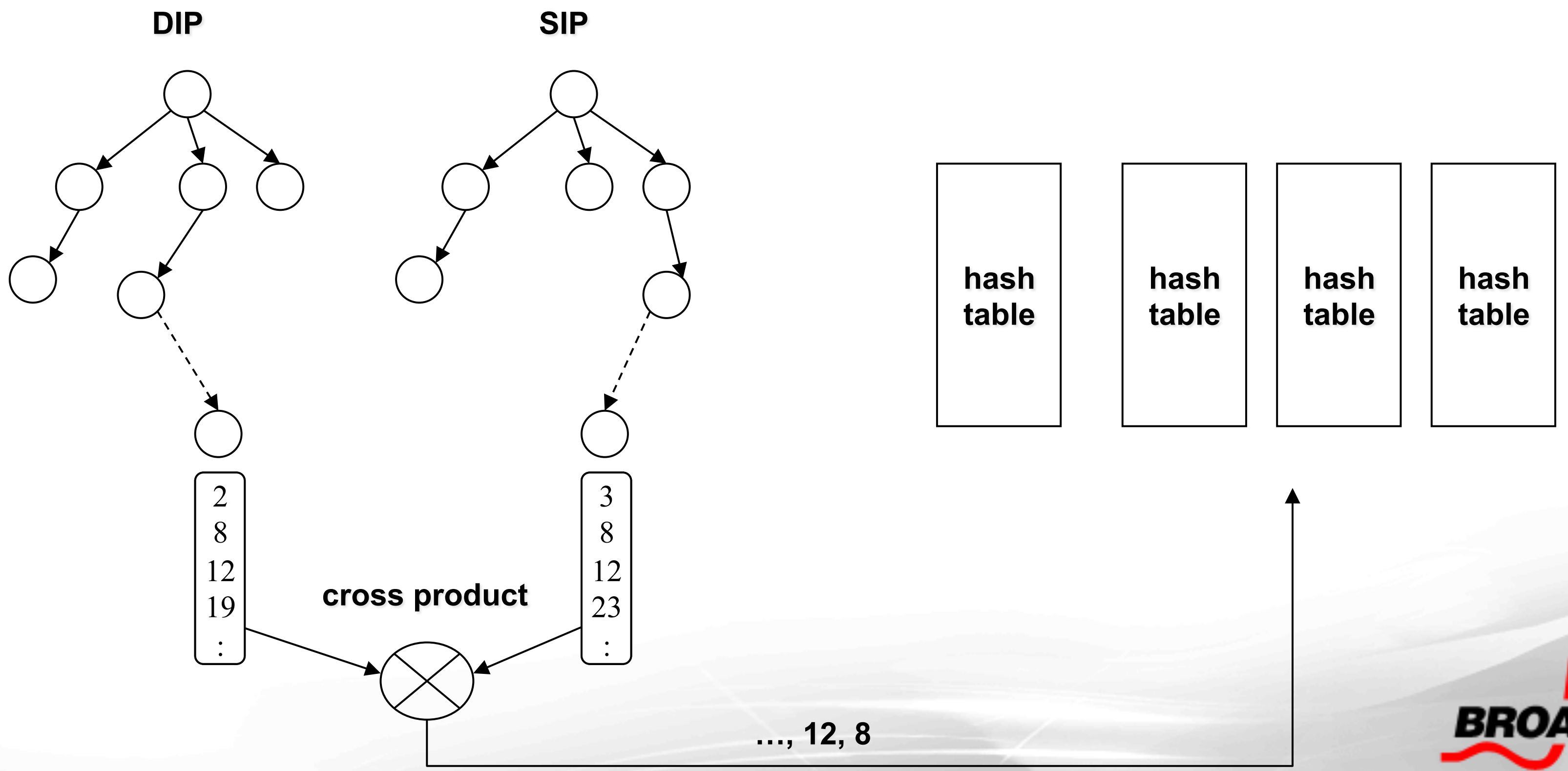


The real rules, stored in a consecutive array

Tuple Space

Average number of accessed tuples per lookup

FW1	ACL1	IPC1	FW-5K	ACL-5K	IPC-5K	FW-10K	ACL-10K	IPC-10K
72.95	6.30	11.45	68.2	10.68	9.24	67.76	6.73	8.69



BROADCOM. Connecting everything

HaRP Search Performance

	LuHa Search				ASI Search	
	ρ = 2, HaRP		ρ = 1.5, HaRP*		ρ = 2, HaRP	ρ = 1.5, HaRP*
	Mean number of prefix pair				Mean number of entries	
	Checked	Matched	Checked	Matched	Checked	Checked
FW1	14.32	1.28	10.42	1.20	2.22	2.20
ACL1	25.67	1.52	21.81	1.53	1.85	1.88
IPC1	39.47	2.03	34.50	1.98	1.73	1.73
FW-5K	16.69	1.01	11.71	1.01	1.20	1.20
ACL-5K	18.31	1.17	12.88	1.22	3.38	3.25
IPC-5K	21.13	1.39	19.03	1.58	1.66	1.74
FW-10K	19.37	1.00	14.76	1.01	1.00	1.00
ACL-10K	17.57	1.14	13.53	1.13	1.64	1.65
IPC-10K	21.64	1.36	17.94	1.53	1.64	1.69

BROADCOM. Connecting everything®

Conclusion

- We propose an innovative hash table-based design
- A two stage method is shown to be effective
- The transitive property of prefixes allow migration of elements in the hash table for more even distribution
  - simple data structures
  - simple operations
  - the smallest amount of storage among existing methods
  - easy incremental update

BROADCOM.
Connecting everything®

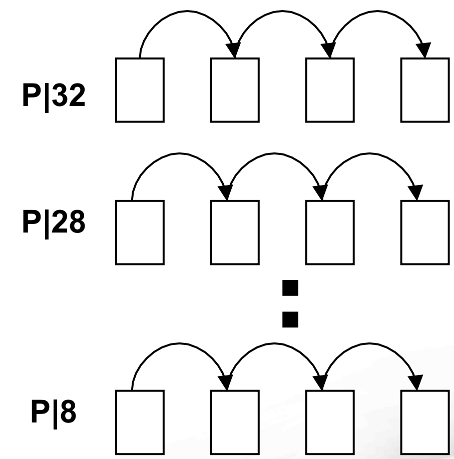
Q&A

Thank You!

Comparison Between HaRP* and d-left (Multiple) Hashing

- d-left Hashing or Multilevel Hashing
  - d hash tables, [s1, s2,… sd]
  - Use *d* hash functions to identify *d* buckets
  - Use the least loaded bucket
  - Tie breaker goes to sj with lower number j
- HaRP* ≈ d-left with subtle differences

	HaRP*	d-left
#hash functions	1	d (>=2)
#hash tables	1	m*d (d per tread)
#hash probes	2*m	2*m*d



BROADCOM.
Connecting everything