SAIL Based FIB Lookup in a Programmable Pipeline Based Linux Router

MD Iftakharul Islam, Javed I Khan

Department of Computer Science
Kent State University
Kent, OH, USA.

Outline

Longest Prefix Matching

- A router needs to perform longest prefix matching to find the outgoing port.

Table: Routing table (also known as FIB table)

Prefix	Outgoing port
10.18.0.0/22	*eth*1
131.123.252.42/32	*eth*2
169.254.0.0/16	*eth*3
169.254.192.0/18	*eth*4
192.168.122.0/24	*eth*5

- 169.254.198.1 $\implies$ *eth*4
- 169.254.190.5 $\implies$ *eth*3

Explosion of Routing Table

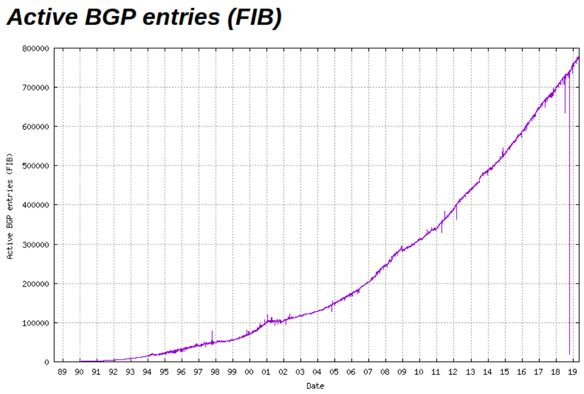


Figure: The number of routes in the Internet backbone routers

- A backbone router needs to perform around 1 billion routing table lookups per second to sustain the line rate.
- Performing FIB lookup at such a high rate in such a large routing table is particularly challenging.

FIB lookup in a Linux router

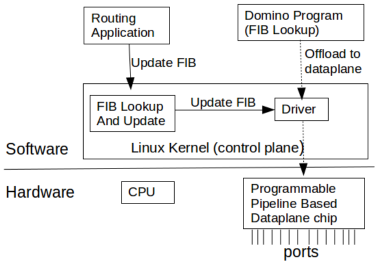


Figure: Linux Router

- Here Linux kernel works as a control plane and a programmable pipeline based VLIW processor works as a dataplane.
- We have implemented our FIB lookup in Linux kernel.
- We also have implemented the FIB lookup in Domino which would be executed on the dataplane.

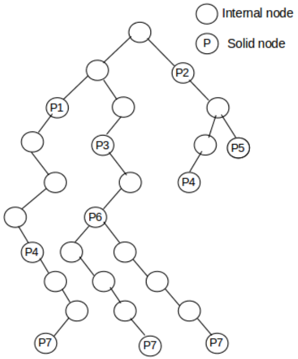
SAIL based FIB Lookup

- Recently several FIB lookup algorithms have been proposed that exhibit impressive lookup performance.
  - These include SAIL *[SIGCOMM 2014]*, Poptrie *[SIGCOMM 2015]*.
  - We chose SAIL as a basis of our implementation as it outperforms other solutions
- The main drawback of SAIL is its very high memory consumption. For instance, it consumes 29.22 MB for our example FIB table with 760K routes.
- We have used *population-counting* (a data structure) that reduces memory consumption up to 80%.
- SAIL has two variants namely SAIL_L and SAIL_U.
- We have implemented both variants with *population-counting* in both Linux kernel and Domino.
- Our implementation shows that SAIL is able to perform FIB lookup at line rate in a VLIW processor.
- We also have compared the performance of SAIL_L and SAIL_U (with *population-counting*) in Linux kernel and Domino.

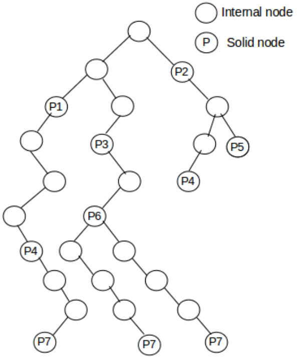
SAIL based FIB lookup

- We first show how SAIL_U constructs its data structure.
- SAIL divides a routing table into three levels: level 16, 24 and 32.
- However for simplicity, in this example, we divide the routing table into level 3, 6 and 9.
- We then show how *population-counting* is used on the data structure.

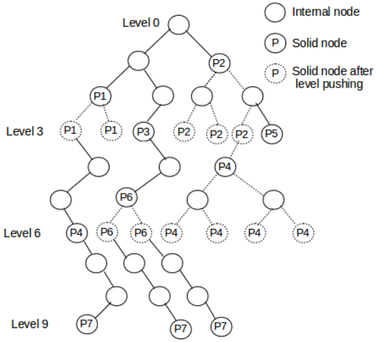
Prefix (in binary)	Next-hop
00*	P1
1*	P2
010*	P3
1100*	P4
111*	P5
000101*	P4
01010*	P6
000101110*	P7
010100111*	P7
010101111*	P7



SAIL based FIB lookup (level pushing)



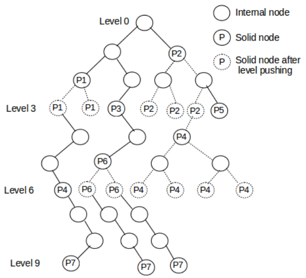
(a) Binary tree



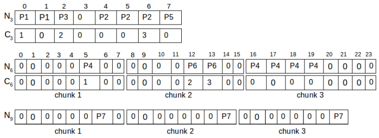
(b) Solid nodes in level 1 – 2 are pushed to level 3; solid nodes in level 4 – 5 are pushed to level 6; solid nodes in level 7 – 8 are pushed to level 9

Figure: Tree construction in SAIL

SAIL based FIB lookup (array construction)



(a) Tree



	0	1	2	3	4	5	6	7
N_3	P1	P1	P3	0	P2	P2	P2	P5
C_3	1	0	2	0	0	0	3	0

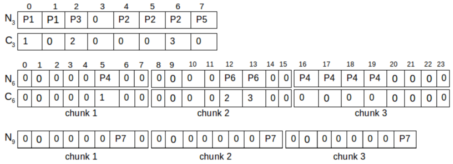
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
N_6	0	0	0	0	0	P4	0	0	0	0	0	0	P6	P6	0	0	P4	P4	P4	P4	0	0	0	0
C_6	0	0	0	0	0	1	0	0	0	0	0	0	2	3	0	0	0	0	0	0	0	0	0	0
	chunk 1								chunk 2								chunk 3							

N_9	0	0	0	0	0	0	P7	0	0	0	0	0	0	0	0	P7	0	0	0	0	0	0	0	P7
	chunk 1								chunk 2								chunk 3							

(b) *N* is the next-hop array and *C* is the chunk ID array. There will be a chunk in level 6 for each prefix in level 3 which has a longer prefix. **Most of the entries in C_6 remains 0 in practice. However it consumes around 23.16 MB in a real backbone router**

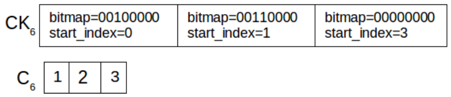
Population counting

- It's a data structure which was presented in the book *Hacker's Delight (2002)*.



N_3: P1 P1 P3 0 P2 P2 P2 P5
C_3: 1 0 2 0 0 0 3 0
N_6: 0 0 0 0 0 P4 0 0 | 0 0 0 0 P6 P6 0 0 | P4 P4 P4 P4 0 0 0 0
C_6: 0 0 0 0 0 1 0 0 | 0 0 0 0 2 3 0 0 | 0 0 0 0 0 0 0 0
chunk 1
chunk 2
chunk 3
N_9: 0 0 0 0 0 0 P7 0 | 0 0 0 0 0 0 0 P7 | 0 0 0 0 0 0 0 P7
chunk 1
chunk 2
chunk 3

(a) *N* and *C* array



CK_6
bitmap=00100000 start_index=0
bitmap=00110000 start_index=1
bitmap=00000000 start_index=3
C_6
1 2 3

(b) C_6 is encoded with bitmap and a revised C_6 where all the zero entries are eliminated. This reduces the memory consumption of SAIL by up to 80% in a real backbone router

Population counting

- As SAIL processes 8 bits in every step of the way (level 16, 24 and 32), we maintain a 256-bit bitmap.

```
struct chunk {
u64 bitmap[4]; //256-bit bitmap.
//Index to C24 where the chunk is started.
u64 start_index[4];
};
```

Figure: Chunk structure

- During FIB lookup, we need to find out how many 1-bit (*population-count*) are there before $i^{th}(0 < i < 255)$ bit.
- This will generally require calling POPCNT CPU instruction $4(\frac{256}{64})$ times because POPCNT can process only 64 bit at once.
- To avoid that, we divide the 256-bitmap into four parts. Each part maintains its own start index. The start index contains the pre-calculated population count prior to that part. This is why, we don't need to calculate the POPCNT for the whole chunk. Instead we need to calculate the POPCNT for a part of the chunk.
- We map *i* to a part by simply dividing it by 64. This is why we only require calling POPCNT only once and a DIVISION operation.

Population counting in Poptrie

- Population counting was also used in Poptrie. However they use 64-bit bitmap.
- This is why, they can apply POPCNT directly. However they will require visiting more levels (16, 22, 28, 34) than SAIL which reduces its lookup performance.
- Our implementation of SAIL uses *population-counting* while visiting just three levels (16, 24, 32).

SAIL based FIB Lookup with *population counting*

```
procedure LOOKUP($ip$)
    $nexthop \leftarrow def\_nh$
    /*Extract 16 bits from $ip$ */
    $idx \leftarrow ip >> 16$
    if $N_{16}[idx] > 0$ then
        $nexthop \leftarrow N_{16}[idx]$
    if $C_{16}[idx] \mathrel{!=} 0$ then
        /*Extract bit 17-24 from $ip$ */
        $ck\_off \leftarrow ((ip \ \& \ 65280) >> 8)$
        $ck\_id \leftarrow C_{16}[idx]$
        $idx24 \leftarrow (ck\_id - 1) * CK\_SZ + ck\_off$
    else
        return $nexthop$
    if $N_{24}[idx24] > 0$ then
        $nexthop \leftarrow N_{24}[idx24]$
    /*Check if there is a chunk in level 32 for this IP*/
    $part\_idx \leftarrow \frac{ck\_off}{64}$
    $part\_off \leftarrow ck\_off \ \% \ 64$
    $bitmap \leftarrow CK_{24}[ck\_id - 1].bitmap[part\_idx]$
    $idx \leftarrow CK_{24}[ck\_id - 1].start\_index[part\_idx]$
    if $bitmap \ \& \ (1ULL << part\_off)$ then
        /*Calculate the index to $C_{24}$*/
        $c24i \leftarrow idx + BITCOUNT(bitmap, part\_off)$
        /*Extract 8 bits from the MSB of $ip$ and calculate index to $N_{32}$*/
        $idx32 \leftarrow (C_{24}[c24i] - 1) * CK\_SZ + ip \ \& \ 255$
    else
        return $nexthop$
    if $N_{32}[idx32] > 0$ then
        $nexthop \leftarrow N_{32}[idx32]$
    return $nexthop$
/*counts the number of 1s in left-most $n$ bits of $X$*/
procedure BITCOUNT($X, n$)
    return $POPCNT(((1ULL << n) - 1) \ \& \ X)$
```

Implementation

- We have implemented SAIL_L and SAIL_U (with *population counting*) in Linux kernel 4.19 (contains around 2500 lines of C code).
- Our implementation include FIB lookup, FIB update, FIB delete and FIB flush.
- We also have implemented test code in linux kernel to evaluate the performance of our algorithms (around 400 lines of C and assembly code).
- Finally we have implemented SAIL_L and SAIL_U (with *population counting*) using Domino programming language (around 150 lines).
- We have made our implementation publicly available in Github.

SAIL in a Programmable Pipeline

- Domino programming language enables us to develop programs for programmable pipeline based VLIW processors.
- A Domino program successfully compiled by domino-compiler is guaranteed process packets at line rate (processing 1 billion packets per second on a 1 GHz VLIW processor).
- Our Domino implementation is successfully compiled by domino-compiler.
- This shows that a programmable pipeline based a VLIW processor can run SAIL with population-counting at line rate.

SAIL in a Programmable Pipeline

- A Domino compiler enables us to evaluate a Domino program without needing actual hardware
- Actual hardware doesn't exist yet (although Verilog implementation exists).
- Domino compiler generates a dependency graph that shows how the program would be executed on a pipeline (We have made the graph publicly available)

Table: Comparison between SAIL_U and SAIL_L (with *population-counting*)

	SAIL_U	SAIL_L
Number of pipeline stages	15	32
Maximum # of atoms (ALU) per stage	5	6
Processing latency (for each packet)	15 ns	32 ns

Dataset

- We have evaluated our Linux kernel implementation with FIBs from real backbone router (obtained from RouteView project)
- RouteView project provide us with RIB in MRT format. We then convert the MRT RIB to FIB using BGPDump and our custom Python script (*both data and the scripts are publicly available*).
- We conducted our experiment in a Laptop. We have created 32 virtual ethernet to emulate a router.

Name	AS Number	# of prefixes	# of next-hops	Prefix length
fib1	293	759069	2	0 – 24
fib2	852	733378	138	0 – 24
fib3	19016	552285	236	0 – 32
fib4	19151	737125	2	0 – 32
fib5	23367	131336	178	0 – 24
fib6	32709	760195	140	0 – 32
fib7	53828	733192	223	0 – 24

Impact of Population Counting

Table: Impact of population counting on memory consumption (for *fib6*)

	Without Population Counting		With Population Counting	
Array	Length	Size	Length	Size
N_{16}	65536	64 KB	65536	64 KB
C_{16}	65536	128 KB	65536	128 KB
N_{24}	6071808	5.79 MB	6071808	5.79 MB
CK_{24}			366	22.87 KB
C_{24}	6071808	23.16 MB	366	1.42 KB
N_{32}	93696	91.50 KB	93696	91.50 KB
Total		29.22 MB		6.09 MB

- The memory consumption primarily differs for C_{24}.
- 98.5% routes in backbone routers are $0 - 24$ bit long. This is why most of the entries in C_{24} remains 0. Population counting eliminates those entries results significant reduction in memory consumption.

Impact of Population Counting

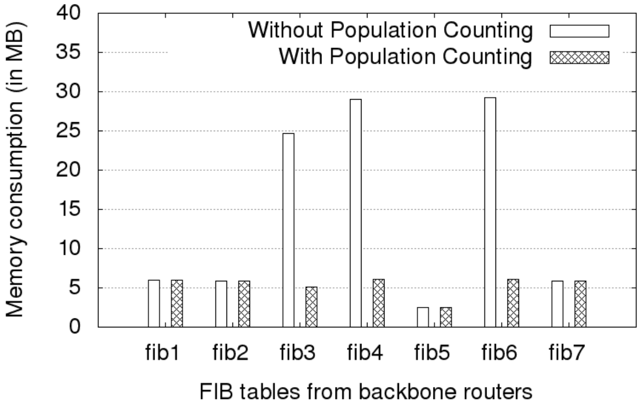
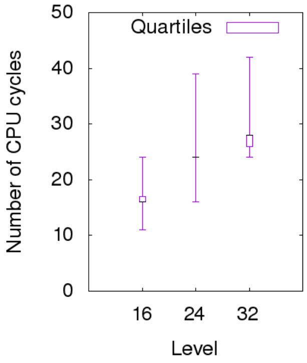
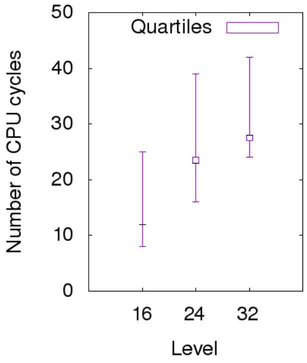


Figure: Memory consumption for different FIBs

Lookup Cost



(a) SAIL_U



(b) SAIL_L

Figure: Lookup cost for different levels .

Lookup Cost (Lesson Learned)

- The result shows that a general purpose CPU fail to exhibit deterministic performance.
- It also shows that both SAIL_U and SAIL_L (with *population-counting*) exhibit comparable lookup performance.
- The result also shows that lookup cost increases for higher level. For instance, the lookup cost is maximum when the longest prefix is found in level 32. Again the lookup cost is minimum when it is found in level 16.

Lookup Cost (Lesson Learned)

- It is noteworthy that we disable *hyper-threading* and *frequency scaling* while conducting the experiemnt. This avoids unnecessary cache thrashing.
- Here only considered the data where SAIL is stored in CPU cache (so that DRAM latency doesn't affect the actual performance of the algorithm)
- It is noteworthy that FIB lookup in Linux kernel will not act as a dataplane in a Linux router (it will work as a slow path).

Update cost

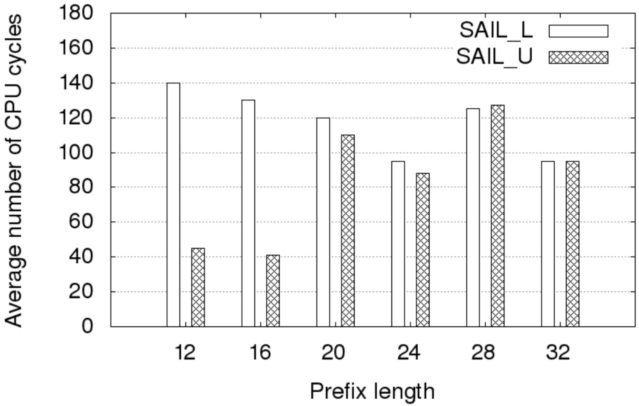


Figure: Update cost for different prefix lengths

Update Cost (Lesson Learned)

- The result shows SAIL_U performs slightly better than SAIL_U for FIB update (when *population-counting* is used).
- It also shows that our implementation can perform fast incremental update which is needed for the control plane of a Linux router.

Thank You