DUO–Dual TCAM Architecture for Routing Tables with Incremental Update [*]

Tania Mishra and Sartaj Sahni
Department of Computer and Information Science and Engineering,
University of Florida, Gainesville, FL 32611
{tmishra, sahni}@cise.ufl.edu

Abstract

We propose a dual TCAM architecture-DUO- for routing tables. Four memory management schemes for TCAMs also are proposed and evaluated. DUO and our memory management schemes support control-plane incremental updates without delaying data-plane lookups. Compared to other TCAM architectures such as CAO_OPT [23] that support incremental updates without delaying lookups, DUO offers reduction in power consumption and/or improvement in worst-case performance for update operations.

Keywords
IP routing table, consistent lookup, incremental updates, power.

1 Introduction

The primary function of an Internet router is to forward packets using a table of rules. A packet forwarding rule (P, H) comprises a prefix P and a next hop H. A packet with destination address d is forwarded to H where H is the next hop associated with the rule that has the longest prefix that matches d. We refer to the set of rules as the rule table or forwarding table. Packet forwarding is performed in the data plane while route updates are done in the control plane. Whereas the data plane receives tens or even hundreds of millions of packets per second, the control plane receives only thousands of update requests per second. Figure 1 illustrates the high level functions of control and data planes.

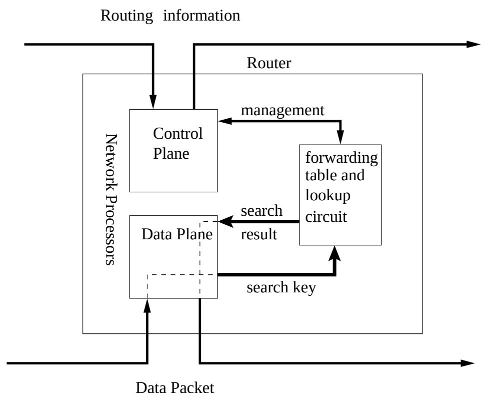


Figure 1. Control and Data Planes in Routers

With the rapid global spread of the Internet, the forwarding table size at each router is growing fast as is the number of route updates that are received by a router due to extensive interconnections. Presently, the largest forwarding tables have about one million rules and the number of updates peaks at about 10,000 updates per second. At a line rate of 10Gbps and a minimum packet size of 40 bytes, the number of data plane lookups per second exceeds 30 million.

[*]This research was supported, in part, by the National Science Foundation, under grant 0829916.

A number of fast lookup schemes have been proposed in literature that use TCAMs as the main hardware component, because TCAMs are simple to use and provide high-speed table lookup [12, 13] ([12, 13] also survey non-TCAM approaches to routing table management). A TCAM is a special type of content addressable memory (CAM) that allows each memory bit to store one of the three values: 0, 1, x (don't care). The prefix of a rule is stored in a word of TCAM and the next hop is stored in the corresponding word of an associated SRAM. The entries of a TCAM may be searched in parallel for a prefix that matches a given destination address. If multiple matching entries are found then the best match is selected by a priority encoder. The best match is quite frequently identified as the first entry that matches. Using the index of the best matched TCAM entry, we access the corresponding SRAM word to determine the next hop. When the prefixes are stored in decreasing order of prefix length, it is possible to determine the TCAM index of the longest matching prefix for any destination address in one TCAM cycle. We note that, a TCAM word will be 32 bits for IPv4 applications. The described TCAM scheme is referred to as the *simple TCAM scheme* [4].

The main drawback of using TCAMs in a router's forwarding engine is that a TCAM consumes a high amount of power for each lookup operation since every TCAM cell in the array is activated for each lookup. There has been a significant amount of research in trying to reduce the power consumption in TCAMs [4, 18, 19, 23, 21, 22]. Lu and Sahni in [4] propose a technique that utilizes wide SRAMs to store portions of prefixes along with their next hops in each SRAM word. This scheme reduces the TCAM size and power requirement drastically. The Simple TCAM with Wide SRAM (STW) organization is the basic scheme in [4] that demonstrates the potential of saving TCAM space and power by utilizing wide SRAM words. One drawback of the STW scheme is that incremental update algorithms are complex because of the need to handle covering prefixes that may be replicated many times. On the other hand, batch update algorithms require twice the memory footprint so forwarding and updating can be applied on two separate copies of the forwarding table [22].

Wang et al. [18] propose a consistent table update scheme that eliminates the need to lock the forwarding table during an update, preserving the correctness of rule matching at all times. Since lookups can proceed at their usual speed even as updates are being carried out, there is no need to minimize the number of rule moves required to incorporate an update as long as the rate of processing keeps up with the arrival rate for updates. However, this does not undermine the advantage of a fast update process requiring a smaller number of rule moves since with a faster process fewer packets will be forwarded to non-optimal next hops.

Wang and Tzeng [19] use leaf pushing to transform the prefixes in the routing table into a set of independent prefixes, which are then stored in a TCAM (in any order). Their consistent update scheme, however, delays data plane lookups that match TCAM slots whose next hop information is being updated. Although, on average, each insert or delete request results in a very small number of insert/delete operations on the set of independent prefixes stored in the TCAM, worst-case inserts and deletes require $\Omega(n)$ insert/delete operations on the set of independent prefixes, where n is the number of independent prefixes. Hence, an adversary can significantly compromise the router by maliciously injecting a sequence of worst-case updates. Further, the method of [19] uses a TCAM search to find a free TCAM slot for an insert and this search interrupts the lookups taking place in the data plane.

In this paper we present three versions of our novel dual TCAM architecture, generally referred to as DUO, along with advanced memory management schemes for performing efficient and consistent incremental updates without degrading lookup speed. The first version of the architecture is DUOS – dual TCAM with simple SRAM, where both the TCAMs have a simple associated SRAM that is used for storing next hops. The second version of the architecture is DUOW – dual TCAM with wide SRAM, where one or both the TCAMs have wide associated SRAMs that are used to store suffixes as well as next hops. The third version is IDUOW – indexed dual TCAM with wide SRAM, in which either or both TCAMs have an associated index TCAM. The advantages of the dual TCAM architecture and the memory management schemes presented in this paper are:

1. Like the TCAM schemes CAO_OPT [23] and MIPS [19], DUO supports incremental updates. This means that we may do updates on the forwarding table one by one efficiently in DUO. In particular, DUO allows us to do an update with no slow down in data plane lookups. In contrast, TCAM schemes such as those of [4, 21] support batch updates only. These latter schemes employ two TCAMs. At any time, one of these is active and the other inactive. The active TCAM is used for data plane lookups. Updates are accumulated, in the control plane, as they arrive over a pre-specified interval. At the end of each accumulation period, the control plane constructs a new forwarding

table in the inactive TCAM. Note that during this construction process that takes place in the control plane, data plane lookups are unaffected as these use the active TCAM. Following the construction of the forwarding table in the inactive TCAM, the roles of active and inactive TCAMs are switched incurring minimal data plane lookup delay. As can be seen, batch schemes require twice the TCAM memory required by incremental schemes and have a significantly larger latency between the arrival of an update request and the time this update is incorporated into the active forwarding table.

2. Incremental updates in DUOS require far fewer rule moves than required by the simple TCAM scheme. The total TCAM and SRAM space used by DUOS is the same as that used by the simple TCAM scheme.

3. The wide SRAM scheme of [4], which is a batch scheme, may be coupled with DUOS to arrive at DUOW and IDUOW, which provide considerable reduction in TCAM memory and power while preserving the efficient incremental update capability of DUOS.

4. Employing memory management scheme DLFS_PLO (Scheme 3) to manage the memory of a simple TCAM enables the simple TCAM to outperform the CAO_OPT scheme (Scheme 4) of [23] with respect to the time required to complete update sequences that arise in practice. Compared to the PLO_OPT memory management scheme (Scheme 1) proposed in [23], however, CAO_OPT is superior (as expected by the analysis of [23]).

The rest of the paper is organized as follows. Section 2 presents related research work. The DUOS architecture and our memory management schemes are described in Section 3 , DUOW is described in Section 4, and IDUOW is described in Section 5. An experimental evaluation of DUO is presented in Section 6 and we conclude in Section 7.

2 Background and Related Work

The high-speed table lookup property of TCAMs is a key feature for implementation of fast engines to be used in packet forwarding. Research on TCAM routers has focused on lowering the power consumption [11, 8, 4, 9, 10, 2, 22, 21, 15, 14, 16, 17], creating new router architectures involving multiple TCAMs that achieve even faster lookup [27, 26], and developing efficient strategies for incremental updates [23, 18, 19]. Since our focus in this paper is to develop router architectures that have efficient support for incremental updates, we present work related to TCAM incremental updates in some detail.

Shah and Gupta [23] describe incremental update algorithms for TCAMs using two different strategies to place prefixes in the TCAM. In PLO_OPT, the prefixes are placed in the TCAM in decreasing order of length. Unused TCAM slots/words are in the middle of the TCAM. So, prefixes of length $W, \cdots, W/2+1$ are above the free slots and the remaining prefixes are below the free slots, where $W = 32$ for IPv4. An insert or delete requires at most $W/2$ prefix moves in PLO_OPT. In CAO_OPT, the prefixes are placed in the TCAM so that if two prefixes are nested, the longer prefix precedes the shorter one. If we start with the binary trie representation of the prefixes of the routing table, the prefixes along any path from the trie root to a trie leaf are nested. So, every root to leaf path in the trie defines a chain of nested prefixes. In CAO_OPT, the prefixes on every chain appear in reverse order in the TCAM. This placement ensures that the first prefix in the TCAM that matches a destination address is the longest matching prefix. The TCAM free slots are in the middle of the TCAM. If the maximum number of prefixes in a nested chain is q, then at most $\lceil q/2 \rceil$ prefixes of a chain are above the free slots. An insert or delete in CAO_OPT requires at most $\lceil q/2 \rceil = W/2$ moves. Since q is about 6 in practical routing tables, CAO_OPT gives a performance improvement over PLO_OPT in practice (though the worst-case performance of both is the same).

Wang et al. [18] define a *consistent rule table* to be a rule table in which the rule matched (including the action associated with the rule) by a look up operation performed in the data plane is either the rule (including action) that would be matched just before or just after any ongoing update operation in the control plane. Wang et al. [18] develop a scheme for consistent table update without locking the TCAM at any time, essentially allowing a search to proceed while the table is being updated. Consistency is ensured by avoiding overwriting of a TCAM entry. Their CoPTUA algorithm can be applied to the PLO_OPT and CAO_OPT schemes of [23] so that rule updates can be carried out without locking the table for data plane lookups under suitable assumptions for TCAM operation [18].

Wang and Tzeng [19] also propose a consistent TCAM scheme. Their scheme, MIPS, however delays data plane lookups that match TCAM slots whose next hop information is being updated. In MIPS, the TCAM stores a set of independent prefixes (i.e., disjoint). This set of independent prefixes is obtained from the original set of prefixes by using the leaf pushing technique [25] followed by a compression step. Since the prefixes in the TCAM are independent, at most one prefix matches any given destination address. Hence, the independent prefixes may be placed in the TCAM in any order and we may dispense with the priority encoder logic of the TCAM, which results in a reduction in TCAM lookup latency by about 50% [24]. Further, a new prefix may be inserted into any free slot of the TCAM and an old prefix deleted by simply setting the associated slot's *valid* bit to 0. While the use of an independent prefix set simplifies table management, leaf pushing replicates a prefix many times. In the worst case, an insert or delete, requires changes to $\Omega(n)$ TCAM entries, where n the number of independent prefixes in the TCAM (Figure 2). Furthermore, the number of independent prefixes that result from leaf pushing and compression can be quite large as, in the worst-case, the compression step may fail to do any reduction in the prefix set following leaf pushing. Experimental results presented in [19] suggest, however, that, on practical rule sets, leaf expansion and compression actually reduce the number of prefixes by 20% to 68% because of the prevalence of a large number of redundant prefixes in practical rule sets. Further, each update operation results in between one and two accesses to the TCAM on average. Wang and Zheng [19] do not use any memory management scheme to keep track of the free slots in the TCAM and instead rely on a TCAM search operation to find an empty slot when such a slot is needed. Since a TCAM cannot perform a data plane search concurrent with a control plane search, update operations delay data plane lookups. In practice, since the number of updates per second is quite small and since each routing table update results in only one or two TCAM update operations (on average) the delay caused by control plane lookups on data plane lookups is quite small. As TCAM lookups consume a significant amount of energy relative to that consumed by TCAM read/write operations, using lookups to locate free TCAM slots increases total energy consumption for updates significantly.

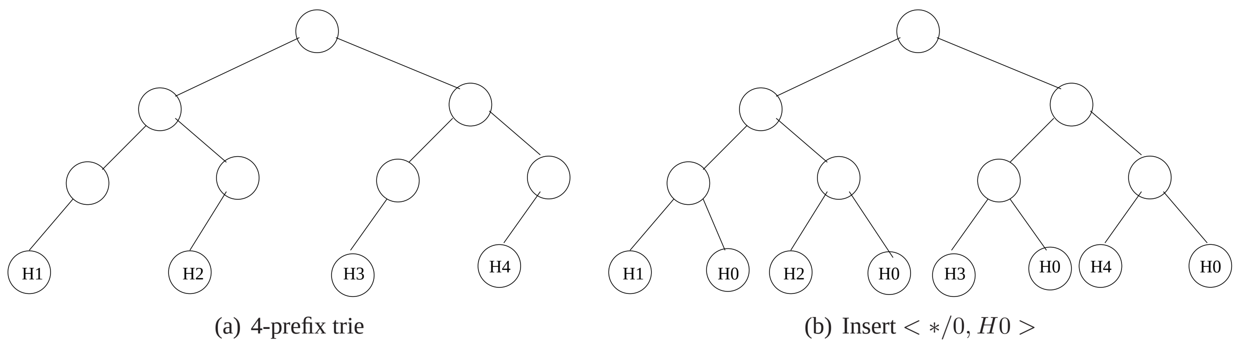


Figure 2. Insertion of the root prefix into (a) requires the insertion of 4 new independent prefixes into the TCAM. Similarly, the deletion of the root prefix from (b) requires the withdrawal of these 4 prefixes from the TCAM

Zane et al. [21] propose an indexed TCAM scheme to reduce the total TCAM power used to search routing tables of a given size. The indexed TCAM schemes of [21], however, increase the total TCAM size needed relative to non-indexed TCAMs. Lu and Sahni [4] couple indexed TCAMs with wide SRAMs to reduce both power and TCAM memory by a significant amount. Although the strategies of [4] are power and memory efficient, they are not well suited to incremental update. Similarly the prefix compaction methods of [11, 8, 22], while resulting in power and memory reduction, do not lend themselves well to incremental update. Chang [2] proposes a TCAM partitioning and indexing scheme in which the TCAM index is stored in a pivot prefix SRAM and an index SRAM. In Chang's scheme [2], the TCAM index is searched using a binary search that makes $O(\log K)$ SRAM accesses to determine the TCAM bucket that is to be searched. On the other hand, the scheme of Zane et al. [21] stores its index in a TCAM enabling the determination of the bucket for further search by a query on the index TCAM. As a result, a lookup takes 2 TCAM searches when the scheme of [21] is used and take 1 TCAM search plus $O(\log K)$ SRAM accesses when the scheme of [2] is used.

3 Simple Dual TCAM – DUOS

DUOS uses any reasonably efficient data structure to store the routing-table rules in the control plane. For example, a simple data structure such as a binary trie or 1-bit trie stored in a 100ns DRAM, permits about 300K IPv4 lookups, inserts, and deletes per second. This performance is quite adequate for the anticipated tens of thousands of control plane operations. For concreteness, we assume that a binary trie is used, in the control plane, to store the routing-table rules. Additionally, DUOS uses two TCAMs each with an associated SRAM. The TCAMs are labeled ITCAM (Interior TCAM) and LTCAM (Leaf TCAM) in Figure 3. The associated SRAMs are similarly labeled. Prefixes stored in leaf (non leaf or interior) nodes of the control plane trie are stored also in the LTCAM (ITCAM) and their associated next hops are stored in the LSRAM (ISRAM). Since the LTCAM stores only leaf prefixes, the prefixes in the LTCAM are disjoint and at most one may match a given destination address. Consequently, the LTCAM prefixes, even though of varying length, may be stored in any order. Further, the LTCAM does not require a priority encoder and, as a result, the latency of an LTCAM search is up to 50% less than that of a search in a TCAM with a priority encoder [28]. A data plane lookup is performed by doing a search for the packet's destination address in both ITCAM and LTCAM. The ITCAM search yields the next hop associated with the longest matching non-leaf prefix while the LTCAM search yields the next hop associated with at most one leaf prefix that matches the destination address. Additional logic shown in Figure 3 returns the next hop (if any) from the LTCAM search; the next hop from the ITCAM search is returned only if the LTCAM search found no match. Note that since the LTCAM has no priority encoder, its search completes sooner than that in the ITCAM. The combining logic of Figure 3 can take advantage of this and abort the ITCAM search whenever the LTCAM search is successful, thereby reducing average lookup time. The correctness of the lookup is readily established. Figure 4 shows a 4-prefix

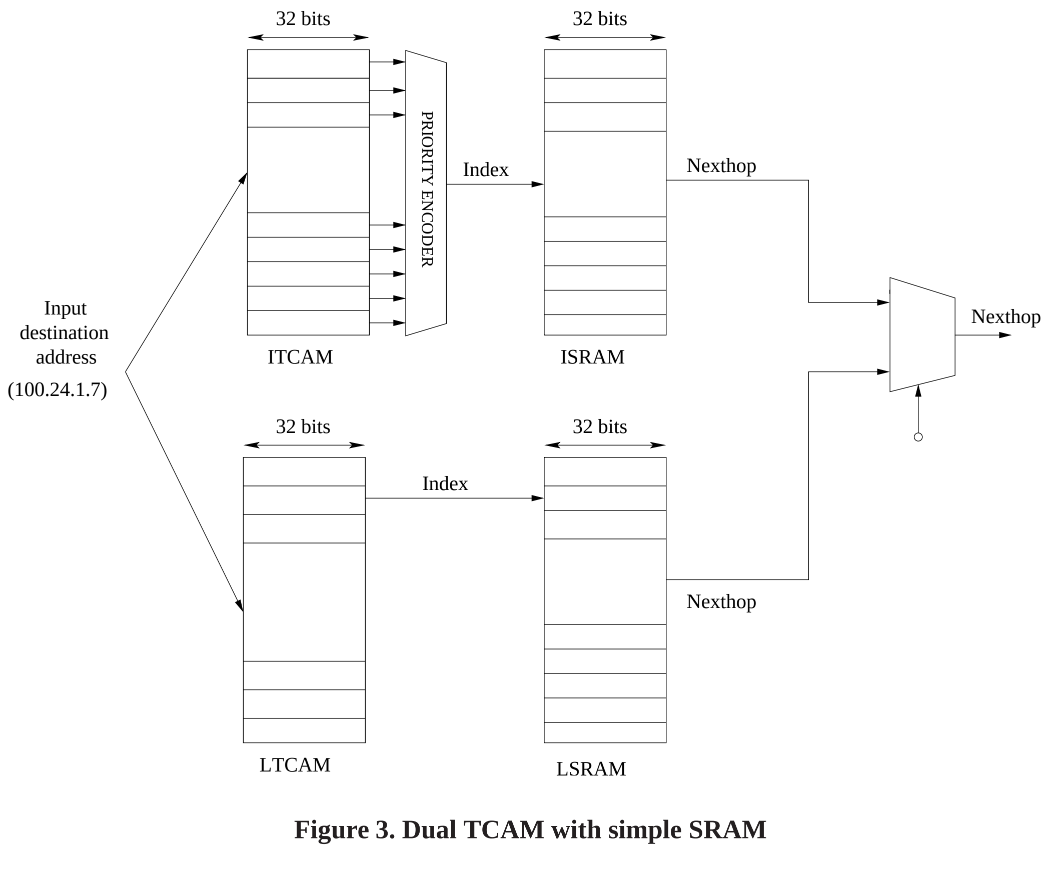


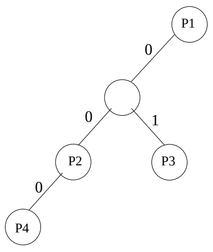
Figure 3. Dual TCAM with simple SRAM

forwarding table together with its corresponding binary trie that is stored in the control plane as well as the content of the two TCAMs and the two SRAMs of DUOS.

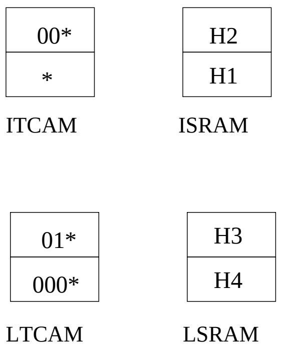
Each node of the control plane trie has fields such as *prefix*, *slot*, *nexthop* and *length* in which the prefix (if any) stored at this node is recorded along with the ITCAM or LTCAM slot in which the prefix is stored and the nexthop and length

	Prefix	Nexthop
P1	*	H1
P2	00*	H2
P3	01*	H3
P4	000*	H4

(a) A 4-prefix forwarding table



(b) Its binary trie representation



(c) DUOS representation

Figure 4. DUOS for an example 4-prefix forwarding table. Note that prefixes in ITCAM are stored in length order, whereas those in LTCAM are stored arbitrarily since the prefixes are disjoint.

of the prefix. Functions for basic operations on the control plane trie (hereinafter simply referred to as trie) are assumed (see Figure 5).

Function: Trie.insert
(a, b) = Trie.insert(prefix, length, nextHop);
This function inserts a prefix given its length and next hop into the control-plane binary trie. It returns the trie node a which stores the new prefix and a's nearest ancestor node b that contains a prefix.
Function: Trie.delete
(a, b) = Trie.delete(prefix, length);
This function deletes a prefix from the control plane trie and returns the trie node a that used to store the prefix just deleted and a's nearest ancestor node b that contains a prefix.
Function: Trie.change
a = Trie.change(prefix, length, newHop);
This function changes the next hop associated with a prefix and returns the trie node a that contains the prefix.

Figure 5. Table of control-plane trie functions

As the control plane will modify the ITCAM, LTCAM, ISRAM, and LSRAM while the data plane performs lookups, the TCAMs need to be dual ported. Specifically, we make the following assumptions:

1. Each TCAM has two ports, which can be used to simultaneously access the TCAM from the control plane and the data plane.

2. Each TCAM entry/slot is tagged with a valid bit, that is set to 1 if the content for the entry is valid, and to 0 otherwise. A TCAM lookup engages only those slots whose valid bit is 1. The TCAM slots engaged in a lookup are determined at the start of a lookup to be those slots whose valid bits are 1 at that time. Changing a valid bit from 1 to 0 during a data plane lookup does not disengage that slot from the ongoing lookup. Similarly, changing a valid bit from 0 to 1 during a data plane lookup does not engage that slot until the next lookup.

We assume the availability of the function $waitWriteValidate$ which writes to a TCAM slot and sets the valid bit to 1. In case the TCAM slot being written to is the subject of ongoing data plane lookup, the write is delayed till this lookup completes. During the write, the TCAM slot being written to is excluded from data plane lookups[1]. This is equivalent to the requirement that "After a rule is matched, resetting the valid bit has no effect on the action return process" [18], and to setting the valid entry to "hit" [19]. Similarly, we assume the availability of the function $invalidateWaitWrite$, which sets the valid bit of a TCAM slot to 0 and then writes an address to the associated SRAM word in such a way that the outcome of the ongoing lookup is unaffected.

We note that $waitWriteValidate$ may, at times, write the prefix and nexthop information in the TCAM and associated SRAM slot and validate it, without any wait. This happens, for example, when the writing is to be done to a TCAM slot that is not the subject of the ongoing data plane lookup. The wait component of the function $waitWriteValidate$ is said to be null in this case.

Figure 6 lists the various update algorithms we define later in this section for DUOS and its associated ITCAM and LTCAM. The indentation represents the hierarchy of function calls. A function at one level of indentation calls one or more functions below it at the next level of indentation or at the same level of indentation.

```
dual−TCAM:
insert
delete
change
    ITCAM (with simple SRAM):
    insert
    delete
    change
        getSpace
        freeSpace
            movesFromAbove
            movesFromBelow
            getFromAbove
            getFromBelow
    LTCAM (with simple SRAM)
    insert
    delete
    change
    LTCAM (with wide SRAM)
    insert
    delete
    change
        addSuffix
        split
        carve
```

Figure 6. Table of functions used for incremental update

3.1 DUOS Incremental Update Algorithms

3.1.1 Insert

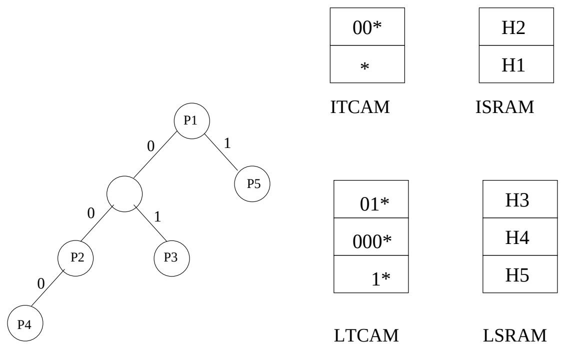
Figure 7 gives the algorithm to insert a new prefix p of length l and nexthop h. For simplicity, we assume that p is, in fact new (i.e., p is not already in the rule table). First, p is inserted into the trie using the trie insertion algorithm, which returns nodes m and n, where m is the trie node storing p and n is the nearest ancestor (if any) of m that has a prefix. When m is a leaf of the trie, there is a possibility that the insertion of p transformed a prefix that was previously a leaf prefix into a non-leaf prefix. If so, this prefix is moved from the LTCAM to the ITCAM. Regardless, p is inserted into

[1]A possible mechanism to accomplish this exclusion is to set the valid bit to 0 before commencing the write and to change this bit to 1 when the write completes.

```
Algorithm: insert ($p$, $l$, $h$)
 ($m$, $n$) = Trie.insert($p$, $l$, $h$)
 if $m$ is a leaf then begin
     if $n$ exists and $n \rightarrow prefix$ was a leaf prefix then
         $slot$ = ITCAM.insert($n \rightarrow prefix$, $n \rightarrow nexthop$, $n \rightarrow length$); // $n \rightarrow prefix$ is no longer a leaf
         LTCAM.delete($n \rightarrow slot$);
         $n \rightarrow slot$ = $slot$;
     endif
     $m \rightarrow slot$ = LTCAM.insert($p$, $h$, $l$);
 else $m \rightarrow slot$ = ITCAM.insert($p$, $h$, $l$);
 endif
```

Figure 7. Algorithm to insert into DUOS

the LTCAM. When m is not a leaf, p is inserted into the ITCAM. Figure 8, 9 and 10 illustrate the insertion of rules P5, P6 and P7 respectively starting with the initial prefix trie in Figure 4.



(a) updated trie (b) updated DUOS

Figure 8. Insert rule P5 - $\{$1*, H5$\}$ to the initial table in Figure 4. P5 is a leaf and hence is added to the LTCAM

3.1.2 Delete

Figure 11 gives the algorithm to delete the prefix p from DUOS. For simplicity, we assume that p is, in fact, present in the rule table and so may be deleted. First, p is deleted from the trie. The trie deletion function returns nodes m and n, where m is the trie node where p was stored and n is the nearest ancestor (if any) of m that has a prefix. If m was a leaf, then p is to be deleted from the LTCAM. In this case, the prefix (if any) in n may become a leaf prefix. If so, the prefix in n is to be moved from the ITCAM to the LTCAM. When m is not a leaf, p is deleted from the ITCAM. Figure 12, 13 14 illustrate the delete procedure of prefixes P7, P4 and P5 respectively starting with the prefix trie in Figure 10.

3.1.3 Change

To change the nexthop of an existing prefix to $newH$, we first change the next hop of the prefix in the trie and return the node m that contains p. Then, depending on whether m is a leaf or non leaf, we invoke the change function for the corresponding TCAM. Figure 15 gives the algorithm.

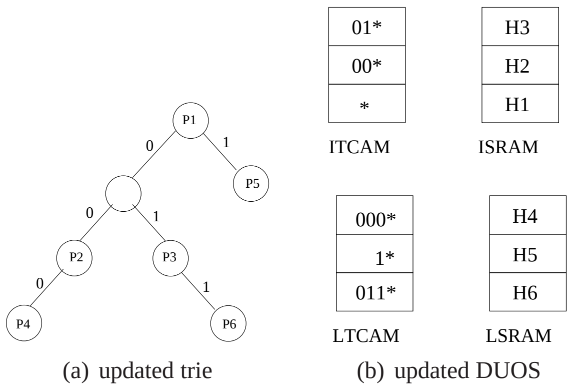


Figure 9. Insert rule P6 - $\{$011*, H6$\}$ to the prefixes in Figure 8. P6 is added to the LTCAM, while P3, which is no longer a leaf, is deleted from LTCAM and added to ITCAM.

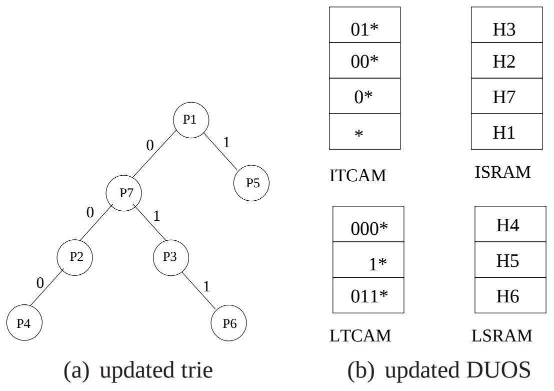
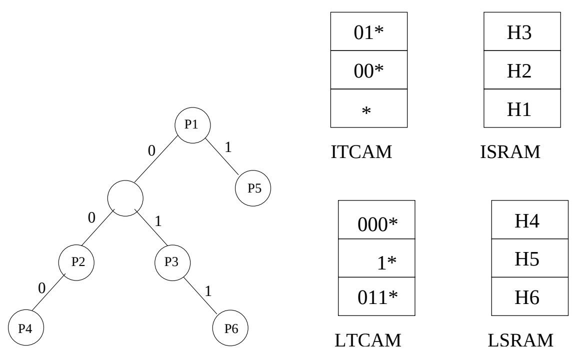


Figure 10. Insert rule P7 - $\{$0*, H7$\}$ to the prefixes in Figure 9. P7 is added to the ITCAM since it involves an intermediate prefix.

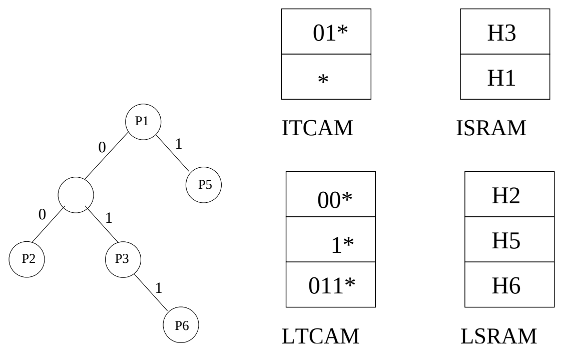
```
Algorithm: delete ($p$, $l$)
    ($m$, $n$) = Trie.delete($p$, $l$)
    If $m$ is a leaf then
        LTCAM.delete($m \rightarrow slot$)
        If $n$ exists and $n$ is now a leaf then
            $slot$ = LTCAM.insert($n \rightarrow prefix$, $n \rightarrow nexthop$, $n \rightarrow length$)
            ITCAM.delete($n \rightarrow slot$, $n \rightarrow length$) // since $n$ is now a leaf prefix
            $n \rightarrow slot$ = $slot$;
        endif
    else
        ITCAM.delete($m \rightarrow slot$, $m \rightarrow length$)
    endif
```

Figure 11. Algorithm to delete from DUOS



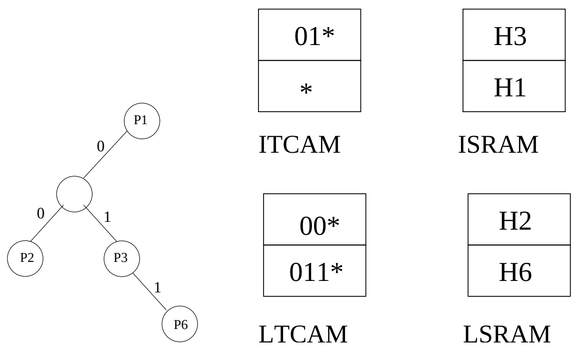
(a) updated trie (b) updated DUOS

Figure 12. Delete rule P7 - $\{$0*, H7$\}$ from the prefixes in Figure 10. P7 is deleted from ITCAM.



(a) updated trie (b) updated DUOS

Figure 13. Delete rule P4 - $\{$000*, H4$\}$ from the prefixes in Figure 12. P4 is deleted from LTCAM. P2 is inserted to LTCAM and deleted from ITCAM as P2 is now a leaf.



(a) updated trie (b) updated DUOS

Figure 14. Delete rule P5 - $\{$1*, H5$\}$ from the prefixes in Figure 13. P5 is deleted from LTCAM.

```
Algorithm: change ($p$, $length$, $newH$)
    $m$ = Trie.change($p$, $l$, $newH$)
    If $m$ is a leaf then
        $m \rightarrow slot$ = LTCAM.change($p$, $m \rightarrow slot$, $newH$);
    else
        $m \rightarrow slot$ = ITCAM.change($p$, $m \rightarrow slot$, $newH$, $length$);
```

Figure 15. Algorithm to change a next hop in DUOS

```
Algorithm: insert(prefix, nexthop, length)
    slot = getSlot(length);
    ITCAM.$waitWriteValidate$(slot, prefix, nexthop);
    return slot;

Algorithm: delete(slot, length)
    freeSlot(slot, length);

Algorithm: change(prefix, oldSlot, nexthop, length)
    slot = insert(prefix, nexthop, length);
    delete(oldSlot, length);
    return slot;
```

Figure 16. ITCAM algorithms

3.2 ITCAM Algorithms

The prefixes in the ITCAM are stored in such a manner as to support determining the longest matching prefix (i.e., in any topological order that conforms to the precedence constraints defined by the binary trie–$p1$ must come before $p2$ whenever $p1$ is a descendent of $p2$ [23]). Decreasing order of length is a commonly used ordering. The function $getSlot(length)$ returns an ITCAM slot such that insertion of the new prefix into this slot satisfies the ordering constraint in use provided the new prefix has the specified length; the function $freeSlot(slot, length)$ frees a slot previously occupied by a prefix of the specified length and makes this slot available for reuse later. These functions, which are described in Section 3.4, are used in our ITCAM insert, delete, and change algorithms (Figure 16), which are self explanatory.

Notice that following the first step of the change algorithm, the prefix whose next hop is being changed is in two valid slots of the ITCAM–$oldSlot$ and $slot$. This duplication does not affect correctness of data plane lookups as whichever one is matched by the ITCAM, we return the next hop that is valid either before or after the change operation. On the other hand, if we attempted to change the next hop in $ISRAM[oldSlot]$ directly, an ongoing lookup may return a garbled next hop. Similarly, if we delete first and then insert, lookups that take place between the delete and the insert may return a next hop that doesn't correspond to the routing table state either before or after the change. If a $waitWriteValidate$ is used to change $ISRAM[oldSlot]$ to nexthop, $oldSlot$ becomes unavailable for data plane lookups during the write operation and inconsistent results are returned in case the prefix in TCAM[$oldSlot$] is the longest matching prefix.

3.3 LTCAM Algorithms

The prefixes in the LTCAM are disjoint and so may be stored in any order. The unused (or free) slots of the LTCAM/LSRAM are linked together into a chain using the words of the LSRAM to build this chain. We use AV to store the index of the first LSRAM word on the chain. So, the free slots are AV, $LSRAM[AV]$, $LSRAM[LSRAM[AV]]$, and so on. The last free slot on the AV chain has $LSRAM[last] = -1$. The LTCAM algorithms to insert, delete, and change are given in Figure 17. These algorithms are self explanatory.

3.4 ITCAM Memory Management

In this section, we describe four possible memory management schemes for an ITCAM. The description of each memory management scheme includes an implementation of the $getSlot$ and $freeSlot$ functions used in Section 3.2 to get and free ITCAM slots. The implementations employ the function $move$ (Figure 18) that moves the content of an in-use ITCAM slot to a free ITCAM slot in such a way as to maintain data plane lookup consistency. Our memory

```
Algorithm: insert(prefix, nexthop, length)
    if ($AV == -1$) throw NoSlotException;
    slot = AV;
    AV = LSRAM[slot];
    LTCAM.$waitWriteValidate$(slot, prefix, nexthop);
    return slot;

Algorithm: delete(slot)
    LTCAM.$invalidateWaitWrite$(slot, AV); // AV is stored in LSRAM[slot] after waiting
                                           // for an ongoing lookup to complete
    AV = slot;

Algorithm: change(prefix, oldSlot, nexthop, length)
    slot = insert(prefix, nexthop, length);
    delete (oldSlot);
    return slot;
```

Figure 17. LTCAM algorithms

```
Algorithm: $move$ ($src$, $dest$)
    ITCAM.$waitWriteValidate$($dest$, ITCAM[$src$], ISRAM[$src$]);
```

Figure 18. Move from ITCAM[src] to ITCAM[$dest$]

management algorithms maintain the invariant that an ITCAM slot has its valid bit set to 0 iff that slot wasn't matched by the ongoing data plane lookup (if any); that is, iff the slot isn't involved in the ongoing data plane lookup.

3.4.1 Memory Management Scheme 1

This scheme, which is the PLO_OPT scheme of [23], is shown in Figure 19(a), the ITCAM slots are indexed 0 through N. The prefixes are stored in decreasing order of length in the TCAM, which ensures that the longest matching prefix is returned as the first matching prefix. The pool of free slots is kept at the logical center of the TCAM, that is, the first free slot in the pool appears after all blocks of prefixes of length $W/2+1$ or more and the last free slot appears before all blocks of prefixes of length $W/2$ or less, where W is the width of the IP address (32 in the case of IPv4). As noted in [23], this scheme requires at most $W/2$ moves for each $getSlot$ and $freeSlot$ request. Our contribution is to provide an implementation that maintains consistency of data plane lookups.

Our lookup consistent implementation of $getSlot$ and $freeSlot$ employ the following variables:
W = prefix length (32 for IPv4)
top[i] = first slot used by block i, $1 \le i \le W/2$
bot[i] = last slot used by block i, $W/2+1 \le i \le W$

The following invariants are maintained:
top[i] = top[i-1] iff block i is empty, $1 \le i \le W/2$
bot[i] = bot[i+1] iff block i is empty, $W/2+1 \le i \le W$

Initially, all blocks are empty and top[$0 : W/2$] = N+1 and bot[$W/2+1 : W+1$] = -1 (recall that the ITCAM slots are indexed 0:N). Figures 20 and 21, respectively, give the $getSlot$ and $freeSlot$ algorithms for Scheme 1. Their correctness and the fact that data plane lookup consistency is preserved are easily established.

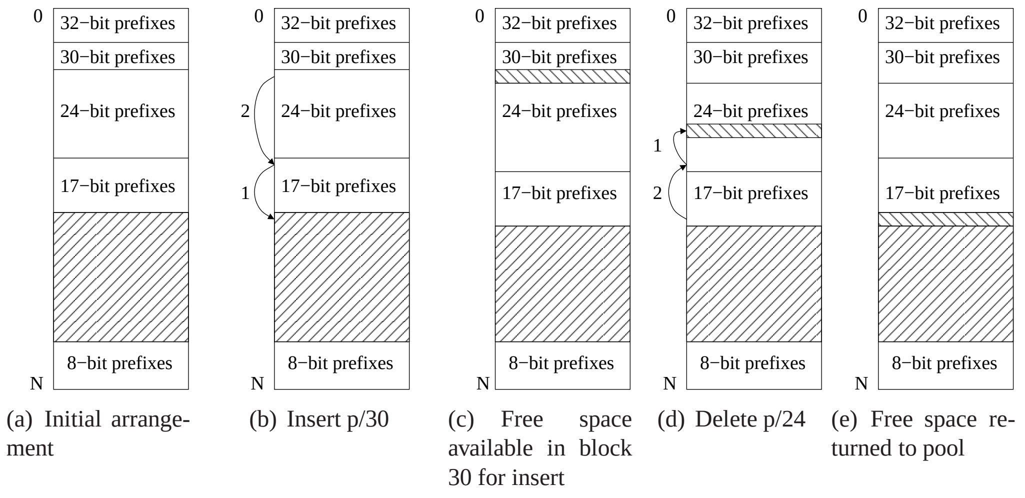


Figure 19. Prefix arrangement in ITCAM for Scheme 1 for IPv4. The free space pool is indicated by hatched lines. Numbers 1, 2 by the curved arrow correspond to the first and second move, respectively.

3.4.2 Memory Management Scheme 2

This scheme is a variation of Scheme 1 in which the free slots are in the boundary between two prefix blocks (Figure 22). This scheme is also called DFS_PLO (Distributed Free Space with Prefix Length Ordering Constraint). At the time the ITCAM is initialized, the available free slots are distributed in proportion to the number of prefixes in a block with the caveat that an empty block gets 1 free slot at its boundary. In this scheme, top[i] is the slot where the first prefix of length i is stored and bot[i] is the slot where the last prefix of length i is stored, $0 \leq i \leq W$ (i.e., these variables define the start and end of block i). Note that top[i] $\leq$ bot[i] for a non-empty block i and top[i] $>$ bot[i] for an empty block. For convenience, we define top[0]=bot[0]=$N+1$ and top[$W+1$]=bot[$W+1$] = -1. For an empty ITCAM, top[i] = $N+1$ for $1 \leq i \leq W$; bot[i] = -1 for $1 \leq i \leq W$.

Our $getSlot$ algorithm (Figure 23) provides a free slot from either block boundary when there is a free slot on the block boundary. Otherwise, it moves a free slot from the nearest block boundary that has a free slot. This algorithm utilizes several supporting algorithms that are given in Figure 24. The algorithm $movesFromAbove$ ($movesFromBelow$) returns the number of prefix moves that are required to get the nearest free slot from above (below) the block where it is needed and $getFromAbove$ and $getFromBelow$, respectively, get the nearest free slot above or below the block where the free slot is needed.

The algorithm to free a slot (Figure 25) simply moves the slot to be freed to the block boundary unless this slot is at the boundary to begin with. Again, correctness and consistency are established easily. Although the worst-case performance of the Scheme 2 algorithms is the same as that of the Scheme 1 algorithms, we expect the Scheme 2 algorithms to have better performance on average.

3.4.3 Memory Management Scheme 3

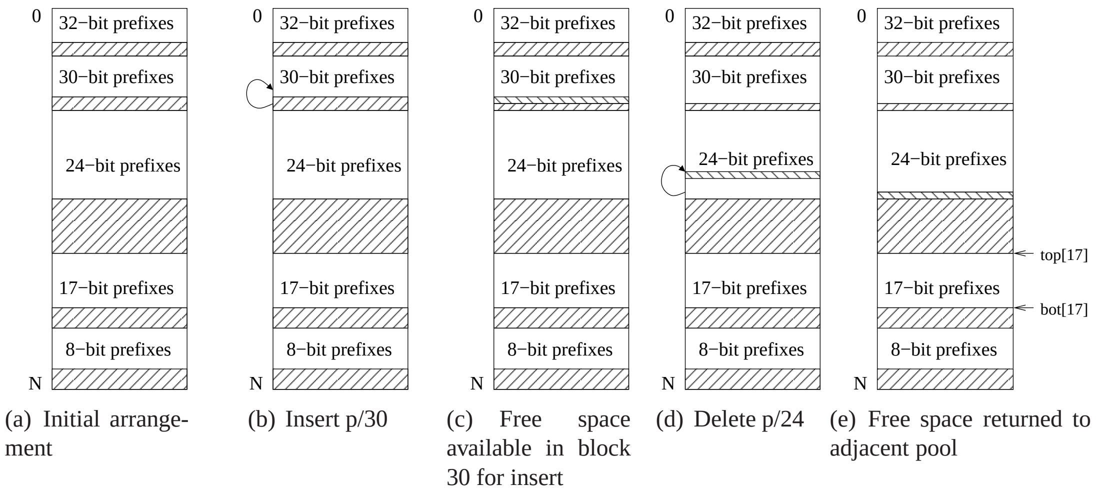
This is an enhancement of Scheme 2 in which we maintain a doubly-linked list of free slots within each block in addition to contiguous free slots at the block boundaries (Figure 26). This scheme is also called DLFS_PLO (Distributed and Linked Free Space with Prefix Length Ordering Constraint). The lists of free slots within a block enable us to avoid the move that is done by the Scheme 2 $freeSlot$ algorithm of Figure 25. The forward links, called next[], of the doubly-linked list are maintained using the ISRAM words corresponding to the free ITCAM slots with AV[i] recording the first slot on the list for the ith block. The backward links, called prev[], are maintained in these ISRAM words in case an ISRAM word is large enough to accommodate two links and in the control plane memory otherwise. All variables, including the array AV[], are, of course, stored in the control plane memory.

```
**Algorithm: getSlot($len$)**
//$len$: length of prefix to be inserted.
// returns free slot for prefix insertion
if (bot[$W/2+1$] $==$ top[$W/2$] - 1) throw NoSpaceException;
if ($len \geq W/2+1$)
    $d$ = ++bot[$W/2+1$];
    for ($i = W/2+2$; $i \leq len$; ++$i$)
        if (bot[$i$] == $d$-1) // block $i-1$ is empty
            bot[$i$] = bot[$i-1$];
        else // move from top of $i-1$ to d
            $s$ = ++bot[$i$];
            $move(s,d)$;
            $d$ = $s$;
        endif
else
    $d$ = $--$top[$W/2$];
    for ($i = W/2-1$; $i >= len$; $--i$)
        if (top[$i$] == $d$+1) // block $i+1$ is empty
            top[$i$] = top[$i+1$];
        else // move from bottom of $i+1$ to $d$
            $s$ = $--$top[$i$];
            $move(s,d)$;
            $d$ = $s$;
        endif
endif
```

Figure 20. Scheme 1 algorithm to get a free slot to insert a prefix whose length is len

```
**Algorithm: freeSlot($slot$, $len$)**
// free ITCAM[$slot$] which had a prefix of length $len$
if ($len \geq W/2+1$) // free space from the top half.
    if ($slot$ != bot[$len$])
        $move$ (bot[$len$], $slot$); $slot$ = bot[$len$];
    endif
    bot[$len$]$--$;
    for ($i = len - 1$; $i > W/2$; $--i$)
        if (bot[$i$] != $slot$) // block $i$ is not empty
            move (bot[$i$], $slot$); $slot$ = bot[$i$];
        endif
        bot[$i$]$--$;
    endfor
else // free space from the bottom half.
    if ($slot$ != top[$len$])
        $move$ (top[$len$], $slot$); $slot$ = top[$len$];
    endif
    top[$len$]++;
    for ($i$=$len$+1; $i$≤$W/2$; ++$i$)
        if (top[$i$] != $slot$) // block $i$ is not empty
            $move$ (top[$i$], $slot$); $slot$ = top[$i$];
        endif
        top[$i$]++;
    endfor
    ITCAM[slot].valid = 0;
endif
```

Figure 21. Scheme 1 algorithm to free a slot previously occupied by a prefix of length len



(a) Initial arrangement (b) Insert p/30 (c) Free space available in block 30 for insert (d) Delete p/24 (e) Free space returned to adjacent pool

Figure 22. ITCAM layout for Scheme 2

```
Algorithm: getSlot($len$)
    ma = movesFromAbove($len$, &aPos);
    mb = movesFromBelow($len$, &bPos);
    if (ma $<$ mb)
        $d$ = getFromAbove($len$, aPos);
        if (top[$len$] $>$ bot[$len$]) bot[$len$] = $d$;
        top[$len$] = $d$;
    else
        if (mb == $W$) throw NoSpaceException;
        $d$ = getFromBelow($len$, bPos);
        if (top[$len$] $>$ bot[$len$]) top[$len$] = $d$;
        bot[$len$] = $d$;
    endif
    return $d$;
```

Figure 23. Scheme 2 algorithm to get a free slot to insert a prefix whose length is len

The scheme 3 $getSlot$ algorithm (Figure 27) first attempts to make available a slot from the doubly-linked list for the desired block. When this list is empty, the algorithm behaves like the $getSlot$ algorithm for Scheme 2 and the supporting algorithms of Figure 28 are similar to the corresponding supporting algorithms for Scheme 3.

The algorithm to free a slot (Figure 29) differs from that for Scheme 2 in that when the slot being freed is inside a block it is added to the doubly-linked list of free slots. Again, correctness and consistency are established easily. Although the worst-case performance of the Scheme 3 algorithms is the same as that of the algorithms for the first two schemes, we expect the Scheme 3 algorithms to have better performance on average.

3.4.4 Scheme 4

This is the CAO_OPT scheme presented in [23]. Here, prefixes are arranged in chain order, with the free space pool in the middle of the ITCAM. Figures 30–32 give the necessary algorithms. The interfaces are different from those used by the first 3 schemes. The input to $getSlot$ is p, which is the node in the trie where the prefix being inserted is stored. Each trie node stores wt, wt_ptr, $hcld\_ptr$, $lchild$, $rchild$, which are explained in [23]. In addition to these we use the following variables:

```
Algorithm: movesFromAbove($len$, *$pos$) // returns number of moves needed to acquire free space from above the block of length $len$
    moves=0;
    for (p=$len$; top[p] $>$ bot[p]; p$--$); // find max p $\le len$ such that block p is not empty
    for (c=$len$+1; c$\le W+1$; c++) // find min c $> len$ with space just below it
        if (top[c] $\le$ bot[c]) // not empty
            if (bot[c]+1 $<$ top[p]) *$pos$ = p; return moves; endif
            moves++;
            p = c;
        endif
    return $W$;

Algorithm: movesFromBelow($len$, *$pos$) // returns number of moves needed to acquire free space from below the block of length $len$
    moves=0;
    for (p=$len$; top[p] $>$ bot[p]; p++); // find min p $>= len$ such that block p is not empty
    for (c=$len$-1; c$>=$0; c$--$) // find min c $> len$ with space just below it
        if (top[c] $\le$ bot[c]) // not empty
            if (top[c]-1 $>$ bot[p]) *$pos$ = p; return moves; endif
            moves++;
            p = c;
        endif
    return $W$;

Algorithm: getFromAbove($len$, $pos$) // get free space from above
    $d$ = top[$pos$]-1;
    for (c=$pos$; c$>$ $len$; c$--$)
        if (top[c] $\le$ bot[c])
            $d$ = bot[c];
            $move$(bot[c]$--$, $--$top[c]);
        endif
    return $d$;

Algorithm: getFromBelow($len$, $pos$) // get free space from below
    $d$ = bot[$pos$]+1;
    for (c=$pos$; c$<$ $len$; c++)
        if (top[c] $\le$ bot[c])
            $d$ = top[c];
            $move$(top[c]++, ++bot[c]);
        endif
    return $d$;
```

Figure 24. Supporting algorithms used by the algorithm of Figure 23

```
Algorithm: freeSlot($d$, $len$)
    if (top[$len$] == $d$) ITCAM[top[$len$]++].valid = 0;
    else if (bot[$len$] == $d$) ITCAM[bot[$len$]−−].valid = 0;
    else
        $move$ (bot[$len$], $d$);
        ITCAM[bot[len]−−].valid = 0;
    endif
```

Figure 25. Scheme 2 algorithm to free a slot

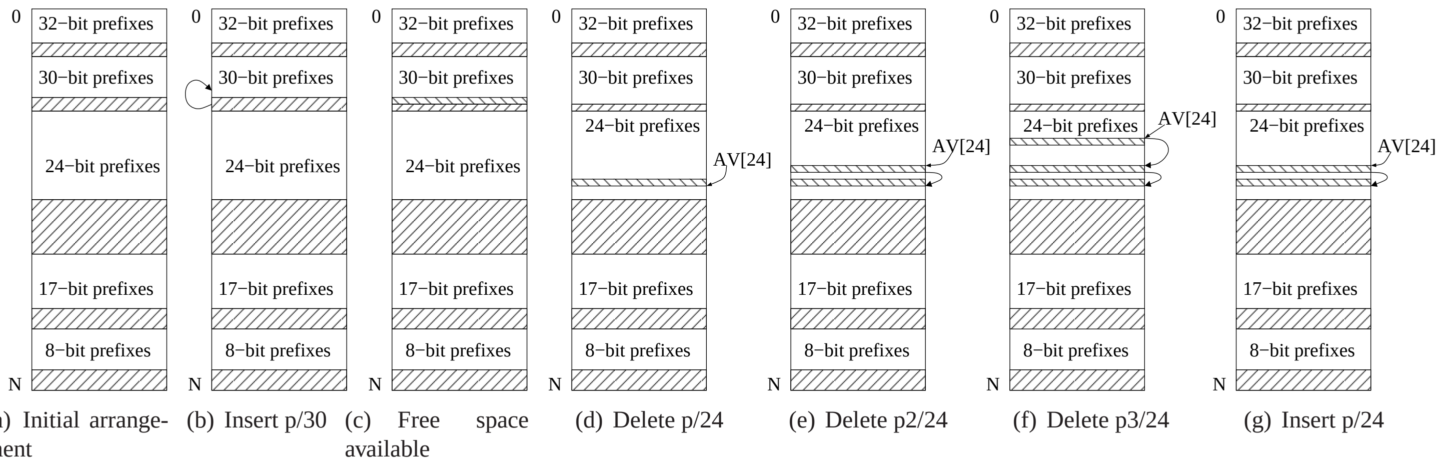


Figure 26. ITCAM layout for Scheme 3, with moves for insert and delete. The curved arrows on the right show the forward links in the list of free spaces.

$slot$: address of ITCAM slot in which prefix is entered. If prefix has not yet been entered, then this variable is set to -1.
$firstFree$: first free space
$lastFree$: last free space
$shift$[0:W/2]: temporary array of nodes
Also needed is an array of nodes, say $nodeMap$[0:N] for ITCAM[0:N] that contains the node address of each valid prefix in the ITCAM, so that they can be located in the trie.

4 Wide Dual TCAM–DUOW

In this section we extend our DUOS scheme to the case when wide SRAMs (say, 144-bit words or larger) are in use. We describe the extension only for the case when the LSRAM is wide. The case when the ISRAM is wide uses techniques almost identical to those used in [4] while for a wide LSRAM, we need to modify these techniques. As in [4], a wide LSRAM word is used to store a subtree of the binary trie of a forwarding table. However, instead of beginning with the binary trie for all prefixes as is done in [4], we begin with the binary trie, leaf trie, for only the leaf prefixes. When a subtree of the leaf trie is stored in an LSRAM word, that subtree is removed from (or carved out of) the leaf trie before another subtree is identified for carving. Let N be the root of the subtree being carved and let $Q(N)$ be the prefix defined by the path from the root of the trie to N. $Q(N)$ is stored in the LTCAM, and $|P_i| - |Q(N)|$ suffix bits, of each prefix P_i in the carved subtree rooted at N, are stored in the LSRAM word. Note that each suffix stored in the LSRAM word is a suffix of a leaf prefix that begins with $Q(N)$. By repeating this carving process, all leaf prefixes are allocated to the LTCAM and LSRAM. To obtain the mapping of leaf prefixes to the LTCAM and LSRAM, we need a carving algorithm that ensures that the $Q(N)$s stored in the LTCAM are disjoint. Since the carving algorithm of [4] does not

```
Algorithm: getSlot($len$)
    aP=0; bP=0; aC=0; bC=0;
    if (AV[$len$] == -1) // AV[$len$] stores the first free space in block of length $len$
        ma = movesFromAbove($len$, &aP, &aC);
        mb = movesFromBelow($len$, &bP, &bC);
        if (ma $<$ mb)
            $d$ = getFromAbove($len$, aP, aC);
            if (top[$len$] $>$ bot[$len$]) bot[$len$] = $d$;
            top[$len$] = $d$;
        else
            if (mb == $W$) throw NoSpaceException; // no space
            $d$ = getFromBelow($len$, bP, bC);
            if (top[$len$] $>$ bot[$len$]) top[$len$] = $d$;
            bot[$len$] = $d$;
        endif
    else
        $d$ = AV[$len$];
        AV[$len$] = next[$d$];
    endif
    return $d$;
```

Figure 27. Scheme 3 algorithm to get a free slot to insert a prefix whose length is len

ensure disjointedness, a new carving algorithm is needed. As an example, consider the binary trie of Figure 33(a), which has been carved using a carving algorithm that ensures that each carved subtree has at most 2 leaf prefixes. The LTCAM will need to store $Q(N1)$, $Q(N2)$ and $Q(N3)$. Even though the prefixes in the binary trie are disjoint, the $Q(N)$s in the LTCAM are not disjoint (e.g., $Q(N1)$ is a descendant of $Q(N2)$ and so $Q(N2)$ matches all IP addresses matched by $Q(N1)$). To retain much of the simplicity of the LTCAM management scheme of DUOS it is necessary to carve the leaf trie in such a way that all $Q(N)$s in the LTCAM are disjoint.

As in [4], we carve via a postorder traversal of the binary trie. However, we use the visit algorithm of Figure 34 to do the carving. In this algorithm, w is the number of bits in an LSRAM word and $x \rightarrow$size is the number of bits needed to store (1) the suffix bits corresponding to prefixes in the subtrie rooted at x, (2) the length of each suffix, (3) the next hop for each suffix, (4) the number of suffixes in the word, and (5) the length of $Q(x)$, which is the corresponding prefix stored in the LTCAM. Algorithm $splitNode(q)$ (not specified in this paper) does the actual carving of the subtree rooted at node q. The basic idea in our carving algorithm is to forbid carving at two nodes that have an ancestor-descendent relationship. This ensures that the $Q(N)$s are disjoint. Figure 33(b) shows the subtrees carved by our algorithm. As can be seen, $Q(N1)$, $Q(N2)$, $Q(N3)$ are disjoint. Although our carving algorithm generally results in more $Q(N)$s than when the carving algorithm of [4] is used, our carving algorithm allows us to retain the flexibility to store the $Q(N)$s in any order in the LTCAM as the $Q(N)$s are independent.

The LTCAM algorithms to insert, delete, change, and necessary support algorithms are given in Figures 35–39.

The function $carve$ is invoked by both the insert and delete algorithms under different contexts that we analyze below. When a prefix is deleted, the LSRAM word storing its suffix (corresponding to the LTCAM word for $Q(cNode)$) may have remaining suffixes that can be merged with another LSRAM word. This merge is accomplished by the $carve$ function, by carving the trie at $tNode$, which is the nearest ancestor with two children, of $cNode$. Thus $carve$ helps to reduce the LTCAM entries by one. When a prefix is inserted, it may be possible to add the suffix bits of the new prefix in the LSRAM word that corresponds to the LTCAM slot for $Q(cNode)$. If there is no $cNode$ in the path between the new prefix node and the root, then we try to carve at $tNode$, which is the nearest degree 2 ancestor of the new prefix node, and therefore includes the new prefix along with other existing prefixes. So, in this case, using $carve$ we prevent

Algorithm: movesFromAbove(len, *pos, *cur)
```
moves=0;
for (p=$len$; top[p] $>$ bot[p]; p$--$); // find max p $\leq len$ such that block p is not empty
for (c=$len$+1; c$\leq W$+1; c++) // find min c $> len$ with space just below it
    if (top[c] $\leq$ bot[c]) // not empty
        if (bot[c]+1 $<$ top[p] || !valid[bot[c]])
            *$cur$ = c; *$pos$ = p;
            return moves;
        endif
        moves++;
        if (AV[c] $>=$ 0) *$pos$ = c; *$cur$ = c; return moves; endif
        p = c;
    endif
return $W$;
```

Algorithm: movesFromBelow(len, *pos, *cur)
```
moves=0;
for (p=$len$; top[p] $>$ bot[p]; p++); // find min p $>= len$ such that block p is not empty

for (c=$len$-1; c$>=$0; c$--$) // find min c $> len$ with space just below it
    if (top[c] $\leq$ bot[c]) // not empty
        if (top[c]-1 $>$ bot[p] || !valid[top[c]])
            *pos = p; *cur = c;
            return moves;
        endif
        moves++;
        if (AV[c] $>=$ 0) *$pos$ = c; *$cur$ = c; return moves; endif
        p = c;
    endif

return $W$;
```

Algorithm: getFromAbove(len, p, c)
```
if (top[$p$] $>$ bot[$c$]+1) $d$ = top[$p$]-1; $c$ = $p$;
else
    if (!valid[bot[$c$]])
        $d$ = bot[$c$]$--$;
        if ($d$ == AV[$c$]) AV[$c$] = next[AV[$c$]];
        else
            next[prev[$d$]] = next[$d$];
            if (next[$d$] != -1) prev[next[$d$]] = prev[$d$];
        endif
    else
        $d$ = AV[$c$];
        AV[$c$] = next[AV[$c$]];
        $move$(bot[$c$], $d$);
        $d$ = bot[$c$]$--$;
    endif
    $c--$;
endif
for (; $c > len$; $c--$)
    if (top[$c$] $\leq$ bot[$c$])
        $move$(bot[$c$]$--$, $--$top[$c$]);
        $d$ = bot[$c$]+1;
    endif
return $d$;
```

```
**Algorithm: freeSlot($d$, $len$)**
    if (top[$len$] == $d$) ITCAM[top[$len$]++].valid = 0;
    else if (bot[$len$] == $d$) ITCAM[bot[$len$]$--$].valid = 0;
    else
        ITCAM.$invalidateWaitWrite$($d$, AV[$len$]); // AV[$len$] is stored in ISRAM[$d$].
        if (AV[$len$] != -1) prev[AV[$len$]] = $d$;
        AV[$len$] = $d$;
    endif
```

Figure 29. Scheme 3 algorithm to free a slot

```
**Algorithm: getSlot($p$)**
    $d$ = isTopHeavy($p$) ? $lastFree--$ : $firstFree$++;
    if (parent($p$) and $p$$\rightarrow$parent($p$)$\rightarrow slot < firstFree$) // Case I: Insert prefix on top.
        c = parent($p$); i=0;
        while (c and c$\rightarrow slot < firstFree$)
            $shift$[i++] = c;
            c = parent(c);
        endwhile
        for (j=i-1; j$>=$0; $--$j)
            tmp = $shift$[j]$\rightarrow slot$ ;
            $move$($shift$[j]$\rightarrow slot$, $d$);
            $d$ = tmp;
        endfor
    else if (child($p$) and child($p$)$\rightarrow slot > lastFree$) // Case II: Insert prefix on bottom.
        c = child($p$); i=0;
        while (c and c$\rightarrow slot > lastFree$)
            $shift$[i++] = c;
            c = child($p$);
        endwhile
        for (j=i-1; j$>=$0; –j)
            tmp = $shift$[j]$\rightarrow slot$ ;
            $move$($shift$[j]$\rightarrow slot$, $d$);
            $d$ = tmp;
        endfor
    endif
    return $d$;
```

Figure 30. Scheme 4 $getSlot$ algorithm

```
Algorithm: freeSlot($d$)
    if ($d < firstFree$)
        c = $nodeMap$[$firstFree$-1]; i=0;
        while (c and c$\rightarrow slot > d$)
            $shift$[i++] = c;
            c = child(c);
        endwhile
        for (j=i-1; j$>=$0; –j)
            tmp = $shift$[j]$\rightarrow slot$ ;
            $move$($shift$[j]$\rightarrow slot$, $d$);
            $d$ = tmp;
        endfor
        $firstFree - -$;
    else
        c = $nodeMap$[$lastFree$+1]; i=0;
        while (c and c$\rightarrow slot < d$)
            $shift$[i++] = c;
            c = parent(c);
        endwhile
        for (j=i-1; j$>=$0; –j)
            tmp = $shift$[j]$\rightarrow slot$ ;
            $move$($shift$[j]$\rightarrow slot$, $d$);
            $d$ = tmp;
        endfor
        $lastFree$++;
    endif
    ITCAM[$d$].valid = 0;
```

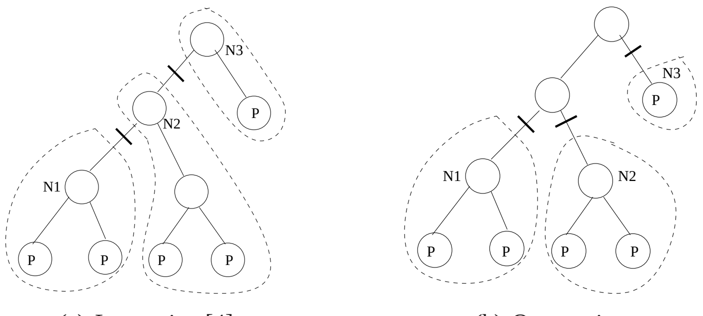
Figure 31. Scheme 4 $freeSlot$ algorithm

```
**Algorithm: isTopHeavy($p$)**
    top=bot=0;
    for (c = parent($p$); c != NULL; c = parent(c))
        if (c$\rightarrow slot > lastFree$) bot++;
        else top++;
    for (c = $p\rightarrow wt\_ptr$; c != NULL; c = c$\rightarrow wt\_ptr$)
        if (c stores a prefix)
                if (c$\rightarrow slot > -1$) // prefix is already placed in TCAM
                    if (c$\rightarrow slot > lastFree$) bot++;
                    else top++;
                else k++;
                endif
        endif
    n = top+bot+k;
    return (top $>$ n/2) ? 1 : 0
```

```
**Algorithm: parent($p$)**
    c = $p\rightarrow$parentNode;
    while (c and !c$\rightarrow$valid) c = c$\rightarrow$parentNode;
    return c;
```

```
**Algorithm: child($p$)**
    c = NULL;
    // don't return a LTCAM prefix as child.
    if ($p\rightarrow hcld\_ptr$ and $p\rightarrow hcld\_ptr\rightarrow tcam == 1$) c = $p\rightarrow hcld\_ptr$;
    return c;
```

Figure 32. Supporting control plane trie algorithms used by the Scheme 4 $getSlot$ and $freeSlot$ algorithms



(a) Lu carving [4]   (b) Our carving

Figure 33. Carving using the method of [4] and our method

```
Algorithm: visit_postorder($x$)
    if (!$x$) return 0;
    isSplit = visit_postorder($y$);
    isSplit $|=$ visit_postorder($z$);
    if (isSplit $||x\rightarrow$size $>$ w) then
        splitNode($y$);
        splitNode($z$); // where $y$ and $z$ are children of $x$
        return 1;
    else if ($x\rightarrow$size == w) then
        splitNode($x$);
        return 1;
    endif
    return 0;
```

Figure 34. Algorithm to carve a leaf trie to obtain disjoint $Q(N)$s

```
Algorithm: insert($node$, $cNode$, $tNode$)
// $node$: node in leaf trie for new prefix to be inserted.
// $cNode$: nearest carved ancestor in leaf trie of $node$ (may be NULL).
// $tNode$: nearest degree 2 ancestor of node.
    if ($cNode$)
        $d$ = $cNode\rightarrow slot$;
        addSuffix($d$, $cNode$, $node$);
        LTCAM.$invalidateWaitWrite$($d$, AV);
        AV = $d$;
    else if (!carve($tNode$, $node$)) // create new suffix node with 1 sufix
        $d$ = AV;
        AV = next[$d$];
        LTCAM.$waitWriteValidate$($d$, Q(node), suffix);
        node$\rightarrow$slot = $d$;
    endif
```

Figure 35. DUOW algorithm to insert a prefix into the LTCAM

```
Algorithm: addSuffix($slot$, $cNode$, $node$)
    if (suffix does not fit in LSRAM[$slot$]) // need another suffix node
        split($cNode$);
        $cNode\rightarrow slot$ = -1;
    else // add suffix to LSRAM[slot]
        $d$ = AV;
        AV = next[$d$];
        LTCAM.$waitWriteValidate$($d$, LTCAM[$slot$], LSRAM[$slot$] + suffix);
        $cNode\rightarrow slot$ = $d$;
    endif
```

Figure 36. Algorithm to add a suffix to a wide LSRAM word

```
Algorithm: split ($cNode$)
    if ($cNode$→y and $cNode$→z) // carve at children y & z of cNode
        cNode→y→slot = AV;
        cNode→z→slot = next[AV];
        AV = next[next[AV]];
        LTCAM.$waitWriteValidate$(cNode→y→slot, Q(cNode→y), suffixes(cNode→y));
        LTCAM.$waitWriteValidate$(cNode→z→slot, Q(cNode→z), suffixes(cNode→z));
    else if (cNode→y) split (cNode→y);
    else split(cNode→z);
    endif
```

Figure 37. Algorithm to split a wide LSRAM word into two

```
Algorithm: delete($node$, $cNode$, $tNode$)
// $node$: node in leaf trie corresponding to the prefix to be deleted
// $cNode$: nearest carved ancestor of node cannot be NULL
// $tNode$: nearest degree 2 ancestor node of cNode.
    oldSlot = $cNode$→slot;
    p = number of suffixes in LSRAM[oldSlot];
    if (p $>$ 1 and !carve($tNode$, $cNode$)) // delete suffix from its suffix node
        d = AV;
        AV = next[d];
        LTCAM.$waitWriteValidate$(d, LTCAM[oldSlot], LSRAM[oldSlot] - suffix);
        cNode→slot = d;
    else cNode→slot = -1;
    endif
    LTCAM.$invalidateWaitWrite$($oldSlot$, AV);
    AV = oldSlot;
Algorithm: carve($tNode$, $cNode$)
    if (!$tNode$) return 0;
    if (suffixes($tNode$) fit in a suffix node) // carve at tNode
        $d$ = AV;
        AV = next[$d$];
        LTCAM.$waitWriteValidate$($d$, Q(tNode), suffixes(tNode));
        tNode→slot = d;
        otherNode = the carvedNode in subtree rooted at tNode that is not cNode;
        LTCAM.$invalidateWaitWrite$(otherNode→slot, AV);
        AV = otherNode→slot;
        otherNode→slot = -1;
        return 1;
    endif
    return 0;
```

Figure 38. DUOW algorithm to delete a leaf prefix

```
Algorithm: change(prefix, cNode, $nexthop$)
    oldSlot = cNode→slot;
    $d$ = AV;
    AV = next[$d$];
    newWord = LSRAM[oldSlot] with next hop for $cNode \rightarrow prefix$ set to $nexthop$;
    LTCAM.$waitWriteValidate$($d$, prefix, newWord);
    cNode→slot = d;
    LTCAM.$invalidateWaitWrite$(oldSlot, AV);
    AV = oldSlot;
```

Figure 39. DUOW algorithm to change the next hop of a leaf prefix

the addition of a new LTCAM entry for the new prefix.

Next we show that $tNode$ is indeed an appropriate node to carve and the algorithm preserves the property of carving at only one node along any path from the root. $tNode$ is carved only if the number of bits needed to store all suffixes in the subtree rooted at $tNode$ is less than the size of an LSRAM word. In this case there is a single $otherNode$ that is a descendant of $tNode$ and for which $Q(otherNode)$ is in the LTCAM. To see that there cannot be more than one $otherNode$, suppose there are q such nodes with $Q(q)$ in the LTCAM. All of these q nodes must be in the subtree of $tNode$ that does not contain the target node, which is $cNode$ for a delete and the new prefix node for an insert. This is because, if there was one carved node t among the q nodes in the subtree of $cNode$, for a delete, then t must occur either in the path between $cNode$ and $tNode$, or as a descendant of $cNode$, given that $tNode$ is the nearest ancestor of $cNode$ with two children. In either case, t violates the property of a single carving along any path from the root. Similarly for an insert, if there were a carved node t in the same subtree that contained the newly added prefix, then t would have served as the $cNode$ and we would not have started the $carve$ algorithm in the first place. Since all q nodes must appear in the same subtree rooted at either the left or right child of $tNode$, and the sum of their sizes is small enough to fit in an LSRAM word, our carving algorithm would have carved that child of $tNode$. Thus there is only one $otherNode$. Since we delete $Q(cNode)$ and $Q(otherNode)$ right after adding $Q(tNode)$, the property of carving only once along any path is maintained.

Figure 40 shows a possible assignment of the 5-prefix example in Figure 8. The intermediate prefixes P1 and P2 are stored in the ITCAM, while the leaf prefixes P3, P4 and P5 are stored in the LTCAM using a wide LSRAM. The suffix nodes begin with the prefix length field of 2 bits in this example followed by the suffix count field of 2 bits. Next comes the (length, suffix, nexthop) triplet for each prefix encoded in the suffix node, the number of allocated bits being (2bits, 4 bits, 6 bits) respectively for the three fields in the triplet.

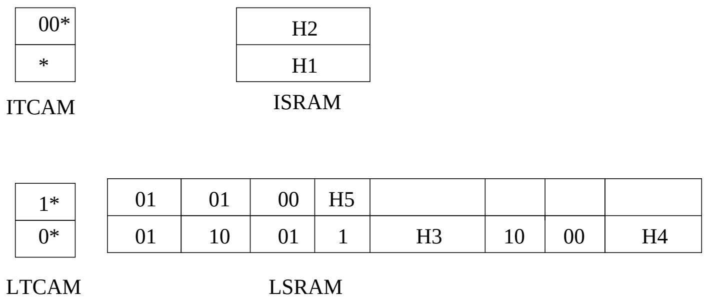


Figure 40. Assignment of prefixes of Figure 8 to the two TCAMs in the dual TCAM architecture

5 Indexed DUOW–IDUOW

Zane et al. [21] introduced the concept of an indexed TCAM that reduces significantly the power consumed by a TCAM lookup. This concept was refined by Lu and Sahni [4] to reduce both the TCAM power and space requirements substantially. We show how to incorporate an index TCAM in conjunction with an LTCAM that uses a wide LSRAM (i.e., an index for the LTCAM of DUOW). Adding an index to the ITCAM of DUOW follows easily from [4]. When the LTCAM is indexed, we have two TCAMs replacing the LTCAM–a data TCAM referred to as DLTCAM and in index TCAM referred to as ILTCAM. The associated SRAMs are DLSRAM and ILSRAM.

We consider the two most effective index TCAM strategies of [4]–1-12Wc and M-12Wb. The former is best for power whereas the latter is the best overall scheme consuming least TCAM space and low power for lookups [4]. Both 1-12Wc and M-12Wb organize the DLTCAM into fixed size buckets that are indexed using the ILTCAM and ILSRAM, which also is a wide SRAM that stores suffixes and associated information.

5.1 Memory Management for DLTCAM and ILTCAM

Each DLTCAM bucket is assigned a unique number between 0 and $totalSlots/bucketSize$, where $totalSlots$ is the total number of DLTCAM slots. The unique number so assigned to a bucket is called its index. A bucket index is stored in the trie node (in field $bIndex$) that is carved and represents an index prefix enclosing the DLTCAM prefixes in the bucket. The free slots in a bucket are linked through the associated DLSRAM. The first several bits (32 should be enough) of a DLSRAM word store the address of the next free DLTCAM slot in the same bucket. The last free slot in a bucket stores -1 in bits 0-31 of the corresponding DLSRAM word. For each bucket we keep one free slot at all times. This free slot is used for consistent updates, to copy the new prefix before deleting the old one. The first free slot in a bucket is stored in an array AV indexed by the bucket index. The array AV is initialized and maintained in the control plane. A list of free buckets is maintained in the DLSRAM using additional bits of each DLSRAM word (12 bits are sufficient when the number of buckets is at most 4096). The first available slot in a free bucket stores the bucket index of the next free bucket in the DLSRAM bits and so on. The free bucket chain is terminated by a -1 in the bits used to store the index of the next free bucket. The variable $bucketAV$ keeps track of the first bucket on the free bucket chain. In our algorithms we use the array $nextBucket$ to represent the forward links in the bucket list.

When the prefixes in an ILTCAM are disjoint, we may use the simple memory management scheme used for the LTCAM of DUOS and when these prefixes are not disjoint, they must be ordered and any of the memory management schemes proposed for the ITCAM of DUOS in Section 3.4 may be used.

The update algorithms (Figures 41–46) are almost identical for 1-12Wc and M-12Wb. We explain the differences in the next two subsections.

5.2 1-12Wc

This two-level TCAM organization in [4] employs wide SRAMs in association with both the data and index TCAMs as shown in the Figure 47. The strategy adopted in [4] to fill up the TCAMs and the SRAMs is summarized as follows. Firstly, suffix nodes are created for prefixes in the 1-bit trie, as described in Section 4, using Lu's carving heuristic. Secondly, every $Q(N)$ to be entered in the data TCAM, is treated as a prefix and the subtree split algorithm [4] is applied to carve index nodes in the trie. The carving is done so that the number of data TCAM prefixes enclosed by the node being carved, is less than or equal to the size b of a data TCAM bucket. A new bucket is assigned to every index node. An enclosed data TCAM prefix and the corresponding suffix node are entered in a new entry in the bucket. When an index node encloses fewer than b prefixes, the remaining entries in the bucket are padded with null prefixes. Finally, the index nodes are treated as prefixes, the algorithm to create suffix nodes is run on the trie containing only index prefixes. The newly carved index $Q(N)$ prefixes and the corresponding suffix nodes are entered in the index TCAM and the associated wide SRAM respectively. Using this strategy, the bucket numbers corresponding to the suffixes in an index SRAM suffix node, happen to be consecutive. Hence, the index SRAM omits the bucket number for all suffixes except the starting suffix, as shown in the Figure 47.

```
Algorithm: insert($node$, $cNode$, $tNode$, $isramNode$, $itcamNode$, $itNode$)
// $node$: node in leaf trie for new prefix to be inserted.
// $cNode$: nearest carved ancestor in leaf trie of $node$ (may be NULL).
// $tNode$: nearest degree 2 ancestor of node.
// $isramNode$: index node enclosing $cNode$
// $itcamNode$: node whose prefix exists in ILTCAM.
// $itNode$: nearest degree 2 ancestor of $isramNode$.
    if ($cNode$)
        bucketIndex = $isramNode \rightarrow bIndex$;
        $d$ = $cNode \rightarrow slot$;
        addSuffix($d$, $cNode$, $node$, $isramNode$, $itcamNode$, $itNode$);
        DLTCAM.$invalidateWaitWrite$($d$, AV[bucketIndex]);
        AV[bucketIndex] = $d$;
    else if (!carve($tNode$, $node$, bucketIndex)) // create new suffix node with 1 sufix
        if ($isramNode$) then
            bucketIndex = $isramNode \rightarrow bIndex$;
            $d$ = AV[$bucketAV$];
            AV[$bucketAV$] = next[$d$];
            if (AV[$bucketAV$] == -1) then
                splitBucket($isramNode$, $itcamNode$, $itNode$);
                if ($node \rightarrow slot == -1$) // slot has not been assigned in splitBucket
                    N = descendant of $isramNode$ pointing to a DTCAM bucket and enclosing $node$.
                    $newd$ = AV[N→bIndex];
                    AV[N→bIndex] = next[$newd$];
                    AV[$bucketAV$] = $d$;
                    $d$ = $newd$;
                endif
            endif
            if ($node \rightarrow slot == -1$)
                DLTCAM.$waitWriteValidate$($d$, Q(node), suffix);
                decrementRoom(bucketIndex);
            endif
        else
            assignNewBucket($node$);
            $d$ = AV[$node \rightarrow bIndex$];
            AV[$node \rightarrow bIndex$] = next[$d$];
            DLTCAM.$waitWriteValidate$($d$, Q(node), suffix);
            node→slot = $d$;
            ILTCAM.insert($node$, $NULL$, $NULL$);
        endif
    endif
```

Figure 41. DLTCAM insert algorithm

```
Algorithm: addSuffix($slot$, $cNode$, $node$, $isramNode$, $itcamNode$, $itNode$)
    if (suffix does not fit in DLSRAM[$slot$]) // need another suffix node
        split($cNode$, $isramNode$, $itcamNode$, $itNode$);
        $cNode \rightarrow slot$ = -1;
    else // add suffix to DLSRAM[slot]
        bucketIndex = $isramNode \rightarrow bIndex$;
        $d$ = AV[bucketIndex];
        AV[bucketIndex] = next[$d$];
        DLTCAM.$waitWriteValidate$($d$, DLTCAM[$slot$], DLSRAM[$slot$] + suffix);
        $cNode \rightarrow slot$ = $d$;
    endif
```

Figure 42. Add a suffix to a DLSRAM word

During incremental updates, if a bucket overflows then assigning a new bucket immediately next to the overflowing bucket may require a large number of moves. Hence the suffix node format in IDUOW stores the bucket number for each suffix, which makes it possible to assign any empty bucket in case of an overflow. The suffix node format for the ILSRAM for 1-12Wc is shown in Figure 48. Also, in keeping with the main idea of storing independent prefixes in the LTCAM, the $visit\_postorder$ algorithm is used instead of the subtree split algorithm in [4] while filling out the TCAMs. The prefix assignment algorithm for 1-12Wc is given below.

1. Suffix nodes corresponding to prefixes in the forwarding table are created using the $visit\_postorder$ algorithm on the 1-bit leaf prefix trie as shown in Section 4.
2. Each $Q(N)$ prefix resulting from Step 1 is to be entered into DLTCAM and is marked as a DLTCAM prefix in the trie.
3. The $visit\_postorder$ algorithm is applied to carve the index prefix nodes. The symbols used in the $visit\_postorder$ algorithm have slightly different meaning now: $x \rightarrow size$ represents the number of DLTCAM prefixes enclosed by node x, and w is $b - 1$, where b is the size of a DLTCAM bucket with one free slot for consistent updates. As an index node is carved, the enclosed DLTCAM prefixes are entered in a new DLTCAM bucket, and the bucket index is stored in the trie node, corresponding to the index, in field $bIndex$.
4. Each $Q(N)$, for the index nodes carved in Step 3, is marked as an index prefix in the trie.
5. Suffix nodes are created for the index prefixes using the $visit\_postorder$ algorithm on the 1-bit trie containing the index prefixes. The $Q(N)$ prefixes corresponding to the carved nodes are entered in the ILTCAM. Suffixes for the index prefixes are entered in ILSRAM along with their bucket indexes, in the ILSRAM suffix node format as shown in the Figure 48.

The functions $incrementRoom$ and $decrementRoom$ are not relevant for 1-12Wc and are null functions. The $assignNewBucket$ function is outlined in Figure 49.

The 1-12Wc scheme loses space efficiency as we carve out independent index prefix nodes and use a single bucket to store the DLTCAM prefixes enclosed by a single index prefix. The M-12Wb scheme doesn't have this deficiency as DLTCAM prefixes from index prefixes are stored in the same bucket.

5.3 M-12Wb

The characteristic of the many-1 schemes in [4] is that all DTCAM buckets, except the last one, can be completely filled. Thus multiple index nodes use the same bucket to store their enclosed data TCAM prefixes. The configuration for M-12Wb is shown in Figure 50. The algorithm for carving and prefix assignment follows:

```
Algorithm: split ($cNode$, $isramNode$, $itcamNode$, $itNode$)
    if ($cNode$→y and $cNode$→z) // carve at children y & z of cNode
        bucketIndex = $isramNode$→$bIndex$];
        if (next[AV[bucketIndex]] == -1 || next[next[AV[bucketIndex]]] == -1)
            splitBucket($isramNode$, $itcamNode$, $itNode$); // AV[bucketIndex] should get reset here
            if ($isramNode$ is not the same as cNode) then
                if (cNode→y→slot == -1) then
                    N = descendant of $isramNode$ pointing to a DTCAM bucket and enclosing $node$.
                    cNode→y→slot = AV[N→$bIndex$];
                    cNode→z→slot = next[AV[N→$bIndex$]];
                    AV[N→$bIndex$] = next[next[AV[N→$bIndex$]]];
                    decrementRoom(N→$bIndex$, 2);
                endif
                incrementRoom(bucketIndex, 1);
            else
                if (cNode→y→slot == -1)
                    cNode→y→slot = AV[$isramNode$→$child[0]$→$bIndex$];
                    AV[$isramNode$→$child[0]$→$bIndex$] = next[cNode→y→slot];
                    decrementRoom($isramNode$→child[0]→$bIndex$, 1);
                endif
                if (cNode→z→slot == -1)
                    cNode→z→slot = AV[$isramNode$→$child[1]$→$bIndex$];
                    AV[$isramNode$→child[1]→$bIndex$] = next[cNode→z→slot];
                    decrementRoom($isramNode$→child[1]→$bIndex$, 1);
                endif
                incrementRoom(bucketIndex, 1);
            endif
        else
            cNode→y→slot = AV[bucketIndex];
            cNode→z→slot = next[AV[bucketIndex]];
            AV[bucketIndex] = next[next[AV[bucketIndex]]];
            decrementRoom(bucketIndex, 1);
        endif
        DLTCAM.$waitWriteValidate$(cNode→y→slot, Q(cNode→y), suffixes(cNode→y));
        DLTCAM.$waitWriteValidate$(cNode→z→slot, Q(cNode→z), suffixes(cNode→z));
    else if (cNode→y) split (cNode→y);
    else split(cNode→z);
    endif
```

Figure 43. Split a DLSRAM word

```
Algorithm: delete($node$, $cNode$, $tNode$, $isramNode$, $itcamNode$, $itNode$)
// $node$: node in leaf trie corresponding to the prefix to be deleted
// $cNode$: nearest carved ancestor of $node$, cannot be NULL
// $tNode$: nearest degree 2 ancestor node of cNode.
// $isramNode$: index node enclosing $cNode$
// $itcamNode$: node whose prefix exists in ILTCAM.
// $itNode$: nearest degree 2 ancestor of $itcamNode$.
    bucketIndex = $isramNode \rightarrow bIndex$;
    oldSlot = $cNode \rightarrow$slot;
    p = number of suffixes in DLSRAM[oldSlot];
    if (p $>$ 1 and !carve($tNode$, $cNode$, bucketIndex)) // delete suffix from its suffix node
        d = AV[bucketIndex];
        AV[bucketIndex] = next[d];
        DLSRAM[d] = DLSRAM[oldSlot] - suffix;
        DLTCAM[d].prefix = DLTCAM[oldSlot].prefix;
        DLTCAM[d].valid = 1;
        cNode$\rightarrow$slot = d;
    else
        cNode$\rightarrow$slot = -1;
        if ($isramNode \rightarrow$size == 0) then // bucket becomes empty
            deleteBucket($isramNode$, $itcamNode$, $itNode$);
        endif
        decrementRoom(bucketIndex);
    endif
    DLTCAM.$invalidateWaitWrite$($oldSlot$, AV[bucketIndex]);
    AV[bucketIndex] = oldSlot;
Algorithm: carve($tNode$, $cNode$, bucketIndex)
    if (!$tNode$) return 0;
    if (suffixes($tNode$) fit in a suffix node) // carve at tNode
        $d$ = AV;
        AV = next[$d$];
        DLSRAM[$d$] = suffixes(tNode);
        DLTCAM[$d$].prefix = Q(tNode);
        DLTCAM[$d$].valid = 1;
        tNode$\rightarrow$slot = d;
        otherNode = the carvedNode in subtree rooted at tNode that is not cNode;
        DLTCAM.$invalidateWaitWrite$(otherNode$\rightarrow$slot, AV[bucketIndex]);
        AV[bucketIndex] = otherNode$\rightarrow$slot;
        otherNode$\rightarrow$slot = -1;
        return 1;
    endif
    return 0;
```

Figure 44. Delete a leaf prefix

```
Algorithm: change(prefix, cNode, $nexthop$, isramNode)
    oldSlot = cNode→slot;
    bucketIndex = $isramNode \rightarrow bIndex$;
    $d$ = AV[bucketIndex];
    AV[bucketIndex] = next[$d$];
    newWord = DLSRAM[oldSlot] with next hop for $cNode \rightarrow prefix$ set to $nexthop$;
    DLTCAM.$waitWriteValidate$($d$, prefix, newWord);
    cNode→slot = $d$;
    DLTCAM.$invalidateWaitWrite$(oldSlot, AV[bucketIndex]);
    AV[bucketIndex] = oldSlot;
Algorithm: deleteBucket($isramNode$, $itcamNode$, $itNode$)
    bucketIndex = $isramNode \rightarrow bIndex$;
    nextBucket[bucketIndex] = bucketAV;
    bucketAV = bucketIndex;
    $isramNode \rightarrow bIndex$ = -1;
    ILTCAM.delete($isramNode$, $itcamNode$, $itNode$);
Algorithm: splitBucket ($isramNode$, $itcamNode$, $itNode$)
    if ($isramNode \rightarrow$y and $isramNode \rightarrow$z) // carve at children y & z of isramNode
        // We want to move the split child that contains fewer prefixes.
        if ($isramNode \rightarrow y$ contains fewer prefixes)
            $isramNode \rightarrow y \rightarrow bIndex$ = $isramNode \rightarrow bIndex$;
            assignNewBucket($isramNode \rightarrow z$);
            $node$ = $isramNode \rightarrow z$;
        else
            assignNewBucket($isramNode \rightarrow y$);
            $isramNode \rightarrow z \rightarrow bIndex$ = $isramNode \rightarrow bIndex$;
            $node$ = $isramNode \rightarrow y$;
        endif
        ILTCAM.insert($isramNode \rightarrow y$, $itcamNode$, $itNode$);
        ILTCAM.insert($isramNode \rightarrow z$, $itcamNode$, $itNode$);
        ILTCAM.delete($isramNode$, $itcamNode$, $itNode$);
        deletePrefixes($node$);
    else if ($isramNode \rightarrow$y) splitBucket($isramNode \rightarrow$y, $itcamNode$, $itNode$);
    else splitBucket($isramNode \rightarrow$z, $itcamNode$, $itNode$);
    endif
```

Figure 45. Change the next hop of a leaf prefix

```
Algorithm: deletePrefixes($node$)
    bucketIndex = $node \rightarrow bIndex$;
    len = length of deleteList;
    for (i=0; i<len; ++i)
        slot = deleteList[i];
        DLTCAM.$invalidateWaitWrite$(slot, AV[bucketIndex]);
        AV[bucketIndex] = slot;
        incrementRoom(bucketIndex);
    endfor
    clear deleteList;
```

Figure 46. Delete prefixes

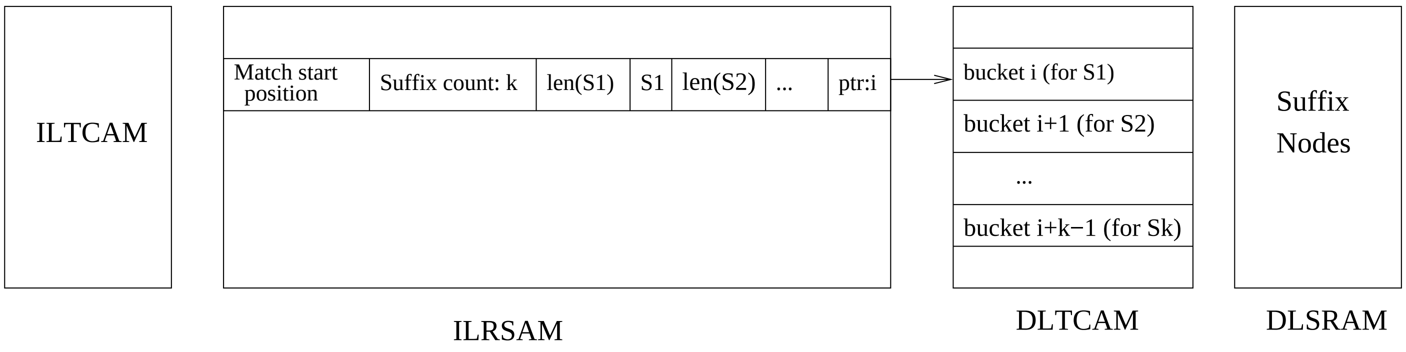


Figure 47. 1-12Wc configuration in [4]

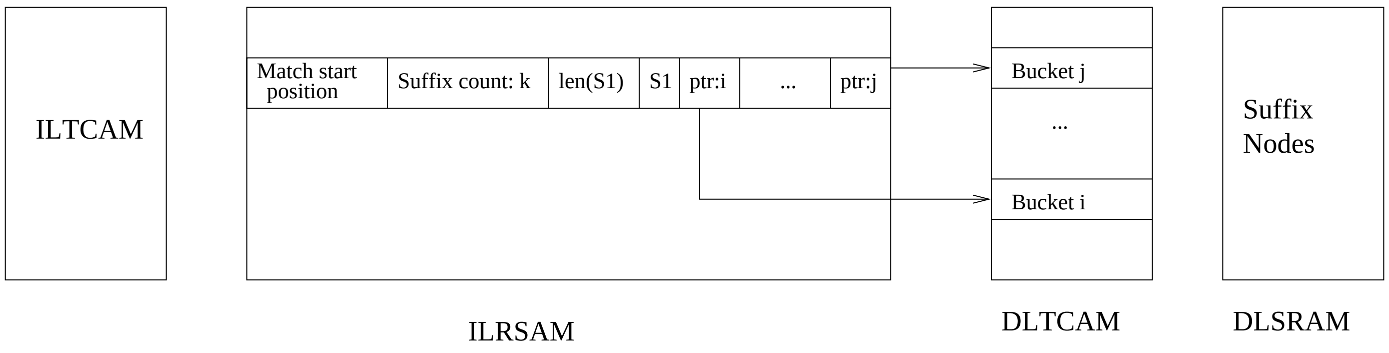


Figure 48. Our 1-12Wc configuration

1. Step 1: [Seed the DLTCAM buckets]
   Run $feasibleST2(T, b-1)\lceil n/(b-1)\rceil$ times. // $b-1$, since one free slot is needed in a bucket for consistent updates.
   Each time call $splitNode$ to carve the found $bestST$ from T (thereby updating T) and pack $bestST$ into a new DLTCAM bucket.
   The function $splitNode$ adds one or more prefixes to the ILTCAM.

2. Step 2: [Fill the buckets]
   While there is a DLTCAM bucket that is not full and T is not empty, repeat Step 3.

```
Algorithm: assignNewBucket($node$)
    $node \rightarrow bIndex$ = bucketAV;
    if (bucketAV == -1) throw NoBucketsException;
    bucketAV = nextBucket[bucketAV];
```

Figure 49. Assign a new bucket in 1-12Wc

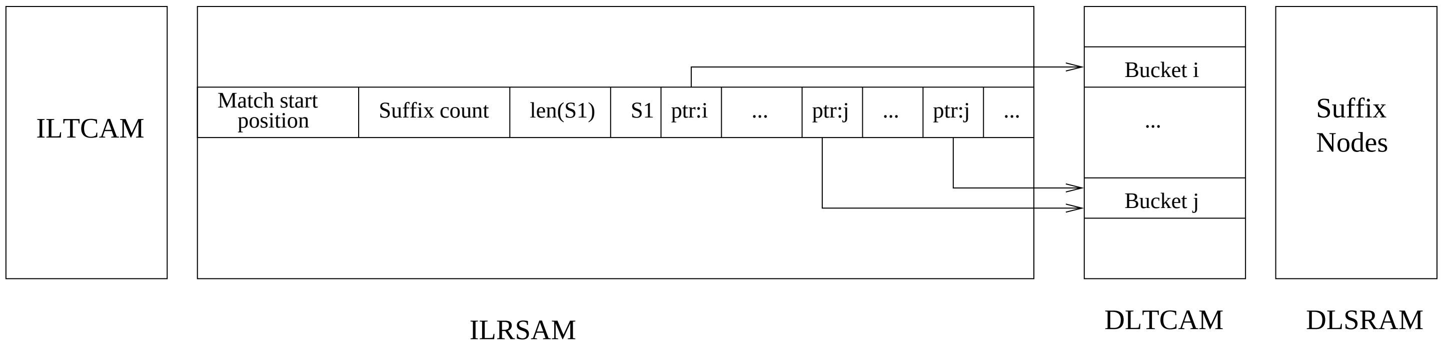


Figure 50. M-12Wb configuration in [4]

3. Step 3: [Add to a bucket]
   Let B be the DLTCAM bucket with the fewest number of prefixes.
   Let s be the number of prefixes in B.
   Run $feasibleST2(T, b - s)$.
   Using $splitNode$ carve the found $bestST$ from T (thereby updating T) and pack $bestST$ into B.
   The function $splitNode$ adds one or more prefixes to the ILTCAM.

4. Step 4: [Use additional buckets as needed]
   While T is not empty, fill a new DLTCAM bucket by making repeated invocations of $feasibleST2(T, q)$, where q is the remaining capacity of the bucket.
   Add ILTCAM prefixes as needed.

There are three main differences between this algorithm and the PS2 algorithm in [4]. The first difference is reflected in the $visit2$ algorithm (invoked by $feasibleST2$) in that covering prefixes are not stored in the TCAMs. The second difference is in supplying $b - 1$ as available space in an empty bucket of size b, reserving one free slot for consistent updates. The third difference is in the use of carving function $splitNode$ which helps to create independent prefixes for IDUOW.

Apart from the data structures already defined for the two-level indexing schemes, the M-12Wb requires a doubly linked list of used buckets to keep track of the buckets and the available spaces in them. An instance of a class BList is maintained in the control plane which contains the doubly linked list of buckets as well as an array to get to the right bucket quickly using a bucket index. Each bucket in the list has fields $room$ to indicate available bucket slots and $index$ to indicate the index of the bucket. The room in a bucket decreases from $head$ to $tail$ of the list. BList uses function add to add a new bucket to the list and the array and $getBucket$ to get the appropriate bucket based on bucket index.

6 Experimental Results

We evaluated the performance of the different versions of DUO using 21 IPv4 routing tables and update sequences downloaded from [6] and [7]. Figure 55 gives the characteristics of these datasets. The update sequences for the first 20

Algorithm: visit2(x)

```
    d = count(x); // returns the number of DLTCAM prefixes.
    if (d $\leq$ q and d $>$ bestCount)
        bestST = ST(x);
        bestCount = d;
    endif
    // check T - ST(x)
    d = count(root(T)) - count(x);
    if (d $\leq$ q and d $>$ bestCount)
        bestST = T - ST(x);
        bestCount = d;
    endif
```

Figure 51. Visit algorithm

Algorithm: splitNode(N, NoN)

NoN is trie node x if bestST = T-ST(x), otherwise NoN is passed as NULL.

```
    if (!$N$ || $N == NoN$) return;
    if ($N$→istouched == 0)
        $N$→istouched = 1;
        $N$→bIndex = BList.$head$→index;
        fill bucket with DLTCAM prefixes in $N$.
        Let $s$ = number of DLTCAM prefixes in $N$.
        BList.$head$→room = BList.$head \rightarrow room$ - $s$.
    endif
    splitNode($N \rightarrow y$);
    splitNode($N \rightarrow z$);
```

Figure 52. Split a node

Algorithm: assignNewBucket($node$)

```
    Let $s$ = number of DLTCAM prefixes in $node$.
    if ($s < BList.head \rightarrow room$)
        $node \rightarrow bIndex$ = BList.$head \rightarrow index$;
        BList.$head$→room -= $s$;
    else if (bucketAV $>$ -1)
        $node \rightarrow bIndex$ = bucketAV;
        BList.add(bucketSize, bucketAV);
        BList.$head$→room -= $s$;
        bucketAV = nextBucket[bucketAV];
    else
        if (BList.$head$→room == 1) throw NoSpaceException;
        Run step 3 in PS2 while $node$ still has a prefix;
    endif
```

Figure 53. Assign a new bucket

```
**Algorithm: incrementRoom(bucketIndex)**
    $b$ = BList.getBucket(bucketIndex);
    $b \rightarrow room$++;
    if $b \rightarrow prev$ and $b \rightarrow room > b \rightarrow prev \rightarrow room$ then relocate $b$ to restore order.

**Algorithm: decrementRoom(bucketIndex)**
    $b$ = getBucket(bucketIndex);
    $b \rightarrow room$–;
    if $b \rightarrow next$ and $b \rightarrow room < b \rightarrow next \rightarrow room$ then relocate $b$ to restore order.
```

Figure 54. Increment and decrement room

routing tables were captured from files storing update announcements from 12am on February 1, 2009 for the stated number of hours; the update sequence for the last routing table rrc00May20 was captured from files storing eight hours of activity starting from 12am on May 20, 2008. The columns labeled $\#RawInserts$, $\#RawDeletes$ and $\#RawChanges$,

DataSet	#Prefixes	Collection Period (hours)	$\#RawInserts$	$\#RawDeletes$	$\#RawChanges$
rrc00	294098	75.7	39553	40051	368013
rrc01	276795	75.2	41692	41988	492315
rrc03	283754	42.7	27702	27914	292454
rrc04	288610	17	16086	15977	193392
rrc05	280041	103	20276	18285	439647
rrc06	278744	235	157549	157547	289272
rrc07	275097	0.417	247	218	179835
rrc10	278898	105	21620	22473	326720
rrc11	277166	80.2	58115	58378	290621
rrc12	278499	62.3	33196	33572	410464
rrc13	284986	57.8	23920	23713	284710
rrc14	276170	83.6	56598	56810	203955
rrc15	284047	134	95790	93750	183131
rrc16	282660	672	3338	937	8896
route-views2	294127	56.5	13882	15552	679100
route-views4	275737	95	69627	69754	526302
route-views.eqix	275736	70.3	51104	51066	253693
route-views.isc	281095	68.2	44286	44444	292323
route-views.linx	278196	49.1	23137	23413	384344
route-views.wide	283569	174	101821	103862	372035
rrc00May20	266185	8	5392	5322	45542

Figure 55. Datasets used in the experiments

respectively, give the number of insert, delete, and change next hop requests in the update sequences. Using consistent updates, a next hop change request is implemented (see Figure 16 for example) as an insert (of the prefix with the new next hop) followed by a delete (of the prefix with the old nexthop). Therefore, all results henceforth are in terms of the effective inserts and deletes. Note that the number of effective inserts (#Inserts) and deletes (#Deletes) is given by the following equations.

$$\#Inserts = \#RawInserts + \#RawChanges; \quad (1)$$

$$\#Deletes = \#RawDeletes + \#RawChanges; \quad (2)$$

6.1 Evaluation of Memory Management Schemes

We first ran a set of experiments on the simple TCAM [4] to compare our memory management schemes–Schemes 1-4. The simple TCAM we instantiated for the experiments has 300,000 slots. Figures 56 and 57, respectively, give the total and average number of prefix moves (i.e., number of invocations of $move()$) required for an insert (includes raw inserts change next hop inserts) and a delete in our test update sequences (the data in Figure 57 is obtained from that in Figure 56 by dividing by #Inserts or #Deletes). Note that the theoretical worst-case number of moves for an insert/delete in IPv4 for the four memory management schemes is, respectively, 16, 32, 32 and 16. Figures **??** and 59, respectively, give the maximum number of moves per insert/delete and the standard deviation. From our experiments, we make the following observations:

1. Scheme 1 (PLO_OPT) required the maximum number of moves (sum of moves for inserts and deletes) for all our test sets and Scheme 3 required the least. In fact, the disparity among the 4 schemes is very significant with Scheme 3 requiring a total number of moves that is orders of magnitude less than that required by the remaining schemes. Schemes 2 is comparable to Scheme 4 and Scheme 1 requires 10 times (or more) as many moves as required by Schemes 2 and 4.

2. The number of moves due to inserts in Scheme 2 is lower than those in Scheme 4 (CAO_OPT) by orders of magnitude. For some of our test sets, inserts required no moves when Scheme 2 was used.

3. The number of moves due to deletes in Scheme 2 is comparable to that in Scheme 4 (CAO_OPT).

4. The number of moves due to inserts in Scheme 3 is lower than that in Scheme 4 (CAO_OPT) by orders of magnitude. For the inserts in some of our test sets, Scheme 3 required no moves at all.

5. The number of moves due to deletes is 0 in Scheme 3 because in this scheme the slot within a block, freed by a delete is simply appended to the free space list for the block.

6. The maximum number of moves per insert/delete about the same for Schemes 2, 3 and 4, and about half that for Scheme 1. We note that Scheme 4 has a better worst-case performance for inserts than Schemes 2 and 3 but is worse for deletes.

7. The standard deviation is very small for Schemes 2 and 3. The number of moves, needed for an insert operation using Scheme 3, has low average and standard deviation values. So, the number of TCAM moves for any insert operation is, with a good probability, very low as well when Scheme 3 is used. The number of moves, needed for a delete operation using Scheme 3, has zero average and standard deviation values since Scheme 3 does not involve any move for a delete operation.

We also note that for Schemes 2 and 4, the number of moves due to deletes is much more than that due to inserts. For Scheme 4 this is because a delete rarely occurs adjacent to either of the two boundaries of the free space pool and non-boundary deletes require at least one move to shift the empty slot to the free space pool. However, since the prefix trie is shallow and the free space pool cuts each root to leaf path in the middle, many of the inserts in an update sequence are expected to occur at a boundary of the free space pool. So, inserts take much less than 1 move, on average, when Scheme 4 is used. Similarly, when Scheme 2 is used, most deletes are from within a block rather than at a block boundary. These non-boundary deletes require 1 move each. However, an insert requires no moves if there is a free slot at the top or bottom of its block, a likely occurrence.

Figure 60 shows the number of $waitWrites$ (sum of invocations of $waitWriteValidate()$ and $invalidateWaitWrite()$), which is the equal to the sum of inserts, deletes and moves for the simple TCAM and reflects the update performance for the four memory management schemes. As expected, Scheme 3 requires the least number of operations, due to the small number of moves. For Scheme 3, the average number of $waitWrites$ per insert and delete (number of $waitWrites$/(#Inserts + #Deletes)) ranged from a low of 1 for rrc01, rrc07, rrc16, route-views.linx to a high of 1.0072 for rrc03. Figure 61(a) shows the normalized average number of moves for each scheme on a logarithmic scale. For this figure, we computed the average number of moves per Insert/Delete for each data set. Then the average of these averages was computed and normalized by the average of averages for Scheme 3. Figure 61(b) shows the normalized average $waitWrite$s invoked by the different schemes. For this figure, we computed the average number of $waitWrite$s per Insert/Delete for each data set, then computed the average of these averages for each memory management scheme and finally normalized by the average of the averages for Scheme 3.

Dataset	Scheme 1		Scheme 2		Scheme 3		Scheme 4	
	insert	delete	insert	delete	insert	delete	insert	delete
rrc00	2527899	2938315	395	404839	401	0	28621	405733
rrc01	3106800	3641827	0	531397	0	0	57619	529312
rrc03	1933994	2254451	4622	317445	4630	0	35144	321432
rrc04	1260636	1467502	0	208142	2	0	61507	221368
rrc05	2765874	3210264	543	455214	541	0	48677	461485
rrc06	2785323	3228450	8	435997	8	0	11452	439501
rrc07	973836	1153713	0	179987	0	0	35256	206623
rrc10	2090263	2444704	658	347529	671	0	30037	355326
rrc11	2100218	2449182	266	343898	245	0	17726	342096
rrc12	2657748	3101916	4665	438759	4659	0	44243	448979
rrc13	1784545	2090102	1035	304541	989	0	53433	560835
rrc14	1517650	1778785	4	255964	4	0	18265	255381
rrc15	1682880	1940303	2986	266885	2769	0	22314	286126
rrc16	71864	67329	0	9777	0	0	580	11075
route-views2	4140653	4844342	14	691240	14	0	92948	697924
route-views4	3584127	4177510	141	590235	141	0	39481	586756
route-views.eqix	1813054	2115841	33	300259	33	0	14235	301296
route-views.isc	2003320	2338493	12	331570	12	0	13537	326232
route-views.linx	2440442	2848138	0	404276	0	0	39136	403168
route-views.wide	2918481	3402684	1	462801	1	0	18559	466695
rrc00May20	311588	361323	19	50512	22	0	6380	50946

Figure 56. Number of moves for the simple TCAM

Effect of TCAM Size on Memory Management Schemes

The number of moves required by an update sequence is independent of the size of the TCAM (provided there are enough slots to accommodate all prefixes) when Schemes 1 and 4 are used. This, however, is not the case for Schemes 2 and 3. Because of the relatively poor performance of Scheme 2 in our earlier test (Figure 56), we did not study the impact of TCAM size on the number of moves using this scheme. Figure 62 gives the number of moves required by the inserts (effective) in each of our test update sequences for varying TCAM size. The column labeled #Prefixes gives the initial number of prefixes in the routing table while that labeled #MaxPrefixes gives the maximum size attained by the routing table during the course of the update sequence. The TCAM occupancy is defined to be #MaxPrefixes/(TCAM size)*100%. For our experiment, we selected TCAM size so as to have occupancies of 80%, 90%, 95%, 97%, and 99%. As can be seen, even with an occupancy of 99%, our Scheme 3 does very well. In fact, its nearest competitor, Scheme 4 (CAO_OPT), requires between 72 and 241879 times as many moves (for inserts and deletes combined) as required by Scheme 3 (see Figure 56 for the number of moves required by Scheme 4).

6.2 Evaluation of DUOS

In DUOS, each prefix in the forwarding table occupies a slot in either the ITCAM or the LTCAM. Columns 2 and 5 of Figure 63 give the initial prefix distribution between the 2 TCAMs of DUOS. Columns 3 and 6 give the distribution of

Dataset	Scheme 1		Scheme 2		Scheme 3		Scheme 4	
	insert	delete	insert	delete	insert	delete	insert	delete
rrc00	6.20	7.20	0.000969	0.9921	0.000984	0	0.0702	0.994
rrc01	5.8	6.8	0	0.9945	0	0	0.1078	0.9906
rrc03	6.04	7.04	0.0144	0.9909	0.0145	0	0.1098	1.0033
rrc04	6.01	7.01	0	0.9941	0.00001	0	0.293	1.0573
rrc05	6.01	7.01	0.00118	0.994	0.00118	0	0.1058	1.0078
rrc06	6.23	7.23	0.000018	0.976	0.00002	0	0.0256	0.9836
rrc07	5.41	6.41	0	0.9996	0	0	0.1958	1.1476
rrc10	6.00	7.00	0.00189	0.9952	0.00193	0	0.0862	1.0176
rrc11	6.02	7.01	0.000762	0.9854	0.0007	0	0.0508	0.9802
rrc12	5.99	6.99	0.0105	0.9881	0.0105	0	0.0997	1.0111
rrc13	5.78	6.77	0.00335	0.9874	0.0032	0	0.1731	1.0351
rrc14	5.82	6.82	0.000015	0.9816	0.000015	0	0.0701	0.979
rrc15	6.03	7.01	0.0107	0.963	0.0099	0	0.08	1.0333
rrc16	5.87	6.84	0	0.9943	0	0	0.0474	1.1263
route-views2	5.97	6.97	0.00002	0.9951	0.00002	0	0.1341	1.0047
route-views4	6.01	7.01	0.000236	0.99	0.000236	0	0.0663	0.9843
route-views.eqix	5.94	6.94	0.000108	0.98523	0.000108	0	0.0467	0.9886
route-views.isc	5.95	6.94	0.000036	0.9846	0.000036	0	0.0402	0.969
route-views.linx	5.98	6.98	0	0.9914	0	0	0.096	0.9887
route-views.wide	6.15	7.15	0.000002	0.9725	0.000002	0	0.0392	0.9807
rrc00May20	6.12	7.10	0.00037	0.99308	0.000431	0	0.1253	1.0016

Figure 57. Average number of moves for the simple TCAM

the inserts (i.e., number of non-leaf inserts and number of leaf inserts) while columns 4 and 7 give the distribution of the deletes. We note that a leaf insert/delete may trigger additional insert and/or delete operations on the TCAMS of DUOS. These additional inserts/deletes are accounted for in Figure 63. As a result,

$$ITCAM.\#inserts + LTCAM.\#inserts \geq \#Inserts \qquad (3)$$

It is interesting to note that more than 90% of the prefixes in each data set are leaf prefixes and that more than 90% of the inserts and deletes in each update sequence are directed at the LTCAM.

Given the distribution of the prefixes and insert and delete operations, we instantiated an LTCAM with 300,000 slots and an ITCAM with 28,000 slots for our DUOS experiments. Since the performance of DUOS is determined by the number of $waitWrite$ operations, we measure this quantity for our datasets. In addition, since the number of moves directly impacts the number of $waitWrite$ operations, we measured the number of moves separately so to compare the effect of the four memory management schemes for ITCAM. Figure 64 gives the number of ITCAM moves for inserts and deletes. The number of moves shown in Figure 64 includes the ITCAM moves resulting from ITCAM operations triggered by LTCAM inserts and deletes as well (for example, when inserting a leaf prefix, we insert into the LTCAM and delete its parent prefix (if any) from the LTCAM and reinsert this parent prefix into the ITCAM). The relative performance of the 4 memory management schemes for ITCAM is quite similar to that observed for a simple TCAM organization and Scheme 3 outperforms the remaining schemes handily. Figure 65 shows the number of $waitWrites$ generated in the ITCAM and we find that Scheme 3 is the best for this metric as expected from the smaller number of moves required by Scheme 3. Figure 66 gives the number of LTCAM moves required by the test update sequences. As expected, the number of LTCAM moves is zero[2]. The total number of moves for the simple TCAM is between 14-24 times that for DUOS using Scheme 1 (PLO_OPT), between 9-15 times using Scheme 2, 7-227 times using Scheme 3, and 9-16 times using Scheme 4

[2]Recall that, in an LTCAM, an insert may be done in any free slot and a slot freed by a delete is simply linked to the free space list.

Dataset	Scheme 1		Scheme 2		Scheme 3		Scheme 4	
	insert	delete	insert	delete	insert	delete	insert	delete
rrc00	15	16	8	1	8	0	4	5
rrc01	15	16	0	1	0	0	3	5
rrc03	14	15	7	1	7	0	3	5
rrc04	14	15	0	1	1	0	5	5
rrc05	14	15	7	1	7	0	4	5
rrc06	11	12	1	1	1	0	4	5
rrc07	11	12	0	1	0	0	4	5
rrc10	15	16	3	1	3	0	3	6
rrc11	14	15	5	1	5	0	3	5
rrc12	15	16	8	1	8	0	3	5
rrc13	15	16	8	1	8	0	4	5
rrc14	15	16	2	1	2	0	3	5
rrc15	14	15	7	1	7	0	3	5
rrc16	14	13	0	1	0	0	2	5
route-views2	15	16	2	1	2	0	3	5
route-views4	13	14	6	1	6	0	2	6
route-views.eqix	14	15	2	1	2	0	3	5
route-views.isc	15	16	1	1	1	0	2	6
route-views.linx	15	16	0	1	1	0	2	6
route-views.wide	14	14	1	1	1	0	3	5
rrc00May20	15	16	4	1	4	0	3	6

Figure 58. Maximum number of moves for the simple TCAM

(CAO_OPT). Thus there is a reduction of more than 90% in the total number of moves for any scheme. This is due to the DUO architecture, as the reduction is observed for PLO_OPT and CAO_OPT also. Note that the number of $waitWrites$ in an LTCAM equals the number of inserts and deletes on the LTCAM and $waitWriteValidate$s in an LTCAM, have null wait as no invalid slot is involved in an ongoing lookup. This is ensured by using $invalidateWaitWrite$ to free a slot. Note that $invalidateWaitWrite$ waits till an ongoing lookup is complete and then invalidates the slot. Since updates are done serially in the control plane, $invalidateWaitWrite$s from an LTCAM delete must complete before the next update operation begins.

6.3 Evaluation of DUOW

In evaluating DUOW, we used a wide SRAM in conjunction with the LTCAM only as the ITCAM has relatively few (about 10%) prefixes. We instantiated an LTCAM with 100,000 slots and used the same configuration for the ITCAM as used in our evaluation of DUOS. For the DUOW evaluation, we used only Scheme 3 for memory management in the ITCAM. Figure 67 gives the number of LTCAM prefixes carved by Lu's carving heuristic [4] and our carving heuristic of Section 4. The carving by both methods is done only on the trie of leaf prefixes as only leaf prefixes are stored in the LTCAM and its associated wide SRAM. Surprisingly, the number of prefixes that result when our method is used is fewer than when the method of [4] is used. This is surprising because our method carves out independent prefixes while the method of [4] may carve any set of prefixes. The approximately 1% drop in the number of prefixes when our carving method is used results from the observation that when our method is used we do not need to supplement the carving prefixes with covering prefixes while covering prefixes need to be added to the set of carving prefixes generated by the method of [4]. Since covering prefixes account for approximately 8% of the prefixes generated by the method of [4], a 1% drop in the total number of prefixes when our method is used implies a roughly 7% increase in carving prefixes before accounting for covering prefixes.

Dataset	Scheme 1		Scheme 2		Scheme 3		Scheme 4	
	insert	delete	insert	delete	insert	delete	insert	delete
rrc00	1.582	1.578	0.064	0.088	0.064	0	0.276	0.196
rrc01	1.957	1.951	0	0.0735	0	0	0.314	0.212
rrc03	1.750	1.747	0.311	0.095	0.311	0	0.337	0.212
rrc04	2.434	2.421	0	0.076	0.003	0	0.554	0.276
rrc05	1.675	1.669	0.077	0.077	0.077	0	0.333	0.170
rrc06	1.457	1.453	0.004	0.154	0.004	0	0.163	0.274
rrc07	2.168	2.168	0	0.019	0	0	0.412	0.407
rrc10	1.888	1.882	0.067	0.069	0.069	0	0.288	0.246
rrc11	1.699	1.697	0.04	0.12	0.0377	0	0.223	0.262
rrc12	1.896	1.89	0.27	0.1083	0.27	0	0.307	0.281
rrc13	2.144	2.1365	0.117	0.111	0.114	0	0.449	0.245
rrc14	1.834	1.833	0.005	0.134	0.005	0	0.286	0.287
rrc15	2.066	2.0428	0.216	0.186	0.212	0	0.279	0.310
rrc16	1.7722	1.8366	0	0.9943	0	0	0.215	0.406
route-views2	1.746	1.744	0.005	0.07	0.005	0	0.371	0.14
route-views4	1.756	1.754	0.034	0.098	0.034	0	0.251	0.218
route-views.eqix	1.722	1.720	0.0107	0.1206	0.0107	0	0.213	0.243
route-views.isc	1.691	1.686	0.006	0.1232	0.006	0	0.1992	0.2627
route-views.linx	1.933	1.928	0	0.092	0	0	0.306	0.212
route-views.wide	1.502	1.5	0.0015	0.164	0.0015	0	0.198	0.266
rrc00May20	1.7479	1.7286	0.0284	0.0829	0.0313	0	0.3417	0.228

Figure 59. Standard deviation in number of moves for the simple TCAM

Dataset	Scheme 1	Scheme 2	Scheme 3	Scheme 4
rrc00	6281844	1220864	816031	1249984
rrc01	7816937	1599707	1068310	1655241
rrc03	4828969	962591	645154	997100
rrc04	3146985	626989	418849	701722
rrc05	6893993	1373612	918396	1428017
rrc06	6907413	1329645	893648	1344593
rrc07	2487684	540122	360135	602014
rrc10	5232500	1045720	698204	1082896
rrc11	5247135	1041899	697980	1057557
rrc12	6647360	1331120	892355	1380918
rrc13	4491700	922629	618042	989752
rrc14	3817753	777286	521322	794964
rrc15	4178985	825673	558571	864242
rrc16	161260	31844	22067	33722
route-views2	10372629	2078888	1387648	2178506
route-views4	8953622	1782361	1192126	1818222
route-views.eqix	4538451	909848	609589	925087
route-views.isc	5015189	1004958	673388	1013145
route-views.linx	6103818	1219514	815238	1257542
route-views.wide	7270918	1412555	949754	1435007
rrc00May20	774709	152329	101820	159124

Figure 60. Number of $waitWrites$ for the simple TCAM

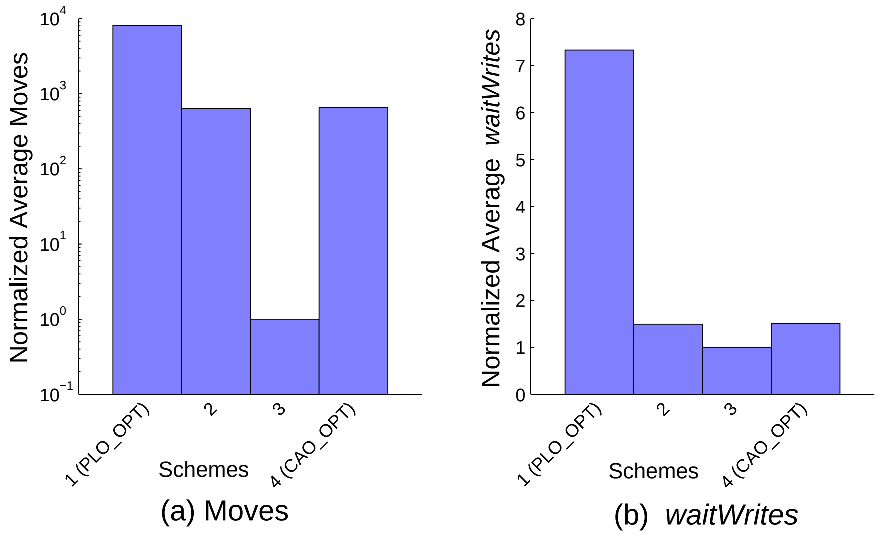


Figure 61. Comparison of performance between the different memory management schemes

Dataset	#Prefixes	#MaxPrefixes	Occupancy				
			80%	90%	95%	97%	99%
rrc00	294098	294318	0	0	96	227	554
rrc01	276795	277002	0	0	0	31	307
rrc03	283754	284464	3412	4311	4641	4774	4916
rrc04	288610	288915	0	0	2	5	1144
rrc05	280041	282223	180	383	584	696	810
rrc06	278744	279202	0	5	11	17	97
rrc07	275097	275130	0	0	0	0	1
rrc10	278898	280158	381	490	798	1044	1355
rrc11	277166	277391	39	168	362	514	668
rrc12	278499	279155	2588	4174	4966	5212	5448
rrc13	284986	285621	120	592	973	1107	1272
rrc14	276170	276385	0	2	14	51	146
rrc15	284047	286467	1652	2392	2736	2838	2982
rrc16	282660	285170	0	0	0	0	0
rviews2	294127	294598	0	0	0	4	28
rviews4	275737	276035	12	106	178	201	222
rviews.eqix	275736	276230	12	29	97	166	269
rviews.isc	281095	281430	0	5	14	20	131
rviews.linx	278196	278283	0	0	0	34	348
rviews.wide	283569	284569	0	1	1	1	19
rrc00May20	266185	267344	13	32	101	173	430

Figure 62. Number of Scheme 3 moves for inserts

Dataset	ITCAM			LTCAM		
	#Prefixes	#inserts	#deletes	#Prefixes	#inserts	#deletes
rrc00	27381	27454	27483	266717	388106	388575
rrc01	24787	39796	39789	252008	501753	502056
rrc03	26116	30461	30464	257638	296757	296966
rrc04	25137	20549	20535	263473	192299	192204
rrc05	25375	36852	36518	254666	426613	424956
rrc06	25207	36518	36453	253537	438945	439008
rrc07	24441	15946	15944	250656	164143	164116
rrc10	24832	26364	26520	254066	324914	325611
rrc11	24787	29399	29382	252379	328358	328638
rrc12	24894	35725	35713	253605	418259	418647
rrc13	26320	33025	32985	258666	282274	282107
rrc14	24485	27455	27428	251685	240669	240908
rrc15	26184	26356	25932	257863	267119	265503
rrc16	25586	1368	837	257074	11292	9422
route-views2	26883	64285	64540	267244	633843	635258
route-views4	24543	49773	49750	251194	562837	562987
route-views.eqix	24423	28188	28137	251313	284179	284192
route-views.isc	25459	36595	36565	255636	311072	311260
route-views.linx	25072	36247	36280	253124	376990	377233
route-views.wide	26410	37266	37567	257159	456074	457814
rrc00May20	24407	3738	3722	241778	47696	47642

Figure 63. Distribution of prefixes, inserts, and deletes for DUOS

Figure 68 gives the number of inserts and deletes applied on the LTCAM of DUOW as well as the number $waitWrites$. We observe that the number of $waitWrites$ for the LTCAM of DUOW is more than the number of inserts and deletes done in the LTCAM. This is in contrast to DUOS where the number of $waitWrites$ is the same as the number of inserts and deletes. This is because additional writes are needed in DUOW to maintain lookup consistency when the contents of an SRAM word are split or merged or when a suffix is added to or deleted from an existing SRAM word.

We note that the number of ITCAM inserts and deletes as well as the number of ITCAM $waitWrites$ are unaffected by the coupling of a wide SRAM to the LTCAM. So, the numbers shown in Figure 64 are valid for the DUOW ITCAM as well as for the DUOS ITCAM.

6.4 Evaluation of IDUOW

As was the case for our DUOW evaluation, for IDUOW too, we used a wide SRAM only in conjunction with the LTCAM. Further, an index TCAM (ILTCAM) with an associated wide SRAM was added only to the LTCAM. Our instantiated DLTCAM and ILTCAM had 200,000 and 20,000 slots, respectively. The DLTCAM bucket size was set to 512 slots for both schemes discussed in Section 5. Figures 69 and 70 give the number of inserts and deletes as well as the number of $waitWrites$ for the ILTCAM and DLTCAM using 1-12Wc while Figures 71 and 72 give these numbers for the M-12Wb indexing scheme. As can be seen, the 1-12Wc scheme required between 203 to 227 buckets, thereby using up between 103936 and 116224 DLTCAM slots. The number of moves resulting from bucket splits varied from 0 to 1085. The M-12Wb scheme is more space efficient requiring between 128 and 153 buckets, thereby using up between 65536 and 78336 DLTCAM slots. However, the number of moves is between 800 and 16603 when M-12Wb is used. (We shall see later that the worst-case number of moves for these two schemes is comparable). Just as in DUOW, the number of $waitWrites$ is more than the number of inserts and deletes and for DLTCAM there is an additional source for writes–prefix moves resulting from bucket overflows.

Dataset	Scheme 1		Scheme 2		Scheme 3		Scheme 4	
	insert	delete	insert	delete	insert	delete	insert	delete
rrc00	102529	129102	11	26064	12	0	1185	27826
rrc01	125384	164002	1	38410	1	0	2016	41086
rrc03	115788	144517	604	27691	614	0	1470	30230
rrc04	57004	76867	0	19686	0	0	1598	21880
rrc05	119429	153774	8	35401	8	0	1930	37533
rrc06	124810	156191	0	29826	0	0	3065	37407
rrc07	42336	58283	0	15919	0	0	1063	17123
rrc10	85401	111943	3	25892	3	0	1737	27999
rrc11	104737	131628	9	26124	9	0	1033	27615
rrc12	126858	159951	605	31608	627	0	1840	35267
rrc13	119635	151585	7	31627	9	0	2762	34074
rrc14	81599	107038	5	24795	5	0	2133	26717
rrc15	88563	110812	112	23127	102	0	1232	25863
rrc16	4502	3374	0	794	0	0	84	865
route-views2	213075	276846	5	62422	6	0	2594	64460
route-views4	135689	182428	0	45788	0	0	1330	48228
route-views.eqix	87653	113690	4	25349	4	0	740	26410
route-views.isc	118409	152221	5	32703	4	0	821	34062
route-views.linx	135956	170540	3	33718	5	0	1537	35311
route-views.wide	136375	172235	0	32563	0	0	2548	37307
rrc00May20	13182	16674	16	3458	19	0	142	3479

Figure 64. Number of moves for inserts and deletes in the ITCAM of DUOS

Dataset	Scheme 1	Scheme 2	Scheme 3	Scheme 4
rrc00	286568	81012	54949	83948
rrc01	368971	117996	79586	122687
rrc03	321230	89220	61539	92625
rrc04	174955	60770	41084	64562
rrc05	346573	108779	73378	112833
rrc06	353972	102797	72971	113443
rrc07	132509	47809	31890	50076
rrc10	250228	78779	52887	82620
rrc11	295146	84914	58790	87429
rrc12	358247	103651	72065	108545
rrc13	337230	97644	66019	102846
rrc14	243520	79683	54888	83733
rrc15	251663	75527	52390	79383
rrc16	10081	2999	2205	3154
route-views2	618746	191252	128831	195879
route-views4	417640	145311	99523	149081
route-views.eqix	257668	81678	56329	83475
route-views.isc	343790	105868	73164	108043
route-views.linx	379023	106248	72532	109375
route-views.wide	383443	107396	74833	114688
rrc00May20	37316	10934	7479	11081

Figure 65. Number of $waitWrites$ in the ITCAM of DUOS

Dataset	#moves	$\#waitWrites$
rrc00	0	776681
rrc01	0	1003809
rrc03	0	593723
rrc04	0	384503
rrc05	0	851569
rrc06	0	877953
rrc07	0	328259
rrc10	0	650525
rrc11	0	656996
rrc12	0	836906
rrc13	0	564381
rrc14	0	481577
rrc15	0	532622
rrc16	0	20714
route-views2	0	1269101
route-views4	0	1125824
route-views.eqix	0	568371
route-views.isc	0	622332
route-views.linx	0	754223
route-views.wide	0	913888
rrc00May20	0	95338

Figure 66. Number of LTCAM moves and $waitWrites$ for DUOS

Dataset	Lu [4]	Our
rrc00	68876	68196
rrc01	65068	64672
rrc03	66567	66060
rrc04	67895	67327
rrc05	65726	65319
rrc06	65411	65014
rrc07	64737	64322
rrc10	65566	65199
rrc11	65187	64766
rrc12	65564	65133
rrc13	66832	66366
rrc14	64955	64575
rrc15	66544	65982
rrc16	66353	65859
rviews2	68939	68300
rviews4	64839	64435
rviews.eqix	64881	64466
rviews.isc	66079	65664
rviews.linx	65372	64957
rviews.wide	66319	65910
rrc00May20	62638	62014

Figure 67. Number of prefixes to be stored in the LTCAM and associated wide SRAM

Dataset	#inserts	#deletes	$\#waitWrites$
rrc00	391398	391867	840029
rrc01	507076	507379	1071516
rrc03	302359	302568	641880
rrc04	194212	194117	412859
rrc05	429065	427408	884675
rrc06	442145	442208	1121870
rrc07	164213	164186	328698
rrc10	329469	330166	693637
rrc11	336674	336954	750563
rrc12	423898	424286	892330
rrc13	282676	282509	594906
rrc14	251034	251273	577811
rrc15	268558	266942	667692
rrc16	11297	9427	23902
route-views2	635210	636625	1290565
route-views4	572678	572828	1252536
route-views.eqix	289812	289825	648771
route-views.isc	315905	316093	694299
route-views.linx	379967	380210	789002
route-views.wide	466474	468214	1070171
rrc00May20	48036	47982	104402

Figure 68. Number of $waitWrites$ in the LTCAM of DUOW

Dataset	#inserts	#deletes	$\#waitWrites$
rrc00	10	5	34
rrc01	8	4	28
rrc03	4	2	14
rrc04	10	5	37
rrc05	0	0	0
rrc06	15	7	48
rrc07	0	0	0
rrc10	2	1	7
rrc11	6	3	21
rrc12	4	2	15
rrc13	0	0	0
rrc14	4	2	15
rrc15	6	3	20
rrc16	10	5	34
route-views2	2	1	6
route-views4	2	1	8
route-views.eqix	4	2	15
route-views.isc	4	2	15
route-views.linx	6	3	20
route-views.wide	2	1	7
rrc00May20	4	2	15

Figure 69. Number of $waitWrites$ for the ILTCAM of IDUOW using 1-12Wc

Dataset	#inserts	#deletes	#numBuckets	$\#waitWrites$
rrc00	391398	391867	221	849709
rrc01	507076	507379	215	1081031
rrc03	302359	302568	215	649739
rrc04	194212	194117	227	417069
rrc05	429065	427408	214	887978
rrc06	442145	442208	219	1151447
rrc07	164213	164186	209	328706
rrc10	329469	330166	218	696321
rrc11	336674	336954	215	760370
rrc12	423898	424286	213	902959
rrc13	282676	282509	215	600789
rrc14	251034	251273	213	585663
rrc15	268558	266942	218	683039
rrc16	11297	9427	219	26241
route-views2	635210	636625	223	1295688
route-views4	572678	572828	211	1268931
route-views.eqix	289812	289825	213	656627
route-views.isc	315905	316093	213	705386
route-views.linx	379967	380210	215	796007
route-views.wide	466474	468214	218	1089092
rrc00May20	48036	47982	203	105453

Figure 70. Statistics for the DLTCAM of IDUOW using 1-12Wc

Dataset	#inserts	#deletes	$\#waitWrites$
rrc00	170	85	533
rrc01	174	87	550
rrc03	134	67	424
rrc04	176	88	564
rrc05	210	105	656
rrc06	284	142	893
rrc07	20	10	63
rrc10	182	91	574
rrc11	214	108	673
rrc12	150	75	472
rrc13	224	117	693
rrc14	154	77	485
rrc15	298	152	936
rrc16	224	112	702
route-views2	158	79	498
route-views4	276	138	868
route-views.eqix	270	135	848
route-views.isc	156	78	493
route-views.linx	260	130	816
route-views.wide	265	134	835
rrc00May20	104	52	323

Figure 71. Statistics for the ILTCAM of IDUOW using M-12Wb

Dataset	#inserts	#deletes	#numBuckets	$\#waitWrites$
rrc00	391398	391867	149	867908
rrc01	507076	507379	142	1099815
rrc03	302359	302568	144	664596
rrc04	194212	194117	148	436057
rrc05	429065	427408	146	912990
rrc06	442145	442208	149	1180372
rrc07	164213	164186	128	330282
rrc10	329469	330166	142	716458
rrc11	336674	336954	145	784772
rrc12	423898	424286	142	920560
rrc13	282676	282509	149	624757
rrc14	251034	251273	140	602928
rrc15	268558	266942	153	714549
rrc16	11297	9427	147	49978
route-views2	635210	636625	149	1313783
route-views4	572678	572828	148	1299622
route-views.eqix	289812	289825	147	685458
route-views.isc	315905	316093	142	723400
route-views.linx	379967	380210	149	823707
route-views.wide	466474	468214	153	1120072
rrc00May20	48036	47982	131	117622

Figure 72. Statistics for the DLTCAM of IDUOW using M-12Wb

6.5 Comparison with MIPS [19] and CAO_OPT [23]

MIPS [19] and an update consistent version of CAO_OPT [23] obtained using the method of [18] are the competitors of DUO. In this section, we compare the consistent update TCAM schemes MIPS, CAO_OPT, and DUO. In MIPS, a data plane lookup is delayed if the lookup matches a TCAM slot whose next hop information is being updated. To avoid this delay while changing the nexthop of a prefix, we first insert a new entry with latest nexthop, and then delete the existing entry, in our experiments for MIPS. This ensures that data plane lookups are consistent and correct and are not delayed by control plane operations. Also as noted earlier, the MIPS scheme as described in [19] uses no memory management scheme and free slots are determined using TCAM lookups that delay data plane lookups. To avoid these data plane lookup delays, for our experiments, we augmented the MIPS scheme of [19] with the memory management scheme employed by us for the LTCAM (Section 4). For the ITCAM of DUO, memory management is done using Scheme 3. Since the performance of the 3 TCAM schemes is characterized by the total number of the $waitWrite$ operations required by an update sequence as well as the maximum number of operations for an individual update request, our experiments measured these quantities.

Figure 73 gives the total number of $waitWrite$s required to perform our test update sequences. We see that our DUO architecture requires fewer write operations than MIPS and CAO_OPT. The average number of $waitWrite$s per operation (Insert or Delete) ranged from a low of 1.565 to a high of 6.72 for MIPS, from 1.505 to 6.51 for CAO_OPT, from 1.000039 to 1.0641 for DUOS, from 1.00126 to 1.33705 for DUOW, from 1.00128 to 1.3702 for IDUOW with 1-12Wc and from 1.00583 to 2.3966 for IDUOW with M-12Wb. Since the various DUO schemes require a similar number of writes, M-12Wb is to be preferred because of its lower TCAM memory and power requirement. Figure 74(a) shows the normalized average $waitWrite$s for the different architectures. For this figure, we first computed the average number of $waitWrites$ per Insert/Delete for each dataset. Then, the average of the averages was computed for each architecture and normalized by the average of the averages for DUOS.

Figure 75 gives the maximum number of write operations required by an insert or delete in our test update sequences.

Dataset	MIPS [19]	CAO_OPT [23]	DUOS	DUOW	IDUOW(1-12Wc)	IDUOW(M-12Wb)
rrc00	1442078	1249984	831630	894978	904692	923390
rrc01	1798445	1655241	1083395	1151102	1160645	1179951
rrc03	1159357	997100	655262	703419	711292	726559
rrc04	887877	701722	425587	453943	458190	477705
rrc05	1436610	1428017	924947	958053	961356	987024
rrc06	2074384	1344593	950924	1194841	1224466	1254236
rrc07	783637	602014	360149	360588	360596	362235
rrc10	1168964	1082896	703412	746524	749215	769919
rrc11	1352758	1057557	715786	809353	819181	844235
rrc12	1602375	1380918	908971	964395	975039	993097
rrc13	1191824	989752	630400	660925	666808	691469
rrc14	993155	794964	536465	632699	640566	658301
rrc15	1208090	864242	585012	720082	735449	767875
rrc16	45895	33722	22919	26107	28480	52885
route-views2	2242123	2178506	1397932	1419396	1424525	1443112
route-views4	2304065	1818222	1225347	1352059	1368462	1400013
route-views.eqix	1278271	925087	624700	705100	712971	742635
route-views.isc	1172542	1013145	695496	767463	778565	797057
route-views.linx	1306298	1257542	826755	861534	868559	897055
route-views.wide	1988152	1435007	988721	1145004	1163932	1195740
rrc00May20	683608	663306	102817	111881	112947	125424

Figure 73. Total number of TCAM $waitWrite$ operations

As can be seen, MIPS uses a larger number of writes in the worst case than any of the remaining schemes. We notice that the worst-case number of writes for rrc00May20 is particularly large for MIPS. This is because the update sequence for rrc00May20 contains announcements and withdrawals of routes for prefixes of small lengths, such as 2 and 4. Each of these translates into a very large number of inserts/deletes of independent prefixes.

Our DUOS and DUOW architectures have better worst-case performance (on a per update basis) than MIPS. DUOS is generally better than CAO_OPT and DUOW, while inferior to CAO_OPT, is often competitive. Even though, the worst-case number of writes with IDUOW is more than that for CAO_OPT, the number of writes is bounded by the size of a bucket. Thus, the worst-case writes may be reduced by using a smaller bucket size than the 512 size used in our experiments. For example, when the bucket size as 32, the maximum number of write operations in DLTCAM of IDUOW is also 32. This is because when an index node is split, we relocate the split node that has the smaller number of DLTCAM prefixes. Thus at most 16 prefixes are moved, and hence there are 32 write operations at most.

Theoretically, it is possible for each update in MIPS to require a number of TCAM writes equal to the number of prefixes in the table. This happens for example when there is a trie in which no leaf prefix has a sibling after the leaf pushing and prefix compression steps, and to that trie if a default prefix of length 0 is inserted or deleted (see Figure 2). On the other hand, CAO_OPT requires at most $W/2$ moves per update(W = 32 for IPv4). Hence, CAO_OPT requires $W/2$ writes per update in the worst case. For DUOS, the worst case writes occur when a prefix is to be inserted to LTCAM and this requires a prefix deletion from LTCAM and a prefix insertion at ITCAM. The two LTCAM operations require 2 writes, whereas the ITCAM operation requires W writes in the worst case using Scheme 3. Thus DUOS requires $(W + 2)$ writes in the worst case. For DUOW, the worst case scenario is same as that for DUOS, except that a LTCAM insert can require 3 writes when a SRAM word is split (1 delete to remove the split word and 2 inserts for the new words). Similarly, a LTCAM delete can also require 3 writes when a SRAM word is merged (2 deletes for the two words merged and 1 insert for the new word). Thus, DUOW requires $(W + 6)$ writes in the worst case. For IDUOW, the worst case combination involves the ITCAM, ILTCAM and DLTCAM. IDUOW requires at most W writes for ITCAM and 6 writes for ILTCAM and $bucketSize$ writes for DLTCAM, with a maximum of $(W + bucketSize + 6)$ writes for

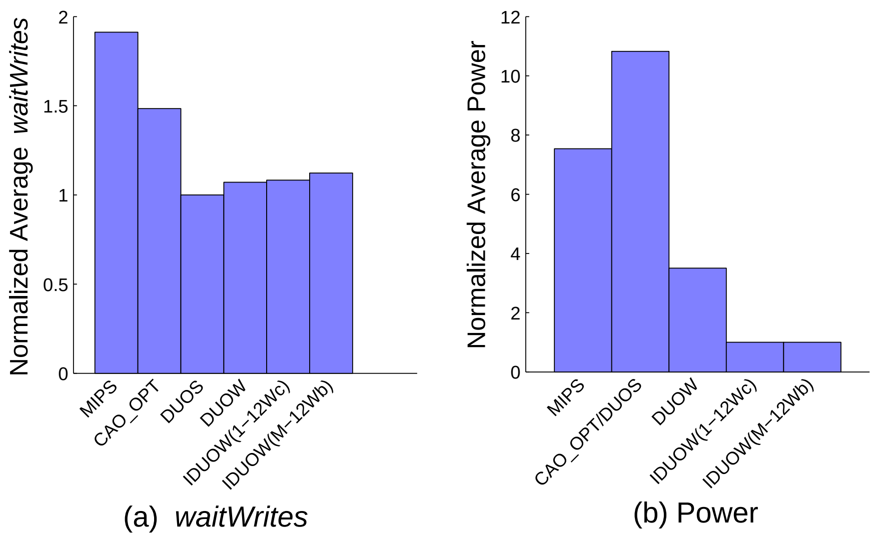


Figure 74. Comparison of TCAM performance and power consumption between MIPS, CAO_OPT, DUO

a single update.

Dataset	MIPS [19]	*CAO_OPT* [23]	DUOS	DUOW	IDUOW(1-12Wc)	IDUOW(M-12Wb)
rrc00	266	5	6	9	512	512
rrc01	296	5	3	7	505	505
rrc03	1186	5	9	10	505	505
rrc04	2682	6	3	7	505	511
rrc05	383	5	3	7	7	500
rrc06	1278	5	3	7	505	505
rrc07	6389	5	3	6	6	368
rrc10	304	6	4	7	222	503
rrc11	546	5	4	7	507	507
rrc12	7099	6	10	11	505	505
rrc13	1071	5	3	7	7	499
rrc14	306	6	5	7	505	505
rrc15	5938	4	4	7	510	508
rrc16	198	5	3	7	507	507
route-views2	568	5	5	7	399	497
route-views4	377	5	3	7	505	505
route-views.eqix	260	7	4	7	505	505
route-views.isc	386	5	3	7	505	505
route-views.linx	306	6	3	7	510	508
route-views.wide	278	4	3	7	332	503
rrc00May20	102249	6	5	7	378	475

Figure 75. Maximum number of TCAM writes for a single raw insert/delete

Figure 76 gives the power consumption characteristics of MIPS, CAO_OPT and DUO in terms of the number of entries enabled during a search operation. The TCAM entries are counted based on the initial layout of prefixes for the input routing table. MIPS, CAO_OPT, DUOS and DUOW enable all valid TCAM entries during a search operation. IDUOW, on the other hand, enables all valid TCAM entries for ITCAM and ILTCAM, and only a bucket of entries for

DLTCAM. Column 2 gives the number of enabled entries for MIPS, while column 3 gives the number of enabled entries for CAO_OPT on the simple TCAM and also for DUOS which is obtained by summing up the number of ITCAM and LTCAM entries. Both CAO_OPT and DUOS have the same number of entries in TCAM since they store each prefix in a single TCAM entry. Column 4 gives the number of enabled entries for DUOW, which is obtained as the sum of valid ITCAM and LTCAM entries. Columns 5 and 6 give the number of enabled entries for IDUOW with 1-12Wc and M-12Wb, respectively. This number is obtained as the sum of valid entries in ITCAM, ILTCAM and the number of entries in a bucket in DLTCAM (fixed to 512 for our experiments). We observe that for MIPS, the leaf pushing and prefix compression steps have reduced the number of TCAM entries, and hence the power compared to CAO_OPT and DUOS. MIPS requires about 1.5 to 2 times the power required by DUOW for all the tests, except rrc06 and rrc15. In the case of rrc06, MIPS requires about 7% more power than DUOW while it requires about 7% less power on rrc15. MIPS consumes between 3 to 10 times the power consumed by IDUOW. Figure 74(b) shows the normalized average power for the different schemes. For this figure, we first computed the average number of enabled entries for every TCAM search for each architecture. Then, the average was normalized by the average number of enabled entries for IDUOW with 1-12Wc. Note that the power requirement for DUOW can be reduced further by using a wider SRAM than the 144 bit wide SRAM used for our experiments. The power requirements for IDUOW may be reduced by increasing SRAM width and by adding an index TCAM and a wide SRAM to the ITCAM. For example, the power consumed by DLTCAM and ILTCAM of IDUOW was less than 560 for the 1-12Wc scheme and less than 630 for the M-12Wb scheme. When an index TCAM and wide SRAM is added to the ITCAM to our experimental IDUOW, the power requirement for the ITCAM is expected to approximate that for the LTCAM (assuming the same bucket size is used). So, the IDUOW power requirement would drop to about 1120 for 1-12Wc and about 1260 for M-12Wb. So, with the addition of an index TCAM and a wide SRAM to the ITCAM of IDUOW, the power required by MIPS is between 68 to 248 times that required by IDUOW.

Dataset	MIPS	CAO_OPT/DUOS	DUOW	IDUOW(1-12Wc)	IDUOW(M-12Wb)
rrc00	245875	294098	95577	27938	27989
rrc01	200733	276795	89459	25343	25387
rrc03	272046	283754	92176	26672	26714
rrc04	203375	288610	92464	25701	25759
rrc05	261067	280041	90694	25933	25987
rrc06	96479	278744	90221	25762	25839
rrc07	188373	275097	88763	24997	25055
rrc10	178987	278898	90031	25390	25434
rrc11	188527	277166	89553	25343	25399
rrc12	203440	278499	90027	25450	25493
rrc13	234053	284986	92686	26877	26935
rrc14	172096	276170	89060	25041	25084
rrc15	85463	284047	92166	26741	26813
rrc16	212282	282660	91445	26143	26198
rviews2	277560	294127	95183	27439	27502
rviews4	140962	275737	88978	25099	25145
rviews.eqix	175659	275736	88889	24980	25054
rviews.isc	193800	281095	91123	26018	26091
rviews.linx	202254	278196	90029	25631	25688
rviews.wide	150427	283569	92320	26966	27037
rrc00May20	220067	266185	86421	24961	25001

Figure 76. A comparison of power consumed by MIPS, CAO_OPT and DUO in performing TCAM search

7 Conclusion

We have proposed a dual TCAM architecture-DUO-for routing tables. Four memory management schemes also have been evaluated extensively for the ITCAM of DUO. Of these memory management schemes, Scheme 2 and Scheme 3 are the ones proposed by us. Our experiments showed that Scheme 3 is far better than any of the other schemes in terms of the number of moves per update operation.

DUO provides incremental update facility to the low power lookup schemes in [4], without locking the TCAM at any time for performing updates. Supporting incremental updates to the schemes in [4] was problematic since each prefix in the TCAM stored a corresponding covering prefix in the wide SRAM, and such covering prefixes could be shared by a number of TCAM entries. The covering prefixes are a subset of the intermediate prefixes, and by putting all intermediate prefixes in a separate TCAM(ITCAM) and enabling a parallel match on ITCAM and LTCAM, as in DUO, we completely bypass the problem with covering prefixes in the performance of incremental updates.

DUO is fast and power efficient when it comes to incorporating the updates. The speed and power efficiency of DUO are due to both its architecture and its memory management schemes. From our datasets we found that over 90% of the updates are directed to the LTCAM of DUO, which stored disjoint prefixes. Thus, no prefix move in involved for most of the updates. The less than 10% of updates that are directed to the ITCAM involve very small number of moves when the memory management Scheme 3 is used. Memory management Scheme 3, proposed by us, requires between 1/74000 and 1/93 times the number of moves required by CAO_OPT. The low average values for the number of moves using Scheme 3 are backed by very low standard deviation for all the tests in our dataset.

Our DUO architectures, like those based on the CoPTUA [18], provide for consistent data-plane lookups and incremental control-plane updates that do not delay data-plane lookups. While the MIPS architecture of [19] provides consistent data-plane lookups, these lookups may encounter delays by ongoing control-plane operations that, for example, change the next hop associated with a prefix. These delays may be eliminated by implementing a next hop change as an insert followed by a delete as suggested in [19]. Delays caused by control-plane operations that require a free slot to be found may be eliminated using one of our proposed memory management schemes, preferably Scheme 3. Making these two modifications to MIPS results in a delay-free MIPS.

Experiments with delay-free MIPS and a consistent lookup version of CAO_OPT indicate that these two architectures make, on average, between 1.5 and 5 times as many TCAM writes as made by any our DUO architectures to perform control-plane updates. In terms of the worst-case number of writes needed for an insert or delete, MIPS requires as many writes as prefixes in the table while CAO_OPT requires 16 for IPv4, DUOS requires 34, DUOW requires 38, and IDUOW requires 38+$bucketSize$. On our test data, MIPS required up to 102,249 writes for a single insert/delete while CAO_OPT required at most 7 writes, DUOS required at most 10 writes, DUOW required at most 11 writes, and IDUOW required at most 512 writes. The maximum number of writes for IDUOW may be reduced by reducing the bucket size. The very large number of worst-case writes for MIPS is a serious problem as this makes the router very susceptible to malicious users who inject a stream of worst-case inserts/deletes into the update stream. While this also is an issue, though to a lesser extent, for IDUOW, IDUOW offers power advantages over the remaining DUO schemes.

On our test data, MIPS reduced power consumption for a TCAM search by 4% to 69% relative to CAO_OPT and DUOS, which take the same amount of power. However, MIPS generally required between 1.5 and 2 times the power required by DUOW and between 3 and 10 times that required by our experimental version of IDUOW. However, by adding an index TCAM and a wide SRAM to the ITCAM of IDUOW, the power required by MIPS is between 68 and 248 times that required by the enhanced IDUOW. Further reduction in power required by DUOW and IDUOW result from using a wider SRAM than the 144-bit wide SRAM used in our experiments.

Between DUOW and IDUOW, IDUOW is recommended for least power consumption during lookups whereas DUOW is recommended for a lower worst-case delay in incorporating the updates to the forwarding table while still providing significant power benefits during a TCAM lookup.

References

[1] M. Akhbarizadeh, M. Nourani, R. Panigrahy and S. Sharma, A TCAM-based parallel architecture for high-speed packet forwarding, *IEEE Trans. on Computers*, 56, 1, 2007, 58-72.

[2] Y. Chang, Power-efficient TCAM partitioning for IP lookups with incremental updates, *ICOIN Proceedings*, Lecture Notes in Computer Science, Springer Verlag, 3391, 2005, 531-540.

[3] H. Lu, Improved Trie Partitioning for Cooler TCAMs, *ACST*, 2004.

[4] W. Lu and S. Sahni, Low Power TCAMs For Very Large Forwarding Tables, *Proceedings of INFOCOM*, 2008.

[5] W. Lu and S. Sahni, Succinct representation of static packet classifiers, *International Conference on Computer Networking*, 2007.

[6] *http://bgp.potaroo.net*, 2007.

[7] *http://www.ripe.net/projects/ris/rawdata.html*, 2008.

[8] H. Liu, Routing Table Compaction in Ternary-CAM, *IEEE Micro*, 22, 3, 2002.

[9] V.C. Ravikumar, R. N. Mahapatra, and L. N. Bhuyan, EaseCAM: An Energy And Storage Efficient TCAM-Based Router Architecture for IP Lookup, *IEEE Transactions on Computers*, 54, 5, May 2005, 521-533.

[10] V.C. Ravikumar, R. N. Mahapatra, and L. N. Bhuyan, TCAM architecture for IP lookup using prefix properties, *IEEE Micro*, 24, 2, March 2004, 60-69.

[11] R. Daves, C. King, S. Venkatachary, and B.Zill, Constructing Optimal IP Routing Tables, *Proceedings of INFOCOM*, 1999.

[12] M. Ruiz-Sanchez, E. Biersack, and W. Dabbous, Survey and taxonomy of IP address lookup algorithms, *IEEE Network*, 2001, 8-23.

[13] S. Sahni, K. Kim, and H. Lu, Data structures for one-dimensional packet classification using most-specific-rule matching, *International Journal on Foundations of Computer Science*, 14, 3, 2003, 337-358.

[14] C. A. Zukowski, and S. Wang, Use of Selective Precharge for Low-Power Content-Addressable Memories, *IEEE International Symposium on Circuits and Systems*, 1997.

[15] N. Mohan, and M. Sachdev, Low Power Dual Matchline Ternary Content Addressable Memory, *IEEE International Symposium on Circuits and Systems*, 2004.

[16] H. Miyatake, M. Tanaka, and Y.Mori, A design for high-speed low-power CMOS fully parallel content addressable memory macros, *IEEE Journal of Solid State Circuits*, 36, 6, June 2001, 956-968.

[17] C.-S. Lin, J.-C. Chang, and B.-D Liu, A low-power pre-computation based fully parallel content addressable memory, *IEEE Journal of Solid State Circuits*, 38, 4, April 2003, 654-662.

[18] Z. Wang, H. Che, M. Kumar, and S.K. Das, CoPTUA: Consistent Policy Table Update Algorithm for TCAM without Locking, *IEEE Transactions on Computers*, 53, 12, December 2004, 1602-1614.

[19] G. Wang and N. Tzeng TCAM-Based Forwarding Engine with Minimum Independent Prefix Set (MIPS) for Fast Updating, *IEEE International Conference of Communications* Volume 1, June 2006, 103-109

[20] M. Wang, S. Deering, T. Hain, and L. Dunn, Non-random Generator for IPv6 Tables, *12th Annual IEEE Symposium on High Performance Interconnects*, 2004.

[21] F. Zane, G. Narlikar and A. Basu, CoolCAMs: Power-Efficient TCAMs for Forwarding Engines, *INFOCOM*, 2003.

[22] T. Mishra and S.Sahni, PETCAM – A Power Efficient TCAM For Forwarding Tables, *IEEE Symposium on Computers and Communications*, 2009

[23] D. Shah and P. Gupta, Fast Updating Algorithms on TCAMs, *IEEE Micro* Volume 21, Issue 1, Jan-Feb 2001, 36-47

[24] M. Akhbarizadeh and M. Nourani, Efficient Prefix Cache For Network Processors, *IEEE Symp. on High Performance Interconnects*, 41-46, 2004.

[25] V. Srinivasan and G. Varghese, Faster IP lookups using controlled prefix expansion, *SIGMETRICS*, 1998.

[26] K. Zheng, C. Hu, H. Lu and B. Liu, An Ultra High Throughput and Power Efficient TCAM Based IP Lookup Engine, *Proceedings of INFOCOM*, 2004

[27] M. Akhbarizadeh, M. Nourani, R. Panigrahy and S. Sharma, A TCAM-based parallel architecture for high-speed packet forwarding, *IEEE Trans. on Computers*, 56, 1, 2007, 58-2007.

[28] M. Akhbarizadeh and M. Nourani, Efficient Prefix Cache for Network Processors, *IEEE Symposium on High Performance Interconnects*, August 2004.

[29] T. Mishra and S. Sahni, CONSIST - Consistent Internet Route Updates, *To be posted before the conference*.