TCAM Enabled On-Chip Logic Minimization

Seraj Ahmad
seraj@tamu.edu

Rabi Mahapatra
rabi@cs.tamu.edu

Department of Computer Science
Texas A & M University
College Station, Texas-77843

ABSTRACT

This paper presents an efficient hardware architecture of an on-chip logic minimization coprocessor. The proposed architecture employs TCAM cells to provide fastest and memory efficient implementation suitable for emerging on-chip minimization applications. The paper presents a detailed design of the on-chip minimizer and shows that it requires very little hardware resources to achieve acceptable quality of minimization. An incremental insertion and bulk deletion is achieved in 0.25 μs and 3.8 ms respectively and a compaction of 100000 entries in 25 ms using just 300 TCAM entries.

Categories and Subject Descriptors

B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Algorithms implemented in hardware*

General Terms

Algorithms, Design

Keywords

TCAM, Logic Minimization,On-Chip

1. INTRODUCTION

Logic Minimization has been traditionally applied to problems (such as logic synthesis) where the table to be minimized is statically defined. However, several emerging applications require logic minimization on a rapidly changing table. In addition, they also have very stringent timing and memory constraints. Example of such applications include routing table compaction, network access control list minimization and to some extent dynamic hardware/software partitioning. Logic minimization problem is intractable making the traditional algorithm unsuitable to these class of applications. Due to frequent update on the target table, the algorithm must be suitable for performing fast incremental minimization. Also, in order to optimize the response time, the algorithm is kept closer to the application and share the limited memory and computing resources. This makes software oriented approaches inadequate to fulfill all the constraints.

There has been several attempts to use hardware-based coprocessors to accelerate general purpose logic minimization algorithms. Cong et al. investigate the acceleration of tautology checking using FPGA [4]. Their approach is however restricted to 8 or fewer variables and computes all the minterms. Authors in [3] suggests a coprocessor design based on FPGA to evaluate binate covering problem. These FPGA based approaches are reported to achieve a speedup factor ranging from 1.36 to 2.94 compared to their desktop counterparts in a co-designed implementation of Espresso-II [3]. Roman Lysecky et al. propose hardware acceleration of ROCM, a lightweight minimizer targeted for on-chip logic minimization due to its compact size and lower memory requirement [8]. In co-designed ROCM implementation, the six most frequently used loops accounting for 90% of the execution time are pushed to hardware. These loops mainly involve determination of tautology, cofactor and cube covers. The implementation requires about 19000 gates and offers a 10 to 18 times speedup for reduction of routing tables. However the hardware design and implementation details of the functional blocks are not provided for practical evaluation.

Although these acceleration strategies significantly speedup the execution time, they are still not suited for minimization of large routing tables expected to contain more than several hundred thousand entries. For example, to minimize a routing table containing 79743 entries using a co-designed ROCM with a hardware speedup factor of 20 on a 400 MHz XScale platform will still take 420 seconds, quite an overhead for a backbone router which is either starting-up or recovering from a failure. Also, the hardware assisted version of these algorithm will still be unable to fulfill the peak-rate and worst-case BGP update requirements. Authors in [1] discuss a trie-based data structure which uses a combination of containment and d_1 merge techniques to achieve minimization. Their technique achieves fast and memory efficient compaction by employing a dynamic table partitioning and garbage collection scheme at an acceptable compaction loss. Their approach provides 100-1000 times speedup over Espresso-II or ROCM for routing table compaction while requiring about one tenth of the data memory. For example, their implementation can minimize the same table containing 79743 entries in 2.38 seconds. Although m-Trie offers decent performance but it may still be insufficient to provide adequate number of updates without hardware acceleration.

This paper discusses a novel architecture to implement m-Trie in hardware. The proposed architecture utilizes the ternary content addressable memory(TCAM) cells to achieve parallelization in m-Trie implementation. Most of the tables targeted for dynamic minimization are usually stored in TCAM modules for either lookup or classification purposes. This presents a good case for embedding the proposed hardware-based minimizer in the TCAM modules itself. Additionally, this work describes three techniques used in the implementation of the hardware-based minimizer, which can be adopted in commercial TCAMs to offer more powerful lookup and classification capabilities required for pattern matching applications.

The rest of this paper is organized as follows. Section 2 describes the architecture and operation of standard ternary content addressable memories. Section 3.1 discusses enhancements needed for TCAM lookup architecture to support more powerful matching semantics. Section 3.2 discusses the proposed logic minimizer architecture. Section 3.3 discusses the cube Insertion/Deletion algorithms used by the minimization controller. Section 4 discusses the case study and experimental results. The conclusion and future directions are discussed in Section 5.

2. PRELIMINARIES

Two level logic minimization problem involves exploring the minimal representation of a boolean function written in *sum of product* (or product of sum) form. Each of the product term(also known as *cubes*) contains a number of binary variables(or *literals*) in complemented or un-complemented form selected from the set $\{x_1, x_2, x_3 \cdots x_W\}$ of W variables. If the cube contains all the W variables, then it is called a *minterm*. Thus each of the cubes can be mapped to a fixed length ternary string in $\{0, 1, x\}^*$ by encoding the complemented, un-complemented and absent literals by 0, 1 and x respectively. The proposed logic minimizer architecture utilizes ternary content addressable memory (TCAM) cells to support storage and comparison of minterms expressed as fixed length ternary strings.

A NOR based TCAM cell is shown in Figure 1. It uses two SRAM based storage cell to store states 0, 1 and x based on the encoding scheme given in Table 1. Each TCAM cell is provided with four transistor switches to assist comparison.

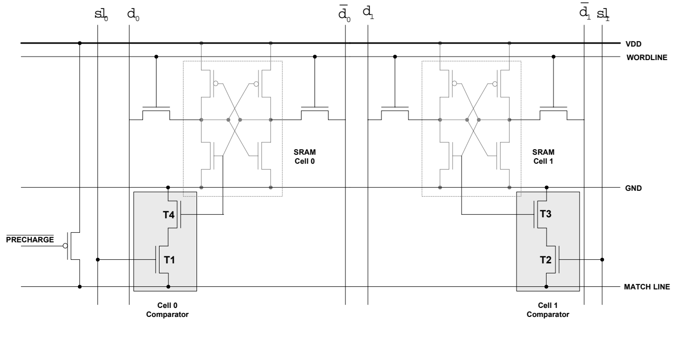

Figure 1: A NOR-based TCAM Cell

These transistor switches prevent matchline from getting shorted to ground when a match occurs. For example a state 0 in TCAM cell will turn off the transistor $T3$. A search for 0 applied on search lines sl_0 and sl_1 will turn off transistor $T1$ blocking match line from getting shorted to ground. However a search for 1 will turn on the transistor $T1$ creating a path to ground through transistor $T4$. On the other hand a state x in TCAM will turn off both $T3$ and $T4$ blocking all path to ground thus matching all search keys applied to TCAM cell giving a don't care match semantics needed to encode the *absence* of a literal.

Table 1: Encoding of Ternary Symbols in TCAM

d_1	d_0	T
0	0	x
0	1	0
1	0	1
1	1	x_{sr}

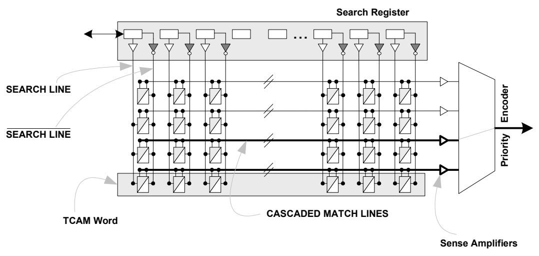

Figure 2: On-Chip Minimizer Architecture

A simplified TCAM architecture is shown in Figure 2. Here an array of TCAM cells are arranged to form a TCAM word. In order to perform word comaprison, all cells belonging to a single word share a common match-line.

Since data being searched can match multiple words in TCAM due to variable length matching, all the matchlines are connected to a priority encoder. The priority encoder selects the word at the lowest address among all the matched words.

To initiate a search in TCAM, the *matchline* is pre-charged to a high level. The data to be searched is stored in search register and asserted. TCAM words which do not match the search data causes the matchline to be discharged, which is detected with the aid of a sense amplifier. Since the search data is fed to large number of cells, TCAM uses *searchline drivers* to handle the capacitive sink load caused by each cell.

3. ON-CHIP MINIMIZER

The proposed logic minimizer architecture uses a TCAM based lookup architecture to achieve parallel comparisons. However, it adds several enhancements to existing TCAM lookup architecture to implement the minimizer. We describe these enhancements in the following subsections.

3.1 Enhanced TCAM Lookup

TCAMs are mostly used to perform IP routing table lookup in $O(1)$ time. The lookup operation involves comparing a fixed length binary string against a set of variable length strings to find the a longest match. The variable length strings are called IP prefixes and can be thought of as ternary strings with x's appearing only at the end.

However, it should be noted that TCAMs permits the storage of x at any arbitrary position enabling it to perform

Table 2: Match Behavior For Ternary Symbols

	0	1	x
0	y	n	y
1	n	y	y
x_{sr}	n	n	y

lookup of a binary string amongst any arbitrary ternary strings instead of against only trailing ternary strings. Thus IP lookup only uses a subset of the powerful search capability provided by the TCAMs.

Theoretically, TCAM search semantics is equivalent to searching the smallest cube containing a specified minterm in a set of cubes. For performing logic minimization, TCAM lookup mechanism needs to be enhanced to perform a *cover*, *contain* and *exact lookup* of a ternary string amongst a set of ternary strings. The exact ternary lookup is a hybrid operation and is implemented in terms of *cover* and *contain* operations, which is described next. We will denote the TCAM providing the cover and contain functionality by $T_{\supseteq}$ and $T_{\subseteq}$ respectively.

3.1.1 Cover Lookup ($\supseteq$)

A cover lookup($\supseteq$) operation can be defined as the problem of finding all the cubes covering a specified cube. In order to perform cube cover lookup in TCAM, the search register should be capable of holding 0 , 1 and x. This can be easily accomplished by using 2 bits to represent each symbol of the cube. The corresponding hardware overhead is quite small and doesn't impact the TCAM architecture much. Since the encoding of x in TCAM and search register are not same, we will use x_{sr} for x in search register.

The 0, 1 , x and x_{sr} symbols are encoded using the scheme given in Table 1 for a TCAM cell shown in Figure 1. The opposite encoding of "don't care" symbol x in TCAM and search register results in match behavior shown in Table 2. For example, consider the NOR based TCAM cell shown in Figure 1. If a "don't care" is stored in search register, the encoding scheme implies that it will turn on the transistor T1 and T2. If a "0" or "1" is stored in the cell, it would cause either T3 or T4 to be turned on thereby creating a discharge path to ground (a mismatch). However, if the "don't care" symbol x is stored in TCAM, it will turn off the transistor T3 and T4 blocking any path to ground thereby matching with either of 0, 1 and x. This match behavior enables TCAM to search all the cubes covering the specified cube.

The proposed insertion algorithm ensures that the cubes are always ordered according to their size. Thus a priority encoder is sufficient to select the largest cube amongst all the covering cubes.

3.1.2 Contain Lookup ($\subseteq$)

A *contain lookup*($\subseteq$) operation can be similarly defined as the problem of finding all the cubes contained in a specified cube. In order to perform contain lookup operation, the search register in table $T_{\subseteq}$ is similarly enhanced using 2 bits to allow storage of 0, 1 and x_{sr}. However, it uses a different encoding scheme as shown in Table 3 to achieve *contain lookup* capabilities.

Table 3: Encoding of Ternary Symbols in $T_{\subseteq}$

d_1	d_0	T
0	0	x_{sr}
0	1	0
1	0	1
1	1	x

To illustrate, we again consider the NOR based TCAM cell shown in Figure 1. If a don't care is stored in search register, the encoding scheme implies that it will turn off the transistor T1 and T2 blocking all paths to ground, thus matching a 0, 1 or x stored in TCAM. However, a 0 or 1 stored in the search register matches with only 0 or 1 stored in TCAM respectively. This runs contrary to normal TCAM behavior but gives the desired contain-lookup capability. Further, all the contained cubes are discovered in one cycle.

The algorithms proposed here do not require the capability to find the largest cube contained in the specified cube. Thus we can eliminate the extra hardware needed to select the the largest cube.

3.1.3 Exact Ternary Lookup

An exact ternary lookup operation can be realized in terms of *cover* and *contain* lookup operations. Suppose $A_{\supseteq}$ be the set of cubes in S, which covers a specified cube A. Also suppose $A_{\subseteq}$ be the set of cubes in S, which are contained in the cube A. Thus we can express the cube A as follows:

$$A \overset{x}{=} A_{\supseteq} \cap A_{\subseteq}$$

This relation forms the basis of the proposed O(1) exact ternary lookup. In order to implement this relations, the match lines in cover lookup table($T_{\supseteq}$) and contain lookup table($T_{\subseteq}$) are joined together. Thus the entries which are strictly contained in A or those which strictly covers A are pulled down to zero resulting in a mismatch. This leaves only the entry corresponding to the cube exactly matching the specified cube remain at high level indicating a match.

The three lookup enhancements described here are utilized by the proposed minimizer architecture to support m-Trie based insertion/deletetion mechanisms required to implement minimization algorithm.

3.2 Minimizer Architecture

Figure 3 depicts the high level minimizer architecture. The coprocessor uses two TCAM tables $T_{\supseteq}$ and $T_{\subseteq}$ called *cover* and *contain* lookup tables respectively. Any *insert* or *delete* operation is carried out on both the tables. Together, these tables are used to support cube merge and containment operations. Both the tables are equally sized and contain N TCAM words each. The optimal value of N is discussed in the *experimental results* section. The characterization of these tables are given in the following subsections.

In Figure 3, an entry being searched can match multiple entries in the TCAM. The proposed minimization algorithm requires either iterating over all the matching entries or just the first matched entry. To support this complex behavior, the architecture uses a *Matched Entry* flag for each TCAM entry to remember the result of the last lookup operation. The result of the match is then fed to a *priority encoder*, which generates the location of the first matched entry. Thus the minimization controller can process either all the matched entries sequentially or just the first matched entry.

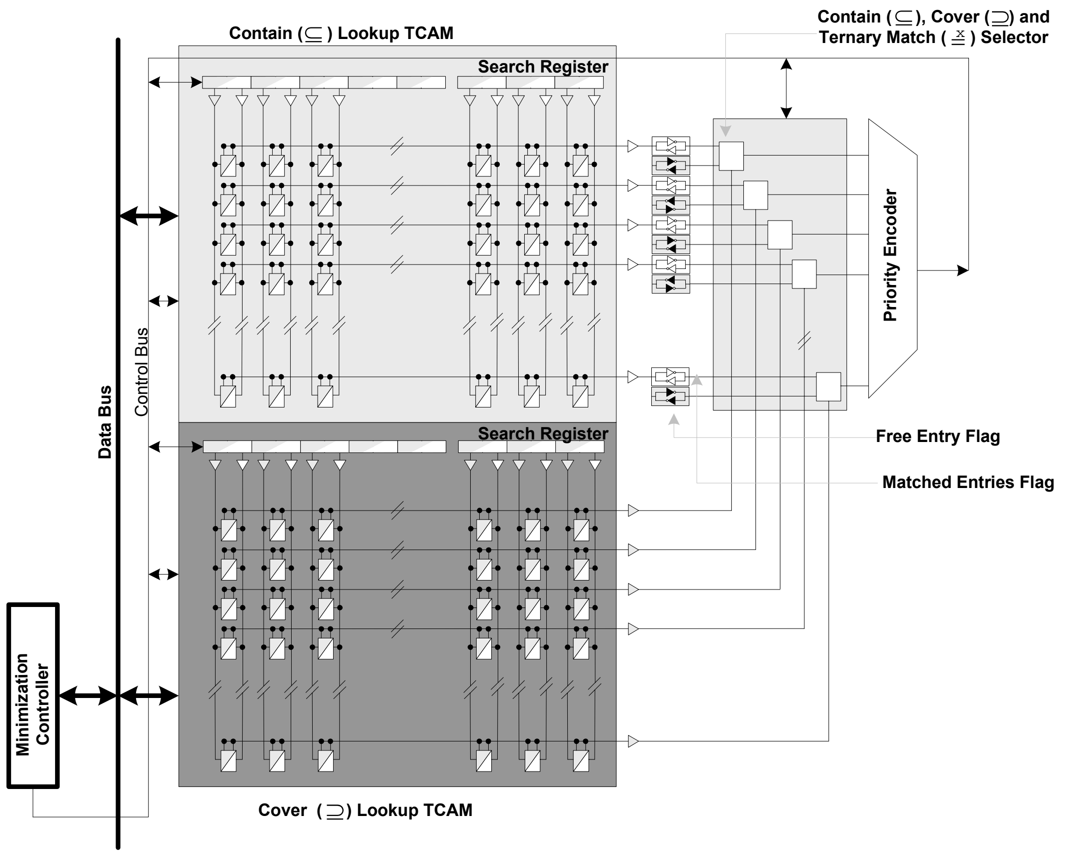

Figure 3: On-Chip Minimizer Architecture

The architecture also uses a *Free Entry* flag for each TCAM entry to implement an *address-input* free write mechanism as proposed in [10]. The flag is set to 1 for each unused location. All the flags are fed to a priority encoder to select the first unused location for a write operation. Since the *TCAM writes*, *cover lookup* and *contain lookup* operations are used sequentially in the minimization algorithm, a single priority encoder is enough to handle these operations. The minimization controller multiplexes the use of priority encoder through a *cover*, *contain*, *exact ternary* and *Free Entry* selector as shown in the Figure 3.

Each of these TCAM table is provided with a search register to support simultaneous search in both the tables. Normally, the search register found in TCAM based coprocessors uses *one bit per symbol* and supports fixed length binary search keys. Here, the search register used here are *enhanced* to utilize two bits per symbol to be able to specify variable length search keys. The search register input is fed through a driver as discussed in Section 2, which helps to overcome the long interconnect and sink load capacitances. Also, the *searchline* may be segmented to improve the lookup delay. Further improvement on lookup delay can be obtained by segmenting the *matchline*.

3.3 Minimization Controller

The TCAM tables, search registers, priority encoder arbitration are manipulated using a minimization controller which implements the minimization algorithm logic. In order to handle the dynamically changing tables, the minimization is treated as a series of cube insertion and deletion. Cube insertion and deletion utilize the enhanced lookup functionality and derive their efficiency from increased search parallelism. Following section describe the insertion/deletion algorithm used by the minimization controller. The algorithms are adapted from m-Trie based insertion/deletion algorithm to work with TCAM tables.

3.3.1 Cube Insertion

In order to insert the cube **c**, the first step involves in finding, if **c** is contained in some other cube. This is accomplished by performing a lookup of the specified cube in the *cover* TCAM table $T_{\supseteq}$. If a covering cube $c_{\supseteq}$ exists, the cube **c** is simply dropped as it is already present inside a larger cube. This step is captured in lines 1-4 in Algorithm 1.

The second step involves determining $\mathbb{C}$, the set of all the cubes which are contained in the cube **c**. This is accomplished by performing a lookup in the *contain* TCAM table $T_{\subseteq}$. The insertion algorithm then iterates over all the cubes $c_{\supseteq} \in \mathbb{C}$ and marks the location corresponding to these entries as free. This step is captured in the lines 5-7.

The third step involves searching the cubes which can be merged into the cube **c**. Please note that a cube c of the form $c_{1\cdots i-1} \cdot {\scriptstyle 0} \cdot c_{i+1\cdots W}$ can be merged with another cube $c_{1\cdots i-1} \cdot {\scriptstyle 1} \cdot c_{i+1\cdots W}$ at i^{th} position and vice versa. The algorithm therefore inverts a 0 or 1 symbols present in the

Table 4: Comparison of HWLM Performance against m-Trie and codesigned-ROCM

	Base Tables		codesigned ROCM		m-Trie(sw)		HWLM		
	Orig.	Pruned	Time (sec)	Table Size	Time (sec)	Table Size	Time (millisec)	Table Size	TCAM size (No Of Entries)
paix	13914	11091	5.992	8984	0.240	8881	2.773	8969	116
pacbell	22165	16124	61.000	11604	0.310	11351	4.031	11642	138
maewest	29585	22042	46.466	16728	0.440	16354	5.511	16686	139
aads	33740	24795	50.671	18898	0.510	18531	6.199	18881	133
att	112412	79743	420.268	57175	2.380	57918	19.936	58772	137
bbn	124538	92773	142.917	70984	2.820	70716	23.193	71782	137

cube **c** to get the cube c'. It then performs a *exact ternary lookup* of the cube c' to find $c'_=$. If the cube $c'_=$ is found, it is simply deleted (i.e. associated entry is just marked free) and the corresponding 0 or 1 symbol in cube **c** is promoted to symbol x. The procedure iterates over all the 0 and 1 symbol present in the cube **c** to explore all possible 0,1 promotions. This is captured in lines 8-17. Finally, the modified cube **c** resulting from third step is written in the TCAM.

The cube insertion algorithm spends one cycle in the lines 1, 2-4 and 5 each. It spends W cycles in the lines 5-7 in the worst case. The lines 8-17 requires 2 cycles and W worst case iterations. Thus the insertion algorithm requires $3 \times W + 4$ cycles.

Please note that lines marked by arrows are needed by the cube deletion algorithm.

Algorithm 1.

InsertCube(c)

```
1   $c_{\supseteq}$ = lookup($c$, $T_{\supseteq}$)
2   if($c_{\supseteq} \neq \varnothing$)
→       count[$c_{\supseteq}$]++
→       $(c_{\supseteq})_{b_0} | = (c_{\supseteq} \oplus c)_0$ , $(c_{\supseteq})_{b_1} | = (c_{\supseteq} \oplus c)_1$
3       return
4   endif
5   $\mathbb{C}$ = lookup($c$, $T_{\subseteq}$)
→   $count[c] = 1$, $(c)_{b_0} = 0$, $(c)_{b_1} = 0$
6   $\forall c_{\subseteq} \in \mathbb{C}$
→       count[$c$] = count[$c$] + count[$c_{\subseteq}$]
→       $(c)_{b_0} | = (c \oplus c_{\subseteq})_0$ , $(c)_{b_1} | = (c \oplus c_{\subseteq})_1$
7       delete($c_{\subseteq}$)
8   for($i = W$; $i > 0$; $i = i - 1$)
9       if($c_i \in 0, 1$)
10          $c' = c_{1 \cdots i-1}.\overline{c}_i.c_{i+1 \cdots W}$
11          $c'_= \overset{x}{=}$ lookup($c'$, $T_{\supseteq}$) $\cap$ lookup($c'$, $T_{\subseteq}$)
12          if($c'_= \neq \varnothing$)
→               count[$c$] = count[$c$] + count[$c'_=$]
13              delete($c'_=$)
14              $c$ =promote($c, i$)
15          endif
16      endif
17  endfor
18  insert($m$)
end
```

3.3.2 Cube Deletion

A cube **c** to be deleted may not be directly present in the TCAM due to the way insertion works. For example, the containment operation or symbol promotion may cause the cube **c** to be merged into another cube which is in fact a *cover* of the cube **c**. Hence the first step in cube deletion algorithm involves lookup of cube **c** in the table $T_{\supseteq}$ to find a covering cube $c_{\supseteq}$. This is captured by the first line in Algorithm 2.

The second step involves recovering the merging and containment information of the cube **c** from the cube $c_{\supseteq}$. The cube deletion algorithm maintains a counter and two bitmaps (b_0 and b_1) for each of the cube stored in TCAM. The counter corresponding to each cube records the number of cubes merged or contained into that cube. The bitmap b_0 records the positions of symbol x which observe containment of 0's. Similarly, the bitmap b_1 records the positions observing containment of 1's. For example, the containment of a cube 01x011xx into a cube 01xx1xxx can be recorded by $b_0 = 0001\,0000$ and $b_1 = 0000\,0100$. These recovery information are collected during the *cube insertion* as is highlighted in Algorithm 1 by arrows.

Algorithm 2.

DeleteCube(c)

```
1   $c_{\supseteq}$ = lookup($c$, $T_{\supseteq}$)
2   if($c_{\supseteq} \neq \varnothing$)
3       count[$c_{\supseteq}$]=count[$c_{\supseteq}$]-1
4       $b = c_{\supseteq} \oplus c$
5       $b = (\overline{b} \oplus (c_{\supseteq})_{b_0}) | (b \oplus (c_{\supseteq})_{b_1})$
6       mask=1
7       for($i = W$; $i > 0$; $i = i - 1$)
8           if($b$ & mask)
9               $\hat{c} = c_{1 \cdots i-1}.\overline{c}_i.c_{i+1 \cdots W}$
10              count[$c_{\supseteq}$]=count[$c_{\supseteq}$]-1
11              insert($\hat{c}$)
12          endif
13          mask = mask $\gg$ 1
14      endfor
15      delete($c_{\supseteq}$)
16  endif
end
```

Thus when the cube **c** is withdrawn from the covering cube $c_{\supseteq}$, the counter corresponding to $c_{\supseteq}$ is decremented. The algorithm then computes a bitmap of the positions of **c** having disagreements with $c_{\supseteq}$. It then excludes the positions corresponding to 0 and 1 containment by xor-ing it with bitmap b_0 and b_1. The resulting bitmap b reflects the positions where a 0 or 1 have been promoted to an x symbol during merge stage of cube insertion. In order to restore the effect of merge operation, x symbol at position i is split into c_i and $\bar{c}_i$ giving rise to two cubes $c_{1\cdots i-1}.\bar{c}_i.c_{i+1\cdots W}$ and $c_{1\cdots i-1}.c_i.c_{i+1\cdots W}$. This step is performed for all the positions indicated by the bitmap b. Finally, the covering prefix $c_{\supseteq}$ is deleted from the TCAM.

The cube deletion algorithm spends one cycle in lines 1, 2, 3-6 and 8-13 and 15. Further, it requires W worst case iteration of lines 8-13. Thus the total number of worst case cycles consumed by cube deletion algorithm is $W + 4$.

4. EXPERIMENTAL RESULTS

To establish the suitability of using the proposed on-chip minimizer (HWLM), we estimated its performance on the standard routing table traces used in [9] and [7]. We have also included two large routing table traces from `bbnplanet` and `attcanada`. The results for all the software approaches were obtained on a cerfcube platform running embedded Linux on a 400 MHz Intel XScale processor [6]. The result for ROCM has been scaled up by a hardware acceleration factor of 20 to obtain the codesigned-ROCM performance. The results for the proposed minimizer was obtained for 32-bit IP prefixes and assuming a 2.5ns lookup cycle which is typical of today's TCAM [2], [5]. The main results have been summarized in Table 4. The first column gives the original and pruned table sizes. The next two columns give the execution time and size of the minimized routing table in that order for codesigned-ROCM and software implemented m-Trie algorithms. For HWLM, we mention the execution time, size of the minimized routing table and the size of TCAM minimization table required in that order. As we can see here that HWLM achieves 100 times faster minimization as compared to m-Trie based minimization, the fastest known approach. The proposed architecture takes only 20 milliseconds to minimize the `attcanada` routing table containing 79743 entries.

The performance of cube insertion and cube deletion is only limited by worst-case lookup time achievable in TCAMs. The cube insertion requires 100 worst case cycles while cube deletion algorithm requires only 36 worst case cycles for 32 bit IP prefixes. On a 2.5ns lookup TCAM, the cube insertion will require 0.25 microseconds and deletion will require 0.09 microseconds. Thus the architecture supports a much higher number of updates (1 million with adequate hardware resources) per seconds as compared to thousands of updates for other approaches.

The main overhead of the architecture is contributed by the size of the TCAM table used. As we can observe from the Table 4, TCAMs $T_{\subseteq}$ and $T_{\supseteq}$, can be implemented with 150 entries each and still achieve 95% of the maximum achievable compaction. However, in order to support 0.09 microseconds deletion, we may need up to 9446 entries as in the case of `attcanada` routing table. Since the insertions are fast enough, the cube deletion can be modelled as re-minimization of the resulting table after withdrawing the entries to be deleted. This will keep the number of entries required to 150 but can still achieve a bulk deletion (containing 100-1000 deletions) in 3.8 ms.

5. CONCLUSIONS

In this paper, we presented a TCAM enabled hardware approach to handle minimization for emerging On-Chip logic minimization applications. In particular, we implemented m-Trie based approach toward logic minimization in hardware using a novel architecture. We obtained the minimization and update performance of the proposed architecture. The performance results indicate that the architecture gives the best performance compared to other known approaches till date. The proposed hardware architecture may be extended to support acceleration of other logic minimization algorithm such as Espresso-II as a future work.

6. ACKNOWLEDGMENTS

We would like to acknowledge R. Lysecky and F. Vahid at UCR for providing the ROCM code and routing table traces to help us compare the performance. We would also like to acknowledge various discussion with Dr. Sunil Khatri to refine the paper.

7. REFERENCES

[1] S. Ahmad and R. Mahapatra. m-Trie - A Fast and Efficient Approach to On-Chip Logic Minimization. In *Proc. Intl. Conf. on Computer Aided Design*. IEEE, November 2004.

[2] S. Choi, K. Sohn, M.-W. Lee, S. Kim, H.-M. Choi, D. Kim, U.-R. Cho, H.-G. Byun, Y.-S. Shin, and H.-J. Yoo. A 0.7fJ/bit/search, 2.2ns search time hybrid type TCAM architecture. In *Solid-State Circuits Conference, 2004. Digest of Technical Papers*, volume 1, pages 498–542, Feb 2004.

[3] J. Cong and J. Peck. On Acceleration of Logic Synthesis Algorithms using FPGA-based Reconfigurable Coprocessors. Technical Report TR-970010, Computer Science Department, University of California, Los Angeles, CA, 1997.

[4] J. Cong and J. Peck. On Acceleration of the Check Tautology Logic Synthesis Algorithm using an FPGA-based Reconfigurable Coprocessor. In *Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM '97)*, page 246. IEEE Computer Society, 1997.

[5] B. Gamache, Z. Pfeffer, and S. P. Khatri. A Fast Ternary CAM Design for IP Networking Applications. In *12th International Conference on Computer Communications and Networks (IC3N-03), Dallas, TX*, volume 1, pages 498–542, October 2003.

[6] Intrinsyc. *Cerfcube 255 with Embedded Linux.* http://www.intrinsyc.com/products/cerfcube.

[7] H. Liu. Routing Table Compaction in Ternary-CAM. *IEEE, Micro*, 15(5):58–64, Jan/Feb 2002.

[8] R. Lysecky and F. Vahid. A Codesigned On-Chip Logic Minimizer. In *Proc. CODES+ISSS*, pages 109–113. ACM, October 2003.

[9] R. Lysecky and F. Vahid. On-chip logic minimization. In *Proc. Design Automation Conference*, pages 334–337. ACM, June 2003.

[10] H. Noda, K. Inoue, H. Mattausch, T. Koide, and K. Arimoto. A Cost-Efficient Dynamic Ternary CAM in 130 nm CMOS Technology with Planar Complementary Capacitors and TSR Architecture. In *Symposium on VLSI Circuits Digest of Technical Papers*, pages 83–84, 2003.