On-Chip Logic Minimization

Roman Lysecky, Frank Vahid*
Department of Computer Science and Engineering, University of California, Riverside
{rlysecky, vahid}@cs.ucr.edu, http://www.cs.ucr.edu/~vahid
*Also with the Center for Embedded Computer Systems at UC Irvine

ABSTRACT
While Boolean logic minimization is typically used in logic synthesis, logic minimization can be useful in numerous other applications. However, many of those applications, such as Internet Protocol routing table and network access control list reduction, require logic minimization during the application's runtime, and hence could benefit from minimization executing on-chip alongside the application. On-chip minimization can even enable dynamic hardware/software partitioning. We discuss requirements of on-chip logic minimization, and present our new on-chip logic minimization tool, ROCM. We compare with the well-known Espresso logic minimizer and show that ROCM is 10 times smaller, executes 10-20 times faster, and uses 3 times less data memory, with a mere 2% quality penalty, for the routing table and access control list applications. We show that ROCM solves real-sized problems on an ARM7 embedded processor in just seconds.

Categories and Subject Descriptors
C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems

General Terms: Algorithms, Performance.

Keywords: Logic minimization, dynamic optimization, on-chip logic minimization, on-chip synthesis, system-on-a-chip, embedded systems.

1. INTRODUCTION
Boolean logic minimization is best known as the main part of logic synthesis, which converts a logic function to a circuit. Logic minimization algorithms were first used to reduce two-level logic functions targeted for programmable logic arrays (PLAs). As programmable logic and synthesis tools matured, two-level logic minimization became used as an individual optimization in multi-level logic synthesis. While many logic minimization algorithms exist, most are very computation intensive and are typically targeted to run on large workstations or servers.

Logic minimization can also be useful in applications other than logic synthesis, such as Internet Protocol (IP) routing table reduction and network access control list (ACL) reduction, which we will describe later. Many of those applications require logic minimization to be run dynamically along with the application, which poses several challenges. Figure 1 highlights two possible methods for performing dynamic logic minimization. In the off-chip approach, the application transmits unoptimized data to a logic minimizer running on a workstation accessible through a communication link such as Ethernet. The workstation optimizes the data and transmits the results back to the application. While an off-chip approach has the benefit of using a powerful workstation-based logic minimizer, the approach's communication overhead can greatly slow the optimization process, especially when optimization is applied frequently.

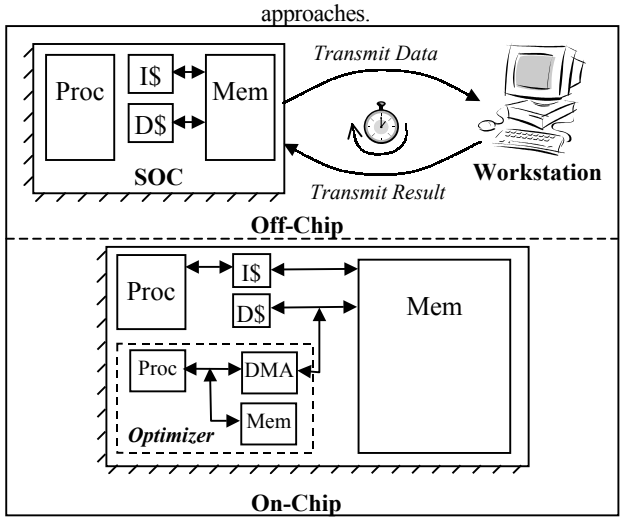
Alternatively, dynamic logic minimization can be performed on-chip by adding an *optimizer* to the chip itself. One approach executes the optimizer as an additional task that shares the same processing resources as the application itself. Another approach implements the optimizer using a separate small embedded processor/memory system, connected to the application processor's memory (perhaps via direct memory access). Both on-chip approaches have the benefit of reducing or even eliminating data transfer between the application and the logic minimizer, resulting in a very fast implementation.

In either on-chip approach, the logic minimizer will have limited processing resources, in terms of processor speed and memory, compared to workstation resources. These limitations create a need for a lean logic minimization tool. In this paper, we describe two modern applications of logic minimization: IP routing table reduction and ACL reduction. We discuss on-chip logic minimization tool requirements. We then describe ROCM (Riverside On-Chip Minimizer), our logic minimization approach targeted for on-chip use, and we compare with Espresso-Exact and Espresso-II. We show further improvements by tuning ROCM to the specific optimization problem of an application.

2. IP ROUTING TABLE REDUCTION
IP network routers route an incoming IP packet to its destination by determining the packet's next hop. The router compares the packet's destination IP address to the router's routing table entries, and selects the entry with the longest matching prefix. For small network routers, searching the routing table entries can be done quickly. However, for larger network routers with tens of thousands of routing table entries, the lookup can be time consuming. Hence, fast IP routing table lookup, either software or hardware-based, has been the focus of much research.

The most common hardware-based lookup techniques incorporate a content-addressable memory (CAM) to perform routing

Figure 1: Comparison of off-chip and on-chip optimization approaches.



DAC 2003, June 2-6, 2003, Anaheim, California, USA.

Figure 2: IP routing table using TCAM's before and after logic minimization.

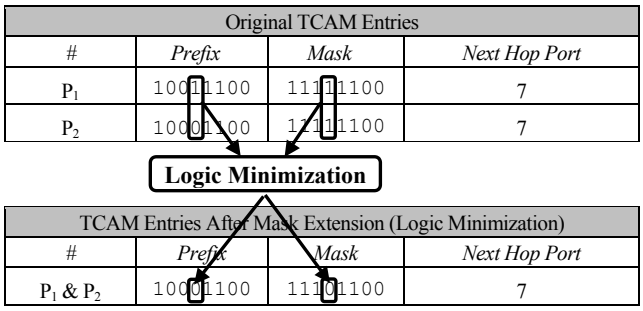


table lookup in parallel. One such approach uses binary CAMs to perform the lookup. Because a binary CAM performs a fixed length match between the input and key, each distinct prefix length requires a separate CAM. Of the matches found within each CAM, the match with the longest prefix is then selected.

Recently, the introduction of ternary CAMs (TCAMs) has enabled hardware-based lookup methods without the need for multiple CAMs. TCAMs operate similarly to binary CAMs, but allow for the storing of a mask that is applied to the input and stored key before comparison. Longest prefix matching can be directly mapped to TCAMs [7]. The approach involves ordering routing table entries from longest to shortest prefix length and storing the entries in the TCAM, where the IP address is stored as the key and the prefix length is stored in the mask as a number of 1s, corresponding to the prefix length, followed by 0s. TCAMs have the drawback of larger size and power than binary CAMs.

To reduce the required TCAM size, Liu [5] used two-level logic minimization to reduce the routing table size, as illustrated in Figure 2, partially taken from [5]. In the original routing table, entries P_1 and P_2 have the same next hop port. Additionally, their prefixes only differ in the fourth bit. Logic minimization combines these two entries by setting the fourth bit of the mask to zero. However, such mask extension cannot be applied to the entire routing table. Instead, the routing table is first pruned to remove redundant entries and partitioned into sets, where each set corresponds to a specific next hop and prefix length. For each set, Liu used Espresso-Exact to perform the mask extension.

Performing mask extension for the entire routing table is time consuming. Therefore, Liu developed an incremental update scheme to handle frequent routing table updates. The incremental update scheme uses the new route being added to the table as the on-set of the logic function and the existing set corresponding to the correct prefix length/next hop port as the don't care set (dc-set). By applying two-level logic minimization as before, two conditions may occur. If the route being added is covered by an existing entry in the table, the new route will be added to the TCAM. Otherwise, the new route will be optimized and all other routes covered by the added route will be removed. Liu's incremental update technique handled an average of 50 updates per second, using Espresso-Exact on a 500 MHz Pentium III processor. However, the update figure did not include the potentially large time for transferring data between the network router and off-chip optimizer. On-chip logic minimization can eliminate the need for such transfers.

3. ACCESS CONTROL LIST REDUCTION

Most commercially available network routers are currently capable of handling Access Control Lists (ACLs). The network routers use information from the incoming IP packet, consisting of *connection type, incoming IP address, incoming port, destination IP address,* and *destination port*, to search the ACL for the first matching entry and the associated action of *permit* or *deny* to be taken. Large ACLs often have thousands of entries, making sequential lookup infeasible. Furthermore, hardware-based parallel lookup approach is limited because the approach must preserve the ordering of the ACL entries.

While discussing the problem of handling large ACLs with a company that designs network router chips, we saw a similarity between ACL processing and IP routing table lookup. ACL entries could be directly stored within a ternary CAM while preserving the list order. However, the size of each ACL entry is over 100 bits. A TCAM large enough to hold tens of thousands of entries requires very large hardware resources. Therefore, we can again use two-level logic minimization to reduce the list size. Before logic minimization, we partition the ACL into sets of non-conflicting entries, corresponding to sequential entries of *permit's* or *deny's*. We further partition these sets into subsets with a maximum of 1000 entries each to achieve faster execution times, as the logic minimization algorithms are super-linear.

4. ON-CHIP LOGIC MINIMIZATION

The Quine-McCluskey method [8] was one of the first exact methods for two-level logic minimization. Most improvements to the method have focused on branch-and-bound techniques and reducing the effort required to generate prime implicants [5] or eliminating the need to explicitly do so [9]. Although exact algorithms are useful, in many cases getting good enough results (near optimal perhaps) in far less time is more important, which has led to the development of heuristic logic minimization tools. These approaches typically start with an initial cover of the logic function and rely upon iterative improvements to achieve good results. While researchers have developed many heuristic minimization methods, one particularly important method is the unate recursive paradigm used by Espresso-II [2].

While researchers have extensively studied both exact and heuristic-based two-level minimization algorithms, they have not explored the usefulness of these algorithms in an on-chip tool. In fact, most of these algorithms strive to reduce the computational time or improve the quality of the results. Most of these algorithms do so at the expense of additional memory requirements and larger code size. The additional memory required is not a big concern in traditional logic minimization that runs on powerful workstations. However, executing logic minimization on-chip imposes new requirements.

4.1 Requirements

Limited available on-chip resources drastically constrain on-chip logic minimizers. Data memory may be very limited. Whereas logic minimizers for desktop processors may use hundreds of megabytes [3], on-chip minimizers must use much less. Instruction memory is also limited. Desktop minimizers may have large program sizes, but on-chip minimizers must be much smaller. On-chip processor speed will also be much slower, as the small embedded processors on which an on-chip minimizer runs is typically about ten times slower than a desktop processor, but on-chip must still execute rapidly.

Obviously, one must expect the results of an on-chip minimizer to be poorer than a desktop minimizer, but those results must still be good enough to satisfy the application's demands.

4.2 ROCM - Riverside On-Chip Minimizer

The general two-level logic minimization problem for heuristic minimizers can be stated as:

> *Given the inputs F (cover of the on-set) and D (cover of the don't care dc-set) of an incompletely specified logic function, determine a cover of F that is minimal, where a minimal cover of F is a cover that is not a proper superset of any other cover of the function.*

In designing our two-level logic minimization algorithm, we employ the techniques used by both Espresso-II [2] and Presto [12]. While Espresso-II produces excellent results, the algorithms employed are not data memory conscious. The most obvious example of large data memory usage is the computation of the off-set. While the use of the off-set for expanding cubes (a binary encoding of the implicants) yields a very efficient algorithm, the size of the off-set can be very large. We therefore chose not to compute the off-set and instead employ a tautology-based approach similar to that used by Presto.

We designed our simple heuristic approach for two-level minimization using a single expand phase implemented with the main goal of very small memory usage and acceptable execution time. Figure 3 presents an overview of our optimization algorithm. First, we order the cubes according to decreasing cube size, under the assumption that larger cubes are more likely to cover other cubes and less likely to be covered by other cubes [2]. The expansion process then selects each cube c of the cover F and expands c to create the cube c'. During the expansion of c, each entry in c is iteratively expanded, ultimately resulting in the expanded cube c'. To check for validity of the expanded cube c', we determine if c' is contained within the current cover F. We perform the validity check by computing the cofactor of F with respect to expanded cube c' ($F_{c'}$), which is then tested to determine if the cofactor is a tautology. If the cofactor is a tautology, the expansion was valid. Otherwise, the expansion was invalid and we revert the expanded cube c' to its previous state. In addition, during the validity check, we will also create a set W' that corresponds to the set of implicants covered by c'. When the expand process is complete, the algorithm returns the resulting cube c' and the set of all cubes covered by the expansion, W. We then compute the current cover F as the union of the previous cover with the expanded cube c' minus the cubes covered by c'.

Most heuristic logic minimizers use the expand operation along with reduce and irredundant transformations to further minimize the cover through repeated execution of an optimization loop. Although a logic minimization tool that uses a single expand phase may not perform well for logic synthesis, the IP routing table reduction application has data sets that contain highly similar entries. The high correlation of the entries results in very good reduction using the expand operation alone. Thus, our first version of ROCM contains only expand, leading to loss of generality, but smaller code size and faster execution.

Figure 3: ROCM optimization algorithm.

```
Optimize(F,D)
{
  OrderCubes(F)
  for i=1 to |F|
  {
    c = F_i
    (c',W) = IterativeExpansion(F,D,c)
         F = (F ∪ c') - W
  }
}

IterativeExpansion(F,D,c)
{
  W = {}
  c' = c
  for i=1 to |c|
    c' = Expand(c',i)
    (val,W') = ValidExpansion(F,D,c')
    if val = true
      W = W ∪ W'
    else
      Revert(c',i)

  return (c',W)
}


ValidExpansion(F,D,c')
{
  CS = F ∪ D
  return Tautology(CS_c')
}
```

5. RESULTS

To determine the feasibility of using ROCM on-chip, we compare ROCM with Espresso-Exact and Espresso-II. Table 1 compares the reduced routing table size, data memory usage, and execution time of the three methods. We performed routing table reduction using mask extension, with routing table information from four large network routers, MaeWest, AADS, Paix, and PacBell [9]. Initially, we obtained all results using a 500 MHz Sun Ultra60 workstation, and we later show results on an embedded processor.

ROCM's code size is an order of magnitude smaller than Espresso, requiring slightly over 1000 lines of C code and only 22 kilobytes of instruction memory. Both Espresso-Exact and Espresso-II are from a single source that is executed using different command line options. Their collective code size is over 11,000 lines of code, resulting in a binary size of 227 kilobytes.

The table also shows that ROCM uses only about one-third the data memory of Espresso-II and Espresso-Exact, using about 1,000 kilobytes compared to over 3,000 kilobytes.

The code and data memory savings come at the expense of ROCM being a bit slower (13%) than Espresso-II. Surprisingly, Espresso-Exact was the fastest method. Normally, the generation of all prime implicants is a very time consuming task, but the high similarity of routing table entries results in relatively few prime

Table 1: Comparison of ROCM with Espresso-Exact and Espresso-II, in terms of data memory usage (*Data*, in kilobytes), execution time (*Time*, in seconds), resulting IP routing table size (*Table size*, in number of entries), and % reduction of the table versus the pruned table (*Reduct.*). Code size is also shown, in kilobytes.

	Initial table size		Espresso-Exact (227 Kb)				Espresso-II (227 Kb)				ROCM (22 Kb)			
	Orig.	**Pruned**	**Data**	**Time**	**Table size**	**Reduct.**	**Data**	**Time**	**Table size**	**Reduct.**	**Data**	**Time**	**Table size**	**Reduct.**
MaeWest	29585	22042	3408	122	16323	26%	3520	217	16327	26%	1048	245	16747	24%
AADS	33740	24795	3520	135	18433	26%	3560	237	18438	26%	1056	259	18898	24%
PacBell	22165	16124	3760	166	11213	30%	3752	271	11221	30%	1080	295	11604	28%
Paix	13914	11091	2056	11	8885	20%	2056	26	8887	20%	1064	30	9000	19%
Average	24851	18513	**3186**	**109**	13714	**26%**	**3222**	**188**	13718	**26%**	**1062**	**207**	14062	**24%**

Table 3: Execution time (in seconds) for incremental routing table updates *(table size in parentheses)* using Espresso-Exact, Espresso-II, ROCM (all on a workstation), and ROCM executing on an ARM7.

	Espresso-Exact	Espresso-II	ROCM	ROCM (ARM7)
24/07 *(420)*	0.42	0.31	0.02	**0.22**
23/01 *(758)*	1.0	0.58	0.04	**0.45**
24/01 *(4080)*	36.6	12.8	0.2	**2.1**

Table 2: ACL reduction (reported in number of entries) and % reduction using Espresso-Exact, Espresso-II and ROCM.

	Orig.	Espresso-Exact		Expresso-II		ROCM	
	Size	Size	Reduct.	Size	Reduct.	Size	Reduct.
Univ	361	229	37%	229	37%	233	35%
Typ1	180	147	18%	147	18%	149	17%
Typ2	128	99	23%	99	23%	99	23%
Bad	359	191	47%	191	47%	200	44%
Long	4425	3406	23%	3406	23%	3536	20%
Average	1091	814	**30%**	814	**30%**	843	**28%**

Table 4: Comparison of data memory usage and execution time for unoptimized ROCM and ROCM-32, and percent savings of ROCM-32 vs. ROCM.

	Data (kilobytes)			Time (seconds)		
	ROCM	ROCM -32	Saving	ROCM	ROCM -32	Saving
MaeWest	1048	936	11%	245	147	40%
AADS	1056	944	11%	259	157	39%
PacBell	1080	984	9%	295	175	41%
Paix	1064	936	12%	30	22	27%
Average	1062	**950**	**11%**	207	**125**	**37%**

implicants compared to the number of entries.

ROCM achieves results very close to the other tools, resulting in only 2% less reduction (24% versus 26%) than Espresso-Exact and Espresso-II. Using ROCM along with the initial pruning step, ROCM achieves an average 43% overall routing table reduction.

Table 1's data is for performing mask extension on the full routing table, which only occurs once, after which we use incremental updates. Table 3 provides execution times for incremental updates for three prefix length/next hop port sets with varying sizes from the optimized AADS routing table. In addition to evaluating the performance on a workstation, we also evaluated ROCM's execution time on an ARM7 processor, a popular and inexpensive embedded 32-bit microprocessor [1], using Triscend's A7 development board that includes a 40 MHz ARM7 processor with an 8 Kbyte instruction/data cache [13]. Due to the generation of all prime implicants by Espresso-Exact and the computation of the off-set and multiple optimization iterations performed by Espresso-II, ROCM outperforms both optimization tools, which required 36.6 seconds and 12.8 seconds respectively for the largest example 24/01 having 4,080 entries. In contrast, the single expand phase of ROCM requires only 0.2 seconds to complete. Interestingly, although the ARM7 clock is over an order of magnitude slower than that of the workstation, ROCM executing on the ARM7 required only 2.1 seconds, which is still significantly less time than both Espresso-Exact and Espresso-II on a workstation.

We also evaluated the usefulness of ROCM for ACL reduction. Table 2 shows the reduced ACL (*number of entries*) and percent savings using Espresso-Exact, Espresso-II, and ROCM for five examples obtained from a router chip design company. *Univ* is an ACL of a university's Computer Science department, *Typ1* and *Typ2* represent typical ACL lists of small size, *Bad* corresponds to an ACL with permit and deny actions frequently interleaved, and *Long* is substantially larger than the other four examples. The ACL reduction by ROCM ranges from 17% to 40%, averaging 28%. Espresso-Exact and Espresso-II provide only an average 2% further reduction.

6. CUSTOMIZING ROCM

In most embedded systems applications, optimizing an algorithm for a particular application is beneficial. One possible optimization is to customize the algorithms and data structures for the particular input size of the application. A customized version of our logic minimizer will require less memory and reduce dynamic memory allocation. We can optimize the algorithms to exploit the known input size to improve performance.

We created a version of ROCM, ROCM-32, optimized for routing table reduction applications that have an input size of 32 bits. Table 4 provides a comparison of data memory usage and execution times for unoptimized ROCM and ROCM-32. ROCM-32 requires an average of 11% less memory and 37% less execution time than the unoptimized version.

7. CONCLUSIONS

We have shown that on-chip logic minimization is feasible. Our ROCM tool has one-tenth the code size and uses one-third the data memory compared to popular on-chip minimizers, while executing fast enough and providing results of sufficient quality (only 2% worse than a powerful desktop logic minimizer) to satisfy our sample networking application's needs. Numerous other applications that can benefit from on-chip logic minimization may exist or evolve. For example, we are presently applying on-chip logic minimization as part of a dynamic hardware/software partitioning approach [11].

8. ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation, grants CCR-9876006 and CCR-0203829, by the Semiconductor Research Corporation, and by a Dept. of Education GAANN fellowship.

9. REFERENCES

[1] Advanced RISC Machines Ltd. ARM7. http://www.arm.com/armtech/ARM7_Thumb/, 2002.

[2] Brayton, R., et al. Logic Minimization Algorithms for VLSI Synthesis. Kluwer Academic Publishers, Boston, MA, 1984.

[3] Cordone, R., F. Ferrandi, D. Scuito, R. Calvo. An Efficient Heuristic Approach to Solve the Unate Covering Problem. Proc. Design Automation and Test in Europe, pp. 364-371, 2000.

[4] Hayashi, T., T. Miyazaki, High-Speed Table Lookup Engine for IPv6 Longest Prefix Match. Proc. IEEE Globecom, Vol. 2, pp. 1576-1581, 1999.

[5] Hlavička, J., P. Fišer. BOOM – A Heuristic Boolean Minimizer. Proc. International Conference on Computer Aided Design, pp. 439-442, 2001.

[6] Liu, H. Routing Table Compaction in Ternary-CAM. IEEE Micro, pp. 58-64, Jan/Feb 2002.

[7] McAuley, A. P. Francis. Fast Router Table Lookup Using CAMs. Proc. Infocom, Vol. 3, pp. 1382-91, 1993.

[8] McCluskey, E. Minimization of Boolean Functions. Bell System Technical Journal, pp. 1417-1444, NY, 1959.

[9] McGeer, P., J. Sanghavi, A. Sangiovanni-Vincentelli. Espresso-Signature: A New Exact Minimizer for Logic Functions. IEEE Transactions on VLSI, Vol. 1, No. 4, pp. 432-440, 1993.

[10] Merit Network, Inc. Internet Routing Table Statistics, http://www.merit.edu/ipma/routing_table/, 2002.

[11] Stitt, G., R. Lysecky and F. Vahid. Dynamic Hardware/Software Partitioning: A First Approach. Proc. Design Automation Conference, 2003.

[12] Svoboda, A., D.E. White. Advanced Logical Circuit Design Techniques. Garland Press, New York, 1979.

[13] Triscend Corporation. A7 CSoC Family. http://www.triscend.com, 2003.