Search Space Optimization for Reversible Logic Synthesis

James Donald and Niraj K. Jha
Department of Electrical Engineering
Princeton University
{jdonald, jha}@princeton.edu

***Abstract*— The Reed-Muller Reversible Logic Synthesis (RMRLS) tool manages to outperform many other reversible synthesis algorithms in terms of speed and scalability. At the same time, its algorithm has been shown to be flexible and extensible. In recent work, it has been used for synthesis with arbitrary gate types. These attributes make RMRLS a favorable foundation for future studies in reversible logic synthesis.**

On the other hand, the algorithm has its imperfections. Although it tends to generate better solutions than other algorithms such as template-matching, even for functions with as few as three variables it can fail to find the optimal solution. This report reexamines some of the basic assumptions of the algorithm to suggest some ideas for better approaching fast and optimal synthesis. We begin by examining results for synthesis of the 40,320 three-variable reversible functions under various configurations. We then use these experiences to formulate several heuristic and fundamental proposals for enhancing RMRLS.

I. Introduction

This study arose from looking into some discrepancies between the results in [1], [5], and the release version of the Reed-Muller Reversible Logic Synthesis (RMRLS) tool. Several of the benchmark synthesis results in [5] were different upon being retested with the release package of RMRLS [4]. However, this was easily explained as the final release of RMRLS had different heuristics from the development version used to obtain various results [3].

Like the synthesis results for various benchmarks, the basic three-variable results were not repeatable either. In this case, all the functions were synthesized using the exhaustive method, so this cannot be explained by a change in pruning heuristics. Although results become non-deterministic if timeouts are encountered, all 40,320 functions synthesize within less than one second, and thus do not come close to reaching the 180 second time limit [5].

Furthermore, the results in Agrawal's earlier paper [1] are actually *better* than those in Gupta's later journal paper [5]. In the development of RMRLS over these past two years, this shift could have been caused indirectly by one of the many changes to the source code.

In attempting to focus on these fundamental aspects of the algorithm, a number of new search options were added to the basic algorithm. These are described below.

- The NOT substitution allows extra substitutions of the form $v \rightarrow v \oplus 1$ to be used regardless of whether the 1 term appears in that variable's PPRM. This was one of the "additional" substitutions described in [5], but after working with the software for a few months it became apparent that such a substitution is not implemented in the release version of RMRLS 0.1.
- The basic algorithm, given an output variable $v_{out,i}$'s PPRM expression, looks for terms in the PPRM expression that do not contain v_i [5]. The $misc$ substitutions even consider terms that do contain v_i, remove the instance of v_i from this term, then generate a Toffoli transformation accordingly. There is not any good fundamental reason to generate such transitions, but this option was easy to implement in the source code.
- The $miniterms$ substitutions were derived when searching for a way to synthesize a minimum quantum cost Fredkin function, which RMRLS would otherwise always fail to do simply because of its search rules. As a result, any candidate factor may create not only one branch, but rather many Toffoli branches using any subset of that candidate factor. This is by the far the most CPU-expensive heuristic. Although it can sometimes reduce gate count, when this heuristic was originally conceived its main intention was to reduce quantum cost while maintaining the same gate count.

This report also looks at the various possibilities for heuristic sorting and pruning. At this time there are at least three known foundations for priority and pruning heuristics:

- Agrawal's settings of $\alpha = 0.2$, $\beta = 0.7$, and $\gamma = 0.1$ [1].
- The settings claimed in Gupta's paper of $\alpha = 0.3$, $\beta = 0.6$, and $\gamma = 0.1$ [5].
- The priority algorithm in the release of RMRLS 0.1, which uses an entirely different system [4].

In Section III we go into further detail on the meaning of α, β, γ, and the other heuristics.

The experiments in this writeup—as well as other insights gathered from experience—lead to several proposals for future research directions in PPRM-based reversible logic synthesis. The rest of this paper is structured as follows. Section II provides an analysis of all three-variable functions under various assumptions for NCT synthesis. Section III examines the priority queue scoring mechanisms in RMRLS. Section IV gives a number of other proposals for future enhancements. Section V concludes.

II. Three-Variable Results

Table I shows the results from synthesizing all 40,320 functions using only NCT gates. It includes some of the various new options, as well as data from prior publications. Only the gate counts are shown; like prior works in reversible logic synthesis [1], [2], [5], [7], we do not consider quantum cost for the 40,320 three-variable functions. All functions were synthesized under exhaustive mode. We increased the timeout limit to one hour when synthesis seemed to be taking on the order of minutes. However, those combinations, which could require as much as a full hour per synthesis instead of merely minutes, are not shown below since it is not expected that their 40,320 synthesis runs will ever finish. After a few weeks, some of the machines running these unfinished tests have crashed. Others have simply remained unfinished.

An additional completed experiment, not shown in the table, was obtained by reenabling Gupta's cleanup-and-restart heuristic in RMRLS 0.1 [5]. The mechanism of this heuristic is for the algorithm to backtrack to an initial state after 2,500 steps. The reason we have not provided an extra column for these results is that the numbers came out to be exactly the same as the default mode (labeled as RMRLS 0.1 in Table I).

Although the first release version of RMRLS is known as both version 0.001 (in the source code) and version 0.1 (in the documentation), throughout this writeup we have opted to refer to it as version 0.1.

There are certainly more combinations to explore, such as $NOT + miniterms$, $misc + miniterms$, or $NOT + misc + miniterms$. Unfortunately, the CPU time can explode when some of these options are used in combination. The $miniterms$ and $NOT + misc$ tests each took several days, which is several times longer than the basic mode which would take about one day to synthesize all 40,320 functions in exhaustive mode. The $NOT + miniterms$ combination took a few weeks, and some other possible combinations such as $NOT + misc + miniterms$ would not be expected to finish within the lifetime one's Ph.D.

The CPU time of $NOT + misc$ is several times that of the basic configuration, but it is still faster than the $miniterms$ substitutions. This configuration, with an average gate count of 6.027, manages to beat Maslov's claimed 6.05 average acquired by template matching applied to RMRLS's synthesis results [5]. Since template matching could be quite CPU-intensive [6], $NOT + misc$ may still be a more viable synthesis method which takes on the order of seconds per benchmark and thus a few days to generate all 40,320 circuits. It is uncertain how this timing compares to that of iterative deepening in [7].

Although the motivation for these experiments was to find sources of discrepancy, none of the modifications came to exactly match either of the earlier published datasets. Thus, these results are inconclusive in that aspect. It is difficult to narrow down such data anomalies. The cause could be anything from a different algorithm rule, special-case heuristic, a bug in the old synthesis tool, a bug in the recent RMRLS, or a bug in the data collection scripts.

One additional issue raised by these experiments is the computational cost of various additional substitutions. In our experience, some very simple substitutions dramatically increase CPU time even for three-variable functions. Furthermore, enabling these substitutions can make larger functions virtually impossible to synthesize. This motivates some better bounding strategies that can tolerate various kinds of substitutions. Section IV details some of these proposals.

III. Priority Queue Heuristics

The previous section examined a number of algorithmic properties of RMRLS for obtaining the best solution in exhaustive mode. In this mode, the use of RMRLS's priority queue policy does not matter significantly except for obtaining the solutions sooner. Regardless of how various nodes are sorted within the priority queue, an exhaustive, non-time-limited mode would be expected to attempt all of the same possible solutions.

When running in time-limited, greedy, or non-exhaustive default modes, however, RMRLS's priority and pruning mechanisms play a significant role. Many of the benchmarks featured in each paper can only be synthesized in greedy mode, and some provide their best circuits in the non-exhaustive default mode. Therefore, the priority and pruning rules are significantly emphasized in [1] and [5].

The priority value for a given node is set at $\alpha(depth) + \beta(elim/depth) - \gamma(num\_literals)$. In [1], $\alpha = 0.2$, $\beta = 0.7$, and $\gamma = 0.1$. In [5], $\alpha = 0.3$, $\beta = 0.6$, and $\gamma = 0.1$. Although in both cases the three factors were intentionally set to add up to 1, there is no good or intuitive reason for this restriction.

The release version of RMRLS, however, does not use either of these configurations [4]. According to the source code, this algorithm is significantly more complicated. For one, there is no depth term, so effectively $\alpha = 0$. The β term is there in a sense, but not as a fixed quantity. When $depth < 10$, $\beta = 1.0$. When $depth \geq 10$, $\beta = 0.3$.

As for γ, the factor multiplied by $num\_literals$, there is actually no fixed value for this purpose. Instead, a record is kept on the best partial solution known for each depth. If the current node is the first at its depth to have a literal count as low as its own, it gets a special bonus of 0.6 added to its score. In addition, if it has the least number of literals out of any partial solution seen so far at any depth, there is an additional bonus of 0.6.

One problem with the above literal counting heuristic is that the priority of a node is time-dependent on when it is first analyzed. This can cause unwanted side-effects of causality. For example, when synthesizing $cycle28\_2$ under the $k = 1$ heuristic with a max-depth of 80, the 56-gate solution is obtained quickly. When using $k = 1$ and a max-depth of 60, one would expect this to be faster as it involves a smaller search space. Under these settings, however, no solution is obtained. One possible explanation is that due to the time-dependency of the priority rules, the larger search space somehow gives priority to the path leading to the 56-gate solution, whereas the smaller search space fails to land on this path.

The heuristics described in the papers, which are not implemented in the release version of RMRLS, do not appear to have such causality problems. On the other hand, the fact that Gupta used different heuristics in the release version suggests that these more complex techniques may have been providing better results. Because the source code for earlier incarnations of RMRLS was not publicly released, we cannot easily compare the simple heuristics to the current ones.

TABLE I

NCT SYNTHESIS RESULTS FOR ALL THREE-VARIABLE REVERSIBLE FUNCTIONS ACCORDING TO VARIOUS RMRLS PUBLICATIONS AND OUR TESTS.

# gates	Agrawal [1]	Gupta [5]	RMRLS 0.1	RMRLS 0.2 NOT	RMRLS 0.2 $misc$	RMRLS 0.2 $miniterms$	RMRLS 0.2 $NOT+misc$	RMRLS 0.2 $NOT+miniterms$	Optimal [7]
9	30	36	92	44	28	36	16	36	
8	3,297	3,351	4,168	3,427	2,598	3,019	2,169	3,019	577
7	12,488	12,476	12,560	12,503	12,578	12,159	12,293	12,141	10,253
6	13,620	13,596	12,939	13,479	14,135	13,952	14,561	13,952	17,049
5	7,503	7,479	7,224	7,485	7,602	7,736	7,857	7,754	8,921
4	2,642	2,642	2,597	2,642	2,639	2,678	2,684	2,678	2,780
3	625	625	625	625	625	625	625	625	625
2	102	102	102	102	102	102	102	102	102
1	12	12	12	12	12	12	12	12	12
0	1	1	1	1	1	1	1	1	1
Avg.	6.100	6.104	6.159	6.108	6.065	6.071	6.027	6.070	5.866

A possible approach for research on improved priority and pruning mechanisms would be exploration of various combinations of the 6 different priority mechanisms covered in this section. To recap, these mechanisms include not only the basic α, β, and γ factors, but also a variable β dependent on depth, the strange 0.6 priority bonus for the least literals at each different depth, and the other 0.6 bonus for the least literals overall.

Thus far we have focused on the priority queue mechanisms. The priority queue rules all affect pruning implicitly, as low-priority nodes are more likely to get pruned. However, in the non-exhaustive default mode there is an additional direct pruning parameter k. This variable sets the maximum number of new nodes to create from each parent. [5] states that k values of 4 or 5 perform best, while the release version uses a value of $k = 4$. It is important to realize that for only a few benchmarks does the default non-exhaustive mode actually do better than exhaustive or greedy modes. Although not explicitly advertised, this was the general experience when obtaining results for [2]. In other words, the pruning with $k = 4$ is a property of an overall policy that does not perform very well.

We additionally experimented somewhat with the case $k = 1$. Our first impression was that this should function identically with greedy mode. This interpretation seems intuitive because greedy mode pushes at most one new node onto the queue at each step. Since results using $k = 1$ with the default mode differ from greedy mode, these two modes are apparently not identical. $k = 1$ enables the 56-gate solution for $cycle28\_2$ and 27-gate solution $cycle15\_2$, which had seemed otherwise unobtainable. Since these good solutions cannot be obtained with the greedy or exhaustive modes (with the default heuristics in the release version of RMRLS 0.1) for at least some functions there is evidently some potential for this unusually simple heuristic option of $k = 1$ in the default mode.

The k heuristic (pruning for fewer branches at each stage) works by sorting all candidates by their priority then only pushing the first k nodes onto the priority queue. One strange characteristic of this routine is that it sorts the candidates from lowest priority value to highest priority value and then prunes away the higher-valued nodes. Since the priority formulas were designed for higher values to symbolize better node potential, this appears to incorrectly prune good nodes. This is perhaps an oversight, and possibly much better results may be achieved by sorting in the opposite direction.

Tweaking the value of k, changing the sort direction, and adjusting the other six priority tuning parameters mentioned earlier form a total of 8 different parameters to tune the existing heuristics. Each one of these has some evidence that it may do good in some situations. Thus, using just the techniques that have been partially tested already, there is already an enormous design space for which exploration could potentially yield a much better combination of techniques for practical priority sorting and pruning.

Before attempting to tune these complex heuristic settings, however, it may be better to first implement some fundamental improvements in the core algorithm such as the enhancement described in Section IV-A.

IV. Other Proposals

While the previous section examined some ideas for improving heuristic sorting and pruning, this section proposes some more bold changes to the algorithm. Many of these are motivated by the poor runtime aspects observed in the experiments from Section II.

A. Duplicate Node Detection

We use the term *bounding* distinctly from *pruning*, where pruning refers to heuristic pruning of potential nodes even though sometimes a poor pruning choice may be made. In algorithm terminology, branch-and-*bound* techniques reduce the search space by applying strict bounds. The use of the $bestDepth$ variable in RMRLS is an example of this, since if we are optimizing for circuit size there is no point in exploring solutions that are guaranteed to have more gates than the $bestDepth$. When configured to minimize the quantum cost, the $bestCost$ variable plays this same role of bounding.

The explosion in CPU time with additional substitutions naturally occurs because of an exponential increase in potential nodes for exploration. However, many of these nodes are probably duplicates created when different paths result in the same PPRM expressions. With the NOT substitution in particular, two NOT gates end up creating the original node with increased depth. If this happens at a depth lower than the current $bestDepth$, it cannot be bounded using the current system. Although the original algorithm's system of Toffoli candidate factors often prevents an excess of duplicates, this cannot be guaranteed with additional gate types.

Thus, we propose duplicate node detection and filtering. Pointers to all nodes should be stored in a hash table where their search key is based on their PPRM expressions. Upon attempted creation of a new node, its PPRM expression is first checked in the hash table. If there is already a matching node with the same or better depth, there is no need to push this new term onto the priority queue. If the new node has a better depth, the old node should be deleted from the queue and replaced in the hash table.

This trimming can be applicable in any of the three modes (greedy, default, or exhaustive), although it will probably provide the most benefit in the default and exhaustive modes.

B. Trimming the Back of the Queue

Whenever the algorithm obtains a new solution, and thus updates $bestDepth$ (or $bestCost$ in the case of minimizing quantum cost), there is an additional opportunity for bounding. At this point, the algorithm should go to the back of the priority queue and trim off all unapplicable nodes which are guaranteed to be useless by the new value of $bestDepth$ (or $bestCost$). Furthermore, by reporting the number of solutions eliminated at these points, the algorithm can give the user a better idea of how the search space is being reduced.

C. Bidirectional and Divide-and-Conquer Synthesis

Many other papers use bidirectional algorithms, yet their algorithms still do not perform as well as RMRLS. It is also possible to extend RMRLS to use coarse-grain bidirectional synthesis. If this results in some better solutions in some of the three-variable cases, it can give some insight into some of RMRLS's unidirectional properties.

The currently known procedure for bidirectional synthesis consists of synthesizing the input function, simulating it for various inputs, using EXORCISM to generate the PPRM expressions for the reverse function, then finally synthesizing the reverse function.

Here we propose something faster which would work well with RMRLS. The PPRM expression can be generated directly from a netlist by calling some existing routines. Just as calling the substitution functions (e.g. $substitute()$, $fredkin\_substitute()$, or $peres\_substitute()$) in succession can turn a PPRM specification into the identity PPRM expression, while calling these routines in reverse can change the identity function back into the original PPRM expression. Furthermore, calling the substitution functions in order, but starting with the unique identity PPRM expression, can actually generate the PPRM expression of the $reverse$ of the original function. There is an amazing duality between *substitution* and *operation* of algebra that we can exploit.

In other words, by using the existing $substitute()$ functions, rather than having to write new Boolean manipulation routines, the reverse synthesis process can be automated.

In addition to synthesizing functions in reverse, this technique enables many more synthesis approaches. One could take an existing specification of many gates generated through the greedy search, generate the PPRM specification for only its first half of gates, then perform a more exhaustive synthesis on just that first half-function. If a better specification is found for the half-function, it can be replaced. This broad approach of taking existing suboptimal circuits and optimizing them resembles the philosophy of template matching, yet it does not require a template library!

D. Garbage Outputs

There could be many naive strategies for efficiently dealing with garbage outputs. The current setup requires that the garbage outputs be preassigned to certain values. Working in this framework, one could simply try various combinations of assignments. However, the number of possible garbage assignments grows quite rapidly with the number of garbage inputs as well as total inputs.

Our idea for assigning garbage outputs involves the synthesis algorithm continually keeping track of garbage lines and exercising the freedom to modify these lines. When an additional gate change the PPRM specifications such that only the garbage lines are affected, this substitution can be considered "for free" and not increase the current $depth$. This way, the garbage outputs will also possibly make it easier to zero in on a solution when substitutions can push potential smart solutions taking advantage of garbage to the top of the priority queue.

One problem with this proposed approach is the directionality of the algorithm. Garbage should be monitored from the start so that these "free" substitutions can be used early. However, RMRLS synthesizes functions from their input to output. Until the output is reached, it may not be known which lines are affected by garbage. Thus, the garbage bit management approach as described here can only be applied by synthesizing functions in reverse. Before attempting the efficient assignment of garbage outputs using the technique proposed in this section, it would be best to first implement the bidirectional synthesis framework as described in the previous section.

V. Conclusions

The numerical studies presented in Section II provide insight into the shortcomings of the basic algorithm. The primary motivation for testing these various configurations was to find the cause of differences between various three-variable results from different versions of RMRLS. Since all of the test options—including one based on a rule presented in [5]—were unable to create results matching either of the original two papers, we can conclude that none of these options alone can fully explain the various discrepancies.

Some additional heuristic options, such as the $misc$ substitution, dramatically increase the CPU time of the algorithm. A major cause of this problem is the number of duplicate expressions in the search tree generated indirectly by such heuristics. The problem of this excessive computational overhead motivates the need for the duplicate detection mechanism described in Section IV. This report has also proposed a number of other enhancements that may dramatically improve the synthesis power of RMRLS, including proper experimentation on the existing heuristics, bidirectional synthesis, and efficient assignment of garbage outputs.

References

[1] A. Agrawal and N. K. Jha, “Synthesis of Reversible Logic,” in *Proc. Design, Automation, & Test in Europe Conf.*, vol. 2, Feb. 2004, pp. 1384–1385.

[2] J. Donald and N. K. Jha, “Reversible Logic Synthesis with Fredkin and Peres Gates,” *ACM Journal on Emerging Technologies in Computing Systems*, 2007.

[3] P. Gupta, personal communication, 2006.

[4] ——, “RMRLS 0.1,” available for download: http://www.princeton.edu/~cad/, 2007.

[5] P. Gupta, A. Agrawal, and N. K. Jha, “An Algorithm for Synthesis of Reversible Logic Circuits,” *IEEE Trans. CAD*, vol. 25, no. 11, pp. 2317–2330, Nov. 2006.

[6] D. Maslov and G. W. Dueck, “Toffoli Network Synthesis With Templates,” *IEEE Trans. CAD*, vol. 24, no. 6, pp. 807–817, June 2005.

[7] V. V. Shende, A. K. Prasad, I. L. Markov, and J. P. Hayes, “Reversible Logic Circuit Synthesis,” in *Proc. IEEE/ACM Conf. Computer-Aided Design*, Nov. 2002.