Xquasher: A Tool for Efficient Computation of Multiple Linear Expressions

Arash Arfaee[†], Ali Irturk[†], Nikolay Laptev[‡], Farzan Fallah[††], Ryan Kastner[†]

[†] Department of Computer Science and Engineering
University of California, San Diego
{aarfaee, airturk, kastner}@cs.ucsd.edu

[‡] Department of Computer Science
University California, Los Angeles
nlaptev@cs.ucla.edu

[††]Engineering Department
Envis Corporation, CA, 95054
Farzan@envis.com

***Abstract*— Digital signal processing applications often require the computation of linear systems. These computations can be considerably expensive and require optimizations for lower power consumption, higher throughput, and faster response time. Unfortunately, system designers do not have the necessary tools to take advantage of the wide flexibility in ways to evaluate these expressions. Therefore, we address the problem of efficiently computing a set of linear systems through a tool, Xquasher, that is developed by us to enable elimination of large common subexpression from expressions with an arbitrary number of terms. Xquasher provides a methodology for efficient computation of both single and multiple linear expressions. We also introduce the concept of power set encoding which helps us to provide an effective optimization method and achieves significant improvement over previously published work. Our tool provides optimized designs with 15% less area with the cost of 3% increase in delay by reducing number of additions on average by 45%.**

I. Introduction

Computation of *multiple linear expressions (MLE)* and its sub-computations: *single constant multiplication (SCM)*, *multiple constant multiplications (MCM)* and *linear expression (LE)* are commonplace in linear systems. Many digital signal processing applications such as Finite Impulse Response Filters and Discrete Fourier Transform use linear systems which are expensive and therefore are the dominant factor in the overall performance. It is often advantageous to convert costly constant multiplications into a corresponding set of shifts and additions which can lead to lower power consumption, higher throughput, and faster response time. By careful common sub-expression elimination over these expressions, one can also reduce the total number of additions.

However, current synthesis techniques perform limited transformations, and are unable to do an adequate optimization of these expressions. A major obstacle to the optimization of linear systems is that system designers do not have the necessary tools to take advantage of the wide flexibility in ways to evaluate these expressions. In most cases, designers rely on hand tuned library routines (for software) or Intellectual Property (IP) blocks (for hardware) to implement these computations. The drawback to using these approaches is that the given library routines and IP blocks may not be ideally suited for the platform with respect to the use and/or the constraints specified by the application. Designing a high level tool for the optimization of the linear system computation is crucial.

DAC'09, July 26-31, 2009, San Francisco, California, USA

Therefore, we developed a tool, Xquasher, which provides a general methodology for linear system optimization. Xquasher performs common subexpression elimination to minimize the area of the resulting hardware of the linear system. Our tool is also efficient for optimization of sub-MLE problems: SCM, MCM and LE. Xquasher utilizes a novel powerset encoding format to enable efficient extraction of large common subexpressions which makes it possible to achieve substantial area improvements over previously published works.

In this paper we are mainly focused on reducing the area by reducing the total number of additions. The optimal solution with the minimum possible number of additions is a well known NP-complete problem. By careful analysis of effective factors in this problem and providing novel solutions for each of them, we generate a heuristic algorithm that efficiently reduces number of additions. Our results shows substantial improvement over existing methods [3][4][5][7].

The remainder of this work is organized as follows. The subsequent section formalizes the problem and its definitions. Section III describes our tool, its methodology which uses powerset encoding. Section IV provides experimental analysis of our results and comparisons with previously published works. We conclude in Section V.

II. Problem Formulation and Definitions

This section is devoted to introduction of multiple linear expression (MLE) and its sub-problems: single constant multiplication (SCM), multiple constant multiplication (MCM) and linear expression (LE) where we illustrate the cost of the computation with a three-dimensional space and define some basic terms: *dot*, *line*, *page* and *space* that are used in our methodology for ease of understanding.

A. Definitions

Canonical Signed digits (CSD) is radix-2 signed digit representation with digit set of $\{-1,0,1\}$ where there is no adjacent non-zero digit in the representation. **Bit-Magnitude (BM)** is an integer with the magnitude of $\pm 2^{\mathbb{N}}$. It can be described as a CSD number with exactly one non-zero digit. **Dot** is an appearance of an input variable in our expression with a relative magnitude of BM. In hardware, this corresponds to rewiring an input to specific point in the circuit with possible some shifts to the right. **Line** is summation of multiple copies of one variable multiplied by a vector of BMs. Therefore, computation of a line is Single Constant Multiplication (SCM). **Page** is formed by a summation of a set of lines. We present examples for page, lines, dots and BMs in

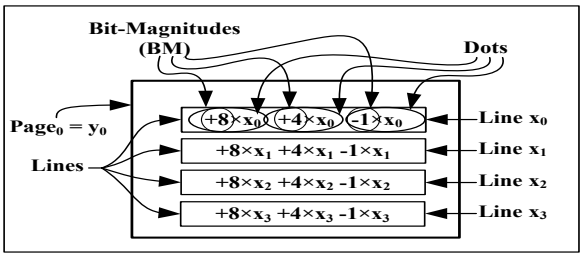

Fig. 1. An example of page and its lines, dots and BMs. Dots are shown with ellipses and in the first line, a circle inside each dot demonstrates the BM part.

Figure 1. **Space** can be created by union of a set of pages. In our problem, a space is a collection of multiple pages. **Cost of Computation (CoC)** is defined as the number of additions that appears in our object. *Example (Figure 1):* $CoC(Line_{x_0}) = 2$. **Object** refers to *a dot*, *a line*, *a page* or *a space*.

B. Problem mapping

In this subsection, we formulize the optimization of a linear system. Our approach is to minimize the number of additions. We first show how to optimize small objects in our space and then apply and improve this methodology for larger objects.

Dot optimization: A dot by itself does not have any cost by definition. They are simply rewiring of copies of input data. We do not perform any optimization over a dot by itself.

Line optimization: Problem of line optimization is known as single constant multiplication (SCM). Modern tools generally use CSD encoding as a solution for this problem. Using a CSD representation with minimum Hamming weight provides a good but not optimal solution in shift and add approach for constant by variable multiplication and therefore is used in our examples as well. Our experiments with Mentor Graphics and Synopsys tools indicate that the area of SCM mirrors the number of additions between non-zero digits in a CSD encoding. Although CSD can be considered as a good solution for optimization of lines, it does not guarantee an optimum solution since further optimization is possible to reduce number of additions inside a line. To see how this is possible, assume a line is given as $Line_1 = 792 \times x_0$ with CSD representation of $(10\bar{1}0010\bar{1}000)_{CSD}$ where a bar in top of a digit means that it is negative. After rewriting corresponding CSD bit values as integer, we have the following vector multiplication:

$$Line_1 = [1024 \quad -256 \quad 32 \quad -8] \times x_0$$
$$= 1024 \times x_0 - 256 \times x_0 + 32 \times x_0 - 8 \times x_0$$

This is the way that modern tools perform this multiplication, and CoC of this multiplication is 3 additions. However, it is possible to optimize this multiplication further by rewriting it:

$$Line_1 = 32 \times (32 \times x_0 - 8 \times x_0) + (32 \times x_0 - 8 \times x_0)$$
$$= (32 + 1) \times (32 \times x_0 - 8 \times x_0)$$

where CoC becomes 2 additions. Elimination of repeated dots by finding common sub-expressions (CS) and Common sub-expression Elimination (CSE) are well known methods for CSM optimization [5][7].

Page optimization: Optimization of a page by applying CSE is similar to the optimization of a line. However searching relatively larger set of dots and applying CSE demand more complex and careful analysis. Therefore, we investigate these issues in more detail and provide a general solution for page optimization. We assume that all lines are encoded with CSD. Assume a page is given as:

$$Page_0 = Line_0 + Line_1 + Line_2 + Line_3$$
$$= +686x_0 + 241x_1 + 991x_2 + 686x_3$$

The un-optimized page results in *4+2+2+4+3 = 15* additions. We can further optimize this page by rewriting it as:

$$Page_1 = +241x_1 + 991x_2 + 686D_0$$
$$D_0 = x_0 + x_3$$

where we can decrease the number of additions to *2+2+4+2+1 = 11* by detecting and eliminating an obvious CS:$+686(x_0 + x_3)$. Further optimization is possible by rewriting each line with equivalent CSD encoded version where more CSs become observable:

1024,0,-256,0,-64,0,16,0,0,-2,0

$$Page_0 = +(-2x_0 + 16x_0 - 64x_0 - 256x_0 + 1024x_0)$$
$$+(+1x_1 - 16x_1 + 256x_1)$$
$$+(-1x_2 - 32x_2 + 1024x_2)$$
$$+(-2x_3 + 16x_3 - 64x_3 - 256x_3 + 1024x_3)$$

If we optimize $Page_0$ by limiting CSs to have exactly two dots, one of the possible results is:

$$Page_2 = +256x_1 + 1024x_2 - 16D_0 - 256D_0 + 1024D_0 + 1D_1 - 1D_2 - 32D_2$$
$$D_0 = +1x_0 + 1x_3, D_1 = +1x_0 - 16x_1, D_2 = +1x_2 - 1D_0$$

where the CoC decreases from 15 to 10. Considering the previous example, one might suggest writing details of lines might lead to a better optimization; however it is more time consuming and hard to determine which CS is the best candidate to eliminate. For example, there are 120 combinations of two dots that could be a CS just in the first step of CSE. In our last attempt over this example, in first and second CSE we ignore details in $Line_0$ and $Line_3$.

$$Page_3 = +256x_1 - 1x_2 - 32x_2 + 1024x_2 + 686D_0 + 1D_1$$
$$D_0 = +1x_0 + 1x_3, D_1 = +1x_1 - 16x_1$$

In the third step we expand $686D_0$ to equivalent CSD encoded version. $Page_4$ is the result of CSE optimization after this step.

$$Page_4 = +256x_1 + 16D_0 - 256D_0 + 1024D_0 + 1D_1 - 1D_2 - 32D_2$$
$$D_0 = +1x_0 + 1x_3, D_1 = +1x_1 - 16x_1, D_2 = +1x_2 + D_0$$

CoC of $Page_4$ is 9, and we have 40% reduction in CoC from $Page_0$'s. Considering that we just needed 3 CSE and much less complexity in the CSE process to gain the same result, it might be a better approach. Here we just have 28 possibilities for CSs with two terms. It seems that CSE process can get much simpler and faster if we can work with dot representation and constant multiplication form of the line.

Space optimization: We design our tool, Xquasher, for space optimization by introducing two new concepts: **1)** *determination of* the *effective factors in optimization of the run-time* and **2)** *common sub-expression fragmentation (CSF).*

1) *Determination of the effective factors in optimization of the run-time:* Run-time is the main challenge in all CSE algorithms. Finding the optimum solution for an NP-complete problem requires an unacceptable run-time. Heuristic algorithms are well known to be useful in tackling NP-complete problems [3][4][5]. Therefore, Xquasher uses a greedy heuristic algorithm where run-time of the algorithm depends on two factors: **1)** Number of iterations and **2)** Run-

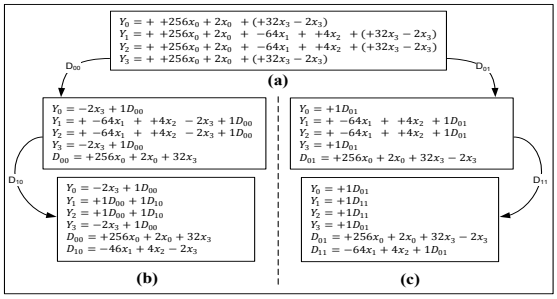

Fig. 2. (a) represents a space with four pages. We show different optimization approaches with (b) and (c). In (b), CS are limited to have exactly three dots. In (c), CSs can have arbitrary large number of dots.

time of each iteration. The algorithm that is used in Xquasher can eliminate larger sets of CSs in a single CSE compared to the algorithms that are used in [4-5]. Our algorithm provides a solution with fewer iteration due to the optimization of larger sets of CSs where [4-5] requires multiple iterations for the same optimization. Xquasher is also capable of seeing both dots and sets of dots in a line as a part of CSs which increases the run-time of each iteration but still an effective factor in the first couple of iterations.

Being able to detect and eliminate a CS which contains large sets of dots causes significantly fast reduction of number of dots in the target space. Therefore, the number of remaining dots will be lower for the next iteration. However, finding proper CSs and CSE at run-time is highly dependent on the number of terms inside CSs. The format of CS in our tool is two sets of dots (two lines) instead of two or three dots. Being able to work with sets of dots, allow us to use a lower order algorithm compared to [4-5] and gain lower CoC.

2) *Common sub-expression fragmentation (CSF):* Our optimization approach is to eliminate CSs with large set of dots. CSE of CS with few limited, two or three, dots can easily prevent elimination of larger CSs and result in series of separated dots such that no optimization over them would be possible. Figure 2 (a, b and c) shows examples of effective and ineffective ways of optimization. Given space consists of four pages (a) and CoC for this space is *16* before performing the optimization.

In (b), we assume that all CSs must contain exactly three dots. We perform CSE for $D_{00} = (-256x_0 + 2x_0 + 32x_3)$ and $D_{10} = (-64x_1 + 4x_2 - 2x_3)$ as the first and second steps respectively. CoC is reduced to *8* after these two optimizations. As mentioned before, elimination of large set of dots may result in better optimization. In (b), we assume that there is no limit for number of dots in a CS. Therefore, we perform CSE for $D_{01} = (-256x_0 + 2x_0 + 32x_3 - 2x_3)$ and $D_{11} = (-64x_1 + 4x_2 + D_{01})$ as the first and second steps respectively. As a result of the CSEs CoC's is reduced to 5 which is a lower cost compared to the first approach of optimization. The reason for this decrease in the cost is that although dot $(-2x_3)$ appeared in all four initial pages (a), in two of these pages dots remain unengaged in all CSEs after the first optimization approach (b). The second approach prevents fragmentation on CSs by considering CSs with more dots.

III. XQUASHER'S METHODOLOGY

The best hand coded optimization for many common linear systems requires the elimination of CSs with large set of dots. Previous works are not able to handle such elimination due to the exponential increase in the cost of considering such CSs. Although our methodology is not able to eliminate all possible large CSs because of the increase in the cost, it is still capable of covering large CSs when CSs' dots are focused inside one or two lines.

A. Power Set Encoding

To convert an integer to PSE, we follow this procedure:

1- Encode an integer, I, to a CSD minimum hamming weight format.

2- Rewrite the CSD encoded number as a series of integer addition where each number has exactly one of the non-zero digits of the original number.

3- Generate power set from step 2's result set.

4- Ignore the null set and generate a new set from the result of step 3 where each element is the sum of numbers inside the related set.

5- Rewrite each number from the result of 4th step as $2^k \times p$, where p is an odd integer, and store these numbers in the following format:

$$[p, k, number\ of\ non-zero\ digit\ in\ CSD\ format]$$

B. Xquasher Algorithm

Here, we present the Xquasher algorithm and describe it below. After encoding all input constants to PSE (1), Xquasher searches for a CS that has the highest number of relevant dots in all appearance of that CS (3-8). Xquasher limits the CS to two PSDs (4-5) and performs CSE for the chosen CS (9), where it updates the related pages to the CS and appends a new page, representing the eliminated CS (10). These steps continue in a loop until there is no more CSE left to be eliminated (2).

```
1  convert all constants to PSE
2  While there is a CS
3     for any page in the space
4        for any PSD1 in any line1 in any page
5        for any PSD2 in any line2 appeared after line1 in any page
6              find number of occurance of CS =
                     (PSD1, PSD2) in the space
7                 if (number of ocuurance) ×
       (number of non − zero digit in PSD1
       +number of non − zero digit in PSD2) is max in the space
8                 best_CF = (PSD1, PSD2)
9     CSE (best_CF)
10    Update PSE values
```

IV. RESULTS

Using the methodology described in Section III along with Synopsys Design Compiler Ultra and TSMC 90nm library, we synthesize several common linear system examples using several techniques. The linear systems that we use as examples are 8-point DCT, 8-point Inverse DCT (IDCT), Discrete Fourier Transform (DFT), Discrete Hartley Transform (DHT), and Discrete Sine Transform (DST) and also three FIR filters: EP24 (6-tap FIR), BT24 (20-tap FIR)and LS24 (41-tap FIR). Xquasher takes a matrix of coefficients as input and then uses our methodology to generate a new set of expressions. Xquasher also generates an HDL code based on these expressions. Therefore, we compare our results with the previously published work in terms of number of additions and synthesis results: area and delay.

Table I presents a comparison between the previously published works [1][2][5] and Xquasher in terms of total number of additions and a comparison between [5] and

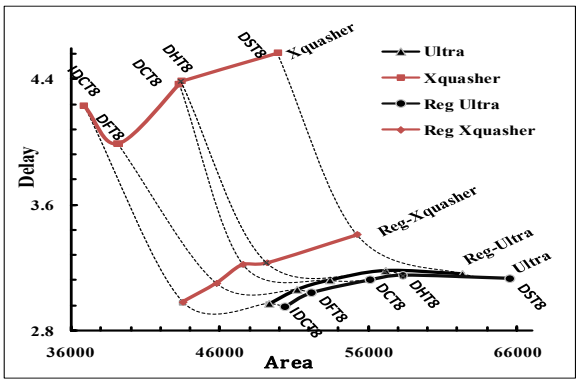

Fig. 3. Synthesis results for benchmarks. Reg Xquasher results shows 15% less area in average with small increase (~3%) in delay compared to the Compile Ultra results[6].

Xquasher in terms of synthesis results: area and delay. As can be seen from Table I, Xquasher provides significant reduction in the number of additions by a careful optimization compared to the previous works. Since the method in [5] is the closest to our results, we take the comparison to another level by presenting the synthesis results of both methods. It can be seen from Table I that Xquasher provides better results in area and as well as delay in our benchmarks.

As can be seen from Table I, previously reported best results [5] are clearly inferior to those provided by Synopsys Compile Ultra since they are significantly worse in both area and delay. To the best of our knowledge and based on our tests, the quality of results for Synopsys' Compile Ultra optimization [6] is the best available. Therefore we compare Xquasher with the results from Compile Ultra in terms of area and delay in Figure 3. Table II also shows this analysis in percentages in terms of area and delay. Xquasher's non-registered results give a significant area reduction (22%) compared to Compile Ultra, though this comes at a price of significant increase in delay (40%).

We also generate a synchronous (Registered) version of our code and compare registered versions of both optimization methods in column 3-4 of Table II. By using registers at the end of each adder tree to save the results by replacing all expressions with registers instead of wires, we compromised partial area saving to gain less delay. Therefore, Xquasher provides 14% decrease in area compared to registered non-optimized version while the average delay penalty is decreased to 3%.

V. Conclusion

Linear systems are prevalent in digital signal processing.

TABLE II
Summary of results of our tool, Xquasher, compared to Synopsys Compiler.

	Xquasher vs. Ultra		Xquasher vs. Ultra (Registered)	
	Delay	**Area**	**Delay**	**Area**
dct8	-41.42%	19.00%	-2.55%	15.23%
dft8	-30.39%	23.49%	-1.97%	12.28%
dht8	-38.49%	23.99%	-3.19%	15.70%
dst8	-46.01%	19.93%	-8.95%	15.76%
idct8	-42.42%	25.21%	-1.02%	13.59%
Average	-39.74%	22.32%	-3.53%	14.51%

We developed a tool, Xquasher, that can effectively find and eliminate higher order common subexpression terms. Our tool provides enormous area reductions at the cost of increasing the critical path; 22% reduction in area with the cost of 40% increase in delay. However, we can easily add registers to our design, which yields 15% decrease in area and a minor increase in delay (3%) compared to registered version of the non-optimized design.

VI. References

[1] M. Potkonjak, M.B. Srivastava, and A.P. Chandrakasan, "Multiple Constant Multiplications: Efficient and Versatile Framework and Algorithms for Exploring Common Subexpression Elimination," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 1996.

[2] H. T. Nguyen and A. Chattejee, "Number-splitting with shift-and-add decomposition for power and hardware optimization in linear DSP synthesis," *IEEE Transactions on Very Large Scale Integration (Vlsi) Systems*, vol. 8, pp. 419-24, 2000.

[3] H. Safiri, M. Ahmadi, G.A. Jullien, and W.C. Miller, "A New Algorithm for the Elimination of Common Subexpressions in Hardware Implementation of Digital Filters by Using Genetic Programming," *IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, 2000.

[4] A. Hosangadi, F. Fallah, and R. Kastner, "Common subexpression elimination involving multiple variables linear DSP synthesis," *IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pp. 202-12, 2004.

[5] A. Hosangadi, F. Farzan, and R. Kastner, "Optimizing high speed arithmetic circuits using three-term extraction," *Design, Automation and Test in Europe*, pp. 6, 2006.

[6] R. Zimmermann, D. Q. Tran, "Optimized Synthesis of Sum-of-Products," *Proceedings 37th Asilomar Conference on Signals, Systems, and Computers*, November 2003

[7] Peter Tummeltshammer, James C. Hoe and Markus Püschel, "Time-Multiplexed Multiple Constant Multiplication", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 26, No. 9, pp. 1551-1563, 2007.

TABLE I
Comparison of previously published work and Xquasher in terms of total of additions. Also synthesis results for [5] and our tool.

Example	Number of Additions					Synthesis Results			
						[5]		XQUASHER	
	Original	**[1]**	**[2]**	**[5]**	**XQUASHER**	**Delay**	**Area**	**Delay**	**Area**
H.264	86	N/A	N/A	63	**53**	4.34	25284.83	**3.50**	**13216.59**
DCT8	274	227	202	188	**161**	5.76	100221.30	**4.37**	**43275.86**
IDCT8	242	222	183	164	**140**	5.33	65135.55	**4.23**	**36907.11**
EP24	26	N/A	N/A	16	**13**	4.49	6394.87	**2.76**	**2272.03**
DST	320	252	238	N/A	**181**	N/A	N/A	**4.57**	**49902.86**
DHT	248	211	209	N/A	**161**	N/A	N/A	**4.39**	**43454.38**
BT24	106	N/A	N/A	48	**48**	6.04	26738.11	**3.12**	**9456.45**
LS24	232	N/A	N/A	112	**99**	5.77	37630.09	**3.17**	**21368.39**