Minimization of Transitions by Complementation and Resequencing using Evolutionary Algorithms

Rolf Drechsler Nicole Drechsler

Institute of Computer Science
University of Bremen
28359 Bremen, Germany
email: drechsle@informatik.uni-bremen.de

ABSTRACT

Power consumption of digital circuits is closely related to the switching activity. With more and more battery driven devices, as PDAs or laptops, the low power aspect is becoming one of the main issues in circuit design. In this context, the *Data Ordering Problem with Inversion (DOPI)* is of importance. Data words have to be ordered and (eventually) negated such that a minimal number of transitions occurs. This problem has several applications, like instruction scheduling, sequencing of test patterns, or cache write-back.

We present an *Evolutionary Algorithm (EA)* for the DOPI. The EA is compared to the best previously published methods. Using EAs results of the same quality as the exact method have been determined, but with much lower run times. In contrast to the exact algorithm, the EA can also be applied to larger examples and furthermore outperforms former heuristic approaches.

KEY WORDS

Evolutionary Algorithms, Digital Circuit Design, Low Power, Data Ordering Problem, Transition Minimization

1 Introduction

Until the late 90s the main optimization goal for circuit and system synthesis was area and delay, i.e. to design small and fast circuits (see e.g. [4]). But with smaller circuit structures and more and more battery driven devices, the power consumption also has to be considered from the very first moment of the design process. For this, recently several approaches for low power design have been proposed [1, 23, 13, 18].

There exist several (completely different) approaches in low power design: On the hardware side designers try to reduce the switching activity in the circuit and also try to reduce the voltage of the circuit [22, 5]. On the other hand the software is also important for the power consumption of a circuit [20, 21]. The improvement only by software is up to 40%.

The power consumption on the software level is closely related to the switching activity that is directly related to the ordering of data words and their (possible) inversion. The first problem of finding an ordering that minimizes the transitions (*DOP=data ordering problem*) has been considered in [15], where it was proven that the related decision problem is NP-complete. Heuristics have been discussed and several extensions of the approach to other applications have been outlined. Motivated by the *bus-invert method* [19] DOP can be extended to also consider inversion of data words. The resulting problem is called DOPI, i.e. *DOP with inversion* [16] (a formal definition is given below in Section 2).

As a general method for solving optimization problems *Evolutionary Algorithms* (EAs) [12] are getting more and more popular. Recently, EAs have successfully been applied to several problems in VLSI CAD (see e.g. [7, 14, 8]). In [9] an EA has been proposed that generates very high quality solutions to the DOP. This approach has been extended to also consider not fully specified data words, i.e. to also consider don't cares, in [6]. But the underlying problems were based on reordering only - similar to the *Travelling Salesman Problem* (TSP).

In this paper we propose a new EA that can solve the more general DOPI. The ordering of the words and the inversion is optimized in parallel. We give comparisons to several alternative methods for DOP and DOPI on experiments. It is shown that the EA approach outperforms all other techniques. Over random orderings, in some cases an improvement of more than a factor of five can be observed, while the EA is never worse than any other technique. This has a direct application in instruction scheduling, where the power consumption is directly related to the switching activity.

The paper is structured as follows: A formal problem definition is given in Section 2. Previously proposed optimization algorithms are briefly reviewed and discussed in Section 3. In Section 4 the new algorithm based on evolutionary techniques is described. Experimental results are given in Section 5, and finally the paper is summarized.

2 Problem Formulation

We start with a formal definition of the *Data Ordering Problem* (DOP) [15] and the *DOP with Inversion* (DOPI) [16].

Let w_i ($i \in \{1,..,m\}$) be a bit-string of length n. Let $w_i(l)$ denote the l-th bit of the string w_i.

Definition 1 The *distance* d (also called the Hamming distance) of two bit-strings w_i and w_j is given by

$$d(w_i, w_j) := |\{l | w_i(l) \neq w_j(l), 1 \leq l \leq n\}|.$$

Example 1 Let w_1, w_2 be two bit-strings of length 4 given by:

$$w_1 = 0011, \quad w_2 = 0101$$

Then the distance is $d(w_1, w_2) = 2$.

The DOP is given as follows:

> *Find a permutation σ of the bit-strings $w_1,..,w_m$, i.e. a 1-to-1 function $\sigma : \{1,\dots,m\} \to \{1,\dots,m\}$ such that*
>
> $$\sum_{i=1}^{m-1} d(w_{\sigma(i)}, w_{\sigma(i+1)}) \qquad (1)$$
>
> *in minimized.*

Definition 2 The *inversion* of a data word w_i is defined as the bit-wise complement of its elements.

Remark 1 The inversion of a data word w_i is also called the *negation* and is denoted as $\overline{w}_i$. The *polarity* (i.e. a phase assignment) of a data word w_i is denoted as α_i and determines whether w_i is inverted ($\alpha_i = 1$) or not ($\alpha_i = 0$).

Example 2 Let w_1 be a bit-string of length 4 given by:

$$w_1 = 0011$$

Then $\overline{w}_1 = 1100$.

Inversion can only be applied to the complete word, not to single bit positions. Then, DOPI is defined as follows:

> *Find a permutation σ of the bit-strings $w_1,..,w_m$ and a polarity α_i for each w_i (that determines whether w_i is inverted or not), such that*
>
> $$\sum_{i=1}^{m-1} d(w_{\sigma(i)}^{\alpha_i}, w_{\sigma(i+1)}^{\alpha_{i+1}}) \qquad (2)$$
>
> *in minimized.*

Here, α_i (for all i) denotes a Boolean value that is one, if the data word is inverted and zero otherwise.

3 Previous Approaches

The DOP is similar to the *Travelling Salesman Problem* (TSP), one of the most intensive studied problems of complexity theory. In both problems a good ordering of elements with respect to a given weight between each two elements has to be determined. For this, several TSP heuristics developed have been applied to the DOP. (All exact methods can only handle small problem instances, since DOP is NP-complete [15].) The heuristics with the corresponding references are given in the following:

1. *Double Spanning Tree heuristic* (DST) [10]
2. *Spanning Tree/Minimum Matching heuristic* (ST-MM) [10]
3. *Greedy Min heuristic* (GM) [15]
4. *Greedy simple* (GS) [9]

DST and ST-MM both guarantee upper bounds that are within a factor of 2 from the minimum. Nevertheless, experiments have shown the greedy heuristics outperform DST and ST-MM on randomly generated examples (see [15, 9]). For this, we restrict ourselves in the following to a comparison to GM and GS. To make the paper self-contained we briefly review these methods:

Greedy Min (GM): Compute the Hamming distance for all (distinct) pairs of given data words and select the pair with the minimum distance. Then construct a sequence of the words by extending the already existing sequence from one element of the pair depending on the minimum Hamming distance and continue until all words are integrated in the sequence. The greedy algorithm GM has quadratic runtime in the number of data words. For a real-time hardware implementation (as suggested in [15] for several applications) GM is in some cases too area consuming and the complexity of the hardware realization becomes too large.

Greedy Simple (GS): As an alternative an other simple greedy algorithm which has a linear runtime in the number of data words can be defined by restricting GM, i.e. apply GM only to the pairs resulting from the initial ordering. Obviously, the quality of GS largely depends on the initial ordering of the data words. But as has been shown in [9], GS obtains better results than random orderings, and in contrast to GM it has a simpler hardware realization.

In [16] GM has been extended to also consider inversion by inverting data word w_i, if $\alpha_i = 1$, $1 \leq i \leq m$. The resulting heuristic is denoted by GMI in the following. For this, in the algorithmic flow of GM α_i is determined by computing the Hamming distances for all pairs of data words with inversion and without. Then the order of the data words is determined analogously to GM.

4 Evolutionary Algorithm

In this section we describe the *Evolutionary Algorithm* (EA) that is applied to the problem given above. In Sections 4.1 to 4.5 the submodules of our EA (like the genetic operators) and the overall structure of the algorithm is described. Finally, the detailed choices for the parameter settings are discussed.

4.1 Representation

Most often EAs work on binary strings and the problem to be solved is encoded in a two-valued representation. In our application we have to store two types of information per data word: the ordering position and the polarity.

1. **Ordering position:** We use an integer encoding to represent the ordering of the data words. (A binary encoding would require special repair operators to avoid the creation of invalid solutions similar to the TSP, see e.g. [24].) Each integer vector of length m represents a permutation of the data words and thus it is a feasible ordering.

2. **Polarity vector:** For each data word a polarity must be chosen. Thus, an m-dimensional binary vector is assigned to each individual. Value **1** (**0**) means that the corresponding data word has (not) to be inverted.

A population is a set of elements. Each element in the population contains two vectors: one for the ordering position called σ-vector and one for the polarity of each word called α-vector. The length of both vectors is given by the number data words m, because for each word the ordering position and polarity has to be stored.

To the vectors σ and α different genetic operators are applied (see Section 4.4).

4.2 Objective Function and Selection

As an *objective function* that measures the *fitness* of an element we use the value determined from Formula (2). $\sigma(i)$ determines the ordering position of data word i and $\alpha(i)$ gives the corresponding polarity.

The selection is performed by *roulette wheel selection* and we also make use of *steady-state reproduction* [3]: Some of the best elements of the old population are included in the new one anyway. This strategy guarantees that the best element never gets lost and a fast convergence is obtained. (EA practice has shown that this method is usually advantageous.)

Example 3 Throughout the example the following data words are considered:

data word	*sequence*	*d*
0	10101	
1	10110	2
2	11101	3
3	11000	2
4	01001	2
5	11101	2
		$\sum$ 11

Column *data word* gives the number of considered data words and column *sequence* shows its sequence. In column *d* the Hamming distances from the considered data word to its predecessor is given. The initial Hamming distance of the set of data words is 11.

In Figure 1 the EA-representation of a solution is illustrated. The first row represents the ordering of the data words, i.e. the σ-vector, and the second row the polarity, i.e. the α-vector, respectively. Value 5 at array position 2 means that data word 5 is at position 2 in the new ordering.

ordering	σ	0	2	5	1	4	3
polarity	α	0	1	0	0	0	0

Figure 1. Representation of data sequence

For fitness computation the sequence given above is reordered and inverted. Notice, that data word 1 is inverted as denoted with an *i* in the following table. The resulting Hamming distance is 5 and the new sequence is given by

data word	*sequence*	*d*
0	10101	
2	11101	1
5	11101	0
i 1	01001	2
4	01001	0
3	11000	2
		$\sum$ 5

4.3 Initialization

At the beginning of each EA run an initial population is randomly generated. First, random permutations for the σ-vectors are generated as follows: We start with an array of numbers from 1 to n in increasing order. To each element in the array a random integer number is assigned. Then the random numbers are sorted using *shell sort*. This results in a permutation of the entries of the initial array. The method guarantees that only feasible solutions are generated and additionally it performs very fast. The initial α-vectors are random bit strings. Then to each element a fitness is assigned corresponding to Formula (2).

Often it is helpful to combine EAs with problem specific heuristics [3, 7, 8]. The resulting EAs are called *Hybrid EAs*. In our application we combined the EA with the greedy algorithm GMI (see Section 3). The initial population in the EA is further optimized by applying a modified version of GMI to $|\mathcal{P}|/4$ elements, where $\mathcal{P}$ denotes the number of elements in the population. This guarantees that the starting points are not too bad and thus the convergence is speeded up. The application of GMI itself is time consuming, but forces a faster convergence.

4.4 Evolutionary Operators

We now introduce the genetic operators that are used. Operators that modify the ordering of the data words are applied to the σ-vector of an element and genetic operators for bit strings are applied to the α-vector. The choice of the operators for the σ- and α-vector depend on probabilities and they are applied independently of each other.

4.4.1 Operators for the σ-vector

In our application we use three different types of crossover operators, i.e. *Partially Matched Crossover* (PMX), *Cycle Crossover* (CX) and *Ordered Crossover* (OX). These methods create two children from two parents. The parents are selected by the method described above. The operators choose two cut positions at random.

Notice that a simple exchange of the parts between the cut positions (as often applied to binary encoded EA problems) is not possible, since this would often produce invalid solutions. The three operators differ in the strategies to *validate* the offsprings after the exchange:

PMX [11]: Construct the children by choosing the part between the cut positions from one parent and preserve the absolute position and order of as many variables as possible from the second parent.

OX [2]: Construct the children by choosing the part between the cut position from one parent and preserve the relative position and order of as many variables as possible from the second parent.

CX [17]: A variable at a position of a child must retain from exactly the same position of one of the parents[1].

For more details about the crossover operators see [11, 2, 17, 7].

Additionally, three different *mutation operators* are used:

Mutation (σ-MUT): Select a parent element at random and choose one position. The value of the variable at this position is used to determine the position of the variable with which it is exchanged.

2-time Mutation (σ-MUT2): Perform σ-MUT two times on the same parent.

Mutation with neighbor (σ-MUT-N): Select a parent element at random and choose one position. Then perform σ-MUT at this and the neighboring position.

Example 4 **σ-MUT:** $p = 321$ and $i = 1$ is the mutation position $\rightarrow c = 123$

[1]The name derives from the fact, that for the construction of the children *cycles* of positions of the parents must be found.

data_ordering_with_inversion (*data words*) {
  generate_random_population () ;
  initialize_population () ;
  apply_GMI_to_initial_population () ;
  {
    apply_operators () ;
    calculate_fitness () ;
    update_population () ;
  } while (*not terminal case*) ;
  return (*best_element*) ;
}

Figure 2. Sketch of basic algorithm

4.4.2 Operators for the α-vector

The operators used for the modification of the polarity are described in the following. Three *standard* crossover operator are applied to the α-vector of an individual:

Crossover (C1): Construct two new elements c_1 and c_2 from two parents p_1 and p_2, where the first (second) part of c_1 (c_2) up to a cut position i is taken from p_1 and the second (first) part is taken from p_2.

2-time Crossover (C2): Construct two new elements c_1 and c_2 from two parents p_1 and p_2, where the first (second) part of c_1 (c_2) up to a cut position i is taken from p_1 (p_2), the second part up to a second cut position $j > i$ is taken from p_2 (p_1) and the last part is again taken from p_1 (p_2).

Uniform Crossover (CU): Construct two new elements c_1 and c_2 from two parents p_1 and p_2, where at each position the element is taken with a given probability from p_1 and p_2, respectively.

Analogously to Section 4.4.1 three mutations are used:

Mutation (MUT): Construct one new element from a parent by copying the whole element and randomly changing its value at randomly chosen position i.

2-time Mutation: Perform MUT two times on the same element.

Mutation with neighbour: Perform MUT at two adjacent positions on the same element.

Obviously, these evolutionary operators only generate feasible solutions if they are applied to binary strings.

Table 1. Small examples

$\|words\|$	n	DOP				DOPI	
		rand	*exact*	GM	EA	GMI	IEA
10	5	20	11	12	11	11	9
10	10	43	25	27	25	23	21
10	20	86	60	63	60	57	57
10	40	170	141	148	141	132	128
10	60	280	230	236	230	227	225
10	80	352	301	306	301	292	289

4.5 Algorithm

Using the operators introduced above our EA works as follows:

- Initially a random population of elements consisting of a σ- and an α-vector is generated and optimized by GMI as described in Section 4.3.
- The crossover operators described in Sections 4.4.1 and 4.4.2 are applied to selected elements and create $|\mathcal{P}|/2$ offsprings. The elements are chosen according to their fitness (see Section 4.2). The offsprings are then mutated by the mutation operators with a given probability. After each iteration the size of the population is constant. Finally, the offsprings are optimized by GMI with a probability of 25%.
- The algorithm stops if no improvement is obtained for 5000 iterations.

A sketch of the algorithm is given in Figure 2.

4.6 Parameter Settings

The size of the population is constantly set to 50. PMX, CX and OX are applied in parallel during the run of the EA. C1, C2 and CU are carried out with probabilities 60%, 30% and 10%, respectively. The different mutation operators, (σ-)MUT, (σ-)MUT2 and (σ-)MUT-N, are carried out on the newly generated elements with a probability of 15%.

5 Experimental Results

The algorithms described above have been implemented in *C++* and all experiments are carried out on a *SUN Ultra-Sparc* workstation. For all experiments randomly generated sets of data words of different size and with varying bit-length are studied. For comparison, we consider DOP and DOPI for all test cases.

In a first series of experiments we considered small problem instances, i.e. instances for which the exact algorithm for DOP could be applied. For DOP the search space size is $|words|!$, since all permutations of the words are considered. For DOPI the search space is larger by a factor of $2^{|words|}$, since all inversions have to be taken into account. In the first two columns of Table 1 the problem size is given by the number of words and the length n of each word. The next two blocks report the results for DOP and DOPI, respectively. For DOP random ordering, the exact method, the greedy algorithm GM and the EA are reported. GM including inversion (column GMI) and INV-EA are given for DOPI in the last two columns. As can be seen, the best results over all methods are determined by INV-EA.

Table 2. Large examples

$\|words\|$	n	DOP			DOPI	
		rand	GM	EA	GMI	IEA
40	5	97	28	23	21	18
40	10	177	92	79	74	71
40	20	396	246	235	220	123
40	40	781	576	570	537	527
40	80	1528	1244	1223	1208	1193
80	10	374	150	144	123	110
80	20	800	449	432	401	393
80	40	1558	1084	1062	1021	1004
80	80	3118	2440	2408	2353	2331
80	100	4062	3122	3088	3044	3018
100	10	471	175	166	134	127
100	20	1001	527	514	483	470
100	40	1963	1316	1296	1252	1232
100	80	3889	3025	2973	2891	2858
100	100	5077	3869	3832	3721	3694
100	150	7387	6098	6050	5951	5921

In a second series of experiments larger problem instances have been considered. For these the exact algorithm cannot be applied due to overlong runtime. The results for the heuristic approaches are given in Table 2. Again, INV-EA determined the best results. Compared to GMI, for each test case the result could be further improved - in some cases by nearly 10%. Of course, this can become significant, e.g. for applications like instruction scheduling, since the codes once generated can be applied frequently in micro controllers or DSPs without any further modification required.

6 Conclusions

We presented an *Evolutionary Algorithm* for transition minimization in data words. For solving the underlying optimization problem, called DOPI, in parallel an ordering and a polarity per vector has to be determined. The EA outperformed all other techniques with respect to quality of the final result.

The techniques finds several applications in low power optimization, like instruction scheduling, test pattern sequencing or scheduling in high-level synthesis. It is focus of current work to apply the developed approach in these domains.

References

[1] A.P. Chandrakasan, M. Potkonjak, J. Rabaey, and R.W. Brodersen. HYPER-LP: A system for power minimization using architectural transformations. In *Int'l Conf. on CAD*, pages 300–303, 1992.

[2] L. Davis. Applying adaptive algorithms to epistatic domains. In *Proceedings of IJCAI*, pages 162–164, 1985.

[3] L. Davis. *Handbook of Genetic Algorithms*. van Nostrand Reinhold, New York, 1991.

[4] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, Inc., 1994.

[5] S. Devadas and S. Malik. A survey of optimization techniques targetting low power VLSI circuits. In *Design Automation Conf.*, pages 242–247, 1995.

[6] N. Drechsler and R. Drechsler. Exploiting don't cares during data sequencing using genetic algorithms. In *ASP Design Automation Conf.*, pages 303–306, 1999.

[7] R. Drechsler. *Evolutionary Algorithms for VLSI CAD*. Kluwer Academic Publisher, 1998.

[8] R. Drechsler and N. Drechsler. *Evolutionary Algorithms for Embedded System Design*. Kluwer Academic Publisher, 2002.

[9] R. Drechsler and N. Göckel. A genetic algorithm for data sequencing. *Electronic Letters*, 33(10):843–845, 1997.

[10] M.R. Garey and D.S. Johnson. *Computers and Intractability - A Guide to NP-Completeness*. Freeman, San Francisco, 1979.

[11] D.E. Goldberg and R. Lingle. Alleles, loci, and the traveling salesman problem. In *Int'l Conference on Genetic Algorithms*, pages 154–159, 1985.

[12] J.H. Holland. *Adaption in Natural and Artifical Systems*. The University of Michigan Press, Ann Arbor, MI, 1975.

[13] S. Iman and M. Pedram. Multi-level network optimization for low power. In *Int'l Conf. on CAD*, pages 372–377, 1994.

[14] P. Mazumder and E. Rudnick. *Genetic Algorithms for VLSI Design, Layout & Test Automation*. Prentice Hall, 1998.

[15] R. Murgai, M. Fujita, and S.C. Krishnan. Data sequencing for minimum-transition transmission. In *IFIP Int'l Conf. on VLSI*, 1997.

[16] R. Murgai, M. Fujita, and A. Oliveira. Using complementation and resequensing to minimize transitions. In *Design Automation Conf.*, pages 694–697, 1998.

[17] I.M. Oliver, D.J. Smith, and J.R.C. Holland. A study of permutation crossover operators on the traveling salesman problem. In *Int'l Conference on Genetic Algorithms*, pages 224–230, 1987.

[18] W.-Z. Shen, J.-Y. Lin, and F.-W. Wang. Transistor reordering rules for power reduction in CMOS gates. In *ASP Design Automation Conf.*, pages 1–6, 1995.

[19] M. Stan and W. Burleson. Limited-weight codes for low-power I/O. *Int'l Workshop on Low Power Design*, 1994.

[20] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: A first step towards software power minimization. In *Int'l Conf. on CAD*, pages 384–390, 1994.

[21] V. Tiwari, S. Malik, A. Wolfe, and M. Lee. Instruction level power analysis and optimization of software. In *VLSI Design Conf.*, 1996.

[22] C. Tsui, M. Pedram, and A.M. Despain. Technology decomposition and mapping targeting low power dissipation. In *Design Automation Conf.*, pages 68–73, 1993.

[23] H. Vaishnav and M. Pedram. PCUBE: A performance driven placement algorithm for low power design. In *European Design Automation Conf.*, pages 72–77, 1993.

[24] D. Whitley, T. Starkweather, and D. Fuquay. Scheduling problems and traveling salesman: The genetic edge recombination operator. In *Int'l Conference on Genetic Algorithms*, pages 133–140, 1989.