VLSI Implementation of Greedy based Distributed Routing Schemes for Ad Hoc Networks

Alberto Aloisio*, Paolo Branchini† Vincenzo Izzo‡, Salvatore Rampone§

Abstract

We describe a VLSI implementation based on a FPGA of a new greedy algorithm for approximating minimum set covering in ad hoc wireless network applications. The implementation makes the algorithm suitable for embedded and real-time architectures.

INTRODUCTION

A wireless ad hoc network consists of a collection of mobile hosts dynamically forming a temporary network without the use of any existing network infrastructure. In such a network, each mobile host can serve as a router all the nodes in the network. Building an infrastructure for ad hoc network that guarantees reliable communication is an important problem, as portable wireless devices continue to grow in popularity and in capabilities. In such networks, all of the nodes are mobile and so the infrastructure for message routing must be self-organizing and adaptive. Finding efficient and effective routing schemes is a challenging problem in the area. Ad hoc wireless networks are here represented by a connected graph where all the links are bi-directional. Several researchers have used minimum connected dominating sets to do routing in ad hoc wireless networks [1], [2], [3], [4], [5], [6], [7]. The dominating set induces a virtual connected backbone. However finding a minimum connected dominating set is an NP-complete problem [8].

Some authors have proposed approximation algorithms for obtaining minimal connected dominating sets [8]. The problem can also be viewed as a minimum set cover approximation [9]. The Set Covering is one of the oldest and most studied NP-hard problems [10]. Given a ground set U of m elements and a collection S of subsets of U, the aim is to find the smallest subcollection of S whose union is U. A weight may be assigned to each subset, and in this case (weighted Set Covering Problem) the aim is to minimize the sum of the subcollection weights. Since SCP is very common in practical applications, a number of exact and heuristic algorithms have been proposed in literature. A well known polynomial time scheme for approximating the SCP is the greedy one [9][11].

B. Liang and Z. J. Haas [12] use a greedy algorithm with redundancy elimination to obtain a d-dominating set. Rather than seeking an optimal solution, they apply the greedy algorithm to obtain a fairly small d-dominating set, and then apply redundancy elimination to reduce this slightly.

Recently, a new greedy algorithm for approximating minimum set cover has been presented [13]. The algorithm, while not randomized, is based on a probability distribution that leads the greedy choice. It shows very good empirical performances and it has successfully been applied in wireless network applications [14] [15], where greedy algorithms for set cover produce sets of nodes that can be used to form a virtual backbone for an ad hoc wireless network. While efficient implementations are given [13], the cost of probability distribution evaluation can still be unaffordable in real-time applications. Therefore a version of the algorithm has been specifically tailored to run on platforms with minimal computational hardware [16].

In this paper we describe a VLSI implementation which makes the algorithm suitable for embedded and real-time architectures in ad-hoc wireless network applications. The implementation is based on a FPGA, short for Field-Programmable Gate Array, a logic programmable chip. It is specially popular for prototyping integrated circuit design. This hardwired implementation makes the algorithm suitable for embedded and real-time architectures [17][18].

This paper is organized as follows. In Section II we introduce a probability distribution over the SCP subsets and we use it to define a greedy approximation algorithm for SCP. In Section III we modify the algorithm selection criteria for real time applications. Then, in Section IV, we introduce the hardware implementation. Test results on a set of benchmarks are described in Section V.

SCP PROBABILITY DISTRIBUTION

Let $U = \{u_1, \ldots, u_n\}$ be a finite set, and let $S = \{S_1, \ldots, S_m\}$ be a collection of subsets of U, such that $\cup S_i = U$. The problem posed is to select $S^* = \{S_{i_1}, \ldots, S_{i_k}\}$, $S^* \in S$, to be a *cover* of U, that is $\cup S_{i_k} = U$, such that $|S^*|$ is minimum.

Greedy Selection Criteria

The classical greedy algorithm for approximating a minimum cover, at each step, simply chooses the subset S_i

* Università di Napoli "Federico II" – Dipartimento di Scienze Fisiche and INFN – Napoli, Italy (e-mail: *aloisio@na.infn.it*).

† Università degli studi "Roma Tre" – Dipartimento di Fisica and INFN – Rome, Italy (e-mail: *branchini@roma3.infn.it*).

‡ Università di Napoli "Federico II" – Dipartimento di Scienze Fisiche and INFN – Napoli, Italy (e-mail: *izzo@na.infn.it*).

§ Università del Sannio – Research Centre on Software Technology (RCOST) and DSGA – Benevento, Italy (corresponding author phone: +39-0824-305151; fax: +39-0824-23013; e-mail: *rampone@unisannio.it*).

which includes (*covers*) the largest number of remaining elements of U, deletes these elements from U, and repeats this process until the ground set $U$is empty. In case of a tie, i.e. when several subsets have the same covering size, the set with smaller subscript is usually chosen.

This greedy choice is in effect random, and it can be viewed as implicitly based on a probability assignment on the subsets, such that

$$P(S_i|U) > P(S_k|U) \Leftrightarrow |S_i \cap U| > |S_k \cap U| \quad (1)$$

for $i, k \in \{1, \ldots, m\}$.

However, in the general case this kind of assignment can be improved. As an intuitive explanation, if there is only one subset S_i in S that covers an element u_j, that subset will be surely included in any cover of U independently of its size.

By extension, a subset S_i covering an element u_j should be included in a cover with a probability decreasing in the number of alternatives, i.e. in the number of subset S_k of S such that $u_j \in S_k$.

Probability-driven Algorithm

The underlying probability distribution can be formalized as follows. For each element u_j and each subset S_i, we define the probability of S_i given u_j as

$$P(S_i|u_j) = I_{S_i}(u_j) / \sum_{k=1}^{m} I_{S_k}(u_j) \quad (2)$$

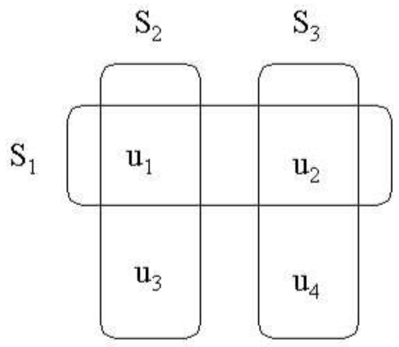


Figure 1: A simple SCP instance

where $I_{S_i}(u_j)$ is the characteristic function of the S_i set, that is 1 if $u_j \in S_i$ and is 0 otherwise.

By taking the mean on U, the probability of a S_i set with respect to the ground set U is

$$P(S_i|U) = (1/n) \sum_{j=1}^{n} P(S_i|u_j) \quad (3)$$

We apply (3) to define a greedy approximation algorithm for SCP as follows: choose the subset S_i whose probability with respect to U is maximum, delete the elements of S_i from U, and repeat this process until the ground set U is empty.

It has been shown that applying the probabilistic choice correspond to approximate the solution of an equivalent SCP instance in a classical greedy way.

```
procedure PGreedy
  Input: $U = \{u_1, u_2, \ldots u_n\}$; $S = \{S_1, S_2, \ldots S_m\}$;
  Init.: Set $den_j = \sum_{k=1}^{m} I_{S_k}(u_j)$, $j \in \{1, \ldots, n\}$;
         $p_i = \sum_{j=1}^{n} \frac{I_{S_i}(u_j)}{den_j}$, $i \in \{1, \ldots, m\}$; $S^* = \emptyset$;
  Step:  repeat
           1.1 Select: Choose $S_q \in S$ such that $p_q$ is maximum;
           1.2 Update: Set $S^* = S^* \cup \{S_q\}$; $p_i = p_i - I_{S_i}(u_j)/den_j$,
               $i \in \{1, \ldots, m\}$, $j \in \{j | u_j \in S_q \cap U\}$;
               $U = U - S_q$;
         until $|U| = 0$
  Output: $S^*$.
```

Figure 2: A scheme of the PGreedy Procedure

Example

Let $U = \{u_1, \ldots, u_4\}$ and $S = \{S_1 = \{u_1, u_2\}; S_2 = \{u_1, u_3\}; S_3 = \{u_2, u_4\}\}$ (Fig. 1). By using (3), we have $P(S_1|U) = 1/4$; $P(S_2|U) = 3/8$; and $P(S_3|U) = 3/8$. The set S_2 is selected, and u_1 and u_3 are erased from U. In the next step $P(S_1|U) = 1/4$; and $P(S_3|U) = 3/4$. The S_3 choice completes the cover, resulting in $S^* = \{S_2, S_3\}$.

On the contrary, the classical greedy algorithm outputs a cover $S^* = \{S_1, S_2, S_3\}$. Actually all the S_i's have the same covering size, hence in the first iteration the set with the smaller subscript, S_1, is chosen. Then, in order to cover u_3 and u_4, both S_2 and S_3 must be chosen.

Implementation: PGreedy

In order to give a low cost implementation, first we note that the quantity $1/n$ in (3) is just a normalization factor, and so we can disregard it in the real computation. In this way the summation (3) does not need to be recomputed in each algorithm iteration, but just to be updated by subtracting the $P(S_i|u_j)$ values related to the covered elements.

Then, by fixing once and for all the quantities

$$den_j = \sum_{k=1}^{n} I_{S_k}(u_j), \quad j = 1, \ldots, n \quad (4)$$

we avoid to store the values $P(S_i|u_j)$. It is easy to see the $P(S_i|u_j)$ values are fixed to $1/den_j$ if $u_j \in S_i$ and to 0 otherwise. A scheme of the resulting procedure (*PGreedy*) is reported in Fig. 2.

REAL TIME VERSION: RTGREEDY

In real time applications the main remaining bottleneck of the described algorithm is the probability computation. In fact, the floating point divisions required by (2) and (3) are too time consuming in environments with limited hardware facilities.

Here, since the method application is mainly devoted to information transmission, without loss of generality, we call the sets S_i's as *Sources* and the elements u_j's as *Destinations*.

For the real time application we substitute the probability distribution by a set of weights. Let us define the set of *Destination Weights*

$$DW_j = 2^{m-\sum_i D_{i,j}} \tag{5}$$

where

$$D_{i,j} = I_{S_i}(u_j) \tag{6}$$

These coefficients resemble the effect of the probability values. In fact, it is easy to see that 2^m in (5) is a multiplication factor of $1/2^{\sum_i I_{S_i}(u_j)}$. By taking the inverse of the latter quantity and applying the logarithm we have $\sum_i I_{S_i}(u_j)$, that is den_j (4).

So $P(S_i|U_j)$ can be represented by $D_{i,j} * DW_j$ and $P(S_i|U)$ by the *Source Weight*

$$SW_i = \sum_j D_{i,j} * DW_j \tag{7}$$

In this way we avoid any floating point arithmetic in the application. We call the resulting algorithm *RTGreedy*.

A scheme of the resulting procedure is reported in Fig. 3.

HARDWARE IMPLEMENTATION

The RTGreedy algorithm has been specifically optimized to solve the Set-Cover problem in real-time applications. It makes only use of bit-wise logic operators and simple finite-state machines and it is suitable to run on platforms with minimal computational resources, like embedded microprocessor or custom logic.

In order to test the achievable performance, we have designed a straightforward hardware implementation based on a Virtex-II XC2V2000 Xilinx FPGA. The logic has been described with synthesizable VHDL language. The code is parametric and it allows us to implement the algorithm for different data-set size.

The input data-set is a matrix of $|S|$ rows and $|D|$ columns where each element i, j is $D_{i,j}$ as in (6).

```
procedure **RTGreedy**

Input:  A matrix of $|S| = m$ rows and $|D| = n$ columns where each element $i, j$ is $D_{i,j}$ as in $(6)$.
Init.:  Set $DW_j = 2^{m-\sum_i D_{i,j}}$, $j \in \{1, \ldots, n\}$;
        $SW_i = \sum_{j=1}^{n} D_{i,j} * DW_i$; $S^* = \emptyset$;
Step:   **repeat**
            1.1 *Select:* Choose $S_q$ such that $SW_q$ is maximum;
            1.2 *Update:* Set $S^* = S^* \cup \{S_q\}$; $SW_i = SW_i - D_{i,j} * DW_i$, $i \in \{1, \ldots, n\}$, $j \in \{j | u_j \in S_q \cap U\}$; $U = U - S_q$;
        **until** $|U| = 0$
Output: $S^*$.
```

Figure 3: A scheme of the RTGreedy Procedure

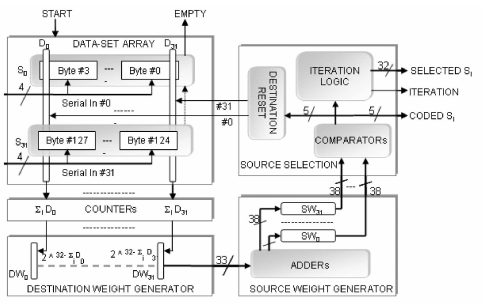


Figure 4: Block diagram of the FPGA implementation

A simplified block diagram of our implementation is shown in Fig. 4. The input data set is stored in an array of 128 bytes aligned in 32 rows by 32 columns. In order to save the FPGA's pins, each byte is loaded serially via a 128-bit input bus. This approach allows us to download the 1-Kbit array in only 8 cycles, while keeping the I/O at a manageable level. Once filled the array, a positive edge on the start input allows the engine to begin the operations.

Each column in the array is concatenated to form a 32-bit shift register, with common reset and enable. The rows are all scrolled downwards in parallel, and the bits set to high increment the pertaining 5-bit counters, obtaining the total number of sources reaching a given destination ($\Sigma_i D_i$).

The algorithm requires computing for each destination a weight DW_j defined as $2^{32-\Sigma_i D_i}$. This is done in the Destination Weight Generator block by simply setting to one the bit at the address $\Sigma_i D_i$ (starting from zero) in the 33-bit Destination Weight word and eventually inverting the more significant with the least significant bit.

The Source Weight can be now be computed by adding together the weights of all the destinations reached by a given Source. The adders in the Source Weight Generator block only have in input binary numbers with just one bit set to high. This allowed us to optimize the logic with a semi-serial handling of the carry. Most of the time, the sum can be obtained just XOR-ing together the partial result with the input data. In presence of carry, input data is shifted by one and the bit-wise XOR is done again. The process stops when the sum does not generate any carry condition. This approach greatly simplifies the adders at the expense of a variable execution time, which on average is negligible with respect to the total elapsed time needed to process the data-set.

The 38-bit Source Weights are compared with each other in order to find the largest value. The row index of the source with the highest score is driven in output on the 5-bit *Coded_S_i* bus. The block also takes care to remove from the data-set array all the destinations reached by the selected source, by resetting the columns with a high at its row index. Every time a source is found, the *Iteration* output pin is pulsed and the 32-bit *Selected_Sources* word is updated, flipping permanently to one the bit at address given by the *Coded_Si* bus. The algorithm stops when all destinations have been removed from the data-set array, and a pulse on the output pin *Empty* is generated.

TEST RESULTS

We have simulated and tested three cases with ($|S| = 8, |D| = 8$), ($|S| = 16, |D| = 16$), ($|S| = 32, |D| = 32$) to study how the logic occupancy and the maximum clock frequency scale with the input data size. The 32x32 implementation is discussed in details, with a description of the main logic blocks and the related timing. We have run on it benchmarks with different source-to-destination density to profile the algorithm execution time versus the data-set complexity.

32x32 Benchmark data

We evaluate empirically the performance of the system on a set of test problems ($|S| = 32, |D| = 32$). Namely, we use 80 randomly generated problems, grouped in the classes *Bench05*, *Bench07*, *Bench09*, and *Bench095* of 20 SCP problems each. Each problem allows at least an admissible solution.

Each class is characterized by a different *density*, i.e. the percentage

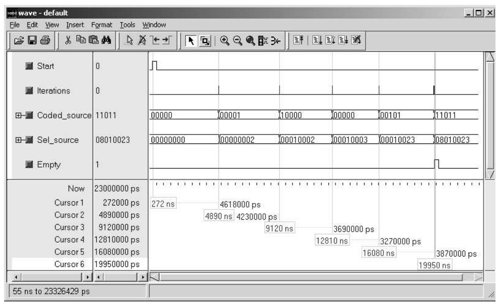
$$\frac{100}{mn}\sum_{i,j} I_{S_i}(u_j) \tag{8}$$

The complement of this quantity is the *zero density* percentage

$$100(1 - \frac{1}{mn}\sum_{i,j} I_{S_i}(u_j)) \tag{9}$$

The latter quantity is 50%, 70%, 90%, and 95% for *Bench05*, *Bench07*, *Bench09*, and *Bench095* respectively.

Evaluations



```
Start          0
Iterations     0
Coded_source   11011
Sel_source     08010023
Empty          1
Now            23000000 ps
Cursor 1       272000 ps
Cursor 2       4890000 ps
Cursor 3       9120000 ps
Cursor 4       12810000 ps
Cursor 5       16080000 ps
Cursor 6       19950000 ps
55 ns to 23326429 ps
```

Figure 5: Timing simulation of a 32x32 run

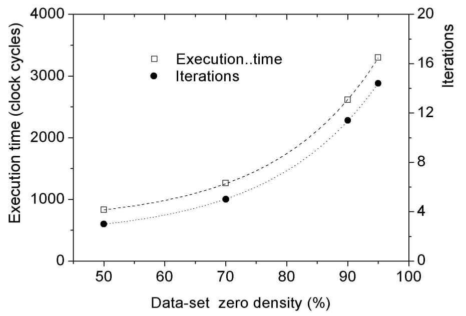


Figure 6: Average values obtained by processing 32x32 data-sets with different zero densities

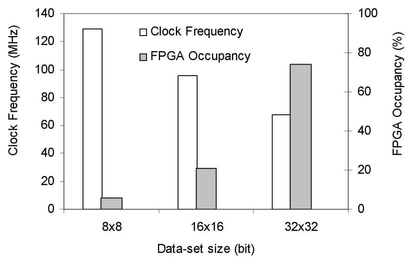


Figure 7: Occupancy and maximum clock frequency for the 8x8, 16x16 and 32x32 implementations

The timing simulation of a 32x32 run is shown in Fig. 5. The output signals allow us to monitor the performance by tracking the run time for each iteration as well as the output sequence of the selected sources. Their index is available on the *Coded_S_i* bus during two iterations and it is also encoded in the *Selected_Sources* word. The single iteration as well as the total execution times are function of the data-set.

The average values obtained processing 32x32 data-sets with different densities of zeros are shown in Fig. 6. As the problem increases in difficulty, i.e. in the range from

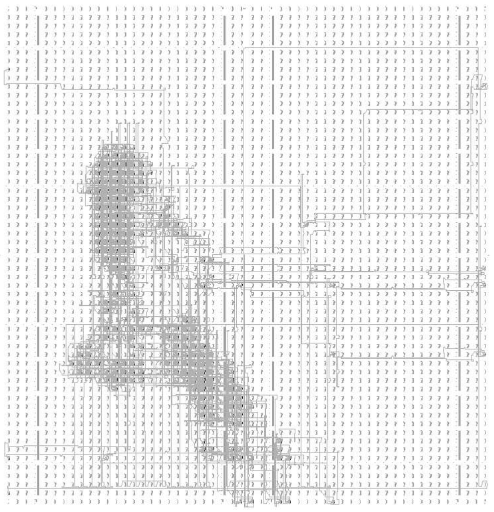


Figure 8: FPGA Occupancy for the 8x8 implementation

50% to 95% of zeros in the data-set, the number of iterations performed by the algorithm increases exponentially (from 3 to 14.4 iterations), approaching the expected (polynomial) worst case bound. The execution times show the same trend, spanning the range 827 - 3298 clock cycles.

Occupancy and maximum clock frequency for the 8x8, 16x16 and 32x32 implementations are shown in Fig. 7 and Fig. 8. The 8x8 only takes 7% of the FPGA and can run up to 130 MHz. The 32x32 implementation almost saturates the FPGA. The maximum clock frequency scales from 130MHz for the 8x8 to half this value for the 32x32. For the latter, the average execution time for a data-set with 95% density of zero is $\sim$50μs.

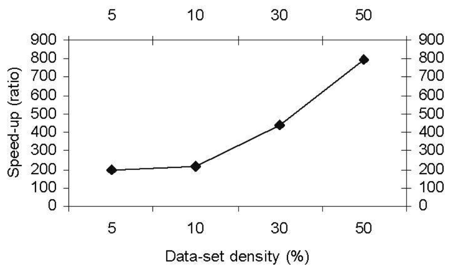


Figure 9: Implementation Speed-up

By comparing the runtimes of the FPGA implementation to the runtimes of the software reference solver the raw speedups appears to be at least of two orders of magnitudes (200×), growing exponentially in the density. In fact differently from the hardware implementation, in the reference solver the major cost of the software implementation, the updating step, does not scales with the density. Fig. 9 reports the speed-up, computed as the ratio between the hardware and software execution times, as function of the data density (8). The reference software data are obtained on an Intel 2.40 GHz Pentium 4 based machine.

CONCLUSION

We described an implementation based on a FPGA of a new greedy algorithm for approximating minimum set covering in ad hoc wireless network applications. The implementation makes the algorithm suitable for embedded and real-time architectures.

For the sum computations we introduced an approach that greatly simplifies the adding steps at the expense of a variable execution time, which on average is negligible with respect to the total elapsed time needed to process the data-set.

The test results show very good empirical performances on the used benchmarks. However, although we have achieved raw speedups at least of two orders of magnitudes, real world experimentation with larger benchmarks is in progress.

REFERENCES

[1] K.M. Alzoubi, P. Wan, O. Frieder. New Distributed Algorithm for Connected Dominating Set in Wireless Ad Hoc Networks. Proceedings of 35th Hawaii International Conference on System Sciences, Hawaii 2002.

[2] A.D. Amis, R. Prakash, T.H.P. Vuong and D.T. Huynh. Max-Min D-Cluster Formation in Wireless Ad Hoc Networks. Proceedings of IEEE INFOCOM'2000, Tel Aviv, March 2000.

[3] Y.P. Chen and A.L. Liestman. Approximating Minimum Size Weakly-Connected Dominating Sets for Clustering Mobile Ad Hoc Networks. Third ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc'02), pp. 157-164, Lausanne, Switzerland, June 2002.

[4] B. Das and V. Bharghavan. Routing in Ad-Hoc Net- works Using Minimum Connected Dominating Sets. IEEE Inter- national Conference on Communications (ICC '97), (1) 1997: 376-380.

[5] M. Q. Rieck, S. Pai, S. Dhar, Distributed Routing Algorithms for Wireless Ad Hoc Networks Using d-hop Connected d-hop Dominating Sets. Proceedings of the 6th International Conference on High Performance Computing: Asia Pacific, Bangalore, December 16-19, 2002.

[6] J. Wu and F. Dai. Broadcasting in Ad Hoc Networks Based on Self-Pruning. IEEE INFOCOM, April, 2003.

[7] J. Wu Extended Dominationg-Set-Based Routing in Ad Hoc Wireless Networks with Unidirectional Links. IEEE Transactions on Parallel and Distributed Systems, Vol. 13, No. 9, September 2002.

[8] S. Guha and S. Khuller. Approximation algorithms for connected dominating sets. Algorithmica, Vol 20, 1998.

[9] D. Johnson. Approximation Algorithms for Combinatorial Problems. Journal of Computer and System Sciences, 9:256-278, 1974.

[10] M.R. Garey, and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, New York, NY: W.H.Freeman, 1979.

[11] T. Asano, K. Iwama, H. Tadaka, and Y. Yamashita, "Designing High-Quality Approximation Algorithms for Combinatorial Optimization Problems", *IEICE Trans. Inf. & Syst.,* vol. E83 (3), 2000, pp. 462-479.

[12] B. Liang and Z.J. Haas. Virtual Backbone Generation and Maintenance in Ad Hoc Network Mobility Management. Proc. 19th Ann. Joint Conf. IEEE Computer and Comm. Soc. INFOCOM, Vol. 3, pp. 1293-1302, 2000.

[13] S. Rampone, Probability-driven Greedy Algorithms for Set Cover. VIII SIGEF Congress ”New Logics for the New Economy” Naples, Italy, September, 2001.

[14] S. Dhar, M. Q. Rieck, S. Pai and E. J. Kim. Various Distributed Shortest Path Routing Strategies for Wireless Ad Hoc Networks. Proceedings of the 5th International Workshop on Distributed Computing (IWDC 2003), Lecture Notes in Computer Science, Springer Verlag, December 2003.

[15] S. Dhar, M. Q. Rieck, S. Pai and E. J. Kim. Distributed Routing Schemes for Ad Hoc Networks Using d-SPR Sets, Journal of Microprocessors and Microsystems, Special Issue on Resource Management in Wireless and Ad Hoc Mobile Networks, Elsevier, Volume 28, Issue 8, October 2004, Pages 427-437.

[16] A.Aloisio, V.Izzo, S.Rampone, FPGA Implementation of a greedy algorithm for set cover, Proc. IEEE RT2005, IEEE Press, 2005.

[17] B. L. Hutchings, and M. J. Wirthlin, “Implementation approaches for reconfigurable logic applications”, in W. Moore and W. Luk, editors *Field-Programmable Logic and Applications*, Oxford, Springer Verlag, 1995, pp. 419-428.

[18] W. Mangione-Smith, B. Hutchings, D. Andrews, A. DeHone, C. Ebeling, R. Hartenstein, O. Mencer, J. Morris, V. Prasanna, and H. Spaanenburg, “Seeking solutions in configurable computing”, *IEEE Computer*, December, 1997, pp. 38-43.