Effective Exploration & Exploitation of the Solution Space via Memetic Algorithms for the Circuit Partition Problem

Shawki Areibi[1]

School of Engineering, University of Guelph sareibi@uoguelph.ca

1 Introduction

Genetic Algorithms (**GA's**) are a class of evolutionary techniques that seek improved performance by sampling areas of the parameter space that have a high probability for leading to good solutions [11]. The evolution program is a probabilistic algorithm which maintains a population of individuals (chromosomes). Each chromosome represents a potential solution within the landscape of the problem at hand. These individuals undergo transformations based on operators to create new populations (solutions). Many evolution programs can be formulated to solve different problems. These programs may differ in the data structures, parameter tuning, specific genetic operators but share some common principles (i) population of individuals (ii) genetic operators to transform individuals into new (possibly better) solutions. The power of GA's comes from the fact that the technique is robust, and can deal successfully with a wide range of problem areas, including those which are difficult for other methods to solve. GA's are not guaranteed to find the global optimum solution to a problem, but they are generally good at finding "acceptably good" solutions to problems. In other words, GA's are considered to be competitive if: the solution space to be searched is large (exploration) and the fitness function is noisy (landscape is not smooth nor unimodal).

Genetic Algorithms are not well suited for fine-tuning structures and incorporation of local improvement has become essential for Genetic Algorithms to compete with other meta-heuristic techniques. Memetic Algorithms [1] apply a separate local search process to refine individuals by hill climbing.

1.1 Motivation and Contributions

Efficient optimization algorithms used to solve hard problems usually employ a hybrid of at least two techniques to find a near optimal solution to the problem being solved. The main motivation for hybridization in optimization practice

is the achievement of increased efficiency (i.e adequate solution quality in minimum time or maximum quality in specified time). From an optimization point of view, Memetic Algorithms combine global and local search by using Evolutionary Algorithms (EA) to perform exploration while the local search method performs exploitation.

The main contributions of this book chapter are (i) investigation of parameter tuning of Genetic Algorithms to solve the circuit partitioning problem effectively, (ii) investigating the balance between exploration and exploitation of the solution space.

1.2 Chapter Organization

The book chapter is organized as follows: Section 2 introduces very briefly the VLSI circuit partitioning problem and terminology used throughout the chapter. The concept of evolutionary computation and Genetic Algorithms will be introduced in Section 3. Section 4 introduces the need behind Memetic Algorithms to further explore the solution space effectively. Results are introduced in Section 5. The chapter concludes with some comments on the issue of effective space exploration and exploitation and possible future work.

2 Background

The last decade has brought explosive growth in the technology for manufacturing integrated circuits. Integrated circuits with several million transistors are now commonplace. This manufacturing capability, combined with the economic benefits of large electronic systems, is forcing a revolution in the design of these systems and providing a challenge to those people interested in integrated system design. Since modern circuits are too complex for an individual designer or a group of designers to comprehend completely, managing this tremendous complexity and automating the design process have become crucial issues.

A large subset of problems in VLSI CAD is computationally intensive, and future CAD tools will require even more accuracy and computational capabilities from these tools. In the combinatorial sense, the layout problem is a constrained optimization problem. We are given a circuit (usually a module-wire connection-list called a *netlist*) which is a description of switching elements and their connecting wires. We seek an assignment of geometric coordinates of the circuit components (in the plane or in one of a few planar layers) that satisfies the requirements of the fabrication technology (sufficient spacing between wires, restricted number of wiring layers, and so on) and that minimizes certain cost criteria. The most common way of breaking up the layout problem into subproblems is first to do *logic partitioning* where a large circuit is divided into a collection of smaller modules according to some criteria, then to perform component *placement*, and then to determine the

approximate course of the wires in a *global routing* phase. This phase may be followed by a *topological-compaction* phase that reduces the area requirement of the layout, after which a *detailed-routing* phase determines the exact course of the wires without changing the layout area.

2.1 Circuit Partitioning

Circuit partitioning is the task of dividing a circuit into smaller parts. It is an important aspect of layout for several reasons. Partitioning can be used directly to divide a circuit into portions that are implemented on separate physical components, such as printed circuit boards or chips. Here, the objective is to partition the circuit into parts such that the sizes of the components are within prescribed ranges and the complexity of connections (nets cut) between the components is minimized. Figure 1 presents a circuit that is partitioned into two blocks (partitions) with a single cut introduced. The inputs/outputs of the circuit represent the terminals (I/O pads) of the circuit. All gates/cells are interconnected by using nets (hyperedges).

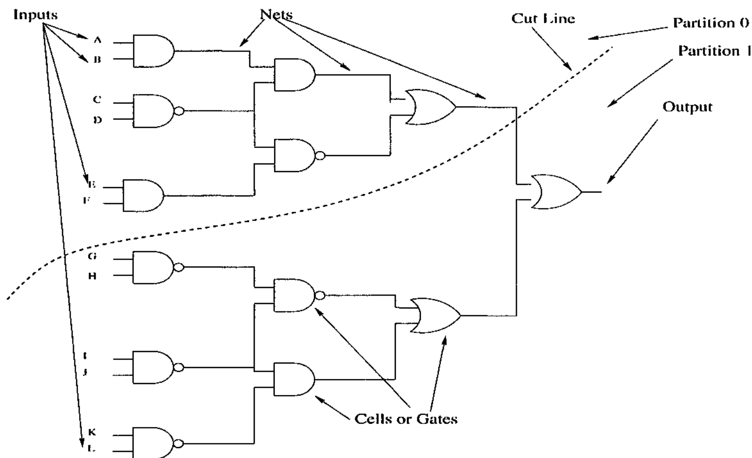


Fig. 1. Circuit Partitioning & Terminology

2.2 Benchmarks

The quality of solutions obtained for the circuit partitioning problem are based on a set of hypergraphs that are part of widely used ACM/SIGDA [12] circuit partitioning benchmarks suite. The characteristics of these hypergraphs are shown in Table 1. The second column of the table shows the number of cells within the circuit. The third column presents the number of nets connecting

the cells within the benchmarks. The total number of pins (i.e connections) within the circuit is summarized in column four. The last two columns summarize the statistics of the circuit (i.e connectivity).

Table 1. Benchmarks Used as Test Cases

Circuit	Cells	Nets	Pins	Cell Degree			Net Size		
				MAX	$\overline{x}$	σ	MAX	$\overline{x}$	σ
Fract	125	147	462	7	3.1	1.6	17	3.1	2.2
Prim1	833	902	2908	9	3.4	1.2	18	3.2	2.5
Struct	1888	1920	5471	4	2.8	0.6	17	2.8	1.9
Ind1	2271	2192	7743	10	3.4	1.1	318	3.5	9.0
Prim2	2907	3029	18407	9	3.7	1.5	37	3.7	3.8
Bio	6417	5742	21040	6	3.2	1.0	861	3.6	20.8
Ind2	12142	13419	48158	12	3.8	1.8	585	3.5	10.9
Ind3	15057	21808	65416	12	4.3	1.4	325	2.9	3.2
Avq.s	21854	22124	76231	7	3.4	1.4	4042	3.4	53.3
Avq.l	25144	25384	82751	7	3.2	1.2	4042	3.2	49.8
Ibm05	29347	28446	126308	9	4.3	2.3	17	4.4	4.2
ibm07	45926	48117	175639	98	3.8	2.4	25	3.6	3.0
ibm10	69429	75196	297567	137	4.2	3.2	41	3.9	3.5
ibm13	84199	99666	357075	180	4.2	3.3	24	3.5	3.0

2.3 Heuristic Techniques for Circuit Partitioning

Heuristic algorithms for combinatorial optimization problems in general and circuit partitioning in particular can be classified as being *constructive* or *iterative*. Constructive algorithms determine a partitioning from the graph describing the circuit or system, whereas iterative methods aim at improving the quality of an existing partitioning solution. Constructive partitioning approaches are mainly based on clustering[3, 6], spectral or eigenvector methods[5], mathematical programming or network flow computations. To date, iterative improvement techniques that make local changes to an initial partition are still the most successful partitioning algorithms in practice. The advantage of these heuristics is that they are quite robust. In fact, they can deal with netlists as well as arbitrary vertex weights, edge costs, and balance criteria.

Constructive Based Techniques (GRASP)

GRASP is a greedy randomized adaptive search procedure that has been successful in solving many combinatorial optimization problems efficiently [8,

4]. Each iteration consists of a construction phase and a local optimization phase. The construction phase intelligently constructs an initial solution via an adaptive randomized greedy function. Further improvement in the solution produced by the construction phase may be possible by using either a simple local improvement phase or a more sophisticated procedure in the form of Tabu Search or Simulated Annealing. The construction phase is iterative, greedy and adaptive in nature. It is *iterative* because the initial solution is built by considering one element at a time. The choice of the next element to be added is determined by ordering all elements in a list. The list of the best candidates is called the restricted candidate list (RCL). It is *greedy* because the addition of each element is guided by a greedy function. The construction phase is *randomized* by allowing the selection of the next element added to the solution to be any element in the RCL. Finally, it is *adaptive* because the element chosen at any iteration in a construction is a function of those previously chosen.

Iterative Improvement

Kernighan and Lin (KL) [10] described a successful iterative heuristic procedure for graph partitioning which became the basis for most module interchange-based improvement partitioning heuristics used in general. Their approach starts with an initial bisection and then involves the exchange of pairs of vertices across the cut of the bisection to improve the cut-size. The algorithm determines the vertex pair whose exchange results in the largest decrease of the cut-size *or* in the smallest increase, if no decrease is possible. A pass in the Kernighan and Lin algorithm attempts to exchange all vertices on both sides of the bisection. At the end of a pass the vertices that yield the best cut-size are the only vertices to be exchanged. Fiduccia and Mattheyses (FM) [7] modified the Kernighan and Lin algorithm by suggesting to move one cell at a time instead of exchanging pairs of vertices, and also introduced the concept of preserving balance in the size of blocks. The FM method reduces the time per pass to linear in the size of the netlist (i.e $O(p)$, where p is the total number of pins) by adapting a single-cell move structure, and a gain bucket data structure that allows constant-time selection of the highest-gain cell and fast gain updates after each move.

Figure 2(a) shows the swap/move of modules between blocks that may lead to a reduction of nets cut. Each module is initially labeled to be free **"F"** to move, but once moved during a pass it is relabeled to be locked **"L"**. The gain of moving a specific module from one partition to another is maintained by using the bucket gain data structure (shown in Figure 2(b)). At the end of a pass only those modules that contribute to the highest gain (i.e reduction in cut size) are allowed to move to their new destination (as illustrated in Figure 2(c)).

Figure 3 shows the basic Fiduccia-Mattheyses (FM) algorithm used for circuit partitioning[7].

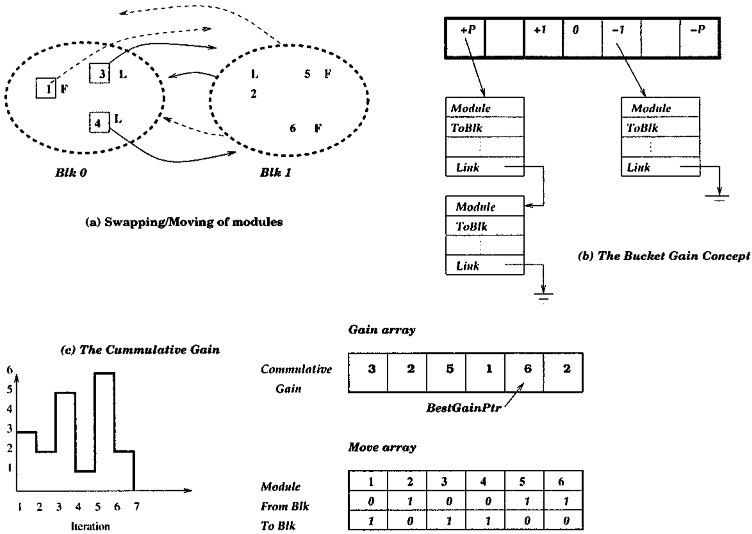


Fig. 2. Basic techniques for Interchange Methods

```
current_solution ← initial_solution
current_cost ← evaluate(current_solution)
Repeat
        initialize partition
        While (can_move(modules))
              choose cell with highest gain
              update gains of all cells
              if (current_gain > previous_gain)
                    bestgain = current_gain
        end while
        move nodes pointed to by bestgain_ptr
        if (no improvement)
              ++noimp_counter
Until((pass > MaxPass) OR
              (noimp > MaxNoImp))
```

Fig. 3. Fiduccia Mattheyses Algorithm

Sanchis [13] uses the above technique for multiple way network partitioning. Under such a scheme, we should consider all possible moves of each free cell from its home block to any of the other blocks, at each iteration during a pass the best move should be chosen. As usual, passes should be performed until no improvement in cutset size is obtained. This strategy seems to offer some hope of improving the partition in a homogeneous way, by adapting the level gain concept to multiple blocks.

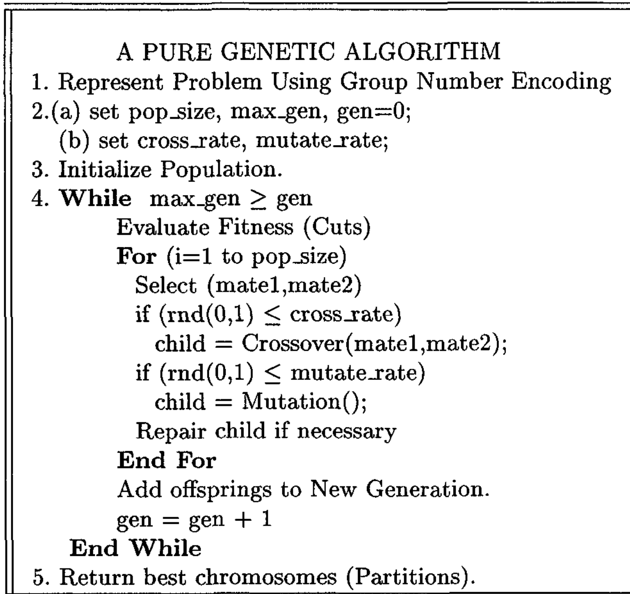
Table 2 presents the results obtained using Sanchis local search technique for two-way and multi-way partitioning. The results are the average of fifty runs. The CPU time increases dramatically as the number of partitions increase in size from 2-way to 4-way and ultimately to 8-way partitioning. In general, node interchange methods are greedy or local in nature and get easily trapped in local minima. More important, it has been shown that interchange methods fail to converge to "optimal" or "near optimal" partitions unless they initially begin from "good" partitions. Sechen [14] shows that over 100 trials or different runs (each run beginning with a randomly generated initial partition) are required to guarantee that the best solution would be within twenty percent of the optimum solution. In order for interchange methods to converge to "near optimal" solutions they have to initially begin from "good" starting points [2].

Table 2. Multi-Way Partitions Based on Local Search

Circuit	2 Blocks		4 Blocks		6 Blocks		8 Blocks	
	Cuts	CPU	Cuts	CPU	Cuts	CPU	Cuts	CPU
Fract	11	0.3	28	0.3	48	0.4	56	0.5
Prim1	58	2.3	148	2.7	171	3.3	189	4.0
Struct	46	5.8	195	6.4	264	8.4	312	10.5
Ind1	30	7.2	245	8.3	364	12.5	374	16.6
Prim2	230	12.4	636	13.3	773	19.1	804	28.0
Bio	91	28.4	532	45.8	726	71.9	806	105.9
Ind2	507	70.4	1759	143.1	2162	272.2	2141	394.4
Ind3	396	63.5	1675	118.4	2636	190.2	2862	280.7
Avq.s	453	126.2	2151	309.9	2436	499.5	2641	674.7
Avq.l	460	178.1	2594	321.8	2728	594.5	3027	857.1
Ibm05	2451	329.4	8922	1618	9629	3719	9894	6059
ibm07	1350	518.3	13527	4437	15922	11820	17011	23185
ibm10	1972	1068	22331	12855	26544	40252	27835	79470
ibm13	1560	1365	26710	16456	31949	53715	34171	105000

3 Genetic Algorithms

As an optimization technique, Genetic Algorithms simultaneously examine and manipulate a set of possible solutions. Figure 4 illustrates a Genetic Algorithm implementation for circuit partitioning.



```
A PURE GENETIC ALGORITHM
1. Represent Problem Using Group Number Encoding
2.(a) set pop_size, max_gen, gen=0;
  (b) set cross_rate, mutate_rate;
3. Initialize Population.
4. While  max_gen ≥ gen
         Evaluate Fitness (Cuts)
         For (i=1 to pop_size)
           Select (mate1,mate2)
           if (rnd(0,1) ≤ cross_rate)
             child = Crossover(mate1,mate2);
           if (rnd(0,1) ≤ mutate_rate)
             child = Mutation();
           Repair child if necessary
         End For
         Add offsprings to New Generation.
         gen = gen + 1
     End While
5. Return best chromosomes (Partitions).
```

Fig. 4. A Genetic Algorithm for Circuit Partitioning

The GA starts with several alternative solutions to the optimization problem, which are considered as individuals in a *population*. These solutions are coded as binary strings, called *chromosomes*. Figure 5 shows a group number encoding scheme to represent the partitioning problem where the jth integer $i_j \in \{1, \ldots, k\}$ indicates the group number assigned to object j.

The initial population is constructed randomly. These individuals are *evaluated*, using the partitioning-specific fitness function. The GA then uses these individuals to produce a new *generation* of hopefully better solutions. In each generation, two of the individuals are selected probabilistically as *parents*, with the selection probability proportional to their fitness. *Crossover* is performed on these individuals to generate two new individuals, called *offspring*, by exchanging parts of their structure. Thus each offspring inherits a combination of features from both parents. The next step is *mutation* where an incremental change is made to each member of the population, with a small probability. This ensures that the GA can explore new features that may not be in the population yet. It makes the entire search space reachable, despite the finite

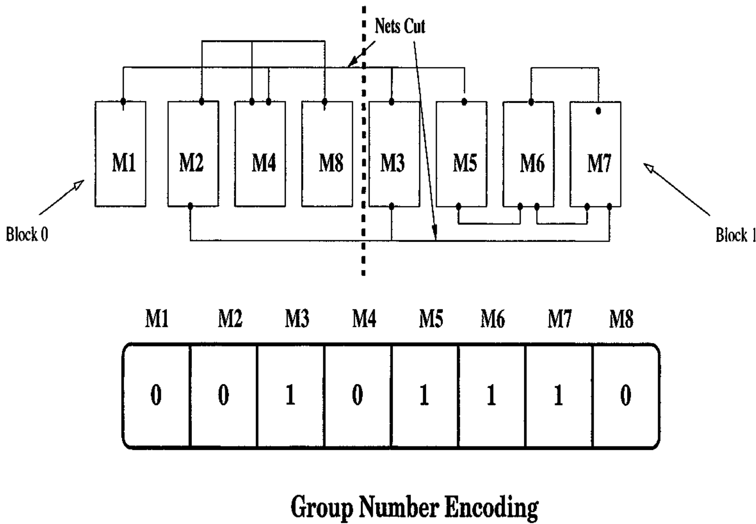


Fig. 5. Chromosome Representation for Circuit Partitioning

population size. However an offspring may contain less than k groups; moreover, an offspring of two parents, both representing feasible solutions may be infeasible, since the constraint of having equal number of modules in each partition is not met. In this case either special *repair heuristics* are used to modify chromosomes to become feasible, or *penalty functions* that penalize infeasible solutions, are used to eliminate the problem.

3.1 Crossover & Mutation

Figure 6 shows the crossover/mutation operators used for the circuit partitioning problem. Operators in the reproduction module, mimic the biological evolution process, by using unary (mutation type) and higher order (crossover type) transformation to create new individuals. *Mutation* as shown in Figure 6(a) is simply the introduction of a random element, that creates new individuals by a small change in a single individual. When mutation is applied to a bit string, it sweeps down the list of bits, replacing each by a randomly selected bit, if a probability test is passed. On the other hand, *crossover* recombines the genetic material in two parent chromosomes to make two children. It is the structured yet random way that information from a pair of strings is combined to form an offspring. Crossover begins by randomly choosing a cut point K where $1 \leq K \leq L$, and L is the string length. The parent strings are both bisected so that the left-most partition contains K string elements, and the rightmost partition contains $L - K$ elements. The child string is formed by copying the rightmost partition from parent P_1 and then the left-most

partition from parent P_2. Figure 6(b) shows an example of applying the standard crossover operator (sometimes called one-point crossover) to the group number encoding scheme. Increasing the number of crossover points is known to be multi-point crossover as seen in Figure 6(c).

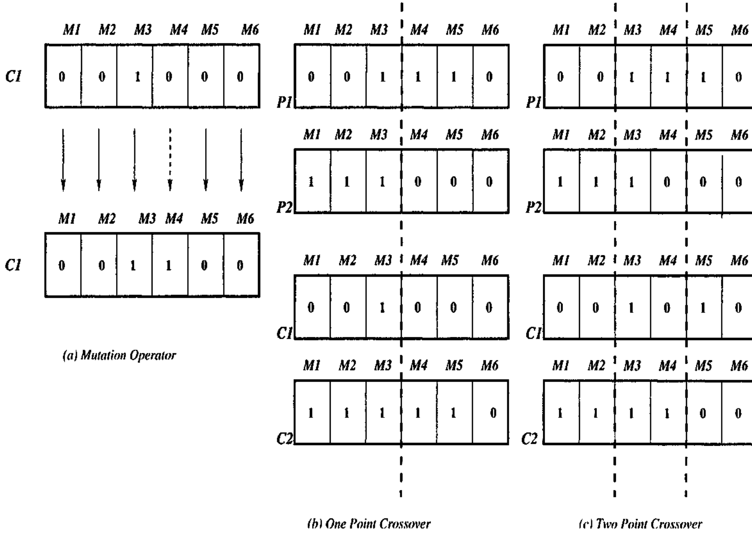


Fig. 6. Mutation & Crossover Operators

Figure 7 and Figure 8 show the affect of mutation rate on the quality of solutions obtained. Figure 9 and Figure 10 highlight the importance of tuning

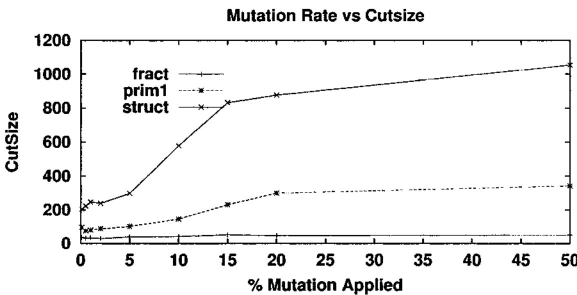


Fig. 7. Mutation Rate (Small Circuits)

the crossover rate and its affect on the solution quality. Figures 11, 12, 13 show the affect of crossover points. It is clear from the figures that multi-point

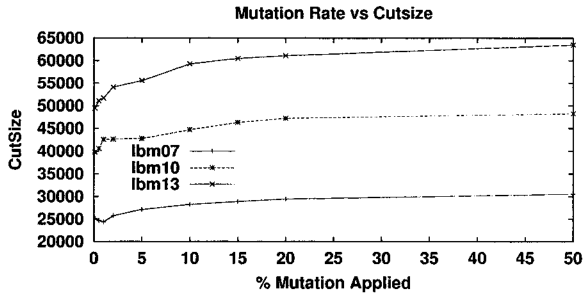


Fig. 8. Mutation Rate (Very Large Circuits)

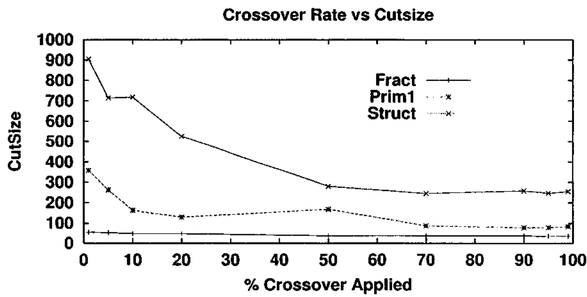


Fig. 9. Crossover Rate (Small Circuits)

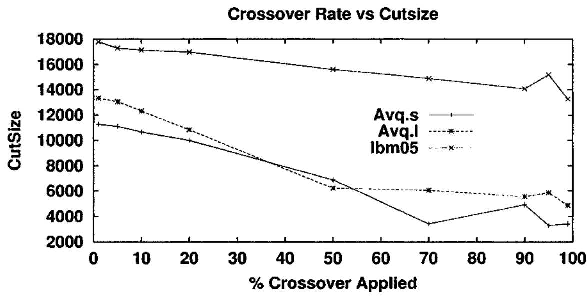


Fig. 10. Crossover Rate (Large Circuits)

crossover performs much better than one-point crossover technique. A 3-point and 4-point crossover works best for our circuit partitioning problem.

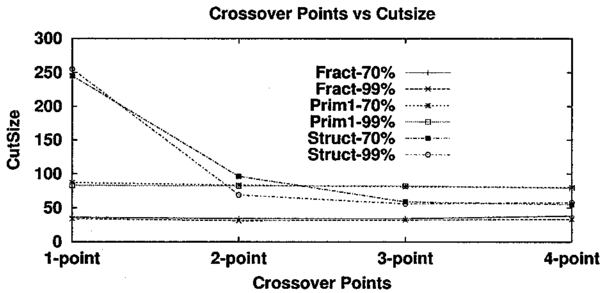


Fig. 11. Crossover Points (Small Circuits)

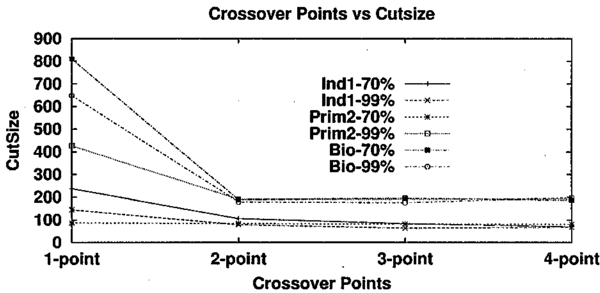


Fig. 12. Crossover Points (Medium Circuits)

3.2 Population/Generation Size

The size of the population is one of the most important choices in implementing any Genetic Algorithm and is considered to be critical for several applications. If the population size is too small then this may lead to early convergence and if it is too large this may lead to huge computation time (i.e waste of computational resources). Figure 14 shows the affect of the population size on the quality of solutions obtained for large circuits. The population in any Genetic Algorithm implementation evolves for a prespecified total number of generations under the application of evolutionary rules. The generation size is crucial in any Genetic Algorithm implementation. As the number of generations increase the quality of solutions improve, but the computation

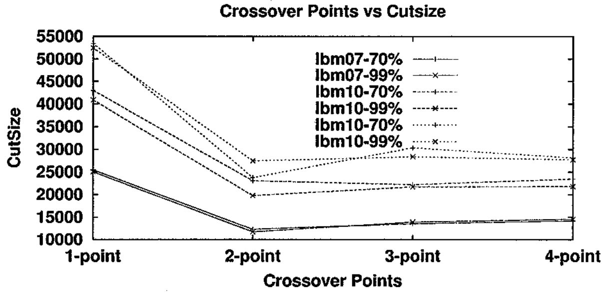


Fig. 13. Crossover Points (Very Large Circuits)

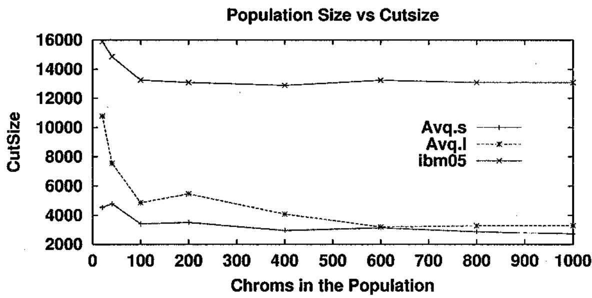


Fig. 14. Population Size (Large Benchmarks)

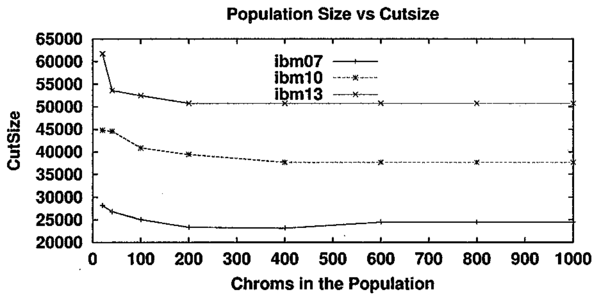


Fig. 15. Population Size (Very Large Benchmarks)

time involved increases dramatically. Figure 16 and Figure 17 show the affect of generation size on the solution quality obtained based on large circuits and very large circuits respectively.

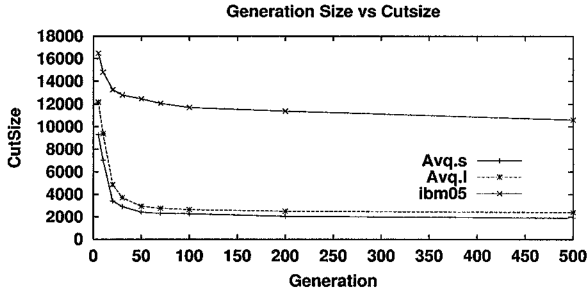


Fig. 16. Affect of Generation Size for Large Benchmarks

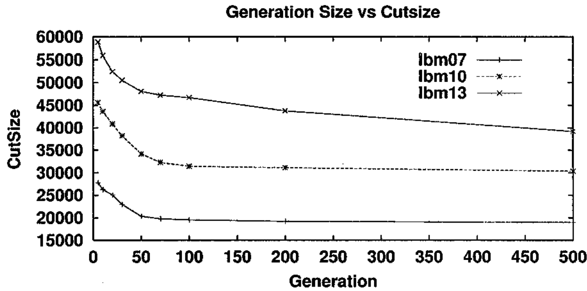


Fig. 17. Affect of Generation Size for Very Large Benchmarks

3.3 Selection Techniques

Strings are selected for mating based on their fitness, those with greater fitness are awarded more offspring than those with lesser fitness. Parent selection techniques that are used, vary from stochastic to deterministic methods. The probability that a string i is selected for mating is p_i, "the ratio of the fitness of string i to the sum of all string fitness values", $p_i = \frac{fitness_i}{\sum_j fitness_j}$. The ratio of individual fitness to the fitness sum denotes a ranking of that string in the population. The Roulette Wheel Selection method (Gsml) is one of the stochastic selection techniques that is widely used. The ratio p_i is used to construct a weighted roulette wheel, with each string occupying an area on

the wheel in proportions to this ratio. The wheel is then employed to determine the string that participates in the reproduction. A random number generator is invoked to determine the location of the spin on the roulette wheel. In Deterministic Selection methods, reproduction trials (selection) are allocated according to the rank of the individual strings in the population rather than by individual fitness relative to the population average. Several selection methods have been implemented as seen in Figure 18 and 19. The technique referred to as *Gsm0* is a deterministic technique where parents are picked uniformly one after the other from the population. *Gsm1* is the stochastic roullette wheel technique. In *Gsm2* the population is sorted according to their fitness each trial the best two in the list are chosen for mating. *Gsm3* is similar to *Gsm2* except that the first half of the sorted list would take higher chances for mating than the rest of the population at the end of the list. *Gsm4* and *Gsm5* are based on a ranking technique. The last two approaches *Gsm6* and *Gsm7* are based on Tournament with replacement and without replacement respectivley. It is clear from Figures 18 and 19 that Tournament Selection with replacement gives the best solution quality compared to other selection techniques.

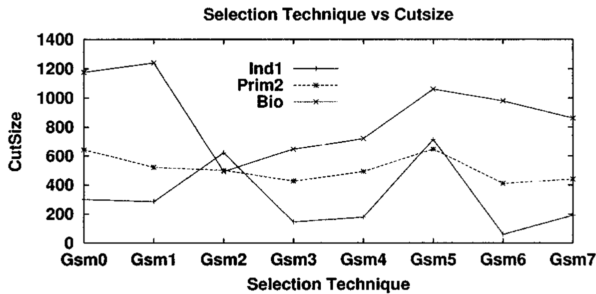


Fig. 18. Selection vs Cutsize (Medium Circuits)

3.4 Replacement Strategy

Generation replacement techniques are used to select a member of the old population and replace it with the new offspring. The quality of solutions obtained depends on the replacement scheme used. Some of the replacement schemes used are based on: (i) deleting the old population and replacing it with new offsprings (R-ap), (ii) both old and new populations are sorted and the newly created population is constructed from the top half of each (R-hp), (iii) replacing parent with the child if newly created member is more fit (R-pc) (iv) replacing the most inferior members (R-mi) in a population by new offsprings. Figure 20 and 21 show the performance of each replacement

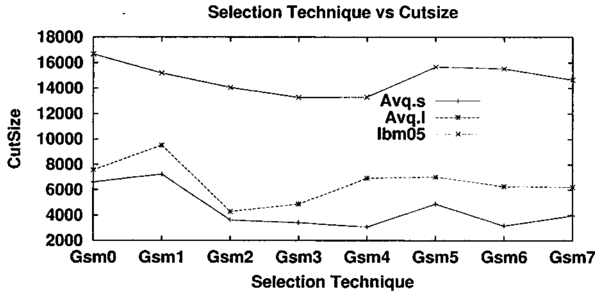


Fig. 19. Selection vs Cutsize (Large Circuits)

technique for large circuits and very large circuits respectivley. It is evident from the Figures that (R-ap) and (R-pc) perform poorly with respect to (R-hp) and (R-mi). Variations to (R-hp) scheme use an incremental replacement approach, where at each step the new chromosome replaces one randomly selected from those which currently have a *below-average* fitness. The quality of solutions improve using (R-hp) and (R-mi) replacement schemes due to the fact that they maintain a large diversity in the population. Our generation replacement technique utilized in both the pure Genetic Algorithm and Memetic Algorithm for circuit partitioning are based on replacing the most inferior member (R-mi) in a population by new offsprings.

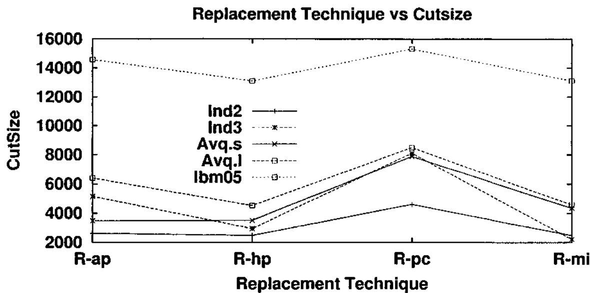


Fig. 20. Replacement Strategy vs Cutsize (Large Circuits)

3.5 Computational Results for GA

Table 3 shows the solution quality for multi-way partitioning and CPU time involved. It is interesting to note that the Genetic Algorithm solution quality compared to Local Search is better for small, medium and large circuits for

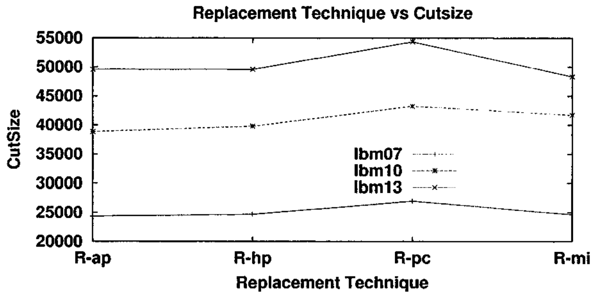


Fig. 21. Replacement Strategy vs Cutsize (Very Large Circuits)

2-way and multi-way partitions. As the size of the circuit increases, the performance of GA deteriorates (as can be seen for benchmarks ibm07, ibm10 and ibm13). On the other hand the complexity of Genetic Algorithm in terms of CPU time is linear as the number of blocks increases. For example, comparing Table 2 and Table 3 for benchmark ibm13, the GA technique cuts the CPU time by almost 50%.

Table 3. Genetic Algorithm Solution Quality for Multi-Way Partitioning

Circuit	2 Blocks		4 Blocks		6 Blocks		8 Blocks	
	Cuts	CPU	Cuts	CPU	Cuts	CPU	Cuts	CPU
Fract	18	19	39	24	49	28	52	38
Prim1	80	126	145	156	136	182	159	234
Struct	52	277	161	344	231	402	255	532
Ind1	70	326	111	408	154	493	159	640
Prim2	186	460	325	581	409	690	557	892
Bio	176	881	266	1122	328	1340	367	1757
Ind2	272	2103	1010	2778	1038	3881	1590	4857
Ind3	491	3106	1337	4645	2130	5753	2341	7801
Avq.s	464	3911	986	4831	1111	7110	1425	9821
Avq.l	465	3999	1002	6336	1093	8066	1426	11272
Ibm05	9248	5585	11890	8158	13026	10918	13704	14690
ibm07	12529	11414	18183	16901	20496	21357	20499	27626
ibm10	20624	22652	29108	30507	31900	37503	32983	47272
ibm13	25876	32610	38186	41371	41452	49693	43139	61771

Comparing results obtained by the Genetic Algorithm with those based on Local Search we can conclude the following. (i) GA's are not guaranteed to find the global optimum solution to a problem, but they are generally good

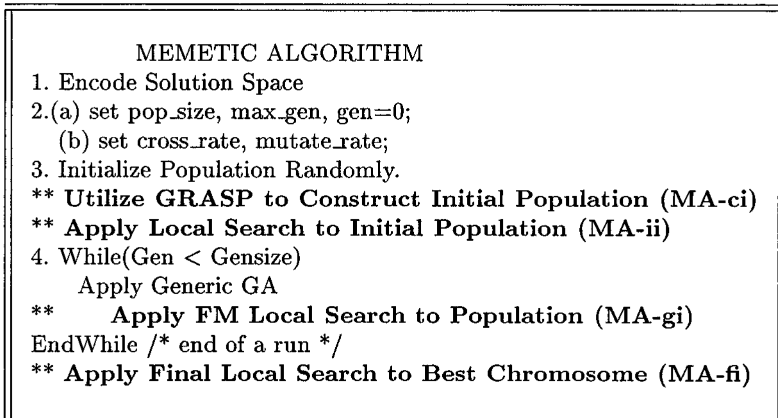
at finding "acceptably good" solutions to problems, (ii) Where specialized techniques exist for solving particular problems, they are likely to out-perform GA's in both speed and accuracy of the final result. Another drawback of Genetic Algorithms is that they are not well suited to perform finely tuned search, but on the other hand they are good at exploring the solution space since they search from a set of designs and not from a single design. Genetic Algorithms are not well suited for fine-tuning structures which are close to optimal solutions [9]. Incorporation of local improvement operators into the recombination step of a Genetic Algorithm is essential if a competitive Genetic Algorithm is desired.

4 Memetic Algorithms

Memetic algorithms (MAs) are evolutionary algorithms (EAs) that apply a separate local search process to refine individuals (i.e improve their fitness by hill-climbing). Under different contexts and situations, MAs are also known as hybrid EAs, genetic local searchers. Combining global and local search is a strategy used by many successful global optimization approaches, and MAs have in fact been recognized as a powerful algorithmic paradigm for evolutionary computing. In particular, the relative advantage of MAs over GA is quite consistent on complex search spaces. Figure 22 shows one possible implementation of a Memetic algorithm based on the Genetic Algorithm introduced earlier in Section 3. We use a simple variation of the Fiduccia and Mattheyses (FM) heuristic [13]. The original FM heuristic has several passes after which the heuristic terminates as presented in Section 2. In the local optimization phase, a single pass is allowed, furthermore a restriction on the number of modules to be moved is set to a certain value. It is to be noted that if local optimization is not strong enough to overcome the inherent disruption of the crossover, more strong local optimization is needed.

4.1 Computational Results for MA

Table 4 shows the results obtained from the Memetic Algorithm. The first column in the table *MA-ii* is the direct application of local search on each chromosome in the population at only the initial stage. The secon column *MA-gi* is the direct application of local search on each chromosome in the population in every generation. It is clear that *MA-gi* performs better fine tuning and exploitation than *MA-ii* which only attempts to fine tune the search at an early stage. *MA-hi* is in affect the combination of *MA-ii* with *MA-gi* such that after an early exploitation of the landscape the system attempts to explore and exploit the solution space simultaneously. The results in the table indicate that the combination does not have a drastic affect on the final solution quality even though an improvement of 2-3% is achieved. The fourth column in the table *MA-ci* is the direct application of GRASP



```
MEMETIC ALGORITHM
1. Encode Solution Space
2.(a) set pop_size, max_gen, gen=0;
  (b) set cross_rate, mutate_rate;
3. Initialize Population Randomly.
** Utilize GRASP to Construct Initial Population (MA-ci)
** Apply Local Search to Initial Population (MA-ii)
4. While(Gen < Gensize)
    Apply Generic GA
**      Apply FM Local Search to Population (MA-gi)
EndWhile /* end of a run */
** Apply Final Local Search to Best Chromosome (MA-fi)
```

Fig. 22. The Memetic Algorithm

to effectively construct good intial solutions for the Genetic Algorithm. The system achieves an improvement of 65% over *MA-ii* and 51% over *MA-gi* for the largest benchmark (ibm13). Experimental results indicate that less than 25% of the population should be injected with good initial solutions for *MA-ci* to perform well. The last column in the table *MA-ci-gi* is a combined *MA-ci* and *MA-gi* approach where good intial solutions are injected into the initial population followed by a balanced exploration (via crossover, mutation) and exploitation (via a single pass of local search) stage. It is quite evident that this Memetic Algorithm approach achieves the best overall results compared to the previously mentioned methods (i.e MA-ii, MA-gi and MA-hi). The overall improvement obtained (over MA-hi) for the largest circuits are: 61% for ibm07, 50% for ibm10 and over 66% for the largest benchmark ibm13.

5 Results & Analysis

In this section we will summarize the results obtained using (i) Local Search (ii) Genetic Algorithms (iii) Memetic Algorithm. Table 5 presents the results obtained by the three techniques mentioned above for four way partitioning. As can be seen in Table 5 the Memetic Algorithm obtains on average better solutions (cuts) than the Local Search technique. As the benchmarks increase in size the quality of solutions obtained using the local search technique deterioates. A comparison between the pure Genetic Algorithm and the Memetic Algorithm reveals the importance of embeding local search within GA to improve its performance. The affect of exploitation shows very clearly for the large benchmarks (ibm07, ibm10 and ibm13).

Table 4. Comparison of Several Memetic Algorithm Implementations

Circuit	MA-ii		MA-gi		MA-hi		MA-ci		MA-ci-gi	
	Cuts	CPU	Cuts	CPU	Cuts	CPU	Cuts	CPU	Cuts	CPU
Fract	47	24	37	24	41	24	35	24	35	24
Prim1	131	157	145	157	123	159	104	158	103	159
Struct	165	344	160	345	165	348	128	348	127	351
Ind1	100	409	97	412	97	414	91	413	90	416
Prim2	303	585	317	588	294	588	265	617	265	621
Bio	267	1120	249	1139	266	1139	234	1155	233	1147
Ind2	1035	2757	710	2841	710	2821	589	2861	587	2832
Ind3	1320	4627	1286	4702	1272	4672	1217	4847	1185	4837
Avq.s	1003	4777	936	4953	963	4804	885	5086	882	5019
Avq.l	999	6351	986	6457	979	6366	968	6386	965	6319
Ibm05	12502	8137	8424	8377	8384	8173	6236	8969	5158	8948
ibm07	18368	16939	12108	17138	12065	17218	8190	17863	6485	18096
ibm10	28765	30569	20239	30820	20296	31238	12307	33421	10119	34322
ibm13	35502	41186	25180	42066	24345	42650	12237	44220	8152	45438

Table 5. Comparison between LS, GA and MA

Circuit	Local Search		Genetic Algorithms		Memetic Algorithms		Improvement	
	Cuts	CPU	Cuts	CPU	Cuts	CPU	LS	GA
Fract	28	0.3	39	24	35	24	-20%	+10%
Prim1	148	2.7	145	156	103	159	+30%	+29%
Struct	195	6.4	161	344	127	351	+34%	+21%
Ind1	245	8.3	111	408	90	416	+63%	+18%
Prim2	636	13.3	325	581	265	621	+58%	+18%
Bio	532	45.8	266	1122	233	1147	+56%	+12%
Ind2	1759	143	1010	2778	587	2832	+66%	+41%
Ind3	1675	118	1337	4645	1185	4837	+29%	+11%
Avq.s	2151	309	986	4831	882	5019	+59%	+10%
Avq.l	2594	321	1002	6336	965	6319	+62%	+4%
Ibm05	8922	1618	11890	8158	5158	8948	+42%	+56%
ibm07	13527	4437	18183	16901	6485	18096	+52%	+64%
ibm10	22331	12855	29108	30507	10119	34322	+54%	+65%
ibm13	26710	16456	38186	41371	8152	45438	+69%	+78%

6 Conclusions

Memetic Algorithms (MAs) are Evolutionary Algorithms (EAs) that apply some sort of local search to further improve the fitness of individuals in the population. This paper provides a forum for identifying and exploring the key issues that affect the design and application of Memetic Algorithms. Several approaches of integrating Evolutionary Computation models with local search techniques (i.e Memetic Algorithms) for efficiently solving underlying VLSI circuit partitioning problem were presented. A Constructive heuristic technique in the form of GRASP was utilized to inject the initial population with good initial solutions to diversify the search and exploit the solution space. Furthermore, the local search technique was able to enhance the convergence rate of the Evolutionary Algorithm by finely tuning the search on the immediate area of the landscape being considered. Future work involves adaptive techniques to fine-tune parameter of the Genetic Algorithm and Local Search when combined to form a Memetic Algorithm. Balancing exploration and exploitation is yet another issue that needs to be addressed more carefully.

References

1. S. Areibi, M. Moussa, and H. Abdullah. A Comparison of Genetic/Memetic Algorithms and Other Heuristic Search Techniques. In *International Conference on Artificial Intelligence*, pages 660–666, Las Vegas, Nevada, June 2001.
2. S. Areibi. An Integrated Genetic Algorithm With Dynamic Hill Climbing for VLSI Circuit Partitioning. In *GECCO 2000*, pages 97–102, Las Vegas, Nevada, July 2000. IEEE.
3. S. Areibi and A. Vannelli. An Efficient Clustering Technique for Circuit Partitioning. In *IEEE ISCAS*, pages 671–674, San Diego, California, 1996.
4. S. Areibi and A. Vannelli. A GRASP Clustering Technique for Circuit Partitioning. 35:711–724, 1997.
5. P.K. Chan, D.F. Schlag, and J.Y. Zien. Spectral K-way Ratio-Cut Partitioning and Clustering. *IEEE Transactions on Computer Aided Design*, 13(9):1088–1096, 1994.
6. S. Dutt and W. Deng. VLSI Circuit Partitioning by Cluster-Removal Using Iterative Improvement Techniques. In *IEEE International Conference on CAD*, pages 194–200. ACM/IEEE, 1996.
7. C.M. Fiduccia and R.M. Mattheyses. A Linear-Time Heuristic for Improving Network Partitions. In *Proceedings of* 19*th DAC*, pages 175–181, Las Vegas, Nevada, June 1982. ACM/IEEE.
8. T. Feo, M. Resende, and S. Smith. A Greedy Randomized Adaptive Search Procedure for The Maximum Ind ependent Set. *Operations Research*, 1994. Journal of Operations Research.
9. D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Publishing Company, Inc, Reading, Massachusetts, 1989.
10. B.W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Technical Journal*, 49(2):291–307, February 1970.

11. Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer-Verlog, Berlin, Heidelberg, 1992.
12. K. Roberts and B. Preas. Physical Design Workshop 1987. Technical report, MCNC, Marriott's Hilton Head Resort,South Carolina, April 1987.
13. L.A. Sanchis. Multiple-Way Network Partitioning. *IEEE Transactions on Computers*, 38(1):62–81, January 1989.
14. C. Sechen and D. Chen. An improved Objective Function for Min-Cut Circuit Partitioning. In *Proceedings of ICCAD*, pages 502–505, San Jose, California, 1988.