

Efficient bipartitioning algorithm for size-constrained circuits

J.-S. Cherng
S.-J. Chen
J.-M. Ho

Indexing terms: VLSI circuits, Bipartitioning algorithm, Module migration

Abstract: A novel module-migration bipartitioner (MMP) for VLSI circuits is proposed. MMP uses an efficient module migration process, which can relax the size constraints temporarily and intensify the capability of escaping from local optima, as its iterative improvement mechanism. Besides evaluating the same module gain when performing the Fiduccia–Mattheyses (FM) algorithm for selecting the module to move, MMP also examines the connection strengths between modules, thus capturing more global implications of module moving. Moreover, MMP is robust with a self-adjusted probabilistic function set which can reduce the sensitivity of some key parameters. Experiments on circuits allowing different deviations from exact bipartition show that MMP is stable on solution quality and that it not only performs much better than FM, but also outperforms many state-of-the-art bipartitioners.

1 Introduction

Partitioning plays an important role in the hierarchical design of electronic systems [1, 2]. In partitioning electronic circuits, minimising interpartition interconnections is most essential. In this paper, we focus attention on finding a size-constrained min-cut bipartitioning algorithm based on the concept of iterative module migration.

1.1 Previous work

Circuit partitioning with size constraints is NP-hard [3]. Hence, various heuristics, such as iterative improvement methods [4–10], clustering-based methods [11–17], and flow-based methods [18], have been developed.

The iterative improvement methods start with a given initial bipartition and try to iteratively improve it by making local changes until the partitioning solution cannot be further improved. In 1970, Kernighan and

Lin [7] proposed the first module interchange algorithm, which was further extended to handle multipin nets by Schweikert and Kernighan [10]. Based on their work, Fiduccia and Mattheyses [5] developed an $O(P)$, time-efficient bipartitioning heuristic (FM), where P is the total number of pins. This is done by moving one module at a time and using an efficient bucket list data structure. Although FM runs fast, it may derive solutions of poor quality due to the lack of an appropriate mechanism for differentiating among highest-gain modules. Thus, Krishnamurthy [8] proposed an extension to FM, which utilises the level gains management to perform a look-ahead (LA) scheme. Furthermore, Sanchis [9] presented a multiway partitioning algorithm based on LA. Recently, Dutt and Deng [4] proposed a probabilistic-gain-based method PROP that computes the gains of modules using much more global and futuristic information than FM and LA.

The clustering-based techniques try to identify natural highly connected groups (i.e. clusters) in a circuit and then assign the clusters to either one of the two partitions. For example, Cheng and Wei [13, 19] combined the ratio-cut scheme [20] with FM [5] to obtain a stable performance bipartitioner (STABLE). First, some clusters are found by using a top-down clustering technique based on the ratio-cut concept. Then FM is applied to rearrange the clusters into two partitions with prespecified size constraints. Similarly, Shin and Kim [17] and Saab [16] adopted bottom-up clustering techniques to find clusters for their respective subsequent bipartitioning. There are also many recent state-of-the-art clustering-based bipartitioners such as EIG1 [14], PARABOLI [15], MELO [12], and WINDOW [11]. Our MMP bipartitioner is compared with some of these in the experimental results.

1.2 Motivation

According to [2, 21], clusters exist in circuit layouts and are formed by placing modules of similar functionality together. Since the intracluster interconnections are much denser than the intercluster interconnections, the goal of a bipartitioner is to keep clusters from being divided as much as possible so that the *cut* can be reduced, where *cut* means the number of nets connecting both partitions.

When moving some modules of a cluster currently being divided to make the cluster entirely fall into one partition, difficulties may occur if FM is applied. First, it is quite possible to select improper modules to move and then, instead of moving them back to their original partitions, FM *locks* these modules [6]. Moreover, even

© IEE, 1998

IEE Proceedings online no. 19981703

Paper first received 25th September 1996 and in revised form 29th August 1997

J.-S. Cherng and S.-J. Chen are with the Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan

J.-M. Ho is with the Institute of Information Science, Academia Sinica, Taipei, Taiwan

when suitable modules are selected to move, size constraints often obstruct the movements of these modules so that the goal of preventing the cluster from being divided becomes hard to achieve.

To avoid division of clusters, many clustering-based bipartitioners [11–15, 17], as mentioned in Section 1.1, have been proposed. However, poor partitioning results may be obtained when using these bipartitioners due to the application of inappropriate clustering strategies or the choice of improper cluster size [22].

Motivated by the above observations, an iterative improvement algorithm has been developed to partition a given circuit with size constraints based on the module migration scheme. This scheme first relaxes the size constraints temporarily when it moves a set of modules from one partition to the other, and reconsiders the original size constraint requirement when another set of modules is moved back. In contrast to the *locking* scheme of FM, we adopt a *nonlocking* scheme; that is, the modules which have been moved forward are allowed to move back again. With this size constraint relaxation and nonlocking scheme, the capability of escaping from local optima is intensified.

For the proposed module choosing strategy, instead of finding clusters explicitly as the clustering-based techniques, we apply the idea of *locality* [16] to implicitly extract natural clusters. The idea is that when a module is moved from one partition to the other, its incident modules have a greater chance of being moved in the following movements, and that after a series of consecutive movements from one partition to the other, a natural cluster will be formed. Consequently, besides using the conventional FM module gain evaluation in our module choosing strategy, we also examine the connection strengths between modules to ensure that the set of modules chosen to be moved can form a natural cluster.

Another unique feature of our algorithm is that it uses a set of self-adjusted probabilistic functions to reduce the sensitivity of some key parameters. Experimental results show that the proposed bipartitioner MMP outperforms many recent state-of-the-art bipartitioners.

2 Problem statement and algorithm description

Suppose a network is represented by a hypergraph $H(V, E)$, where $V = \{v_i | i = 1, 2, \dots, n\}$ denotes the module set and $E = \{e_j | j = 1, 2, \dots, m\}$ denotes the net set. Each net e_j is a subset of V with cardinality $|e_j| \geq 2$. For each module v_i , the set of nets incident to v_i is denoted by $N(v_i)$, and for each net e_j , the set of modules contained by e_j is denoted by $M(e_j)$. We define a module weighting function $s: V \rightarrow R^+$ where R^+ is the set of positive real numbers, used for denoting the actual size of each module, i.e. $s(v_i)$ denotes the size of a module v_i . $S(V) = \sum_{v_i \in V} s(v_i)$ represents the size of the hypergraph H .

A bipartitioning problem is to divide a hypergraph $H(V, E)$ into two nonempty module partitions V_1, V_2 where $V_1 \cap V_2 = \emptyset$ and $V_1 \cup V_2 = V$. The *cut* of a bipartition (V_1, V_2) is the number of hyperedges connecting both partitions and is denoted by $cut(V_1, V_2)$. The objective of bipartitioning is to minimise $cut(V_1, V_2)$ with the size constraints defined as:

$$S_m \leq S(V_k) \leq S_M \quad \text{for } k = 1, 2 \quad (1)$$

where $S(V_k) = \sum_{v_i \in V_k} s(v_i)$ represents the size of partition

V_k for $k = 1, 2$. S_m and S_M are two constraints used to set the size limit for each partition. Here, $0 < S_m \leq S_M < S(V)$ and $S_m + S_M = S(V)$. The *size ratio* of the two partitions is defined as $sr = S(V_1)/S(V_2)$ (or $S(V_2)/S(V_1)$). Feasible bipartitioning solutions with size constraints (S_m, S_M) are found if $(S_m/S_M) \leq sr \leq (S_M/S_m)$. In the experiments as shown in Section 5, we will evaluate MMP with other bipartitioners by finding feasible solutions having different *deviations*, where deviation from the exact bipartition is defined as $\delta = (S_M - S_m)/(S_M + S_m)$.

We now sketch the proposed algorithm as follows. First, an initial partitioning solution is generated as a starting point. Then, by assigning a migration direction, say from V_1 to V_2 , we begin to choose modules from V_1 to be moved to V_2 . This ‘forward’ migration process continues until a stopping criterion is reached. Then, the migration direction is reversed and some modules are moved back to V_1 to ensure the size ratio falls into an acceptable range. We say a *pass* is done when this ‘backward’ migration process is completed. The best feasible solution is used to form a new starting point for the next *pass*. Our algorithm proceeds in a series of *passes*.

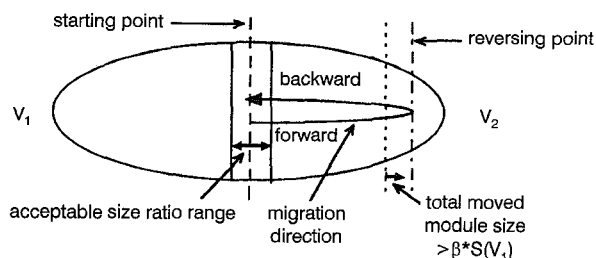


Fig. 1 A pass operation of the MMP algorithm

The basic operation of a *pass* is shown in Fig. 1 where the migration direction is well controlled. In the forward migration, the relaxation of size constraints is controlled by a parameter β , which is devised to determine the size of module set to be moved in the forward migration process of a *pass* operation.

The following *Module Migration Partitioning* (MMP) algorithm consists of three major parts: (1) initialisation (explained in this Section); (2) pass operation and β scaling scheme; and (3) parameter management.

Algorithm *Module Migration Partitioning*

Input: $H(V, E)$: the hypergraph of a given netlist
 num_of_runs : the number of runs performed
 (S_m, S_M) : the size constraints for the two partitions
Output: $(V_1, V_2)_{best}$: the best bipartition among all the runs in terms of minimal *cut*
Variable: $(V_1, V_2)_{run}$: the best bipartition in one run
 $(V_1, V_2)_{pass}$: the best bipartition in one *pass*
 cut_{avg} : the average *cut* for a number of runs

begin

/* q is the net-weighting parameter in eqn. 2, $0 < \beta < 1$ */

randomly select the values of q and β from their respective ranges;

initialise and compute some functions for the *Parameter_Management* procedure;

$(V_1, V_2)_{best} := \text{NULL}$; $cut(V_1, V_2)_{best} := \infty$; $total := 0$;
 $cut_{avg} := 0$;

for $counter_{run} := 1$ **to** num_of_runs **do**

begin

for (each net in E) **do** compute the net weight using eqn. 2 with given q ;

call *Initial_Partition*($H(V, E)$, β , (S_m, S_M)) and return an initial bipartition $(V_1, V_2)_{initial}$;

assign $(V_1, V_2)_{initial}$ to $(V_1, V_2)_{run}$;

/* multiple *pass* operations are performed within the following **for** loop */

for $counter_{pass} := 1$ **to** num_of_passes **do**

begin

call *Pass_Operation*($(V_1, V_2)_{run}$, β , (S_m, S_M)) and return $(V_1, V_2)_{pass}$;

if $cut(V_1, V_2)_{pass} < cut(V_1, V_2)_{run}$ **then** assign $(V_1, V_2)_{pass}$ to $(V_1, V_2)_{run}$;

scale the value of β ;

end; /* end of passes */

if $cut(V_1, V_2)_{run} < cut(V_1, V_2)_{best}$ **then** assign $(V_1, V_2)_{run}$ to $(V_1, V_2)_{best}$;

/* apply *Parameter_Management* procedure to update q and β */

$total := total + cut(V_1, V_2)_{run}$;

$cut_{avg} := total / counter_{run}$;

call *Parameter_Management*($cut(V_1, V_2)_{run}$, cut_{avg}) and return a new q and an initial value of β ;

end; /* end of runs */

report the best bipartition $(V_1, V_2)_{best}$;

end /* end of algorithm */

At the beginning of MMP, the values of q and β are randomly selected from their respective ranges and are updated by the *Parameter_Management* procedure for each run. The initial bipartition for each run is generated by the *Initial_Partition* procedure as follows. First, let $V_1 = V$ and $V_2 = \emptyset$. Then, applying one *pass* operation to derive the initial bipartition. Note that since different values of q and β will influence the result of the *Initial_Partition* procedure, we have a different initial bipartition for each run. Thus it gives MMP the chance of obtaining better optimised results. The initial bipartition is then further improved by a series of *passes* where an efficient module migration scheme is realised.

3 Module migration strategy

In this Section, we introduce the *pass* operation which is the kernel of MMP, and the β scaling scheme.

3.1 Pass operation

The following *Pass_Operation* procedure consists of three steps: (1) forward migration process; (2) reversing point searching; and (3) backward migration process.

Procedure *Pass_Operation*((V_1, V_2) , β , (S_m, S_M))

Input: (V_1, V_2) : the best bipartition obtained so far at the current run

β : is devised to determine the size of modules to be moved in the forward migration process, and $0 < \beta < 1$

(S_m, S_M) : the size constraints for the two partitions

Output: $(V_1, V_2)_{pass}$: the best bipartition at the current *pass*

begin

assign (V_1, V_2) to $(V_1, V_2)_{pass}$;

set the connection strengths of all modules in V_1 to zero;

$total_size := 0$; $ready := \text{FALSE}$;

randomly select a seed v from V_1 ;

/* Step 1: forward migration process is performed within the following **while** loop */

while (V_1 is not empty) **do**

begin

if ($total_size \geq \beta * S(V_1)$ and the gain value of the selected module $v > 0$) **then** $ready := \text{TRUE}$;

/* Step 2: the reversing point is found when the following **if** statement holds */

if ($ready = \text{TRUE}$ and the gain value of the selected module $v < 0$) **then** **break**;

move the selected module v from V_1 to V_2 ;

$total_size := total_size + s(v)$;

call *Update_Information*(v, V_1) to update the gains and the connection strengths of associated modules;

call *Choose_Module*(V_1) to choose the next module v to move;

end;

set the connection strengths of all modules in V_2 to zero;

$sr := S(V_1) / S(V_2)$;

randomly select a seed v from V_2 ;

/* Step 3: backward migration process is performed within the following **while** loop */

while ($sr \leq (S_M / S_m)$) **do**

begin

if ($sr \geq (S_m / S_M)$) **then**

if ($cut(V_1, V_2) < cut(V_1, V_2)_{pass}$) **then** assign (V_1, V_2) to $(V_1, V_2)_{pass}$;

move the selected module v from V_2 to V_1 ;

update size ratio sr ;

call *Update_Information*(v, V_2) to update the gains and the connection strengths of associated modules;

call *Choose_Module*(V_2) to choose the next module v to move;

end;

return the best bipartition $(V_1, V_2)_{pass}$;

end /* end of procedure */

Procedure *Update_Information*($v, Part$)

Input: v : the selected module to be moved in the forward or backward migration process

$Part$: $V_1(V_2)$ in the forward (backward) migration process

Variable: $C(v_k)$: the connection strength of module v_k

$w(e_j)$: the weight of net e_j

begin

update the gain values of the modules contained by $N(v)$;

/* update the connection strengths of associated modules */

for (each net e_j in $N(v)$) **do**

for (each module v_k in $M(e_j)$) **do**

if ($v_k \in Part$) **then** $C(v_k) := C(v_k) + w(e_j)$;

end /* end of procedure */

Step 1. Forward migration process

The module migration direction in this process is set from V_1 to V_2 . To move a cluster (which contains a seed randomly chosen from V_1) entirely into V_2 , the locality property is applied. In other words, besides evaluating the gains of modules in V_1 , the migration of the entire cluster is implicitly promoted by checking the connection strengths of modules in V_1 to those modules which have been moved from V_1 to V_2 in this process. Therefore, the module choosing subprocedure, *Choose Module*, chooses the module which has the highest gain in V_1 (in V_2 for the backward migration process). If a tie exists, the module which has the maximum connection strength to previously moved modules is chosen.

For each module v , the gain calculation is the same as FM [5], and the connection strength of v to the previously moved modules is the sum of the weights of those nets that connect v and the previously moved modules. For net-weight calculation, a weighting function $w: E \rightarrow R^+$ is defined for nets as follows:

$$w(e_j) = \begin{cases} \frac{1}{|M(e_j)|^q} & \text{if } |M(e_j)| > 1 \\ 0 & \text{if } |M(e_j)| = 0, 1 \end{cases} \quad (2)$$

for $j = 1, 2, \dots, m$

where parameter q is a positive real number and $|M(e_j)|$ denotes the cardinality of $M(e_j)$. The more modules a net contains, the more complicated the structure the net has; therefore, for a net e_j containing many modules, the modules in $M(e_j)$ should have little opportunity to move. For this purpose, we set the weight of a net to be in inverse proportion to the number of modules contained by the net. In eqn. 2, besides changing the weight of each net, different values of q influence the connection strengths of modules. In other words, searching for an appropriate q value is essential to the choice of suitable modules to move and to encourage the migration of natural clusters.

Each time after moving a chosen module v from V_1 to V_2 , associated information is modified by the *Update Information* subprocedure where the gains of the modules contained by $N(v)$ are updated and each net in $N(v)$ contributes its weight to the connection strengths for the modules in V_1 that have connections to v .

Step 2. Reversing point searching

When the accumulated size of the modules moved in the forward migration process is greater than a given value $\beta * S(V_1)$, it is time to search for a point to reverse the migration direction. Naturally, we want the *cut* of this reversing point to have fallen to a minimal value to indicate that a cluster has just been moved into V_2 . Under this consideration, two possible cases of reversing points are found as shown in Fig. 2. For both cases, since we do not know where the exact point of

minimal *cut* is, we have to keep a record of the point p where the *cut* begins to decrease and then it is time to reverse the migration direction at the point p' where the *cut* begins to increase. The implementation criterion for reversing point searching, as described in the *Pass Operation* procedure, is simply to check the gain value of the selected module since the gain value can reflect the changing of the *cut*.

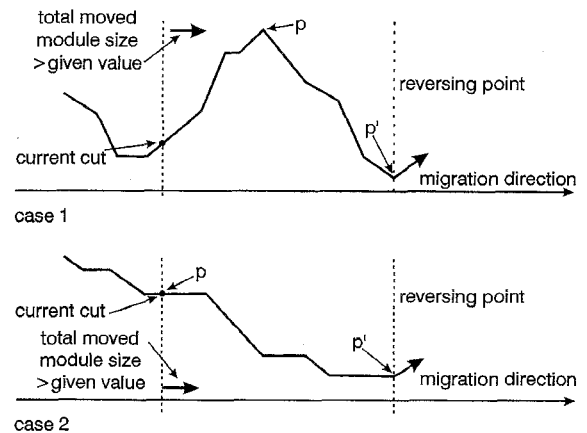


Fig. 2 Reversing point searching

The given value in both cases is set to $\beta * S(V_1)$

Case 1: the cut is increasing when total moved module size is greater than given value

Case 2: the cut is decreasing when total moved module size is greater than given value

Step 3. Backward migration process

Backward migration is essential to search for an acceptable size ratio result during each *pass* operation, and this process is the same as the forward migration process except for the migration direction. When the size ratio of the two partitions generated by the backward migration is no less than the lower bound of an acceptable size ratio range, (S_m/S_M) , we begin to keep a record of the feasible solutions obtained so far until the upper bound (S_M/S_m) is reached. Once the upper bound is crossed, the current *pass* is completed and the best result among the feasible ones is used to form another new starting point for the next *pass*.

3.2 β scaling scheme

In our algorithm, the value of β is set high at the beginning to allow larger clusters to settle down first, and is gradually reduced in the following *passes* to keep smaller clusters moving to their own proper partitions. To properly decide the reduction ratio of β , MMP has been applied to three examples using the following equation:

$$\alpha_i = \beta_{initial} * k^i \quad (3)$$

for $i = 0, 1, \dots, num_of_passes - 1$

where α_i denotes the value of β at $(i + 1)$ th *pass* operation. $\beta_{initial}$ denotes the initial value of β . k is the reduction ratio of β and $0 < k \leq 1$. num_of_passes (also used in MMP) is the number of *pass* operations in one run. In above experiment, num_of_passes was set to 100 and the algorithm was performed ten times for each example. The partitioning results (average *cut* and runtime) are listed in Table 1 for different values of k .

In the Table, on average, MMP produces good results with $k = 0.9$. Although experiment shows that 0.9 is better than the others, it is found to make the algorithm too slow to be effective from a practical point of view. As a result, to enhance the efficiency of

Table 1: Sensitivity of the MMP algorithm to reduction ratio k

primGA1		test02		19kstw		primGA1		test02		19kstw			
k	avg	s/run	avg	s/run	avg	s/run	k	avg	s/run	avg	s/run		
0.05	46.3	2.76	95.4	22.34	128.6	27.70	0.55	44.4	4.03	91.5	30.29	131.7	33.32
0.1	44.7	3.80	94.9	26.74	137.2	27.11	0.6	44.3	4.22	93.1	30.35	122.6	33.76
0.15	44.9	3.45	96.6	26.65	137.2	28.09	0.65	43.5	4.11	91.4	31.31	110.9	35.55
0.2	44.4	3.89	95.0	28.03	137.4	28.28	0.7	42.8	4.34	90.2	32.25	131.6	35.83
0.25	45.3	3.90	93.3	29.11	145.5	29.07	0.75	43.3	5.02	90.9	33.88	123.4	40.34
0.3	45.7	3.93	92.9	28.52	143.5	28.80	0.8	44.4	4.57	91.4	35.60	118.5	38.90
0.35	45.7	4.02	93.1	29.05	131.8	28.71	0.85	43.3	5.28	89.2	40.90	115.1	42.87
0.4	44.9	3.95	92.6	28.88	137.1	29.92	0.9	42.6	5.37	89.2	42.44	106.9	48.45
0.45	45.1	4.29	91.6	29.82	140.0	33.29	0.95	42.9	6.02	93.5	47.63	120.2	58.21
0.5	43.8	4.00	90.0	29.07	128.7	31.56	1.0	44.9	9.21	99.5	59.22	148.7	73.38

MMP, another equation for β was adopted in the current implementation as follows:

$$\alpha_i = \beta_{initial} * 0.9^{\lfloor \frac{i}{d} \rfloor} * 0.6^{i \bmod d}$$

$$\text{for } i = 0, 1, \dots, \text{num_of_passes} - 1 \quad (4)$$

where $\lfloor i/d \rfloor$ denotes the largest integer that is less than or equal to i/d . $i \bmod d$ denotes the remainder when i is divided by d . In the current implementation, d is set to 10.

4 Parameter management

In the MMP algorithm, two parameters, q (defined in eqn. 2) and $\beta_{initial}$ (the initial value of β), strongly influence the partitioning quality. After one run, to obtain a better bipartition for the next run with higher probability, we consider reassignment of the values of both parameters. Instead of a complete random reassignment, a parameter management scheme is developed to automatically assign suitable values to both parameters from given ranges according to the information provided by previous runs.

The basic idea of such a parameter management scheme is as follows: if a value r or a range of values $[r_b, r_r]$ of a certain parameter X used in the algorithm produces mostly good (bad) results, the probability that assigns the next iteration value of X close to r or $[r_b, r_r]$ should be kept high (low). To realise the above concept, we utilise the following set of functions in the parameter management scheme:

$$f(x) : x \rightarrow y \quad x \in [r_1, r_2] \quad (5)$$

$$c(x) : x \rightarrow \int_{r_1}^x f(z) dz \quad x \in [r_1, r_2] \quad (6)$$

$$g(x) : x \rightarrow c^{-1}(x) \quad x \in [0, 1] \quad (7)$$

where $y > 0$, $\int_{r_1}^{r_2} f(x) dx = 1$, and r_1, r_2 are the lower and upper bounds of parameter X , respectively. $f(x)$ is the pdf (probability density function) of X over its range $[r_1, r_2]$ (specified by the user). $c(x)$ is the corresponding cdf (cumulative distribution function) of $f(x)$, and $g(x)$ is the inverse function of $c(x)$. The following shows the *Parameter_Management* procedure.

Procedure *Parameter_Management*($cut_{current}, cut_{avg}$)

Input: $cut_{current}$: the *cut* at the current run
 cut_{avg} : the average *cut* from the first run to the current run

Output: q : the net-weighting parameter defined in eqn. 2

$\beta_{initial}$: the initial value of β

begin

compute the adjusted item J (used in eqn. 8) for $f_q(x)$ and $f_\beta(x)$ based on the difference between cut_{avg} and $cut_{current}$;

modify $f_q(x)$ over its range $[r_1', r_2']$ and $f_\beta(x)$ over its range $[r_1'', r_2'']$ using eqn. 8 with the computed J ;

$f_q(x)$ and $f_\beta(x)$ are divided by $\int_{r_1'}^{r_2'} f_q(x) dx$ and $\int_{r_1''}^{r_2''} f_\beta(x) dx$, respectively, to maintain them as pdfs;

compute $c_q(x)$ and $c_\beta(x)$ according to the modified $f_q(x)$ and $f_\beta(x)$, respectively;

compute $g_q(x)$ and $g_\beta(x)$ according to the computed $c_q(x)$ and $c_\beta(x)$, respectively;

/* assign new values to q and $\beta_{initial}$ by using the computed $g(x)$ function */

randomly select two real numbers t_q and t_β from $[0, 1]$;

$$q := g_q(t_q); \beta_{initial} := g_\beta(t_\beta);$$

end /* end of procedure */

The application of the *Parameter_Management* procedure to a parameter X is illustrated in Figs. 3–5. The corresponding $f(x)$, $c(x)$ and $g(x)$ of X after the $(i-1)$ th run (i.e. the previous run) are plotted in Fig. 3, where $f(x)$ is assumed to be uniform over its ranges for simplicity and r ($r_1 \leq r \leq r_2$) is the value of X generated for the i th run (i.e. the current run). After obtaining the current bipartition with $X = r$, the procedure modifies $f(x)$ according to the quality of the current bipartition. Precisely, $f(x)$ is modified by the following criterion:

$$f(x) = \begin{cases} f(x) * J & \text{if } x \in [r-b, r+b] \\ \frac{f(x)}{J} & \text{if } x \in [r_1, r-b] \cup [r+b, r_2] \end{cases} \quad (8)$$

where b is a small constant, and $J = e^{cut_{avg} - cut_{current}}$ is an adjusted item devised for the modification of $f(x)$ where c is a constant and $c > 1$, cut_{avg} and $cut_{current}$ are defined in the *Parameter_Management* procedure. The difference between cut_{avg} and $cut_{current}$ is used to evaluate the current bipartitioning quality. Based on this difference, there are three possible cases affecting the assignment of the value of X for the next run as follows. For case 1 (case 2) as shown in Fig. 4 (Fig. 5), the current *cut* is relatively good (bad), i.e. $cut_{current} < cut_{avg}$ ($cut_{current} > cut_{avg}$), so $J > 1$ ($J < 1$) and therefore,

the values of $f(x)$ are increased (decreased) for $x \in [r - b, r + b]$ and decreased (increased) for $x \in [r_1, r - b) \cup (r + b, r_2]$. This modification of $f(x)$ leads to the change of $g(x)$ such that the interval $[t_b, t_r]$ on the x -axis of $g(x)$ in Fig. 4 (Fig. 5) is enlarged (shrunk), where $[t_b, t_r]$ is the range of $c(x)$'s value for $x \in [r - b, r + b]$. The effect of the enlargement (shrink) of $[t_b, t_r]$ on the assignment of X 's next value can be observed clearly in $g(x)$ of Fig. 4 (Fig. 5), where the chance of the next iteration value of X , $g(t)$, being close to r is high (low) as a result of randomly selecting a real number t from $[0, 1]$. For case 3, $cut_{current} = cut_{avg}$ such that all functions remain unchanged. In fact, some peaks occur on the plot of $f(x)$ after multiple runs, implying that values of the corresponding parameter close to those peaks usually produce good results.

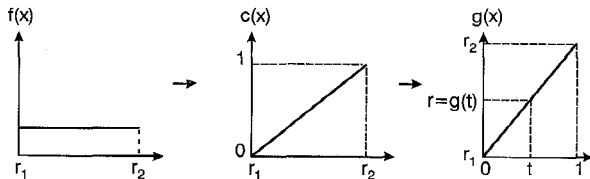


Fig.3 Corresponding functions of X after the $(i-1)$ th run (i.e. the previous run)

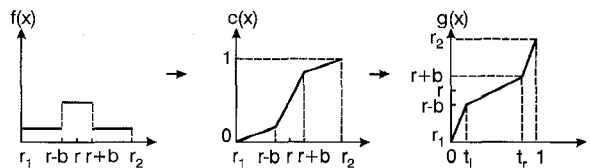


Fig.4 Corresponding functions of X after the i th run (i.e. the current run) if the current cut is relatively good

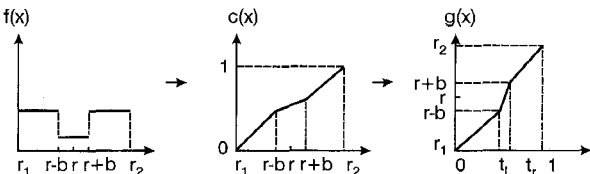


Fig.5 Corresponding functions of X after the i th run (i.e. the current run) if the current cut is relatively bad

Through the above set of self-adjusted probabilistic functions, as long as a parameter range is given, our algorithm can automatically generate optimal parameter subranges in which better results will be obtained with higher probability.

5 Experiments and analysis

5.1 Time complexity

Implementation of MMP is based on the bucket list data structure proposed in [5]. We maintain two such buckets in MMP, one for each partition. The complexity of moving a module and updating the associated module gains and connection strengths is $O(P)$, the same as FM [5], where P is the total number of pins. Let L be the total number of moved modules during one run (which is influenced by the number of *pass* operations and the β scaling scheme) and num_of_runs be the number of runs, then the total time complexity of MMP is thus $O(P * L * num_of_runs)$. One may keep num_of_runs a small constant. Therefore, the complexity can be bounded by $O(P * L)$.

5.2 Benchmark results

MMP was coded in C language and implemented on a SUN SPARC 10 workstation. The ranges of q and β were set to $[0.0, 2.5]$ and $[0.7, 0.9]$, respectively. By experiments on benchmarks, the number of *pass* iterations, num_of_passes (used in MMP), was set to 300 considering both the partitioning quality and the runtime efficiency. We have implemented the FM algorithm [5], and the STABLE algorithm was from [13]. The number of runs num_of_runs in MMP was set to 20. We also performed 20 runs for STABLE and the g value was set to 50 based on [13] (where g is the number of expected clusters). To obtain expected reasonable FM results, we set the number of runs for FM to 500 as suggested in [23]. By a series of experiments on benchmarks based on different deviations from the exact bipartition, we demonstrate the superiority of MMP over FM and STABLE for cases with strictly balanced partition sizes (i.e. deviation $\delta < 2\%$) and for cases with loosely balanced partition sizes (i.e. $\delta \geq 2\%$).

In Table 2, we compare our results to those of FM and STABLE with the size of each partition being allowed to have 0.1%, 1%, and 10% deviations from bipartition, i.e. the size ratios of the two partitions being 1:1.002, 1:1.0202, and 1:1.2222, respectively. Notice that in this Table, each module was given the actual area size. To illustrate the effectiveness of the parameter management, we also list the results (i.e. the numbers in parentheses in the MMP column of Table 2) obtained by MMP without parameter management, where the values of q and β are randomly selected for each run. Although the minimal *cut* results obtained with parameter management are not always better than those without parameter management, a more stable performance can be observed from the average *cut*.

According to Table 2, MMP generates better results than FM and STABLE in terms of the minimal *cut* and the average *cut* for $\delta = 0.1\%$ and $\delta = 1\%$. For $\delta = 10\%$, MMP outperforms FM in the minimal *cut* and the average *cut*, and is competitive with STABLE in either *cut*. On the other hand, on the average of three deviation cases, MMP shows 38% and 45% improvements in the minimal *cut*, and 52% and 36% improvements in the average *cut*, over FM and STABLE, respectively, for four large circuits industry2, industry3, avq.small, and avq.large. Hence, MMP tends to give better performance than FM and STABLE when the problem size becomes large.

From Table 2, it can also be seen that MMP and STABLE have the same time magnitude except for circuits avq.small and avq.large. Although MMP spends approximately five times more runtime than STABLE for avq.small and avq.large, it is worth it to achieve 47% and 44% improvements in the minimal *cut*, and 26% and 25% improvements in the average *cut*, for avq.small and avq.large, respectively, over STABLE (the percentage improvements are the average of three deviation cases). The runtime used in FM is much less than the others. Although FM works very quickly for each run, generally more than 500 runs are required to derive the same solution quality generated by MMP.

The effect of different deviations on the solution quality has been explored through a series of experiments with deviations from 0.1% to 10%. Curves of the average *cut* versus deviation for five tested circuits are shown in Figs. 6 and 7. From the Figures we observe

that the curves derived by STABLE are lower than those of FM for all circuits. However, MMP generates even better results. On average, FM and STABLE show good behaviour in the range 2% to 10%, i.e. the curves remain stable for cases with loosely balanced partition sizes. However, when the deviation is smaller than 2%, the curves for FM and STABLE rise rapidly and become very unstable. So in the case where strictly balanced partition sizes are required, both FM and STABLE have poor performance. As opposed to FM and STABLE, MMP is very stable under all conditions, i.e. MMP is less sensitive to the deviation.

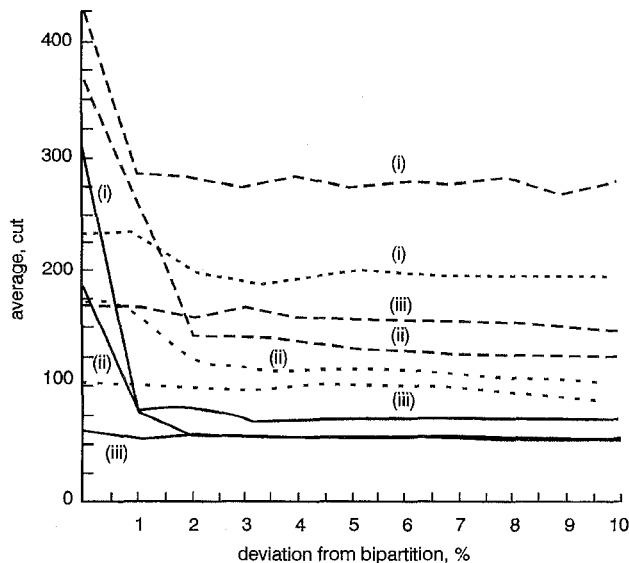


Fig. 6 Curves showing average cut versus deviation
(i) FM, (ii) STABLE, (iii) MMP
5655, test02, --- prim GA2

For further comparison, MMP was also compared with other state-of-the-art bipartitioners EIG1 [14], MELO [12], WINDOW [11], and PROP [4]. In Table 3, we compare the minimal *cut* of MMP (20 runs) with

those of FM (500 runs), EIG1, MELO, WINDOW, and PROP (20 runs) with $\delta = 0\%$ and $\delta = 10\%$. The minimal *cut* results of EIG1, MELO, WINDOW, and PROP are from [14, 12, 11, 4], respectively. Notice that in this Table, each module was given a unit size. From the Table, MMP is competitive with PROP in terms of the minimal *cut*, and MMP performs better than all other bipartitioners.

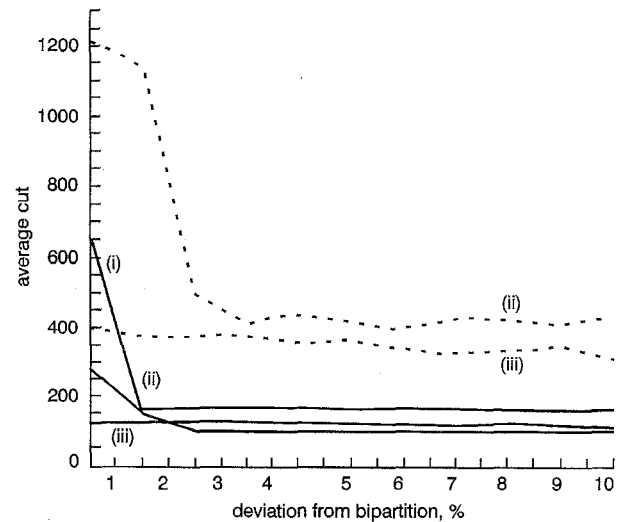


Fig. 7 Curves showing average cut versus deviation
(i) FM, (ii) STABLE, (iii) MMP
19kstw, industry2
The values of the whole curve of FM are above 1980 for circuit industry2

6 Conclusions

A new bipartitioner MMP has been developed. The module migration scheme adopted in MMP, being able to relax the size constraints temporarily and control the module migration direction, was shown to be very efficient. Also, to reduce the sensitivity of key parameters, a set of automatically adjusted probabilistic functions was incorporated in MMP. Experimental results obtained indicate that MMP improves the unstable

Table 3: Comparisons of MMP with other bipartitioners

Example	δ (%)	FM (500 runs) min	EIG1 min	MELO min	WINDOW min	PROP (20 runs) min	MMP (20 runs) min	Improvement over (%)				
								FM min	EIG1 min	MELO min	WINDOW min	PROP min
19ks	0	138			136	120	117	15			14	3
	10	126	179	119		105	104	17	42	13		1
primGA1	0	56			60	59	51	9			15	14
	10	47	75	64		47	45	4	40	30		4
primGA2	0	207			258	154	154	26			40	0
	10	182	254	169		143	134	26	47	21		6
test02	0	119			105	91	99	17			6	-9
	10	113	196	106		90	90	20	54	15		0
test03	0	76			67	58	58	24			13	0
	10	66	85	60		59	53	20	38	12		10
test04	0	87			61	58	53	39			13	9
	10	77	207	61		52	49	36	76	20		6
test05	0	105			101	82	91	13			10	-11
	10	102	167	102		79	78	24	53	24		1
test06	0	71			70	81	64	10			9	21
	10	62	295	90		76	60	3	80	33		21
industry2	0	476			392	254	246	48			37	3
	10	309	525	319		220	206	33	61	35		6
Average for percentage improvements for deviation $\delta = 0\%$								22			17	3
Average for percentage improvements for deviation $\delta = 10\%$								20	55	23		6

property of conventional module migration based bipartitioners such as FM [5], i.e. when considering the effect of different deviations from the exact bipartition, MMP is less sensitive to the deviation. On the other hand, MMP also outperforms FM and many recent state-of-the-art clustering-based bipartitioners. Most significantly, MMP is competitive with PROP [4] which claimed that it can generate better results than many other clustering-based bipartitioners.

In ongoing research, MMP will be extended to handle system partitioning for multichip modules where more performance considerations, such as I/O pin count, thermal, and timing constraints, must be dealt with.

7 Acknowledgments

The authors thank Professors C.K. Cheng and C.W. Yeh for supplying us both the testing benchmarks and the STABLE program.

This work was supported by the National Science Council, Taiwan, under Grant NSC86-2221-E002-066

8 References

- 1 ALPERT, C.J., and KAHNG, A.B.: 'Recent directions in netlist partitioning: a survey', *Integration: The VLSI J.*, 1995, **19**, pp. 1-81
- 2 DONATH, W.E.: 'Logic partitioning' in PREAS, B., and LORENZETTI, M. (Eds.): 'Physical design automation of VLSI systems' (Benjamin/Cummings, Menlo Park, CA, 1988), pp. 65-86
- 3 GAREY, M.R., and JOHNSON, D.S.: 'Computers and intractability: a guide to the theory of NP-completeness' (Freeman, San Francisco, CA, 1979)
- 4 DUTT, S., and DENG, W.: 'A probability-based approach to VLSI circuit partitioning'. Proceedings of ACM/IEEE *Design automation* conference, 1996
- 5 FIDUCCIA, C.M., and MATTHEYSES, R.M.: 'A linear-time heuristic for improving network partitions'. Proceedings of ACM/IEEE *Design automation* conference, 1982, pp. 175-181
- 6 HOFFMANN, A.G.: 'The dynamic locking heuristic - a new graph partitioning algorithm'. Proceedings of IEEE international symposium on *Circuits and systems*, 1994, pp. 173-176

- 7 KERNIGHAN, B.W., and LIN, S.: 'An efficient heuristic procedure for partitioning graphs', *Bell Syst. Tech. J.*, 1970, **49**, (2), pp. 291-307
- 8 KRISHNAMURTHY, B.: 'An improved min-cut algorithm for partitioning VLSI networks', *IEEE Trans. Comput.*, 1984, **C-33**, pp. 438-446
- 9 SANCHIS, L.A.: 'Multiple-way network partitioning', *IEEE Trans. Comput.*, 1989, **38**, (1), pp. 62-81
- 10 SCHWEIKERT, D.G., and KERNIGHAN, B.W.: 'A proper model for the partitioning of electrical circuits'. Proceedings of ACM/IEEE *Design automation* workshop, 1972, pp. 57-62
- 11 ALPERT, C.J., and KAHNG, A.B.: 'A general framework for vertex orderings with applications to circuit clustering', *IEEE Trans. VLSI Syst.*, 1996, **4**, (2), pp. 240-246
- 12 ALPERT, C.J., and YAO, S.Z.: 'Spectral partitioning: the more eigenvectors, the better'. Proceedings of ACM/IEEE *Design automation* conference, 1995, pp. 195-200
- 13 CHENG, C.K., and WEI, Y.C.: 'An improved two-way partitioning algorithm with stable performance', *IEEE Trans. Computer-Aided Des.*, 1991, **10**, (12), pp. 1502-1511
- 14 HAGEN, L., and KAHNG, A.: 'Fast spectral methods for ratio cut partitioning and clustering'. Proceedings of IEEE international conference on *Computer aided design*, 1991, pp. 10-13
- 15 RIESS, B.M., DOLL, K., and JOHANNES, F.M.: 'Partitioning very large circuits using analytical placement techniques'. Proceedings of ACM/IEEE *Design automation* conference, 1994, pp. 646-651
- 16 SAAB, Y.: 'A fast and robust network bisection algorithm', *IEEE Trans. Comput.*, 1995, **44**, (7), pp. 903-913
- 17 SHIN, H., and KIM, C.: 'A simple yet effective technique for partitioning', *IEEE Trans. VLSI Syst.*, 1993, **1**, (3), pp. 380-386
- 18 YANG, H., and WONG, D.F.: 'Efficient network flow based min-cut balanced partitioning'. Proceedings of IEEE international conference on *Computer aided design*, 1994, pp. 50-55
- 19 WEI, Y.C., and CHENG, C.K.: 'A two-level two-way partitioning algorithm'. Proceedings of IEEE international conference on *Computer aided design*, 1990, pp. 516-519
- 20 WEI, Y.C., and CHENG, C.K.: 'Towards efficient hierarchical designs by ratio cut partitioning'. Proceedings of IEEE international conference on *Computer aided design*, 1989, pp. 298-301
- 21 MCFARLAND, M.C.: 'Computer-aided partitioning of behavioral hardware descriptions'. Proceedings of ACM/IEEE *Design automation* conference, 1983, pp. 472-478
- 22 YEH, C.W., CHENG, C.K., and LIN, T.T.Y.: 'A general purpose, multiple-way partitioning algorithm', *IEEE Trans. Computer-Aided Des.*, 1994, **13**, (12), pp. 1480-1488
- 23 YEH, C.W., CHENG, C.K., and LIN, T.T.Y.: 'Optimization by iterative improvement: an experimental evaluation on two-way partitioning', *IEEE Trans. Computer-Aided Des.*, 1995, **14**, (2), pp. 145-153
- 24 SECHEN, C., and CHEN, D.: 'An improved objective function for mincut circuit partitioning'. Proceedings of IEEE international conference on *Computer aided design*, 1988, pp. 502-505