Two Novel Multiway Circuit Partitioning Algorithms Using Relaxed Locking

Ali Dasdan and Cevdet Aykanat

***Abstract*— All the previous Kernighan–Lin-based (KL-based) circuit partitioning algorithms employ the locking mechanism, which enforces each cell to move exactly once per pass. In this paper, we propose two novel approaches for multiway circuit partitioning to overcome this limitation. Our approaches allow each cell to move more than once. Our first approach still uses the locking mechanism but in a relaxed way. It introduces the phase concept such that each pass can include more than one phase, and a phase can include at most one move of each cell. Our second approach does not use the locking mechanism at all. It introduces the mobility concept such that each cell can move as freely as allowed by its mobility. Each approach leads to KL-based generic algorithms whose parameters can be set to obtain algorithms with different performance characteristics. We generated three versions of each generic algorithm and evaluated them on a subset of common benchmark circuits in comparison with Sanchis' algorithm (FMS) and the simulated annealing algorithm (SA). Experimental results show that our algorithms are efficient, they outperform FMS significantly, and they perform comparably to SA. Our algorithms perform relatively better as the number of parts in the partition increases as well as the density of the circuit decreases. This paper also provides guidelines for good parameter settings for the generic algorithms.**

***Index Terms*— Iterative improvement, Kernighan–Lin-based algorithms, move-based partitioning, multiway circuit partitioning, relaxed locking, very large scale integration (VLSI).**

I. Introduction

CIRCUIT partitioning deals with the task of dividing (partitioning) a given circuit into two or more parts such that the total weight of the signal nets interconnecting these parts is minimized while maintaining a given balance criterion among the part sizes. Since circuits can be appropriately represented by hypergraphs [1], we modeled circuits with hypergraphs and will use circuit and hypergraph terms interchangeably. Hypergraph partitioning has many important applications in very large scale integration VLSI layout [2]. The hypergraph partitioning problem is an NP-hard minimization problem [3], [4], and hence, we should resort to heuristic algorithms to obtain a good solution or hopefully a near-optimal solution. Moreover, such algorithms should run in low-order polynomial time because the problem sizes are usually very large.

Kernighan and Lin [5] proposed a two-way graph partitioning algorithm which became the basis for most of the subsequent partitioning algorithms, all of which we call the Kernighan–Lin-based (KL-based) algorithms. Kernighan and Lin's algorithm (KL) operates only on balanced partitions [6] and performs a number of passes over the cells of the circuit where each pass comprises a repeated operation of pairwise cell swapping for all pairs of cells. Schweikert and Kernighan [1] adopted KL to hypergraph partitioning. Fiduccia and Mattheyses [7] obtained a faster implementation (FM) of KL with the help of a new data structure, called the bucket data structure. This data structure basically contains bucket arrays and bucket lists and is explained in Section III-C in detail. FM can operate on unbalanced partitions and employs a single cell move instead of a swap of a cell pair at each step in a pass. Krishnamurthy [8] added to FM a look-ahead ability, which helps to break ties better in selecting a cell to move. Sanchis [9] generalized Krishnamurthy's algorithm to a multiway circuit partitioning algorithm. There are many other approaches to circuit partitioning; the reader is referred to the excellent survey in [10]. The simulated annealing algorithm (SA) [6], [11] is one of the most successful ones. In this paper, we will focus on Sanchis' algorithm (FMS) and SA.

A KL-based algorithm iterates a number of passes over the cells of the circuit until a locally minimum partition is found. Each cell is moved exactly *once* per pass to avoid thrashing or infinite loops [7], [8], and a *locking mechanism* is devised to enforce this restriction. That is, a cell is locked as soon as it is moved in a pass, and it remains locked until the end of the pass. As also independently observed in [12], *we claim that this locking mechanism is too restrictive and that it actually results in poor solution quality*. To remedy this problem, we propose two approaches. Each approach essentially allows each cell to be moved more than once but limits the total number of cell moves per pass. This limit can be more than the total number of cells in the circuit. Our first approach still uses the locking mechanism but in a different way and establishes the basis of the proposed "multiway partitioning by locked moves" algorithm (PLM). Our second approach does not use the locking mechanism at all. It introduces a new property for cell moves and bases the decision of a cell move on this property. This approach establishes the basis of the proposed "multiway partitioning by free moves" algorithm (PFM).

We did experiments on benchmark circuits for the proposed algorithms in comparison with FMS and SA. We compared

Manuscript received January 4, 1995; revised November 26, 1996. This paper recommended by Associate Editor, G. Zimmermann. This work was supported in part by the Commission of the European Communities, Directorate General for Industry under Contract ITDC 204-82166 and The Scientific and Technical Research Council of Turkey under Grant EEEAG-160.

A. Dasdan was with the Department of Computer Engineering and Information Science, Bilkent University, Ankara, Turkey. He is now with the Department of Computer Science, University of Illinois, Urbana-Champaign, IL 61801 USA.

C. Aykanat is with the Department of Computer Engineering and Information Science, Bilkent University, Ankara, Turkey.

Publisher Item Identifier S 0278-0070(97)02683-3.

the algorithms in terms of performance and running time. By the performance of an algorithm, we mean the quality of the solution that the algorithm delivers. In terms of performance, experimental results show that the proposed algorithms outperform FMS significantly and perform nearly as well as SA, although SA yields the best performance. In terms of running time, experimental results show that the running times of the proposed algorithms are far smaller than that of SA but larger than that of FMS. The proposed algorithms seem to perform well for both multiway partitioning and partitioning of sparse circuits.

The rest of the paper is organized as follows. Section II gives the basic definitions related to multiway hypergraph partitioning and introduces the notations. The proposed algorithms are presented in Section III. This section also discusses the data structure and the complexity analysis. The experimental framework giving the details of the experiments on benchmark circuits, and the experimental results for performance and running times are presented in Section IV. This section also includes some experiments on the parameters of our algorithms. Section V contains our conclusions and directions for future work.

II. Definitions and Notations

We model a circuit by a hypergraph $H = (V, E)$ where $V = \{v_i | 1 \leq i \leq n\}$ is the set of cells and $E = \{e_j | 1 \leq j \leq m\}$ is the set of nets. Each net is a subset of V. Each cell has a weight $w_i (w_i > 0)$, and each net has a weight $c_j (c_j > 0)$. The *degree* d_i of v_i is the number of nets connected to v_i, and the *degree* $|e_j| (|e_j| \geq 2)$ of e_j is the number of cells connected to e_j. The total number p of pins denotes the *size* of H where $p = \Sigma_{i=1}^{n} d_i$. The *average cell (net) degree* $D_v (D_e)$ is defined as $D_v = p/n$ $(D_e = p/m)$. The *density* D for $n \geq 2$ is defined as

$$D = \frac{\sum_{j=1}^{m} |e_j|(|e_j| - 1)}{n(n-1)} \tag{1}$$

which is similar to the definition in [13].

A partition Π of H is a *k way partition* if $\Pi = (P_1, \cdots, P_k)$, each *part* P_l is a nonempty subset of V, parts are pairwise disjoint, and the union of k parts is equal to V. A k way partition is also called a *multiway partition* if $k > 2$ and a *bipartition* if $k = 2$.

A net with at least one pin in a part is said to *connect* that part. A net that connects more than one part is said to be *cut*, otherwise *uncut*. The cost $\chi(\Pi)$ of Π, the *cutsize*, is equal to the sum of the weights of all cut nets. As in [9], each net e_j contributes an amount of c_j to the cutsize. The *cutset* of a partition is the set of all cut nets.

The *multiway circuit partitioning problem* involves a k way partitioning of H such that the cutsize is minimized and the partitioning is balanced. A partition Π is *balanced* if each part satisfies the *balance criterion* $L(P_l) \leq w(P_l) \leq U(P_l)$ where $L(P_l) = \lfloor (w(V)/k)(1 - \tau_l) \rfloor$ and $U(P_l) = \lceil (w(V)/k)(1 + \tau_l) \rceil$. Here, $w(P_l)$ is the total weight of the cells in P_l, $w(V)$ is the total weight of all the cells, and τ_l is a parameter satisfying $0 < \tau_l < 1$. We used $\tau_l = 0.1$ in our implementation as in similar works.

All KL-based algorithms select a cell to move based on its move gains. The *gain* $G_i(s, t)$ of the move of v_i from P_s to P_t is equal to the difference between the sum of the weights of the nets that v_i removes from the cutset and the sum of the weights of the nets that v_i adds to the cutset of the partition. Based on this definition, we readily see that the gain of v_i is equal to the decrease or negative increase in the cutsize that would result from moving v_i. The maximum move gain $G_{\max}$ is equal to the product of the maximum cell degree and the maximum net weight. All the gains fall in the interval $[-G_{\max}, G_{\max}]$.

III. Proposed Algorithms

Let N denote the total number of moves in a pass. In our approaches, each cell moves N/n times on the average, which can be more than one when $N > n$. At each step in a pass, a *direct multiway partitioning* algorithm considers all possible moves of a cell from its source part to *any* of the other parts (the target parts) in the partition and chooses the best of them, i.e., the one with the maximum gain. In this respect, FMS and the proposed algorithms are all direct multiway partitioning algorithms. For k way partitioning, there are $k - 1$ possible move directions or target parts for a single cell. We now give the specifics of the proposed approaches.

A. Multiway Partitioning by Locked Moves (PLM)

The generic PLM algorithm is given in Fig. 1. In this algorithm, each pass contains a number of *phases*, and each phase contains a sequence of tentative moves. Let N_{out} denote the number of phases in a pass and N_{in} the number of moves in each phase so that $N = N_{\text{out}} N_{\text{in}}$. In essence, PLM moves a number of cells in a phase, locks each cell as it moves, and unlocks all the cells moved in that phase before starting another phase. Each phase tries to find a better location for the cells, and the final location for a cell is determined only after all the phases, i.e., at the end of each pass. Unlocking a cell at the end of each phase except the last one is to give the cell one more chance of moving in the rest of the pass. The parameters of PLM are N_{out} and N_{in}. Since we have n cells, $N_{\text{in}} \leq n$, but N can be larger than n. The values that we used for these parameters are given in Section IV-A.

Note that step 14 in Fig. 1 finds the best partition encountered during a pass, and steps 15–17 move the cells to their final locations in that partition. The maximum prefix sum in step 14 of a pass is the difference between the cost of the partition at the start of this pass and the cost of the best partition reached. The moves in the maximum prefix subsequence constitute the sequence of the moves that lead to the best partition in this pass. The steps of PLM are almost the same as those of FMS, and PLM actually subsumes FMS for $N_{\text{out}} = 1$ and $N_{\text{in}} = n$. Running FMS with N_{in} moves per pass amounts to running PLM with *only one* phase, and so FMS with N_{in} moves per pass is not equivalent to PLM. The dynamic locking algorithm (DLA) algorithm [12] looks similar to PLM, but DLA is not equivalent to PLM in following major respects: DLA is for bipartitioning, but PLM is for multiway

Algorithm: Multiway Partitioning by Locked Moves.

Input: A k-way partition of $H = (V, E)$ with $|V| = n$, and its cutsize.

Output: A locally minimum k-way partition of H.

```
1  Initialize bucket list pointers
2  repeat /* for each pass */
3     for each of N_out phases do
4        Compute gains of cells, and unlock them
5        Insert cells into bucket lists using their move gains
6        repeat
7           Select a move (and so, a cell) with
            the max. move gain
8           Delete the cell from bucket lists and lock it
9           Tentatively make the move
10          Update gains of all affected cells
11       until N_in times or no move is possible
12       if N_in < n then /* if some buckets are non-empty */
13          Free bucket list nodes
14    Find the max. prefix sum G_L, and determine the max.
      prefix subsequence L of the tentative moves
15    if G_L > 0 then /* if there is a decrease */
16       Make permanent the moves in L
17       Decrease the cutsize by G_L
18 until G_L ≤ 0 /* until no more improvement */
```

Fig. 1. The generic direct multiway partitioning by PLM.

partitioning. DLA uses a different unlocking strategy in that it only unlocks some neighbors of the cell moved, but PLM unlocks all the cells moved in a phase. Finally, DLA imposes an upper bound on the maximum number of moves per cell, but PLM imposes an upper bound on the average number of moves per cell.

B. Multiway Partitioning by Free Moves (PFM)

The generic PFM algorithm is given in Fig. 2. This algorithm does not use the locking mechanism at all. Instead, the decision as to which cell to move is based on a new property of the cells. This new property is called the *mobility*. Each cell has a mobility value for each of its gains. These values determine the move capability of a cell.

The mobility $f_i(s,t)$ of the move of v_i from P_s to P_t is defined as

$$f_i(s,t) = (1 + n_i^{\alpha} \exp(-G_i(s,t)/T))^{-1} \tag{2}$$

where n_i, T, and α are parameters as defined below. The *move count* n_i of v_i counts the moves that v_i makes. When the cells are inserted into the bucket lists for the first time, it is set to one. It is then set to zero and incremented by one with each move. The parameter T is used to expand the range of f values into (0, 1). For a predefined interval $[\epsilon, 1-\epsilon]$ at $n_i = 1$ for f values, T is computed to be

$$\frac{1}{T} = \left(\frac{1}{G_{\max}}\right) \ln\left(\frac{1-\epsilon}{\epsilon}\right) \approx 4.6/G_{\max} \tag{3}$$

using (2), where $\epsilon > 0$ is a very small constant. We used $\epsilon = 0.01$ in our implementation. The mobility of a cell can be considered to be the probability that the cell can be selected for a move. So, the larger the mobility, the larger the chance of being selected for a move. As can be seen from (2), this probability increases as the gain gets larger but decreases as the move count gets larger. That is, the cell is penalized by the number of moves it makes. The parameter α determines the extent of this penalization. We found that $\alpha = 1/2$ is a good choice.

To utilize the bucket data structure, we have to devise a way of indexing the bucket arrays of this data structure using the mobility values. For this, we scale the mobility values to a range larger than (0, 1) and convert them to an integer. Thus, we map a cell with mobility $f_i(s,t)$ to a bucket list indexed by $F_i(s,t) = \lfloor S f_i(s,t) \rfloor$ where S denotes the *scale factor*. Henceforth, by the mobility of v_i, we mean its F value. The flooring in F introduces a slight randomization to the move selection process by mapping some cells with different f values into the same bucket list. The amount of this randomization is controlled by the scale factor in that a small scale factor introduces more randomness. As can readily be seen from the definition of F, F value for a cell can be computed in constant time, given the gain and the move count of the cell. Since each cell has $(k-1)$ possible move gains, each of which is for a target part, each cell also has $(k-1)$ mobility values.

PFM does N moves per pass and does not lock any cell. The same cell can be selected as many times as it has the maximum mobility value among all the cells. The steps of this algorithm are similar to those of PLM with the main difference being that the cells are evaluated on the basis of the mobility values rather than their gains. The parameters of PFM are the move count, α, ϵ, N, and S. In our implementation, we used the move count, α, and ϵ as given in this section. The values that we used for N and S are given in Section IV-A.

C. Data Structure and Initial Partitioning

Since our algorithms are similar to FMS, we adapted the bucket data structure, which was proposed in [9] for a direct multiway partitioning. We will explain this data structure for PFM and give the changes for PLM later. This data structure contains one *bucket array* of size S for each move direction. The bucket arrays are indexed by mobility values. Each move is stored in the arrays at an index corresponding to its mobility value. Since several moves can have the same mobility, each array cell is actually a linked list, called a *bucket list*. For constant time insertion and deletion of moves, the bucket lists are doubly-linked lists. There are $(k-1)$ move directions for each cell and k parts in the partition, so there are a total of $k(k-1)$ bucket arrays. The index of the array cell that contains a nonempty bucket list with the largest mobility value, called the top bucket list, is stored in a special variable to

Algorithm: Multiway Partitioning by Free Moves.

Input: A k-way partition of $H = (V, E)$ with $|V| = n$, and its cutsize.

Output: A locally minimum k-way partition of H.

```
1  Initialize bucket list pointers
2  repeat /* for each pass */
3     Compute gains of cells, and set move counts to 0
4     Insert cells into bucket lists using their mobility values
5     repeat
6        Select a move (and so, a cell) with
         the max. mobility
7        Tentatively make the move
8        Increment the move count of the cell
9        Update gains and mobility values of all affected cells
10    until N times or no move is possible
11    Find the max. prefix sum G_L, and determine the max.
      prefix subsequence L of the tentative moves
12    if G_L > 0 then /* if there is a decrease */
13       Make permanent the moves in L
14       Decrease the cutsize by G_L
15    Free all bucket list nodes
16 until G_L ≤ 0 /* until no more improvement */
```

Fig. 2. The generic direct multiway partitioning by PFM.

ensure constant time access to the best moves in each bucket array. An insertion into a bucket list is done at the head of the list, guaranteeing $O(1)$ time for the operation. To find a move with the maximum mobility, we search all $k(k-1)$ top bucket lists and select the first such move encountered during the process. If there is more than one move with the same maximum mobility, we select the one at the head of the list, obtaining $O(1)$ time for the removal. If the top bucket list becomes empty after the removal, we have to spend $O(S)$ time to update the index of the top bucket list [9]. This scheme is actually called last-in, first-out (LIFO) in [14]. FMS and PLM use the same data structure in the same way except that we should replace S with $2G_{\max}+1$ and mobility with gain in the foregoing discussion.

Like FMS, our algorithms need an initial k way partition as input. We generate an initial k way partition by *randomly* assigning each cell to one of the parts with the minimum size. This algorithm is actually an approximation algorithm [15].

D. Time Complexity Analysis

Since our algorithms are also KL-based, we need one procedure to compute the gains initially and another to update them after a move in such a way that the running time of our partitioning algorithms become linear in the size of the circuit. Our procedures are given in [16] due to lack of space. They can be considered as a straightforward generalization of those in [4] for multiway partitioning or a simplification of those in [9] for the first level gains. For each cell, the initial gain computation procedure computes a move gain for each part by using the definition of the move gain. Its running time is $O(pk)$. The gain update procedure is similar to the one given in [9]. It may end up checking and updating the gains of each cell on the nets that are connected to the cell moved. If locking is used, the total number of updates can be bounded from above as shown in [9], and the running time becomes $O(pkG_{\max})$ for a whole pass or $O(pkG_{\max})/n = O(kG_{\max}D_v)$ per move. If locking is not used as in PFM, we cannot bound the number of times a particular cell moves, and so we have to give a trivial upper bound such that the running time becomes $O(kSD_{v,\max}D_{e,\max})$ per move, where $D_{v,\max}$ is the maximum cell degree and $D_{e,\max}$ is the maximum net degree. FMS, PLM, and PFM use almost the same gain update procedure, the difference being that the gain update procedures for FMS and PLM do not consider locked moves.

Given the running times above, we can derive the total running time of the algorithms as follows. The time complexity of FMS is $O(pk(k+G_{\max}))$ per pass as given in [9]. Since each pass of PLM comprises N_{out} phases, and each phase has a running time of $O(pk(k+G_{\max}))$, PLM runs in $O(N_{\text{out}}pk(k+G_{\max}))$ time per pass. For PFM, we cannot get a simple running time expression due to the difficulty in constraining the total number of moves for a each cell. The dominant steps for PFM's time complexity are steps 1, 6, and 9. These steps are also dominant for PLM but the time complexity of each of these steps is subsumed in the overall running time. Since there are $k(k-1)$ bucket arrays each with size S, the time to initialize all list pointers (step 1) takes $O(k^2S)$ time, and the time to select a cell to move (step 6) takes $O(k^2)$ time. There are N moves per pass, and step 9 takes $O(kSD_{v,\max}D_{e,\max})$ time, so the loop of step 5 takes $O(N(k^2+kSD_{v,\max}D_{e,\max})$ time. Hence, the overall running time of PFM is $O(pk+k^2S+N(k^2+kD_{v,\max}D_{e,\max}S))$ per pass. The total number of passes that each of these algorithms does is not known in advance but usually less than a small constant, and so these per-pass running times also correspond to the total running times. The time complexity of each algorithm can be reduced by using a binary heap to speed up the move selection step, e.g., that of FMS reduces to $O(pk(\lg k+G_{\max}))$ per pass [9].

E. Search Space and Algorithm Behavior

This section comments on the size of each algorithm's search space and gives plots of how they behave during partitioning. By the search space of an algorithm, we mean the set of solutions (partitions) that the algorithm examines during partitioning. The sizes of the search spaces of our algorithms are larger compared to that of FMS, and this is used to give an intuition for their better performance and larger running times.

Every partitioning algorithm developed after FM has used the *move-neighborhood structure*. A partitioning algorithm with a move-neighborhood structure proceeds from one partition to another by means of a single cell move. Our algorithms as well as FMS use the move-neighborhood structure. Let $\mathcal{N}[A]$ denote the number of solutions explored per pass by a

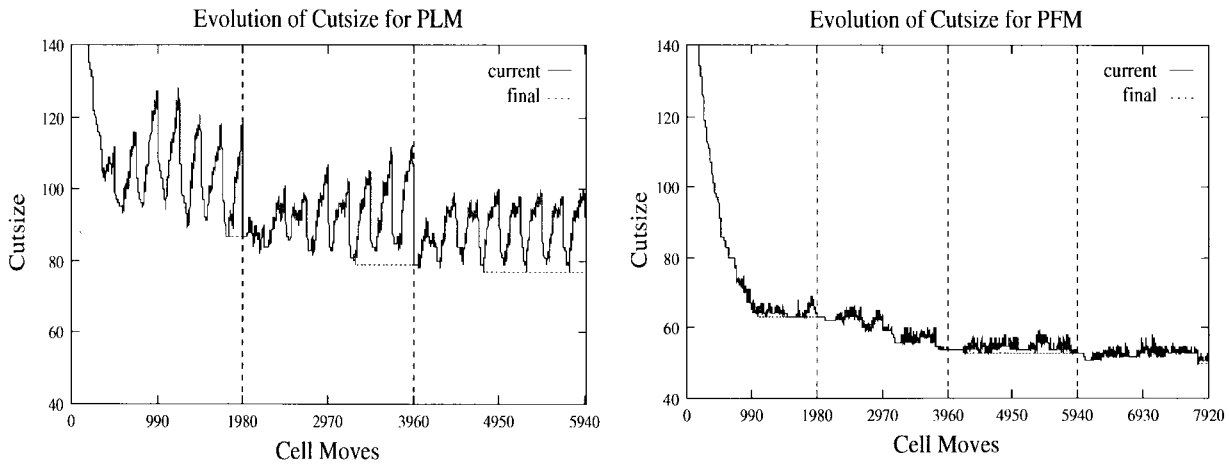

Fig. 3. Evolution of cutsize with cell moves for (a) PLM and (b) PFM on s838 with 495 cells.

KL-based algorithm A. Then, the total number of partitions explored by A is equal to the product of the number of passes that A makes and $\mathcal{N}[A]$. The number of passes is usually less than ten but varies with each choice of both the algorithm and the problem. A move-neighborhood structure for k way partitioning of an n-cell circuit contains at most $n(k-1)$ partitions at each step in a pass, as each of n cells can move to any of the $(k-1)$ target parts. Note that, for an algorithm A using the locking mechanism, only unlocked cells should be considered when computing $\mathcal{N}[A]$. Then, we can obtain the following bounds: $\mathcal{N}[FMS] \leq (k-1)n(n+1)/2, \mathcal{N}[PLM] \leq N(k-1)(2n-N_{\text{in}}+1)/2$, and $\mathcal{N}[PFM] \leq Nn(k-1)$. If all the moves in a pass are possible, these inequalities become equalities.

Intuitively, we expect that the larger the number of partitions explored by an algorithm, the better the quality of the solution delivered by that algorithm as well as the larger the running time of that algorithm. Our experimental observations provide support for this intuitive view, yet they also show that this intuitive fact is not the only factor affecting the performance. Also note that almost all of the partitions explored by FMS per pass are different. However, some of the partitions explored by PLM and PFM per pass may be the same since they allow multiple moves for a cell. Although in general, PFM beats PLM and PLM in turn beats FMS in terms of the total number of solutions explored, we have some exceptions as given in Section IV-B1.

As for how the proposed algorithms behave, Fig. 3(a) and (b) illustrate the evolution of the cutsize with the cell moves in PLM2 and PFM2, respectively, for four-way partitioning of `s838` with 495 cells. This circuit is a small circuit from the `Partitioning93` test suite. PLM2 and PFM2 are two versions of PLM and PFM, respectively, and are presented in Section IV-A. Each interval between two successive vertical lines corresponds to a pass. The "current" cutsize curve is for tentative moves during a pass, and the "final" cutsize curve is for the permanent moves. These two curves usually coincide in the plots. The initial cutsize for both algorithms is 374, and the final cutsizes for PLM2 and PFM2 are 77 and 50, respectively. In Fig. 3(a), each spike roughly corresponds to a phase. In fact, Fig. 3(a) shows the typical behavior of a KL-based algorithm with locking, e.g., FMS has the same behavior. Thus, PLM2 and FMS do not benefit from most of the moves in a pass, indicating that locking does not prevent thrashing. As seen in Fig. 3(b), PFM2, on the other hand, utilize most of them. PFM2 smoothes out the spikes, yielding a more steady convergence.

IV. Experimental Framework, Results, and Discussion

This section presents the details of the experimental framework and gives the experimental results. We evaluated three versions of both PLM and PFM in comparison with FMS and SA on a subset of benchmark circuits.

A. Experimental Framework

By setting the parameters of the generic PLM and PFM algorithms to different values, we generated three versions of each of these algorithms. Henceforth, these versions of PLM and PFM will be referred to as PLMi and PFMi, respectively, for $i = 1, 2, 3$. The values of the parameters and the names of these versions are presented in the following table, where $R = S/(2G_{\max}+1)$ is the ratio of the bucket size in an PFMi algorithm to that of FMS.

Versions of PLM and PFM

N	N_{out}	N_{in}	*Name*	N	R	*Name*
n	2	$n/2$	PLM1	n	2	PFM1
nk	$2k$	$n/2$	PLM2	nk	8	PFM2
nk^2	$2k^2$	$n/2$	PLM3	nk^2	128	PFM3

Let $N[A]$ denote the number of cell moves in a pass of a KL-based algorithm A. Then, for this setting, we have $N[\text{FMS}] = N[\text{PLM1}] = N[\text{PFM1}], N[\text{PLM2}] = N[\text{PFM2}]$, and $N[\text{PLM3}] = N[\text{PFM3}]$. We say that a PFM$i$ algorithm *corresponds* to a PLMj algorithm or vice versa if $i = j$, e.g., PFM2 and PLM2 correspond to each other. Note that N is chosen to be a function of n and k rather than a constant, as the size of each algorithm's search space is proportional to these problem parameters.

TABLE I
PROPERTIES OF BENCHMARK CIRCUITS (n = NUMBER OF CELLS, m = NUMBER OF NETS, p = NUMBER OF PINS, D_v = AVERAGE CELL DEGREE, D_e = AVERAGE NET DEGREE, $D_{v,\max}$ = MAXIMUM CELL DEGREE, $D_{e,\max}$ = MAXIMUM NET DEGREE, AND D = DENSITY)

Full Name	*Short Name*	n	m	p	D_v	D_e	$D_{v,max}$	$D_{e,max}$	D
struct	struct	1888	1888	5375	2.85	2.85	4	16	0.004490
primary2	prim2	3014	3029	11219	3.72	3.70	9	37	0.008204
c7552	c7552	2247	2140	6171	2.75	2.88	5	137	0.008676
c2670	c2670	924	860	2375	2.57	2.76	5	30	0.011444
industry2	ind2	12142	12949	47193	3.89	3.64	12	584	0.011770
primary1	prim1	833	902	2908	3.49	3.22	9	18	0.018056
industry1	ind1	2271	2186	7731	3.40	3.54	9	318	0.038276
biomed	bio	6417	5711	20912	3.26	3.66	6	860	0.062038
test06	test06	1752	1641	6638	3.79	4.05	6	388	0.079830

All the algorithms were coded in C. Our implementation of Sanchis' algorithm, i.e., FMS, is better than Sanchis' original implementation because FMS uses the LIFO tie-breaking scheme, but the original implementation uses the random tie-breaking scheme, which is consistently outperformed by LIFO as advocated in [14]. A comparison of the performance of FMS with that of Sanchis' (even with level 4) as given in [14] on some circuits such as `prim1` and `prim2` also confirms this fact.

All of the experiments were done on a Sun SPARC 10 under SunOS operating system. We used nine benchmark circuits as our test instances from the `LayoutSynth92` and `Partitioning93` test suites in `ACM/SIGDA Design Automation Benchmarks`. The properties of these circuits are summarized in Table I. The circuits in all the tables in Section IV are ordered in ascending density. We deleted certain nonessential features of these circuits as in [1] and [7]. All the nets with only one cell were removed, and each net containing a cell more than once was enforced to contain that cell only once. In order to give to the reader a better interpretation of the experimental results, we set each cell and net weight to one. However, it should be noted that our formulation as well as our implementation allow nonuniformly weighted cells and nets without any change.

In our experiments, we used a slightly modified version of PFM in order to improve performance by eliminating some zero-gain moves. The new version did not select a cell in two successive moves. We used a table lookup technique to speed up the calculation of the exponential function values in (2) as in [6].

We set the number k of parts to 2, 4, 6, and 8 as in similar works. Following [17] and [18], we ran FMS 500 times, each of our algorithms 30 times, and SA ten times on each test instance starting from different initial partitions. The running time of SA on the largest circuit `ind2` for $k = 2$ was so large that we could not obtain any performance data for SA on this circuit. To allow a fair comparison between the algorithms, we used the same initial partition generation algorithm and the same balance criterion for all the algorithms. Moreover, the level parameter of FMS was set to one as the level parameter concept is applicable to our algorithms, but we did not incorporate it. The running time of an algorithm is the sum of its system and user times and includes all the times from that of reading the input circuit up to that of outputting a final locally minimum partition. The parameter settings discussed in this section will be referred to as *the default settings*.

We implemented SA according to the cooling schedule in [6]. This cooling schedule was proposed for bipartitioning and also used in a work [17] similar to ours. We also incorporated the guidelines supplied in [6], [11], and [19]. We made the following three changes in the cooling schedule in [6] to adapt it to multiway partitioning. The starting temperature was set to ten as in [11] where the acceptance rate was larger than 90%, whereas Johnson *et al.* [6] suggested a starting temperature where the acceptance rate was 40% for a speedup. This change did not affect the performance but increased the running time a bit. The termination condition was met when either the acceptance rate was less than 2% as in [6] or the same cutsize was encountered $n/2$ times. This change did not degrade the performance. We used it merely to eliminate unnecessary moves before the convergence. The final change was in the form of the cost function. Johnson *et al.* [6] used a penalty function approach so that their scheme allowed infeasible partitions to be accepted. In order to ensure that each algorithm we compared selects a move in the same way, we did not use the penalty function approach in our implementation of SA. This change may degrade the performance slightly if the balance criterion is tight, but it seems to reduce the running time.

B. Results with Default Settings and Discussion

Table II presents the average and minimum cutsizes found by each algorithm. Table III presents the average running time of each algorithm. The bottom of Table II also includes the average percent improvements of the algorithms with respect to FMS where the averages were taken over all the circuits. We gave these percentages only to give a quick perspective to the reader. In all the tables, the bold values in a row correspond to the best values for that row. Recall that the best cutsize is the smallest cutsize, and the best running time is also the smallest running time. In general, the performance of each algorithm differs when $k = 2$ and $k > 2$. We examine these two cases separately.

1) Results—Performance at Bipartitioning: From Table II, we observe the following for the solution quality at biparti-

TABLE II
AVERAGE (MINIMUM) CUTSIZES FOR BENCHMARK CIRCUITS. BOLD VALUES ARE THE BEST VALUES IN EACH ROW

PROBLEM		AVERAGE (MINIMUM) CUTSIZES							
Name	k	FMS	PLM1	PLM2	PLM3	PFM1	PFM2	PFM3	SA
struct	2	56.8 (40)	62.3 (43)	54.5 (43)	54.7 (47)	99.9 (58)	60.2 (33)	59.2 (33)	67.2 (36)
	4	300.4 (202)	259.3 (206)	230.2 (175)	211.4 (173)	166.6 (136)	126.3 (83)	111.2 (76)	130.0 (121)
	6	408.4 (302)	321.3 (268)	289.5 (231)	267.1 (220)	260.1 (212)	214.0 (159)	183.3 (122)	160.6 (145)
	8	496.8 (406)	432.2 (391)	394.0 (310)	371.6 (280)	369.5 (317)	303.7 (235)	295.1 (257)	180.0 (163)
prim2	2	278.2 (154)	367.7 (238)	312.5 (190)	268.9 (205)	284.2 (218)	259.0 (215)	239.7 (182)	226.0 (182)
	4	828.7 (731)	771.8 (713)	718.2 (635)	671.6 (602)	657.1 (527)	454.1 (409)	416.4 (351)	424.2 (388)
	6	968.5 (883)	873.2 (821)	833.8 (785)	822.8 (765)	839.4 (774)	592.6 (519)	537.4 (479)	508.0 (487)
	8	1043.2 (991)	982.6 (936)	941.0 (883)	901.5 (866)	942.3 (815)	666.5 (598)	626.5 (576)	565.8 (535)
c7552	2	48.6 (21)	61.4 (38)	56.4 (37)	52.6 (32)	107.6 (59)	90.8 (56)	76.9 (37)	83.6 (76)
	4	388.2 (295)	340.8 (298)	350.5 (287)	330.4 (256)	303.7 (243)	184.8 (157)	162.7 (120)	171.4 (159)
	6	498.2 (445)	435.9 (400)	399.6 (359)	372.8 (315)	420.7 (369)	262.6 (211)	218.1 (164)	219.2 (208)
	8	550.7 (501)	517.1 (455)	484.0 (442)	444.0 (409)	478.4 (403)	306.1 (263)	257.8 (200)	259.4 (243)
ind2	2	592.5 (243)	900.2 (667)	756.8 (541)	661.8 (314)	879.8 (690)	768.2 (577)	699.4 (566)	N.A.
	4	2541.4 (2097)	2339.2 (1996)	2159.0 (1984)	2045.2 (1720)	2119.4 (1849)	1258.2 (941)	1020.0 (803)	N.A.
	6	2902.1 (2741)	2535.2 (2373)	2307.2 (2150)	2249.0 (1985)	2664.9 (2427)	1565.5 (1362)	1314.8 (1041)	N.A.
	8	3093.7 (2929)	2809.8 (2544)	2635.2 (2393)	2333.4 (2081)	2816.0 (2607)	1766.4 (1575)	1506.0 (1236)	N.A.
prim1	2	76.4 (48)	82.6 (61)	74.3 (51)	65.9 (49)	84.7 (65)	72.4 (51)	72.8 (47)	73.8 (67)
	4	205.8 (160)	181.5 (159)	161.0 (138)	144.6 (127)	152.6 (125)	123.6 (111)	111.8 (97)	113.6 (109)
	6	244.7 (201)	217.4 (181)	192.7 (170)	172.6 (151)	199.6 (172)	145.4 (123)	130.6 (114)	131.0 (126)
	8	274.0 (224)	253.7 (222)	218.3 (182)	191.5 (161)	227.8 (196)	159.0 (137)	148.7 (126)	144.2 (131)
ind1	2	57.9 (20)	61.5 (30)	62.5 (27)	53.8 (25)	93.9 (30)	82.9 (37)	77.6 (39)	71.2 (48)
	4	438.2 (341)	374.1 (269)	320.0 (235)	295.7 (198)	339.9 (212)	190.7 (118)	148.2 (93)	184.8 (153)
	6	530.6 (438)	475.0 (414)	445.2 (391)	413.0 (348)	461.4 (378)	305.3 (235)	266.4 (200)	263.4 (219)
	8	579.6 (522)	550.6 (506)	521.6 (489)	474.3 (424)	529.5 (455)	381.4 (284)	339.0 (249)	293.2 (278)
bio	2	127.4 (83)	205.1 (171)	185.3 (126)	180.1 (165)	324.2 (277)	223.9 (185)	190.4 (148)	283.2 (272)
	4	729.1 (561)	689.0 (628)	641.4 (577)	570.8 (505)	630.9 (569)	361.6 (322)	296.3 (273)	404.0 (397)
	6	905.1 (779)	738.8 (663)	672.7 (622)	656.3 (613)	786.9 (678)	529.9 (499)	497.1 (461)	485.8 (476)
	8	987.6 (887)	855.2 (748)	798.2 (718)	745.7 (666)	839.8 (727)	631.6 (594)	600.8 (569)	564.8 (517)
test06	2	89.5 (62)	85.6 (73)	84.1 (70)	80.8 (65)	140.2 (91)	93.3 (72)	91.8 (74)	81.8 (75)
	4	289.7 (187)	273.0 (237)	262.5 (219)	248.6 (217)	264.6 (190)	153.8 (104)	137.8 (107)	151.2 (137)
	6	356.5 (297)	328.4 (289)	303.6 (265)	286.0 (248)	321.4 (264)	204.5 (158)	175.1 (144)	173.0 (153)
	8	394.0 (344)	374.4 (345)	349.6 (319)	319.0 (275)	361.2 (299)	233.1 (183)	203.2 (157)	191.8 (170)
Avr % Improvement in Avr (Min) Cutsizes wrt FMS									
	2	0.0 (0.0)	-23.9 (-64.8)	-12.1 (-42.0)	-3.0 (-32.9)	-66.5 (-102.2)	-29.2 (-69.6)	-18.5 (-50.0)	-12.7 (-74.5)
	4	0.0 (0.0)	9.8 (-1.6)	16.5 (7.4)	22.6 (15.0)	21.8 (18.3)	50.0 (48.5)	56.8 (55.2)	50.8 (39.5)
	6	0.0 (0.0)	13.0 (9.4)	20.1 (16.6)	24.5 (22.6)	16.0 (15.4)	43.4 (44.9)	50.4 (52.6)	51.4 (46.9)
	8	0.0 (0.0)	8.1 (6.4)	14.8 (14.5)	21.8 (22.9)	13.3 (15.8)	39.4 (42.5)	45.0 (47.8)	49.8 (47.2)

tioning. For the average performance, PLM3 and FMS deliver the best results, but PLM3 beats FMS on five of the eight circuits. For the minimum performance, FMS outperforms all the others except that for `struct`, the most sparse circuit, PFM2 and PFM3 produce the smallest cutsize. Generally, both the average and minimum performance of PLMi and PFMi gets better as i increases, i.e., as the number of moves per pass increases. Moreover, the PLMi algorithms perform better than the corresponding PFMi algorithms. SA performs nearly as well as PFM3. PLM3 achieves the best average performance, 14% on `prim1`, and PFM3 achieves the best minimum performance, 18% on `struct`, both relative to FMS.

The relatively poor performance of most of our algorithms for bipartitioning seems a bit surprising as we expect that they examine more partitions, and so they must perform better than FMS. The following reasons seem to account for this result. First, FMS executed the largest number of passes, making the size of its search space comparable to that of PLM1 and PFM1. Second, FMS is the most "unstable" algorithm for bipartitioning in the sense that the disparity between the maximum and the minimum cutsizes it found was the largest. The instability of FMS makes its average performance worse but helps it beat all the others for minimum performance. Third, the total number of local minima at bipartitioning is not so large, and so a more greedy strategy like the locking mechanism of FMS pays off. Fourth, the disparity in gain values for bipartitioning is small, making the number of zero-gains larger and so making the move selection process difficult

TABLE III
EXECUTION TIME AVERAGES FOR BENCHMARK CIRCUITS. BOLD VALUES ARE THE BEST VALUES IN EACH ROW

PROBLEM		*EXECUTION TIME AVERAGES (in seconds)*							
Name	k	FMS	PLM1	PLM2	PLM3	PFM1	PFM2	PFM3	SA
struct	2	**2.0**	**1.8**	2.8	4.5	**2.0**	3.2	9.1	155.3
	4	**3.3**	5.4	17.1	61.0	6.6	13.4	67.5	415.1
	6	**4.9**	8.2	35.0	172.4	11.2	31.9	220.9	697.9
	8	**6.9**	9.4	58.2	409.3	13.6	53.7	383.2	950.3
prim2	2	5.6	**4.1**	7.4	14.3	4.3	6.5	22.1	914.0
	4	**7.3**	8.9	42.5	168.7	11.6	37.0	172.5	3565.8
	6	**9.4**	17.2	76.3	337.9	15.8	79.6	453.6	18406.7
	8	**13.2**	21.4	127.4	860.4	20.7	123.1	743.9	37304.2
c7552	2	2.9	2.7	4.1	6.9	**2.3**	3.3	8.8	565.0
	4	**5.0**	8.1	22.4	84.9	6.2	17.6	82.0	3742.7
	6	**6.8**	14.1	53.5	250.4	9.7	43.7	258.1	10692.5
	8	**9.76**	18.1	84.4	646.6	14.5	79.7	494.3	17971.8
ind2	2	55.2	**18.3**	29.2	52.7	38.5	52.8	144.2	>46800
	4	64.6	**51.8**	180.3	702.4	84.4	242.3	1053.7	N.A.
	6	96.7	**92.1**	381.6	1860.5	111.3	541.4	3111.0	N.A.
	8	132.4	**110.4**	637.4	4855.2	189.6	1007.4	7305.0	N.A.
prim1	2	1.0	1.0	1.4	2.6	**0.6**	1.0	3.4	205.1
	4	**1.3**	2.2	6.8	22.1	2.1	4.6	25.0	1235.5
	6	**2.1**	3.6	14.1	70.1	3.4	10.6	84.7	2788.4
	8	**2.8**	5.1	22.4	180.9	4.4	20.6	151.0	2747.8
ind1	2	3.6	3.4	4.6	7.9	**2.6**	4.1	11.0	526.5
	4	**4.8**	8.7	31.7	106.0	6.9	22.3	121.6	4932.2
	6	**6.5**	13.2	51.7	269.2	9.7	48.6	287.1	11504.7
	8	**8.6**	15.3	77.0	604.1	11.2	65.8	479.6	19825.6
bio	2	19.5	**7.2**	10.8	17.8	14.0	15.9	56.6	336.7
	4	24.9	**21.7**	55.1	192.2	28.3	59.5	299.2	837.8
	6	**29.5**	36.2	117.4	538.7	40.6	108.6	582.6	1262.8
	8	**40.8**	50.2	205.4	1416.8	58.9	157.9	948.3	1825.0
test06	2	2.1	2.1	3.6	6.4	**1.8**	2.7	7.5	145.2
	4	**3.3**	4.4	14.5	59.8	3.7	10.7	63.9	3237.8
	6	**4.2**	6.5	30.2	156.7	5.6	25.3	167.4	8134.7
	8	**5.4**	8.3	50.6	375.4	7.1	44.1	390.1	13701.6

especially for the PFMi algorithms. We did an experiment to penalize zero-gain moves more by setting α to one in (2). We observed an overall performance improvement.

2) Results—Performance at Multiway Partitioning: From Table II, we observe the following for the solution quality at multiway partitioning. For the average performance, SA delivers the best results, and PFM3 comes second. For the minimum performance, PFM3 delivers the best performance, and SA comes second. Like the case at bipartitioning, both the average and minimum performance of PLMi and PFMi generally gets better as i increases. Unlike the case at bipartitioning, the PFMi algorithms perform better than the corresponding PLMi algorithms. That even PFM2 beats PLM3 despite $N[\mathrm{PLM3}] > N[\mathrm{PFM2}]$ indicates that N is not the only factor that improves the performance. The mobility concept also helps a lot. The mobility concepts pays off because PFM2 outperforms all the PLMi algorithms. Relative to FMS, PFM3 yields the best average and minimum performance, 66% and 73% both on `ind1`, respectively. Note that the bottom of Table II gives overall relative performance figures in terms of percentages.

As the search space is larger and more difficult to explore at multiway partitioning, better search strategies are needed for a thorough exploration. The superiority of our algorithms with respect to FMS reveals their effectiveness and supports our original claim. Note that the relative performance of our algorithms gets better as we move up in the tables. Since the circuits are ordered in ascending density in the tables, this observation shows that our algorithms perform relatively better as the circuit gets more sparse. There are some anomalies

TABLE IV
CUTSIZE AVERAGES BY PFM3 FOR DIFFERENT VALUES OF THE SCALE FACTOR S. ($G_{\max}$ = MAXIMUM MOVE GAIN POSSIBLE.) BOLD VALUES ARE THE BEST VALUES IN EACH ROW

PROBLEM		$R = S/(2G_{max}+1)$								
Name	k	0.5	1	2	4	8	16	32	64	128
struct	2	207.4	80.0	76.4	71.6	55.4	62.2	59.0	51.0	**42.0**
	4	328.8	137.8	121.6	116.2	107.6	116.4	123.4	121.0	**114.8**
	6	438.8	246.4	236.4	224.0	215.8	210.8	215.8	219.6	**206.4**
	8	485.6	317.0	296.4	289.6	276.6	275.8	264.2	**264.0**	275.6
c2670	2	55.8	44.2	43.4	42.4	42.0	45.6	**38.0**	40.4	40.6
	4	85.4	81.6	74.8	73.8	73.6	72.2	73.4	**70.0**	70.4
	6	109.0	97.2	88.8	88.6	81.8	81.2	78.6	**78.0**	78.6
	8	121.0	103.6	99.2	92.2	92.0	88.8	86.4	89.6	**84.6**

TABLE V
CUTSIZE AVERAGES BY PLM FOR DIFFERENT VALUES OF N_{in} AND N. (N = TOTAL NUMBER OF MOVES PER PASS, N_{in} = NUMBER OF PHASES PER PASS, n = NUMBER OF CELLS, AND k = NUMBER OF PARTS.) BOLD VALUES ARE THE BEST VALUES IN EACH ROW

PROBLEM		$N = n$				$N = nk$				$N = nk^2$			
		N_{in}				N_{in}				N_{in}			
Name	k	n/4	2n/4	3n/4	4n/4	n/4	2n/4	3n/4	4n/4	n/4	2n/4	3n/4	4n/4
struct	2	74.4	75.0	64.4	52.0	61.4	53.2	60.8	50.6	51.6	**49.0**	50.6	50.6
	4	270.0	261.4	197.2	296.6	256.6	232.8	169.6	179.4	255.0	208.2	**150.0**	197.4
	6	380.8	343.2	332.6	421.2	372.2	305.4	322.0	334.8	370.4	**294.6**	314.2	307.2
	8	455.0	403.2	393.2	487.2	460.0	393.4	394.8	405.0	456.6	**356.0**	390.8	405.8
c2670	2	59.0	55.8	56.8	46.4	54.8	55.8	47.4	**45.0**	55.0	51.6	50.4	**45.0**
	4	123.8	123.4	101.0	134.2	119.0	118.0	96.4	**93.4**	116.6	109.0	95.2	96.8
	6	152.2	142.0	135.2	165.0	148.0	124.8	123.2	**112.2**	148.2	119.2	117.0	113.2
	8	177.2	174.8	154.2	186.2	176.4	141.6	147.4	146.2	174.2	**122.2**	129.0	131.0

though, e.g., the performance on `prim2` and `test06`. The variation of the circuits not only in density but in both structure and size seems to account for these anomalies. Through our experiments on randomly generated circuits that varied only in density, we have observed that most of these anomalies disappeared. Our algorithms' superior performance for sparse circuits is very promising as real applications are usually sparse. Also, as the circuit gets denser, even the performance of a simple greedy algorithm becomes comparable to that of KL [20], and so sparse circuits help us assess the performance of an algorithm better.

3) Results—Running Times: As for the running times from Table III, we can say that in general, the algorithms can be ordered according to their running times regardless of the number of parts as $T[\text{FMS}] < T[\text{PFM1}] < T[\text{PLM1}] < T[\text{PFM2}] < T[\text{PLM2}] < T[\text{PFM3}] < T[\text{PLM3}] < T[\text{SA}]$, where $T[A]$ represents the total running time of the algorithm A. Note that the running time of SA is far larger than those of the others, and FMS takes the smallest running time. We derived the following empirical inequality for the (total or per-pass) running time of our algorithms with respect to that of FMS

$$\frac{T[A]}{T[\text{FMS}]} < \frac{2N[A]}{n} \tag{4}$$

where A is any of the PFMi or PLMi algorithms. This inequality shows that the running times of our algorithms in practice are basically directly proportional to the number of cell moves and so they are as efficient as FMS.

C. Experiments on Algorithm Parameters and Discussion

Table IV presents the effects of the scale factor on the performance of PFM3. The results for PFM1 and PFM2 are similar. Table V presents the effects of N and N_{in} on the performance of the PLMi algorithms. The values in the tables are the average of the five best cutsizes in 30 runs. We chose two circuits, `struct` and `c2670`, with different densities. Note that the column for $N = n$ and $N_{\text{in}} = 4n/4$ in Table V corresponds to FMS.

We observe that as S increases, the performance of the PFMi algorithms generally gets better, the reason being that a very small S introduces too much randomness in the move selection process and renders the selection of best moves difficult. Through other experiments, we have also observed that a very large S does not help as it prevents the randomness altogether and prevents the occasional selection of uphill moves. We suggest that $R = 1$, i.e., $S = 2G_{\max} + 1$, is a safe choice but one should use a large S when the search space is difficult to explore as is the case when the circuit is sparse, k is large or $G_{\max}$ is small.

For the PLMi algorithms, we note the following. For multiway partitioning, the PLMi algorithms outperform FMS no matter what N and N_{in} are; however, the results get better as N increases. For bipartitioning, a large N favors a small N_{in}, and a small N favors a large N_{in}, e.g., when $N = n$ or $N = nk, N_{\text{in}} = n$ gives the best results and when $N = nk^2, N_{\text{in}} = 2n/4$ gives the best results. In our experiments mentioned in the previous section, we used $N_{\text{in}} = 2n/4$ as a compromise.

V. Conclusions and Future Work

In this paper, we propose two novel approaches for multiway circuit partitioning to overcome the limitations of the traditional locking mechanism, which has been used by all the previous KL-based algorithms. Each approach allows more moves per pass for each cell. Each approach leads to a generic algorithm whose parameters can be set in different ways such that better performance is usually obtained by spending more time in exploring the search space. We generated three versions of each generic algorithm and evaluated them on a subset of commonly used benchmark circuits in comparison with FMS and SA. The experimental results show that our algorithms outperform FMS significantly especially on multiway partitioning as well as partitioning of sparse circuits. The performance of our algorithms is comparable to that of SA, but the running time of SA is far larger than those of ours. We also did some experiments on the parameters of the generic algorithms and provided some guidelines for good parameter settings. Our approaches can easily be incorporated into existing KL-based algorithms such as those in [9], [13], [17], and [21].

We believe that our approaches are mature and effective enough to use, but there are some areas for further research such as better mobility functions (larger α or larger increments in move count to decrease unnecessary cell moves), design of adaptive schemes to reduce the number of moves per pass, use of phase concept in the PFMi algorithms, incorporation of our approaches with existing approaches, and finally application of our algorithms in other areas like VLSI placement.

References

[1] D. G. Schweikert and B. W. Kernighan, "A proper model for the partitioning of electrical circuits," in *Proc. 9th ACM/IEEE Design Automation Conf.*, 1972, pp. 57–62.

[2] A. E. Dunlop and B. W. Kernighan, "A procedure for placement of standard-cell VLSI circuits," *IEEE Trans. Computer-Aided Design*, vol. 4, no. 1, pp. 92–98, Jan. 1985.

[3] M. R. Garey and D. S. Johnson, *Computers and Intractability*. New York: Freeman, 1979.

[4] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*. Chichester, U.K.: Wiley, 1990.

[5] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell Syst. Tech. J.*, vol. 49, no. 2, pp. 291–307, Feb. 1970.

[6] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon, "Optimization by simulated annealing: An experimental evaluation; Part I, graph partitioning," *Oper. Res.*, vol. 37, no. 6, pp. 865–892, Nov. 1989.

[7] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Proc. 19th ACM/IEEE Design Automation Conf.*, 1982, pp. 175–181.

[8] B. Krishnamurthy, "An improved min-cut algorithm for partitioning VLSI networks," *IEEE Trans. Comput.*, vol. 33, no. 5, pp. 438–446, May 1984.

[9] L. A. Sanchis, "Multiple-way network partitioning," *IEEE Trans. Comput.*, vol. 38, no. 1, pp. 62–81, Jan. 1989.

[10] C. J. Alpert and A. B. Kahng, "Recent directions in netlist partitioning: A survey," *Integr. VLSI J.*, vol. 19, pp. 1–80, Dec. 1995.

[11] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, May 1983.

[12] A. G. Hoffmann, "The dynamic locking heuristic—a new graph partitioning algorithm," in *Proc. IEEE Int. Symp. Circuits Systems*, 1994, pp. 173–176.

[13] C. Park and Y. Park, "An efficient algorithm for VLSI network partitioning problem using a cost function with balancing factor," *IEEE Trans. Computer-Aided Design*, vol. 12, no. 11, pp. 1686–1694, Nov. 1993.

[14] L. W. Hagen, D. J.-H. Huang, and A. B. Kahng, "On implementation choices for iterative improvement partitioning algorithms," in *Proc. European Design Automation Conf.*, pp. 144–149, 1995.

[15] R. L. Graham, "Bounds for certain multiprocessing anomalies," *Bell Syst. Tech. J.*, vol. 45, pp. 1563–1581, 1966.

[16] A. Dasdan and C. Aykanat, "Two novel multiway circuit partitioning algorithms," [WWW], Tech. Rep. BU-CEIS-9609, Bilkent Univ., Ankara, Turkey, May 1996, Available http://www.cs.bilkent.edu.tr/

[17] H. Shin and C. Kim, "A simple yet effective technique for partitioning," *IEEE Trans. VLSI Syst.*, vol. 1, no. 3, pp. 380–386, Sept. 1993.

[18] C.-W. Yeh, C.-K. Cheng, and T.-T. Y. Lin, "Optimization by iterative improvement: An experimental evaluation on two-way partitioning," *IEEE Trans. Computer-Aided Design*, vol. 14, no. 2, pp. 145–153, Feb. 1995.

[19] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon, "Optimization by simulated annealing: An experimental evaluation; Part II, graph coloring and number partitioning," *Oper. Res.*, vol. 39, no. 3, pp. 378–406, May 1991.

[20] T. N. Bui, S. Chaudhuri, F. T. Leighton, and M. Sipser, "Graph bisection algorithms with good average case behavior." *Combinatorica*, vol. 7, no. 2, pp. 171–191, 1987.

[21] Y.-C. Wei and C.-K. Cheng, "Ratio cut partitioning for hierarchical designs," *IEEE Trans. Computer-Aided Design*, vol. 10, no. 7, pp. 911–921, July 1991.

Ali Dasdan received the B.S. degree in computer engineering from Bogazici University, Istanbul, Turkey, in 1991 and the M.S. degree in computer engineering and information science from Bilkent University, Ankara, Turkey, in 1993. He is currently working toward the Ph.D. degree in computer science at the University of Illinois, Urbana-Champaign. He is being supported by a fellowship from The Scientific and Technical Research Council of Turkey.

His current research interests include hardware-software codesign of embedded systems.

Cevdet Aykanat received the B.S. and M.S. degrees from the Middle East Technical University, Ankara, Turkey, in 1977 and 1980, respectively, and the Ph.D. degree from Ohio State University, Columbus, in 1988, all in electrical engineering. He was a Fulbright scholar during his Ph.D. studies.

He worked at the Intel Supercomputer Systems Division, Beaverton, OR, as a Research Associate. Since October 1988 he has been with the Department of Computer Engineering and Information Science, Bilkent University, Ankara, Turkey, where he is currently an Associate Professor. His research interests include parallel computer architectures, parallel algorithms, applied parallel computing, and graph/hypergraph partitioning.