Two Novel Multiway Circuit Partitioning Algorithms Using Relaxed Locking

Ali DAŞDAN*and Cevdet AYKANAT
Department of Computer Engineering and Information Science
Bilkent University
06533 Bilkent, Ankara, Turkey

Abstract

All the previous Kernighan-Lin based circuit partitioning algorithms employ the locking mechanism, which enforces each cell to be moved exactly once per pass. In this paper, we propose for multiway circuit partitioning two novel approaches to overcome this limitation. The proposed approaches are based on our claim that the performance of a partitioning algorithm gets better by allowing each cell to be moved more than once per pass. The first approach introduces the phase concept so that each pass can contain more than one phase, and each cell has a chance to be moved in all the phases. This approach uses the locking mechanism in a relaxed manner in that it does not allow a cell to be moved more than once in a phase. The second approach does not use the locking mechanism at all. It introduces the mobility concept, which allows a cell to be moved more than once per pass. Each approach leads to a Kernighan-Lin based generic algorithm whose parameters can be set in such a way that better performance is obtained by spending more time. By setting these parameters, we generated three versions of each generic algorithm. The proposed algorithms were experimentally evaluated in comparison with Sanchis' multiway circuit partitioning algorithm (FMS algorithm) and the simulated annealing algorithm (SA algorithm) on benchmark circuits. The proposed algorithms outperformed FMS algorithm significantly especially when the number of parts in the partition was large and the circuit was sparse. Their performance is comparable to that of SA algorithm though the running time SA algorithm is far larger than those of the proposed algorithms. We also performed some experiments on the parameters of the proposed algorithms, and provided guidelines for good parameter settings. Besides, we gave a new formulation of multiway circuit partitioning that combined those of the previous approaches, and presented detailed algorithms and their time and space complexity analysis. We observed that experimental results supported our claim. The proposed approaches are effective and efficient, and are applicable to almost all of the previous Kernighan-Lin based algorithms.

1 Introduction

Circuit partitioning deals with the task of dividing (partitioning) a given circuit into two or more blocks (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 [23], we modeled circuits with hypergraphs, and will use circuit and hypergraph terms interchangeably. Hypergraph partitioning has many important applications in VLSI layout, e.g., [1, 6]. The importance of hypergraph partitioning is mostly due to its connection to the divide-and-conquer paradigm. A partitioning algorithm divides a problem into semi-independent subproblems, and tries to reduce the interaction between these subproblems. This division of a problem into smaller subproblems results in a substantial reduction in the search space of the original problem [25]. Hypergraph partitioning problem is an NP-hard combinatorial optimization (minimization) problem [8, 17], 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. We now review the previous work and explain our motivation.

*Current Address: Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1304 W. Springfield, Urbana, IL 61801, USA

1.1 Previous Work

Kernighan-Lin [13] proposed a two-way graph partitioning algorithm which became the basis for most of the subsequent partitioning algorithms, all of which we call the KL-based algorithms. Kernighan-Lin's algorithm (KL algorithm) operates only on balanced partitions [10], and performs a number of passes over the cells of the circuit until it finds a locally minimum partition. Each pass consists of a repeated operation of pairwise cell swapping for all pairs of cells. Schweikert-Kernighan [23] adopted KL algorithm to hypergraph partitioning. Fiduccia-Mattheyses [7] introduced a faster implementation of KL algorithm. Fiduccia-Mattheyses' algorithm (FM algorithm) can operate on unbalanced partitions provided that the part sizes satisfy a particular balance criterion. They also proposed a single cell move instead of a swap of a cell pair at each step in a pass. These modifications as well as proper data structures, e.g., bucket lists, reduced the time complexity of a single pass of KL algorithm to linear in the size of the circuit (the number of pins). Since real circuits are usually very large, KL algorithm is not practical to use because of its high time complexity [24], and so the partitioning algorithms proposed after FM algorithm have utilized all the features of FM algorithm.

Krishnamurthy [16] added to FM algorithm a look-ahead ability, which helps to break ties better in selecting a cell to move. Sanchis [22] generalized Krishnamurthy's algorithm to a multiway circuit partitioning algorithm so that it could handle the partitioning of a circuit into more than two parts. It should be noted that all the previous approaches before Sanchis' algorithm (FMS algorithm) are originally bipartitioning algorithms. Sechen-Chen [24] proposed a new cost measure for the cost of the partition for FM algorithm. Both Wei-Cheng [27] and Park-Park [18] improved FM algorithm by incorporating the balance criterion in the cost measure, but in different ways. Shin-Kim [26] suggested the use of the gradual enforcement of balance criterion in FM algorithm during partitioning.

There are many other approaches to circuit partitioning that are not based on KL algorithm such as Simulated Evolution [21], Spectral Methods [5, 9], Mean Field Annealing [3, 4], and Simulated Annealing [10]. Simulated Annealing algorithm [14] (SA algorithm) is one of the most successful approaches to graph and circuit partitioning. Johnson *et al.* [10] performed an extensive experimental evaluation of SA algorithm on graph partitioning in comparison with KL algorithm and provided some guidelines to optimize the parameters of SA algorithm. We also adopted these guidelines in our implementation of SA algorithm, the performance of which was compared with those of the proposed algorithms.

1.2 Motivation

In this work, we basically propose two different approaches to multiway circuit partitioning. Each approach leads to a different algorithm. Our algorithms are also KL-based algorithms; however, we suggest novel changes in the basic characteristics of the KL-based algorithms such as relaxing the "conventional" locking mechanism. Therefore, we first shortly review how a typical KL-based algorithm employing cell moves works and then present our approaches.

A KL-based algorithm iterates a number of passes over the cells of the circuit until a locally minimum partition is found. All cells are *unlocked* at the start of each pass. At each step in a pass, the move with the maximum *gain* is tentatively performed, and the cell associated with the move is *locked*, where the gain of a move, also a gain of the cell associated with the move, is the decrease (or increase if no decrease is possible) that the move produces in the cutsize. The locking mechanism enforces each cell to be moved exactly *once* per pass. That is, a locked cell is not selected any more for a move until the end of the pass. At the end of the pass, we have a sequence of tentative cell moves and their respective gains. We then construct form this sequence the *maximum prefix subsequence* of moves with the *maximum prefix sum*. That is, the gains of the moves in the maximum prefix subsequence give the maximum decrease in the cutsize among all prefix subsequences of the moves tentatively performed. Then, we permanently realize the moves in the maximum prefix subsequence and start the next pass if the maximum prefix sum is positive. The partitioning process ends if the maximum prefix sum is not positive, i.e., no further decrease in the cutsize is possible, and we then have found a locally minimum partition.

In order to visualize how FMS algorithm works, it is better to look at the curves in Figure 1. These curves show the evolution (change) of the cutsize with the cell moves in FMS algorithm for 4-way partitioning of the circuit `s838` (see Table 1). Each interval between two successive vertical lines corresponds to a pass. The "current cutsize" curve corresponds to the tentative moves, whereas the "final cutsize" curve corresponds to the permanent moves, i.e., those moves in the maximum prefix subsequence. Note that the current cutsize and final cutsize curves coincide up to some point in each pass, and the final cutsize curve stays flat after this point until the end of the pass. The moves up to this point in each pass constitute the maximum prefix

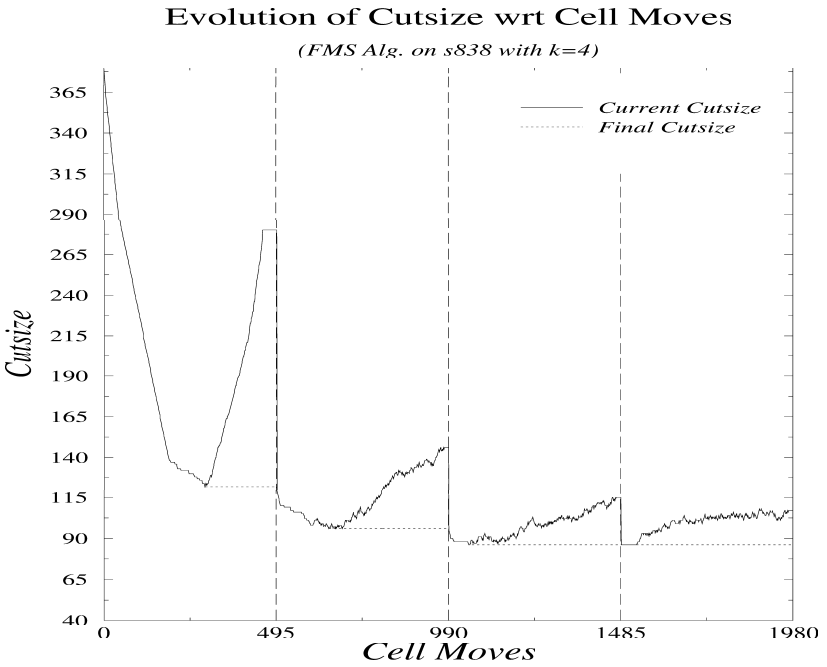

Figure 1: Evolution of cutsize with respect to cell moves for FMS Algorithm on 4-way partitioning of s838 with 495 cells. Each interval between two successive vertical lines corresponds to a pass. The current cutsize and the final cutsize curves correspond to the set of tentative and permanent cell moves, respectively. The initial cutsize is 379, and the final cutsize is 86.

subsequence of the moves in that pass. Also note that this point usually lies in the first half of each pass. That is, FMS algorithm "wastes" about half of the moves in each pass.

Our experiments reveal that a locked cell can later have a move with a gain larger than that of any move of the unlocked cells. Then, we should perform this move as it may decrease the cutsize more. However, such multiple moves for individual cells in a pass are not allowed in all the previous KL-based algorithms due to the locking mechanism. The aim in using the locking mechanism has been to prevent the cell-moving process from thrashing or going into an infinite loop [7, 16]. However, we now argue that we can achieve a smaller cutsize by making such multiple moves without violating the aim above. Thus, we make the following claim, on which all our work is based, and then propose two novel approaches exploiting this claim.

Claim: *Given a hypergraph with n cells, allowing each cell to be moved (possibly) more than once in a pass while preventing the occurrence of a large number of moves with no decrease in the cutsize improves the cutsize more than allowing each cell to be moved exactly once in a pass.*

Our first approach introduces the *phase* concept. A phase contains a sequence of tentative moves and locks. We propose that we can make more than one phase in a pass such that each phase contains n or less than n tentative cell moves. Note that an individual pass of the previous KL-based algorithms can be considered as consisting of a single phase with exactly n tentative cell moves. In the proposed approach, if a phase is not the last one in a pass, then all the cells that are tentatively moved (and so locked) during this phase are unlocked and are made ready to be moved again in the following phase in that pass. If this phase is the last one in a pass, then we perform the same operations as those performed at the end of a pass in a KL-based algorithm such as constructing the maximum prefix subsequence of moves and permanently realizing the cell moves in this subsequence. This approach establishes the basis of the proposed "multiway partitioning by locked moves" (PLM) algorithm.

Our second approach does not use the locking mechanism at all. In this approach, we associate a property, called the *mobility*, with each cell. The decision as to which move to perform is based on the mobility values of the cells but not on their gains. The mobility of a cell is proportional directly to its gain and inversely to the number of moves that the cell has performed. That is, we penalize each cell with the number of moves it has performed. The approach establishes the basis of the proposed "multiway partitioning by free moves" (PFM) algorithm.

In both of the proposed approaches, each cell can perform different number of moves in a pass, and each pass can contain more than n moves. Each cell has a better chance of being in the maximum prefix subsequence of the moves, thereby a chance of contributing more to the reduction in the cutsize. Each of

the proposed algorithms is a generic algorithm whose parameters, e.g., the number of moves in a pass, can be set in such a way that we usually trade better solution quality (smaller cutsize) against larger running time.

We did experiments on benchmark circuits for the proposed algorithms in comparison with FMS algorithm (and also FM algorithm) and SA algorithm. Experimental results show that the proposed algorithms outperform FMS algorithm significantly, and that their performance is closer to that of SA algorithm though SA algorithm yields the best performance. Besides, the running times of the proposed algorithms are far smaller than that of SA algorithm but larger than that of FMS algorithm as they perform more moves per pass. 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 2 gives the basic definitions related to hypergraphs and multiway hypergraph partitioning, and also introduces the notations. The cost, gain, and mobility concepts together with bipartitioning and multiway partitioning are also discussed in this section. The proposed algorithms are presented in Section 3. Besides, this section gives our data structures and initial partitioning algorithm, and a complexity analysis of the proposed algorithms. The experimental framework giving the details of the experiments on benchmark circuits, the size of the search space for each algorithm, and the experimental results are all presented in Section 4. This section also includes some experiments on the parameters of the proposed algorithms. We decided to give the proofs of the propositions and the tables in Appendix so as to increase the readability of the paper.

2 Basic Definitions and Notations

This section presents the basic definitions related to hypergraphs and multiway circuit partitioning, and gives the notations used throughout the paper. The definitions and notations combine those of [7, 16, 17, 18, 22]. Our formulation for a multiway circuit partitioning is more general as we allow non-uniformly weighted cells and nets.

2.1 Preliminaries

A hypergraph (*circuit*) $H = (V, E)$ consists of a finite non-empty set V of vertices (*cells*) and a finite non-empty set $E \subseteq 2^V$ of hyperedges (*nets*), where 2^V is the power set of the vertex set V. A hypergraph $H = (V, E)$ has $|V| = n$ cells and $|E| = m$ nets. Each net e_j in E, $1 \leq j \leq m$, is a subset of V. Each cell v_i, $1 \leq i \leq n$, has a positive integer weight w_i, and each net e_j has a positive integer weight c_j.

A net e_j is said to be *incident to* the cell v_i if $v_i \in e_j$. If a net e_j is incident to a cell v_i, then we say that e_j is on v_i, or v_i is on e_j. Cells on a net are called its terminals, and nets on a cell are called its *pins*. Cells that share terminals are called *neighbor cells*.

The *degree* d_i of a cell v_i is equal to the number of nets incident to v_i. The *degree* $|e_j|$ of a net e_j is equal to the number of its terminals. A net with a degree of 2 is called a *two-pin net*, and a net with a higher degree is a *multi-pin* net. The total number p of pins, or the total number of terminals, in H is defined as

$$p = \sum_{j=1}^{m} |e_j| = \sum_{i=1}^{n} d_i, \tag{1}$$

and is taken as the *size* of the circuit. Without loss of generality, we assume, as in [7], that each net has at least two pins, and each cell has a degree of at least one. Thus, $n = O(p)$ and $m = O(p)$. The *average cell degree* D_v of H is defined as $D_v = p/n$, and the *average net degree* of H is defined as $D_e = p/m$. The *density* D of circuit H with $n \geq 2$ is defined as

$$D = \frac{\sum_{j=1}^{m} |e_j|(|e_j| - 1)}{n(n-1)}, \tag{2}$$

which is similar to the definition in [18]. The density of a circuit determines how sparse the circuit is, and we say that the smaller the density of a circuit, the more sparse the circuit.

Given a circuit $H = (V, E)$, we say that $\Pi = (P_1, \cdots, P_k)$ is a *k-way partition* of H if the following three properties hold: each *part* P_l, $1 \leq l \leq k$, 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 $k = 2$.

Consider a k-way partition $\Pi = (P_1, \cdots, P_k)$ of a circuit $H = (V, E)$. The *size* $w(P_l)$ of a part P_l is equal to the sum of the weights of the cells in P_l. The *total cell weight* of all the cells in V and the *total net weight* of all the nets in E are denoted by $w(V)$ and $c(E)$, respectively. A net that has 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 set $E(l)$ of *external nets* of a part P_l is defined as $E(l) = \{e_j \in E \mid e_j \cap P_l \neq \emptyset \wedge e_j - P_l \neq \emptyset\}$, which consists of those cut nets that connect P_l. The set $I(l)$ of *internal nets* of a part P_l is defined as $I(l) = \{e_j \in E \mid e_j \cap P_l \neq \emptyset \wedge e_j - P_l = \emptyset\}$, which consists of those uncut nets that connect only P_l. We also define the number $\delta_j(l)$ as the number of terminals (pins) of the net e_j that lie in the part P_l, i.e., $\delta_j(l) = |\{v_i \in e_j \mid v_i \in P_l\}|$. This number reflects the distribution of the pins of the nets to the parts.

The cost $\chi(\Pi)$ of a partition Π, also called the *cutsize*, is equal to the sum of the weights of all cut nets. More formally,

$$\chi(\Pi) = c(E) - \sum_{l=1}^{k} \sum_{e_j \in I(l)} c_j = \sum_{j=1}^{m} c_j - \sum_{l=1}^{k} \sum_{e_j \in I(l)} c_j \tag{3}$$

where each cut net e_j contributes an amount of c_j to the cutsize regardless of the number of parts that e connects. This "cutsize definition" is exactly the same as the one in [22]; however, other cutsize definitions are also possible [22]. It should be noted that the proposed PLM and PFM algorithms are independent of the cutsize definition.

We now define the *multiway circuit partitioning problem* as follows. Given a circuit $H = (V, E)$, we regard as solutions those k-way partitions in which the size $w(P_l)$ of each part P_l satisfies the criterion $L(P_l) \leq w(P_l) \leq U(P_l)$. Here, $L(P_l)$ and $U(P_l)$ are positive integer lower and upper bounds on the part size of P_l, respectively. We are then asked to find the k-way partition (or partitions) that has the minimum cutsize over all the solutions. The multiway circuit partitioning problem is an NP-hard combinatorial minimization problem [8, 17].

The upper and lower bounds on part sizes constitute the *balance criterion*. We define these bounds as

$$L(P_l) = \lfloor \frac{w(V)}{k}(1 - \tau_l) \rfloor \text{ and } U(P_l) = \lceil \frac{w(V)}{k}(1 + \tau_l) \rceil \tag{4}$$

where τ_l is a parameter satisfying $0 < \tau_l < 1$. Each τ_l should be chosen in such a way that we have $L(P_l) > 0$ for each part P_l so as not to violate the definition of a k-way partition, and we have $(w(V)/k)\tau_l \geq max_{v_i \in V}(w_i)$ for each P_l so as to allow any cell to be moved from one part to another, i.e., we do not restrict the search space unnecessarily due to a tight balance condition; yet, we may not move any cell from one part to another during later stages of partitioning. In our implementation, we set $\tau_l = \tau$ for each P_l where the value of τ is given in Section 4.2.

We say that a partition is *feasible* if it satisfies a given balance criterion, and *infeasible* otherwise. Besides, a partition is said to be *balanced* if the part sizes are about the same size, and *unbalanced* otherwise [17]. To measure the balance, we say that the smaller the ratio of the maximum part size to the minimum part size in a partition, the more balanced the partition. A partition in which the part sizes are exactly the same is called a *perfectly balanced partition*, but such a partition is highly unlikely at least because $w(V)$ may not be a perfect multiple of k.

2.2 Cost, Gain and Mobility Concepts

The *cutstate of a net* indicates whether the net is cut or uncut. The *cutset* of a partition is the set of all nets that are cut. Notice that a net in the cutset must be in the set of external nets of at least two parts. A net is *critical* if it has a cell such that the cell would change the cutstate of the net if it is moved. Such a move either adds the net to the cutset or removes the net from the cutset. We now give the necessary and sufficient condition for a net to be critical in a k-way partition.

Proposition 2.1 *A net e_j is critical if and only if either there exists a part P_s such that $\delta_j(s) = |e_j|$, or there exist two different parts P_s and P_t such that $\delta_j(s) = 1$ and $\delta_j(t) = |e_j| - 1$.*

The cutstate of a net that is not critical cannot be affected by a cell move by the definition of a critical net, and so such a move cannot have any effect (a decrease or an increase) on the cutsize. A move of a cell can change the cutsize if the cell removes some nets from the cutset, or adds some nets to the cutset, that is, if it alters the cutstate of some nets, which should then be critical nets. Hence, we proved the following proposition.

Proposition 2.2 *The effect of a move of a cell on the cutsize depends only on the critical nets incident to that cell.*

We reflect the effect of a cell in the cutsize in terms of its gains, but the gains of a cell rely on its costs. Let P_s and P_t be two parts. The *cost* $C_i(s,t)$ of a cell v_i in P_s with respect to P_t is called its *external cost* if $s \neq t$ and is defined as

$$C_i(s,t) = \sum_{e_j \in E_i(s,t)} c_j \tag{5}$$

where the set $E_i(s,t) = \{e_j \in E(s) \mid v_i \in e_j \wedge \delta_j(t) = |e_j| - 1\}$ is the subset of external nets of P_s that would be deleted from the cutset if v_i is moved from P_s to P_t. The *cost* $C_i(s,t)$ of a cell v_i in P_s is called its *internal cost* if $s = t$, and is defined as

$$C_i(s,s) = \sum_{e_j \in I_i(s)} c_j \tag{6}$$

where the set $I_i(s) = \{e_j \in I(s) \mid v_i \in e_j\}$ is the subset of internal nets of P_s that would be added to cutset if v_i is moved from P_s to any other part. Note that $e_j \in I(s)$ implies $\delta_j(s) = |e_j|$. Since v_i can change the cutstate of nets in both $E_i(s,t)$ and $I_i(s)$, those nets are all critical nets. In a k-way partition, each cell has only one internal cost but $(k-1)$ external costs, each of which corresponds to a move direction,

The *move gain* (or gain) $G_i(s,t)$ of a cell v_i in P_s with respect to P_t, i.e., the gain of the move of v_i from P_s to P_t, is defined as

$$G_i(s,t) = C_i(s,t) - C_i(s,s) \tag{7}$$

where $s \neq t$. Note that each cell has $k-1$ move gains in a k-way partition. The maximum move gain is denoted by G_{max}, and is equal to the product of the maximum cell degree and the maximum net weight.

Proposition 2.3 *Consider the move of cell v_i from P_s to P_t. Let the cutsize before and after the move be denoted by $\chi(\Pi)$ and $\chi'(\Pi)$, respectively. Then,*

$$\chi'(\Pi) = \chi(\Pi) - G_i(s,t) \tag{8}$$

where $G_i(s,t)$ is the move gain of v_i before the move.

As this proposition shows, the gain of a cell determines the amount of benefit to be obtained by moving that cell. If the gain is positive, the cutsize decreases, but if the gain is negative, it increases.

We now define the proposed mobility concept. PFM algorithm operates on the mobility values of cells. The *mobility* $f_i(s,t)$ of a cell v_i in P_s with respect to P_t, i.e., the mobility of the move of v_i from P_s to P_t, is defined as

$$f_i(s,t) = \frac{1}{1 + n_i^{\alpha} exp(-G_i(s,t)/T)} \tag{9}$$

where n_i is the *move count* of v_i, and T and α are parameters as defined below. Each cell has $k-1$ mobility values each of which corresponds to a different move direction. The move count n_i of v_i is equal to the number of moves that v_i has performed. The mobility of a cell can be considered to be the probability that this cell can be selected for a move; hence, the larger the mobility of a cell, the better chance the cell has to be selected for a move. Moreover, this probability is reduced at each move of the cell because of the inverse proportionality of the mobility of a cell to its move count. This inverse proportionality restricts the number of those moves that the cell, e.g., a cell with a move gain of zero, may unnecessarily perform a large number of times without any significant decrease in the cutsize. The value of α used in our experiments is given in Section 4.2. The parameter T is used to expand the range of mobility values in the interval $(0,1)$. This parameter is computed as

$$\frac{1}{T} = (\frac{1}{G_{max}}) \ln (\frac{1-\epsilon}{\epsilon}) \tag{10}$$

in order to expand the mobility values to a predetermined interval $[\epsilon, 1-\epsilon]$ for $n_i = 1$ where ϵ is a very small positive constant. In this work, we use $\epsilon = 0.01$ for which we obtained $1/T \approx 4.6/G_{max}$. The move counts of cells are taken as 1 while computing the initial mobility values at the start of each pass, and then initialized to zero. Each move of a cell increments its move count by 1.

Each mobility value must be an integer because they are used to index the bucket arrays. Thus, we map a cell with mobility $f_i(s,t)$ to a bucket list with the index

$$F_i(s,t) = \lfloor Sf_i(s,t) \rfloor \tag{11}$$

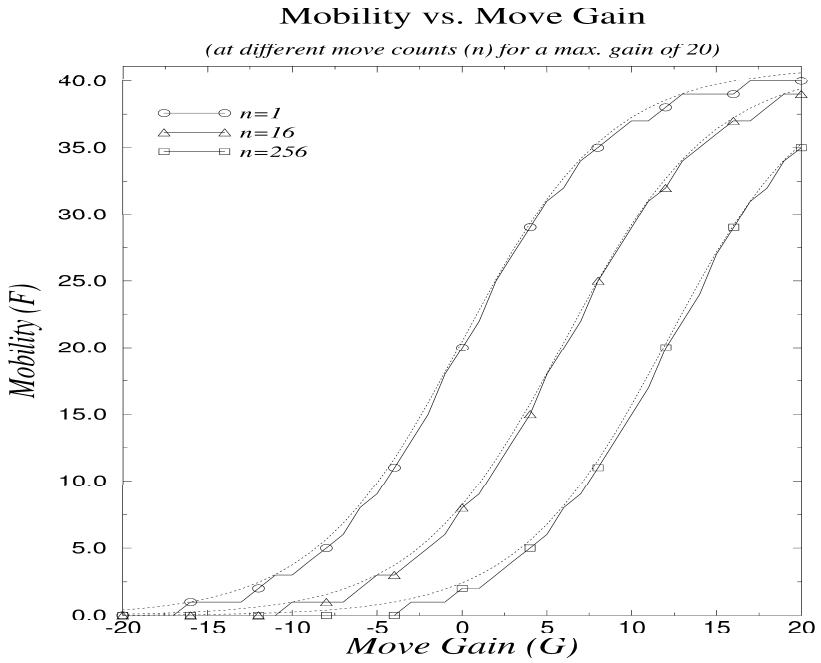

Figure 2: Evolution of mobility (F) with respect to move gain (G) at three different move counts (n). The maximum move gain is $G_{max} = 20$, and the scale factor is $S = 2 * 20 + 1 = 41$. Each dotted curve represents the value of F before flooring, for the nearest solid curve.

where S denotes the *scale factor* and $F_i(s,t)$ denotes the scaled mobility value of v_i for the respective move direction. Henceforth by the mobility of v_i we mean the function F. Note that $0 < f_i(s,t) < 1$ whereas $0 \leq F_i(s,t) < S$. The function f preserves the order of move gains of cells with equal move counts, but the function F does not. In PFM algorithm, two moves with the same F value may not have the same gain, but their gains are very similar if G_{max} is small or S is large. Hence, the function F introduces a slight randomization to the move selection process.

The change of the mobility with the move gain as well as the move count is illustrated in Figure 2. The maximum move gain G_{max} is 20, and the scale factor $S = 2G_{max} + 1 = 41$ where $2G_{max} + 1$ is the bucket size in FMS algorithm. Each pair of curves are for a different move count. The solid curves shows the mobility F, i.e., $\lfloor Sf \rfloor$. Each dotted curve shows the value of Sf, i.e., the F value before flooring, for the nearest solid curve. As seen from the curves, at small move counts, the mobility function does not distinguish very large move gains well, but, at large move counts, it does. Hence, moves selected at earlier stages of a pass do not necessarily correspond to the moves with the largest gain, but the ones selected at later stages usually do. This may prevent PFM algorithm from being stuck in a bad local minimum.

Like other KL-based algorithms, our partitioning algorithms also compute the initial gains of cells before starting a pass and update the gains of the affected cells after each cell move in that pass. We handle these computations and updates in terms of costs rather than in terms of gains because this approach results in simpler algorithms. Hence, we use the terms the "initial cost computation algorithm" and the "cost update algorithm". Since we introduce new concepts such as the mobility concept and do not use the locking mechanism in one of our approaches, we have to present these algorithms. These algorithms are generalized versions of those in [7] in two ways: these algorithms are for multiway partitioning, and these algorithms do not use the locking mechanism. But, we also identify the changes to adapt these algorithms to use the locking mechanism. These algorithms were also used in our implementation of SA algorithm.

The initial cost computation algorithm given in Figure 3 computes the initial costs of each cell in the circuit, assuming an initial feasible partition. The following proposition establishes the correctness of the algorithm, and gives its time complexity.

Proposition 2.4 *The initial cost computation algorithm given in Figure 3 computes costs of each cell as defined in Equations 5 and 6. Furthermore, it has a time complexity of $O(pk)$.*

After the move, the costs of the cell moved, and those of its neighbors should be updated so that they indicate the effect of the move correctly. Besides, the cost updates should be done very carefully so as not to increase the time complexity of the partitioning algorithm. These cost updates are carried out by the cost update algorithm in Figure 4. The following proposition establishes the correctness of the algorithm, and gives its time complexity.

Algorithm: Initial Cost Computation.
Input: A hypergraph $H = (V, E)$ with $|V| = n$, a k-way partition $\Pi = (P_1, \cdots, P_k)$ of H.
Output: Computes all costs of cells in V.

```
1  for each cell v_i ∈ V where let v_i ∈ P_s do
2     for each part P_t in Π do
3        Set C_i(s,t) ← 0 /* Initialize the cost */
4     for each net e_j incident to v_i do
5        if δ_j(s) = 1 then
6           Search for a part P_t such that δ_j(t) = |e_j| − 1
7           if such P_t has been found then
8              Set C_i(s,t) ← C_i(s,t) + c_j
9        else if δ_j(s) = |e_j| then
10          Set C_i(s,s) ← C_i(s,s) + c_j
```

Figure 3: The initial cost computation algorithm.

Proposition 2.5 *The cost update algorithm given in Figure 4 updates the costs of the cell moved and those of its neighbors as defined in Equations 5 and 6. Furthermore, if the locking mechanism is used, then this algorithm takes $O(pkG_{max})$ time for n cell moves. If not, then it takes $O(kG_{max}D_{v,max}D_{e,max})$ time per move where $D_{v,max}$ is the maximum cell degree, and $D_{e,max}$ is the maximum net degree.*

In lines 1, 6, 10, 16, and 20 in the cost update algorithm, the gain (the mobility) of a cell is recomputed as in Equation 7 (Equation 11). The initial cost computation algorithm can be used in a partitioning algorithm that employs the locking mechanism without any change in the algorithm; however, to use the cost update algorithms in such a partitioning algorithm, do the following change in the cost update algorithm: *do an operation for a cell if this cell is unlocked.*

2.3 Bipartitioning versus Multiway partitioning

A bipartitioning algorithm can be adapted to a multiway partitioning algorithm in two ways [13]. One involves a hierarchical use of a bipartitioning algorithm. The other involves successively choosing pairs of parts in the partition and applying a bipartitioning algorithm to these pairs. Both of these ways have serious drawbacks, which are discussed in detail in [13, 22].

An alternative method, as proposed in [22], is creating a multiway partitioning directly, i.e., without using a bipartitioning algorithm. We call this method the *direct multiway partitioning.* Sanchis [22] calls this method the uniform multiway partitioning. In this method, we consider at each step in a pass all possible moves of each cell from its source part to *any* of the other parts (the target parts) in the partition and choose the best such move, i.e., the one with the maximum gain. The moves that do not violate the balance criterion are called *legal* moves, otherwise *illegal* [22]. In this method, only legal moves are considered for a move.

Direct multiway partitioning algorithms are capable of handling partitions involving an arbitrary number of parts, and they do not suffer from the problems that the algorithms based on a bipartitioning algorithm do [22]. In k-way partitioning, each cell has $(k-1)$ possible move directions at each step in a pass where each move direction corresponds to a move from its source part to each of the other parts in the partition. There are $k(k-1)$ possible move directions in total as only the best move is considered in each move direction. A search for the best move should consider all of these move directions. Our algorithms are also direct multiway partitioning algorithms.

3 Proposed Algorithms

This section explains the proposed algorithms in some detail and analyzes their time and space complexity. It also presents data structures and the initial partitioning algorithm.

Algorithm: Cost Update.
Input: A hypergraph $H = (V, E)$ with $|V| = n$, a k-way partition $\Pi = (P_1, \cdots, P_k)$ of H, the move of vertex v_i from P_s to P_t.
Output: Updates all costs of v_i and those of its neighbors.

```
1  Recompute G_i(t,l) (and F_i(t,l)) for each l ≠ t
2  for each net e_j incident to v_i do
3     if δ_j(s) = |e_j| then
4        for each cell v_r ∈ e_j where r ≠ i do
5           Set C_r(s,s) ← C_r(s,s) − c_j
6           Recompute G_r(s,l) (and F_r(s,l)) for each l ≠ s
7     else if δ_j(s) = |e_j| − 1 then
8        Find cell v_r ∈ e_j such that v_r ∈ P_l and l ≠ s
9        Set C_r(l,s) ← C_r(l,s) − c_j
10       Recompute only G_r(l,s) (and F_r(l,s))
11    Set δ_j(s) ← δ_j(s) − 1
12    Set δ_j(t) ← δ_j(t) + 1
13    if δ_j(t) = |e_j| then
14       for each cell v_r ∈ e_j where r ≠ i do
15          Set C_r(t,t) ← C_r(t,t) + c_j
16          Recompute G_r(t,l) (and F_r(t,l)) for each l ≠ t
17    else if δ_j(t) = |e_j| − 1 then
18       Find cell v_r ∈ e_j such that v_r ∈ P_l and l ≠ t
19       Set C_r(l,t) ← C_r(l,t) + c_j
20       Recompute only G_r(l,t) (and F_r(l,t))
```

Figure 4: The cost update algorithm.

3.1 Multiway Partitioning by Locked Moves (PLM)

The proposed PLM algorithm is a direct multiway circuit partitioning algorithm. It employs the phase concept, as discussed in Section 1.2. A cell is locked as soon as it is moved in a phase, and so it is not reselected until the end of that phase.

The generic PLM algorithm is given in Figure 5. The steps of the algorithm are explained in detail when its time complexity is derived in Appendix. PLM algorithm uses two parameters: N_{out} and N_{in}. The parameter N_{out} is the number phases in a pass, and the parameter N_{in} is the number of cell moves in a phase assuming that we perform the same number of moves in each phase. We also denote by N the total number of cell moves in a pass. Thus, we perform $N = N_{out}N_{in}$ cell moves in a pass of PLM algorithm. Notice that PLM algorithm reduces to FMS algorithm for $N_{out} = 1$ and $N_{in} = n$. Based on our main claim, we set $N \geq n$ and $N_{in} \leq n$. We can generate different versions of PLM algorithm by setting these parameters to different values. We report experiments on these parameters in Section 4.4.

3.2 Multiway Partitioning by Free Moves (PFM)

The proposed PFM algorithm is a direct multiway circuit partitioning algorithm. This algorithm does not use the conventional locking mechanism at all. Each cell can make different number of moves as in PLM algorithm. The decision as to which move to select is based on the mobility values of the cells. The mobility of a cell determines its move capability.

The generic PFM algorithm is given in Figure 6. The steps of the algorithm are explained in detail when its time complexity is derived in Appendix. In PFM algorithm, the moves are inserted into the bucket lists on the basis of their mobility values but not on the basis of their gains. PFM algorithm uses the following parameters: N, $f(\cdot)$, S, and ϵ. The parameter N determines the total number of cell moves in a pass. Based on our main claim, we set $N \geq n$. The other parameters are explained in Section 2.2. We report experiments on these parameters in Section 4.4. We can establish an analogy between PFM algorithm and SA algorithm. Both algorithms allow multiple moves for individual cells. The expression for the mobility of a cell is similar to that for the probability of accepting an uphill move in SA algorithm. SA algorithm always accepts a downhill move, but accepts an uphill move with a probability that decreases during partitioning. SA algorithm selects a move randomly, but PFM algorithm selects the move with the maximum mobility.

Algorithm: Multiway Partitioning by Locked Moves.
Input: An initial k-way partition of a hypergraph $H = (V, E)$ with $|V| = n$, an initial cutsize of that partition.
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 initial costs of cells, and initialize cells as unlocked
5        Insert cells into bucket lists on the basis of their move gains
6        repeat
7           Select a legal move (and so a cell) with the maximum move gain
8           Delete that cell from bucket lists, and lock it
9           Tentatively make that move
10          Update costs and gains of all affected cells
11       until N_in times or there is no legal move any more
12       if N_in < n then
13          Free bucket list nodes for remaining unlocked cells
14    Find the maximum prefix sum G_T, and determine the maximum
      prefix subsequence of the tentative moves
15    if G_T > 0 then /* if there is a decrease */
16       Permanently make the moves in the maximum prefix subsequence
17       Decrease the cutsize by G_T
18 until G_T ≤ 0
```

Figure 5: The generic direct multiway partitioning by locked moves (PLM) algorithm.

3.3 Data Structures and Initial Partitioning

We use adjacency lists to store a circuit. We also employ the bucket data structure introduced in [7]. Since our algorithms are also direct multiway partitioning algorithms like FMS algorithm, we adapted the bucket data structure proposed in [22] for a direct multiway partitioning. In this data structure, there is one bucket array for each move direction, and one bucket list containing those cells with the same move gain (or the same mobility) is connected to each slot of a bucket array. Our data structure differs from those in [22] in the following ways. We did not use the level gain concept, and so our data structure is simpler. For FMS algorithm and PLM algorithm, each bucket array has a size of $2G_{max} + 1$; yet, for PFM algorithm, each bucket array has a size of S. Recall from [22] that finding a bucket list for a cell and inserting a cell to a bucket list take constant time, and deleting a cell from a bucket list takes $O(G_{max})$ time. Hence, a cell can be inserted into $(k-1)$ bucket lists in $O(k)$ time in both PLM and PFM algorithms. However a cell can be deleted from $(k-1)$ bucket lists in $O(kG_{max})$ and $O(kS)$ times in PLM and PFM algorithms, respectively. Details of this data structure are given in [22].

Like FMS algorithm, our algorithms need an initial k-way partition as input. We generate an initial k-way partition as follows. The algorithm handles each cell only once, and randomly assigns each cell to one of the parts with the minimum size. For a given balance criterion, if the partition obtained is infeasible, we increase the parameter τ in Equation 4 by a small value such as 0.05 until either we obtain a feasible partition, or τ becomes equal to or larger than 1.0. In the latter case, the algorithm outputs an error message and we do not start the partitioning process. This algorithm runs in $O(pk)$ time. If still a more balanced partition, i.e., the one with a tighter balance criterion, is required, then we have a number partitioning problem [11], and we can use the differencing method proposed in [12], which runs in $O(n \lg n)$ time for 2-way partitioning. If the number of parts is not fixed and we want to minimize it, we can use the FFD algorithm [8] as suggested in [20].

3.4 Complexity Analysis

The following propositions establish the time and space complexity of FMS, PLM, and the PFM algorithms. Each time complexity corresponds to the worst-case time complexity per pass. The time complexity of each algorithm below can be reduced by using a binary heap to speed up the move selection operation [22]. For example, the time complexity of FMS algorithm can be reduced to $O(pk(\lg k + G_{max}))$ [22] by using a

Algorithm: Multiway Partitioning by Free Moves.
Input: An initial k-way partition of a hypergraph $H = (V, E)$ with $|V| = n$, an initial cutsize of that partition.
Output: A locally minimum k-way partition of H.

```
1  Initialize bucket list pointers
2  repeat /* for each pass */
3     Compute initial costs of cells, and initialize their move counts
4     Insert cells into bucket lists on the basis of their mobility values
5     repeat
6        Select a legal move (and so a cell) with the maximum mobility
7        Tentatively make that move
8        Increment move count of the cell
9        Update costs, gains, and mobility values of all affected cells
10    until N times or there is no legal move any more
11    Find the maximum prefix sum G_T, and determine the maximum
      prefix subsequence of the tentative moves
12    if G_T > 0 then /* if there is a decrease */
13       Permanently make the moves in the maximum prefix subsequence
14       Decrease the cutsize by G_T
15    Free all bucket list nodes
16 until G_T ≤ 0
```

Figure 6: The generic direct multiway partitioning by free moves (PFM) algorithm.

binary heap.

Proposition 3.1 ([22]) *The time complexity of FMS algorithm for a k-way partitioning of a circuit with p pins is $O(pk(k + G_{max}))$, and its space complexity is $O(pk + k^2 G_{max})$.*

Proposition 3.2 *The time complexity of PLM algorithm for a k-way partitioning of a circuit with p pins is $O(N_{out} pk(k+G_{max}))$ where N_{out} is the number of phases per pass. Its space complexity is $O(pk+k^2 G_{max} + pN_{out})$.*

We usually set $N = nk^\beta$ and $N_{in} = \Theta(n)$ where $0 \leq \beta \leq 2$. Then, the worst-case time complexity of PLM algorithm per pass becomes $O(pk^{\beta+1}(k+G_{max}))$, and its space complexity becomes $O(pk^\beta + k^2 G_{max})$.

Proposition 3.3 *The time complexity of PFM algorithm for a k-way partitioning of a circuit with p pins is $O(k(p + kS + N(k + D_{v,max} D_{e,max} S)))$ where $D_{v,max}$ and $D_{e,max}$ are the maximum cell and net degrees, respectively, and N is the number of moves per pass. Its space complexity is $O(pk + k^2 S + N)$.*

We usually set $N = nk^\beta$ where $0 \leq \beta \leq 2$. Then, the worst-case time complexity of PFM algorithm per pass becomes $O(k^2 S + pk(k + D_{v,max} D_{e,max} S))$ for $\beta = 0$ and $O(pk^{\beta+1}(k + D_{v,max} D_{e,max} S))$ for $\beta > 0$. Its space complexity becomes $O(pk + k^2 S)$ for $\beta = 0$ and $O(pk^\beta + k^2 S)$ for $\beta > 0$.

4 Experimental 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 algorithms in comparison with FMS algorithm (and so FM algorithm) and SA algorithm.

4.1 Size of Search Space

After FM algorithm, all the partitioning algorithms have used the move-neighborhood structure. In a move-neighborhood structure, we proceed from one partition to another by means of a single cell move. Our algorithms as well as FMS algorithm use the move-neighborhood structure. Let $\mathcal{N}[A]$ denote the number of different solutions (partitions) explored per pass by a KL-based algorithm A. A move-neighborhood structure for k-way partitioning of a n-cell circuit contains at most $n(k-1)$ partitions at each move in a

pass, as each of n cells has $(k-1)$ possible moves at each move in a pass. Note that, for an algorithm A using the locking mechanism, only unlocked cells should be considered when computing $\mathcal{N}[A]$. Then, we have $\mathcal{N}[FMS] \leq (k-1)n(n+1)/2$, $\mathcal{N}[PLM] \leq N(k-1)(2n-N_{in}+1)/2$, and $\mathcal{N}[PFM] \leq Nn(k-1)$ where N denotes the number of cell moves in a pass, and $N = n$ for FMS algorithm. If there are exactly $k(k-1)$ legal move directions at each step in a pass, then these upper bounds become tight. Intuitively, the larger the value of $\mathcal{N}$ for an algorithm, the larger the number of different partitions explored by that algorithm, and so 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 provides support for this intuitive view. Almost all of the solutions explored by FMS algorithm per pass are different; however, some of the solutions explored by PLM and PFM algorithms per pass may be the same since they allow multiple moves for a cell. If we assume that all the solutions explored are different, PFM algorithm explores more solutions per pass than PLM algorithm does per pass even when the number, N, of moves per pass is the same for both. For example, set $N = nk^{\beta}$ and $\beta \geq 0$ for PFM algorithm, and set $N = nk^{\beta}$, $\beta \geq 0$, and $N_{in} = n/2$ for PLM algorithm. Then, $\mathcal{N}[PFM]$ is larger than $\mathcal{N}[FMS]$ by a factor of $2k^{\beta}$, and $\mathcal{N}[PLM]$ is larger than $\mathcal{N}[FMS]$ by a factor of $3k^{\beta}/2$. Moreover, both of our algorithms explore more solutions per pass than the FMS algorithm does per pass even when all perform the same number of moves in a pass. For example, for $\beta = 0$, i.e., when all the algorithms perform the same number of moves in a pass, $\mathcal{N}[PFM]$ is larger than $\mathcal{N}[FMS]$ by a factor of 2, and $\mathcal{N}[PLM]$ is larger than $\mathcal{N}[FMS]$ by a factor of $3/2$.

When computing the total number of solutions explored by an algorithm, we should take into account not only the number of different solutions explored per pass but also the number of passes that the algorithm executes as well as the number of feasible solutions since the number of passes that a KL-based algorithm executes is usually very small but not known in advance, and such an algorithm sometimes cannot explore a solution depending on the tightness of the balance criterion. In general, our algorithms explore more solutions than FMS algorithm especially for multiway partitioning even when the number of passes are considered. Thus, we can say that our algorithms explore the search space more effectively than FMS algorithm.

4.2 Experimental Framework

By setting the parameters of PLM and PFM algorithms to different values, we generated three versions of each of these algorithms. Henceforth these versions of PLM algorithm (PFM algorithm) will be referred to as PLMi algorithms (PFMi algorithms) for $i = 1, 2, 3$. The values of the parameters and the names of these versions are presented in the following table, where

$$R = S/(2G_{max} + 1) \tag{12}$$

is the ratio of the bucket size in a version of PFM algorithm to that of FMS algorithm.

Versions of PLM and PFM Alg.'s					
N	N_{in}	*Name*	N	R	*Name*
n	$n/2$	PLM1	n	2	PFM1
nk	$n/2$	PLM2	nk	8	PFM2
nk^2	$n/2$	PLM3	nk^2	128	PFM3

Thus, each cell makes N/n moves on the average in a pass of each algorithm. Let $N[A]$ denote the number of cell moves in a pass of a KL-based algorithm A. Then, for this setting, we have $N[FMS] = N[PLM1] = N[PFM1]$, $N[PLM2] = N[PFM2]$, and $N[PLM3] = N[PFM3]$. We say that a PFMi algorithm *corresponds* to a PLMj algorithm if $i = j$, e.g., PFM2 algorithm corresponds to PLM2 algorithm.

The level parameter of FMS algorithm in our implementation was set to 1 in order to allow a fair comparison between FMS algorithm and our algorithms since the level gain concept [22] is also applicable to our algorithms, and we did not incorporate it in our algorithms. Moreover, our aim was not to present an extensive experimental evaluation of the previous multiway partitioning algorithms in comparison with our algorithms but to present experimental results enough to show the potential of our algorithms. Since FMS algorithm with the level parameter set to 1 exactly corresponds to a direct multiway implementation of FM algorithm, we will refer to it as FM algorithm for bipartitioning and as FMS algorithm for multiway partitioning but FMS algorithm also subsumes FM algorithm.

All the algorithms were coded in the C programming language. All the experiments were carried out on a SUN SPARC station (SPARC 10) under SunOS operating system. (SUN SPARC station and SunOS operating system are trademarks of Sun Microsystems, Inc.) We used 12 benchmark circuits as our

Table 1: Properties of benchmark circuits. Here, n is the number of cells, m is the number of nets, p is the number of pins, D_v is the average cell degree, D_e is the average net degree, $D_{v,max}$ is the maximum cell degree, $D_{e,max}$ is the maximum net degree, and D is the density of the circuit.

PROBLEM									
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
c1355	c1355	650	618	1745	2.68	2.82	5	11	0.010065
c3540	c3540	1038	1016	3131	3.02	3.08	5	23	0.011204
c2670	c2670	924	860	2375	2.57	2.76	5	30	0.011444
primary1	prim1	833	902	2908	3.49	3.22	9	18	0.018056
s838	s838	495	460	1261	2.55	2.74	5	33	0.019801
industry1	ind1	2271	2186	7731	3.40	3.54	9	318	0.038276
test03	test03	1607	1618	5807	3.61	3.59	54	225	0.050714
playout.balu	balu	701	702	2493	3.56	3.55	9	117	0.052924
test06	test06	1752	1641	6638	3.79	4.05	6	388	0.079830

test instances from `LayoutSynth92` standard cell test suite and `Partitioning93` test suite in `ACM/SIGDA Design Automation Benchmarks` (also known as `MCNC Benchmarks`). The properties of these circuits are summarized in Table 1. The circuits in all the tables in Section 4 are ordered in ascending density. We deleted certain nonessential features of these circuits as in [7, 23]: all the nets with only one terminal were removed, and each net containing a cell more than once as a terminal 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 1; however, it should be noted that our formulation and also our implementation allow non-uniformly weighted cells and nets without any change.

When constructing our test instances, we took the following points into account: we tried to choose those circuits that were used in previous works on circuit partitioning, e.g., `prim1` and `prim2` from [27], to choose circuits having a variety of properties, e.g., `struct` has a regular structure, `s838` is a small circuit, and `prim2` is a large circuit, to choose circuits that were intended specifically for performance evaluation of partitioning algorithms, e.g., those circuits from `Partitioning93` test suite such as `c1355`, to choose some circuits that were not used in previous works, e.g., `balu`, to choose circuits that were not so large that they take a lot of time for such time-consuming algorithms as SA algorithm. Note that the time complexity of each of our algorithms is linear in the size of the circuit so that large circuits do not pose any problem for our algorithms to partition.

We used the initial partitioning algorithm as explained in Section 3.3. We set the parameter τ to 0.10 in the balance criterion, i.e., we allow each part size to be 10% more or 10% less than its value in a perfectly balanced partition. This balance criterion always led to feasible initial partitions. After a number of experiments, we adopted the following to improve the performance of our algorithms. We slightly modified PFM algorithm in that a cell was not selected in two successive moves. All PFMi algorithms used the mobility function in Equation 11. We set the parameter α in Equation 9 to 1/2. We used a table lookup technique to speed up the calculation of the exponential function values in Equation 9. The n_i^α values in Equation 9 can also be obtained with a table lookup technique but we did not try it. Henceforth, the performance of an algorithm refers to the quality of the cutsize it produces.

We set the number k of parts to 2, 4, 6, and 8 as in similar works. Following [26], we ran FMS algorithm 500 times, each of our algorithms 30 times, and SA algorithm 10 times on each test instance starting from different initial partitions. The running times were measured in seconds. The running time of an algorithm included 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 algorithm according to the cooling schedule in [10]. This cooling schedule was also used in a work [26] similar to ours. Since Johnson *et al.* [10] gave an implementation of SA algorithm for graph bipartitioning, we incorporated the guidelines supplied in [10, 11, 14] to adapt SA algorithm for multiway circuit partitioning. We made the following three changes in the cooling schedule in [10]. The starting temperature was set to 10 as in [14] where the acceptance rate was larger than 90%, whereas Johnson *et al.* [10] suggested a starting temperature where the acceptance rate was 40% for a speedup. The termination condition was met when either the acceptance rate was less than 2% as in [10], or the same cutsize was encountered $n/2$ times, which is necessary because the presence of a large number of moves with

zero gain usually prevents the acceptance rate to be less than 2%. The final change was in the form of the cost function. Johnson *et al.* [10] 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 algorithm. However, we should mention that the penalty function approach improves the performance of SA algorithm [10].

4.3 Results with Default Settings and Discussion

Table 2 presents the average cutsizes found by each algorithm. The values in parentheses give the ratio of the average cutsize found by the respective algorithm to that by FMS algorithm. To find the percent improvement done by an algorithm in the average cutsize with respect to FMS algorithm, we should subtract the values in parentheses from 1.00. For example, for 4-way partitioning of `struct`, PFM3 algorithm produced an average cutsize that is 63%, i.e., $1.00 - 0.37 = 0.63$, smaller (better) than the one by FMS algorithm. For 2-way partitioning of `struct`, PFM3 algorithm produced an average cutsize that is -4%, i.e., $1.00 - 1.04 = -0.04$, better than one by the FM algorithm; yet, a negative improvement means that PFM3 algorithm was beaten by FM algorithm. Note that the smaller the value in parenthesis, the better the respective algorithm with respect to FMS algorithm. Table 3 gives the minimum cutsizes found by each algorithm. The values in parentheses give the ratio of the minimum cutsizes found by the respective algorithm to those by FMS algorithm. The interpretation of the values in parentheses is similar to the one explained above for the average cutsizes. Table 4 presents the average running time of each algorithm. In all the tables, the bold values in a row correspond to the best value for that row. Recall that the best cutsize is the smallest cutsize, and the best running time is also the smallest running time.

Table 5 summarizes the average results for each algorithm in terms of seven comparison measures labelled A through G. In this table, the averages for each algorithm were computed by considering the performance of the algorithm on all the circuits. This table also includes the averages for the results in Table 2, Table 3, and Table 4. This table helps us to estimate the relative performance of the algorithms on the whole test suite.

Let $\chi[A]$ and $T[A]$ denote the cutsize (average or minimum cutsize) found by an algorithm A and the running time of an algorithm A, respectively. In general, the performance of each algorithm differ depending on the value of the number, k, of parts, and thus, we usually examine the performance of algorithms for bipartitioning, i.e., $k = 2$, and for multiway partitioning, i.e., $k > 2$, separately.

From Table 5(A), we observe the following. For bipartitioning, the algorithms can be ordered with respect to the quality of the average cutsizes they produce as follows: $\chi[PLM3] < \chi[FM] < \chi[SA] < \chi[PLM2] < \chi[PLM1] < \chi[PFM3] < \chi[PFM2] < \chi[PFM1]$. Thus, PLM3 algorithm outperforms all the others. Also note that FM algorithm is the second in the performance rank. Although FM algorithm performs better than many of the algorithms in terms of the average quality of the average cutsizes, FM algorithm does not display the same relative performance when the circuits are considered separately. For example, PLM3 and SA algorithms yield better performance than FM algorithm on the bipartitioning of 10 and 9 circuits, respectively, out of 12 circuits in Table 2. From Table 5(A), the algorithms for multiway partitioning can be ordered as follows: $\chi[SA] < \chi[PFM3] < \chi[PFM2] < \chi[PLM3] < \chi[PLM2] < \chi[PFM1] < \chi[PLM1] < \chi[FMS]$. Thus, SA algorithm outperforms all the others, and each of the proposed algorithms produces better average cutsizes than FMS algorithm does. The improvement achieved by PFM3 algorithm with respect to FMS algorithm is as large as 66% (on 4-way partitioning of `ind1` in Table 2).

From Table 5(B), we see that FM algorithm outperforms all the other algorithms in terms of the average quality of the minimum cutsizes for bipartitioning. Besides, FM algorithm produces the smallest cutsizes in all the circuits except `struct` and `prim1` as seen in Table 3. For multiway partitioning, PFM3 algorithm delivers the smallest cutsizes. SA algorithm is the second in rank, and each of the proposed algorithms outperforms FMS algorithm. The improvement achieved by PFM3 algorithm with respect to FMS algorithm is as large as 73% (on 4-way partitioning of `ind1` in Table 3).

From Table 5(F), we can say that, in general, the algorithms can be ordered according to their running times regardless of the number of parts as follows: $T[SA] > T[PLM3] > T[PFM3] > T[PLM2] > T[PFM2] > T[PLM1] > T[PFM1] > T[FMS]$. The running time of SA algorithm is far larger than those of the others. FMS algorithm takes the smallest running time. For any of PFMi and PLMi algorithms, we derived the following empirical inequality for the ratio of their running times (total as in Table 5(F) or per pass as in Table 5(E)) to that of FMS algorithm

$$\frac{T[A]}{T[FMS]} < \frac{2N[A]}{n} \tag{13}$$

where A is any of PFMi and PLMi algorithms, and $N[A]$ is the number of cell moves per pass of the algorithm A as mentioned earlier. Recall that $N[FMS] = n$. This inequality shows that the number of cell moves per pass is the dominant factor in determining the running times of the algorithms.

From Table 5(G), we observe that, for bipartitioning, FM algorithm executes the largest number of passes; however, for multiway partitioning, PFM1 algorithm executes the largest number of passes, and PFM3 algorithm executes the smallest number of passes. The number of passes that an algorithm executes decreases as the number of cell moves performed per pass of this algorithm increases. Also note that the average of the average number of passes that any algorithm executes is less than 10.0. Hence, the number of passes for FMS algorithm as well as our algorithms can be considered to be a constant.

We will call an algorithm *stable* if the ratio, the *stability ratio*, of the standard deviation of the average cutsize to the average cutsize that the algorithm produces is small, and *unstable* otherwise. From Table 5(D), we note that, for bipartitioning, FM algorithm is the least stable, and PLM3 algorithm is the most stable. For multiway partitioning, SA and FMS algorithms are the most stable, and our algorithms are less stable than FMS algorithm. Moreover, a PLMi algorithm is more stable than a corresponding PFMi algorithm but margins are very small. Even though FMS algorithm seems to be a stable algorithm, any of our algorithms (except PFM1 algorithm for bipartitioning) and SA algorithm improve upon FMS algorithm in terms of the average of the maximum cutsizes (Table 5(C)). That is, FMS algorithm produces the largest cutsizes.

From the experimental results explained above, we deduce the following. The performance of both PFMi and PLMi algorithms gets better as the number of cell moves increases per pass. These provide support for our claim that allowing each cell to move more than once improves the performance. Since the performance of our algorithms is significantly better than that of FMS algorithm for multiway partitioning, we can say that the phase and the mobility concepts seem to be very beneficial. Moreover, since the performance of each PFMi algorithm is better than the corresponding PLMi algorithm as well as the performance of PFM2 algorithm is better than any PLMi algorithm for multiway partitioning, we can say that the locking mechanism prevents an algorithm to explore the search space of the problem more effectively. It should also be noticed that SA algorithm like PFMi algorithms does not restrict the move capability of cells, and this feature of SA algorithm may count for its superior performance.

We can say that our algorithms outperform FMS algorithm for multiway partitioning as shown in the tables. We can also argue that our algorithms outperform FMS algorithm as the circuit becomes more sparse. To see this, note that the circuits are ordered in ascending density in the tables, and the performance of our algorithms gets better as we go up in the tables displaying the cutsizes. For example, PFM3 algorithm performs 63% better for multiway partitioning of `struct` and 50% better for multiway partitioning of `prim2` than FMS algorithm on the average. However, there do exist some anomalies such as the performance on `prim2` and `test06`. The reason for such anomalies seems to be that the circuits not only vary in their densities but also in their structures as well as their sizes. Such anomalies are expected to decrease if we can generate circuits such that only their densities were different but all the other properties were the same. Our experiments on randomly generated circuits that vary only in their density support the conclusion above. The better performance of our algorithms on sparse circuits is very promising because real applications as well as circuits difficult to partition are usually very sparse. As a test instance gets denser, even the performance of a simple greedy algorithm becomes comparable to that of KL algorithm [2].

For bipartitioning, we have a different situation. FM algorithm usually dominates the other algorithms in terms of the average quality of the minimum cutsizes. It also performs better than the others except for PLM3 algorithm in terms of the average quality of the average cutsizes. Surprisingly, FM algorithm has the largest stability ratio, i.e., the most unstable algorithm, and usually executes the largest number of passes. We think that the following reasons usually account for the dominance of FM algorithm. As mentioned in Section 4.1 and shown by the experimental results in this section, the performance of an algorithm tends to improve as the number of partitions examined per pass of that algorithm increases; however, we should also consider the number of passes that the algorithm executes. The number of partitions examined per pass of each of our algorithms is larger than that of FM algorithm, but we should multiply the number of passes by the number of partitions examined per pass to find the total number of partitions examined by an algorithm. Since FM algorithm has the largest number of passes, it is very likely that it examines more partitions than some of our algorithms. For example, the total number of solutions examined by PLM1 algorithm on the average is less than FM algorithm. Besides, our algorithms may explore the same solutions more than once since they relax the locking mechanism. Thus, even though some of our algorithms, e.g., PFM1 algorithm, may examine more partitions in total than FM algorithm, the total number of different partitions may be the same for both. Note that the performance of our algorithms gets better as we permit more cell moves per pass, thereby increasing the number of different solutions examined per pass. The

moves with zero gain can also be a problem for our algorithms for bipartitioning. Since our algorithms allow a particular move to occur more than once, a move with zero gain may waste a substantial portion of the total number of moves allowed per pass without any decrease in the cutsize. To eliminate this problem at least partially, we did an experiment with PFMi algorithms by setting α in Equation 9 to 1 so that the move capability of moves declined. We observed a performance improvement, suggesting the presence of better mobility functions for bipartitioning. Moreover, since PLM3 algorithm beats FM algorithm on the average quality of the average cutsizes, a greedy strategy like the locking mechanism should be preferred for bipartitioning, and, again, allowing more cell moves per pass yields better results, as claimed. Finally, note that SA algorithm basically selects the cell moves randomly, and so longer runs of SA algorithm tend to give better results than multiple shorter runs of it [10]. Since our algorithms like SA algorithm employs a less greedy strategy in selecting cell moves, their longer runs, i.e., more cell moves per pass, happen to give better results, as shown by the experimental results.

The running times of the proposed algorithms are basically proportional directly to the number, N, of cell moves per pass in practice, as shown by Equation 13. Thus, their running times must be larger than that of FMS algorithm. This results also shows that the proposed algorithms are efficient. The running times of PFMi algorithms are not affected much by such time-consuming operations as the exponential function evaluation due to the table look-up technique used. In general, we trade better solution quality (smaller cutsize) against larger running time in all the algorithms we compared.

For large k and best results, SA algorithm is the best choice if the time is not a hard constraint. However, if we are faced with the problem of finding the best solution in shorter time, then the best choice seems to be PFM2 algorithm. For small k ($k > 2$), we suggest the use of either PFM3 algorithm or PFM2 algorithm. For bipartitioning, we should use FM algorithm to obtain good partitions in very short time, but PLM3 algorithm or SA algorithm if more time is allowed. In short, since any of our algorithms and SA algorithm outperform FMS algorithm for multiway partitioning, and PLM3 algorithm outperforms FM algorithm for bipartitioning, we should choose them by taking into account the tradeoff between the better solution quality and shorter running time. We can conceive that we can run FMS algorithm a large number of times so as to obtain better partitions than any of the other algorithms since the running time of FMS algorithm is small compared to those of the other. However, it seems that we cannot make a substantial improvement in the average cutsize. For example, we obtained almost no improvement in the average cutsize but only a 10% improvement in the minimum cutsize in an experiment with FMS algorithm for 4-way partitioning of `c1355` involving 100,000 runs. Finally, after some experimentation, we observed that a partition that is a local minimum for FMS algorithm was not a locally minimum partition for the proposed algorithms, but a partition that is a local minimum for the proposed algorithms was usually a locally minimum partition for FMS algorithm. Also, any partition that is a local minimum for a PLMi algorithm with $N_{in} = n$ is always a locally minimum partition for FMS algorithm.

Figures 7 and 8 illustrate the evolution of the cutsize with the cell moves in PLM2 and PFM2 algorithms for 4-way partitioning of `s838` with 495 cells. Each interval between two successive vertical lines corresponds to a pass. The "current cutsize" curve corresponds to the tentative moves, whereas the "final cutsize" curve corresponds to the permanent moves, i.e., those moves in the maximum prefix subsequence. In Figure 7, each spike roughly corresponds a phase. The height of each spike is smaller compared to those in Figure 1. The evolution of the cutsize in PLM2 algorithm is similar to that in FMS algorithm since they both use the locking mechanism though in different ways. The evolution of the cutsize in PFM2 algorithm is very different. As seen in Figure 8, almost all the moves in a pass (especially those in initial passes) are included in the maximum prefix subsequence, and there are no high spikes like those in the other figures.

4.4 Experiments on Algorithm Parameters and Discussion

In order to see the effect of the scale factor on the performance of PFMi algorithms, we did a number of runs for each value of R starting with the same initial partitions for each PFMi algorithm. Table 6 presents the average of the 5 best cutsizes for PFM3 algorithm. We also did experiments with PFM1 and PFM2 algorithms but did not report them here in tables due to lack of space. In Table 6, `struct` is very sparse, `balu` is very dense, and `c2670` is in between.

We expect that, as S increases, the performance of a PFMi algorithm gets better because, at small S, a bucket list may contain many moves with the same mobility but have large differences in their move gains, thereby preventing moves with larger gains to be selected during the early stages of a pass. Note that, at $S = 1$, PFM algorithm degenerates to a variant of a random search algorithm. As shown in Table 6, the performance gets better as the scale factor S increases, as expected. Also note that PFM3 algorithm outperforms FMS algorithm even when $R = 1$, i.e., $S = 2G_{max} + 1$. It seems that the performance gets

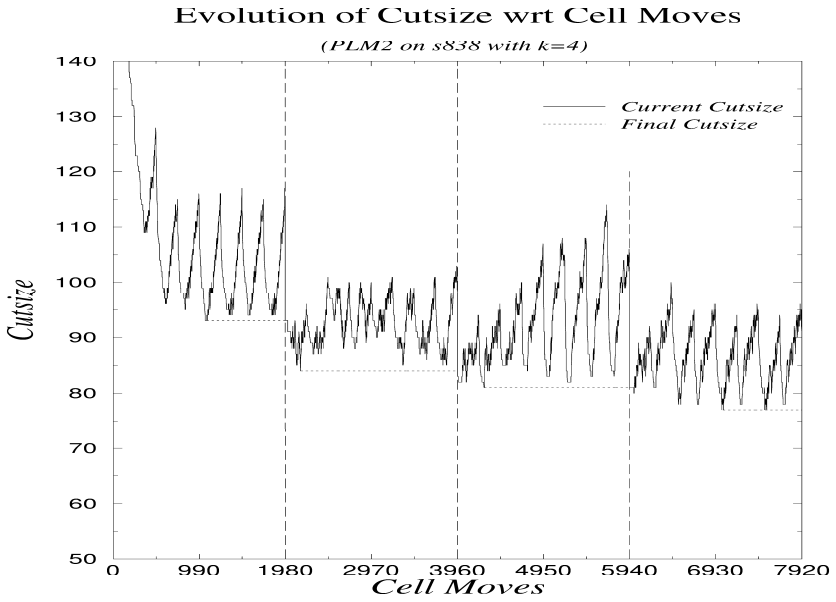

Figure 7: Evolution of cutsize with respect to cell moves for the PLM2 Algorithm on 4-way partitioning of s838 with 495 cells. Each interval between two successive vertical lines corresponds to a pass. The current cutsize and the final cutsize curves correspond to the set of tentative and permanent cell moves, respectively. The initial cutsize is 379, and the final cutsize is 77.

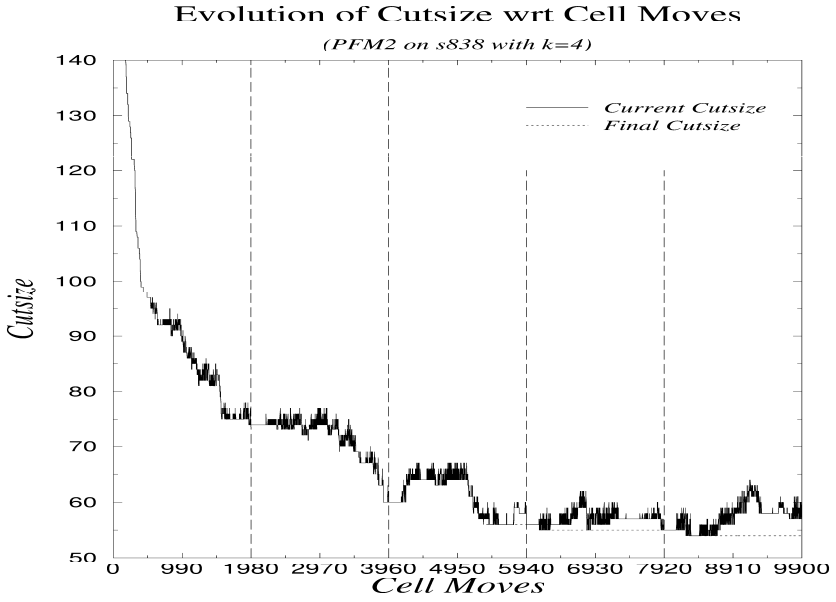

Figure 8: Evolution of cutsize with respect to cell moves for the PFM2 Algorithm on 4-way partitioning of s838 with 495 cells. Each interval between two successive vertical lines corresponds to a pass. The current cutsize and the final cutsize curves correspond to the set of tentative and permanent cell moves, respectively. The initial cutsize is 379, and the final cutsize is 54.

better as the circuit becomes more sparse, but the performance does not improve steadily as the scale factor increases. We think that a very large value for the scale factor prevents the selection of uphill moves, those moves increasing the cutsize, thereby forcing the algorithm to converge prematurely; yet, a very small value for the scale factor tends to forbid the algorithm to find better solutions for difficult test instances such as very sparse circuits. We suggest that a large R should be chosen when the circuit is sparse, when k is large, and when G_{max} is small. Any $R \geq 1$ is a safe choice in that the results are better than those of FMS algorithm.

In order to see the effect of the parameters N and N_{in} on the performance of PLM algorithm, we did a number of runs for different values of these parameters starting with the same initial partitions. Table 7 presents the average of the 5 best cutsizes for each combination of these parameters. Note that the column with $N = n$ and $N_{in} = n$ corresponds to FMS algorithm. In Table 7, `struct` is very sparse, `balu` is very dense, and `c2670` is in between.

We see that, for bipartitioning, the best performance is produced when $N_{in} = n$ for $N = n$ and $N = nk$, and when $N_{in} = n$ or $N_{in} = n/2$ for $N = nk^2$. Thus, for bipartitioning, each phase should contain n moves when N is small, and n or $n/2$ moves when N is larger. For multiway partitioning, the best performance is produced when $N_{in} = 3n/4$ for $N = n$ and $N = nk$, and when $N_{in} = n/2$ or $N_{in} = 3n/4$ for $N = nk^2$. Note that as the number, N, of moves per pass increases, the results get better, and that the overall best results are produced at $N = nk^2$ on `struct` and `test03`, as well as at $N = nk$ and $N = nk^2$ on `c2670`. One of the reasons why we set $N_{in} = n/2$ in our default setting was to allow each cell to be moved twice on the average. It should also be noted that almost all of the cutsizes in these tables are better than those by FMS algorithm, which provides another support to our claim.

5 Conclusions and Future Work

All the KL-based algorithms employ the locking mechanism which enforces each cell to be moved exactly once per pass. In this paper, we proposed two novel approaches for multiway circuit partitioning to overcome this limitation. The proposed approaches are based on our claim that performance gets better by allowing each cell to be moved more than once per pass. The first approach introduces the phase concept, and the second introduces the mobility concept. Each approach leads to a generic algorithm whose parameters can be set in such a way that better performance is obtained by spending more time. By setting the parameters, we generated three different versions of each of our algorithms. The proposed algorithms were experimentally evaluated in comparison with FMS algorithm and the SA algorithm on a subset of benchmark circuits. The experimental results show that the proposed algorithms outperform FMS algorithm significantly especially on multiway partitioning as well as partitioning of sparse circuits. The performance of some of the proposed algorithms is comparable to that of SA algorithm even though the running time of SA algorithm is far larger than those of the proposed algorithms. We performed some experiments on the parameters of the proposed algorithms, and suggested guidelines for good parameter settings. Besides, we gave a time and space complexity analysis of our algorithms. Experimental results and our arguments on the size of the search space explored by the algorithms compared seem to provide support for our claim. The proposed approaches are easily applicable to almost all of the existing KL-based algorithms such as those in [16, 18, 22, 24, 26, 27], and also the approaches in those works such as multilevel gains [22], the ratio cut metric [27], and so on, are easily applicable to our algorithms.

We now mention some areas for further research. First, we think that better mobility functions are possible. But, a simple function such as $f_i(s,t) = (G_i(s,t) + G_{max})/(2n_i^{\alpha}G_{max})$ does not work. Second, a simple way to reduce the running time of the KL-based algorithms is to use an adaptive scheme in that the number of cell moves per pass can vary. Usually, the number of cell moves in the maximum prefix subsequence in a pass is smaller than that in the previous pass. Hence, the number of cell moves in a pass can be set to a fraction of that in the previous pass. This fraction can be determined by experimental analysis for each algorithm, but we think that it depends also on the size and structure of the test instance being partitioned. An adaptive scheme for PLMi algorithms can also change the number of cell moves in each phase. Third, a phase concept similar to the phase concept can be used for the PFM algorithm. Each pass can contain a number of phases, and after each phase, we initialize each move count to zero and start another phase. This phase concept seems promising since the mobility values of cells get smaller and smaller during partitioning, thereby partially preventing the selection of the moves with larger gains. This problem can be avoided by the usage of a large scale factor also. Fourth, we can change the way the move counts are incremented in that we can penalize the moves with smaller gains more and more. Fifth, an application of our algorithms to VLSI placement problem seems to be promising, as there are a number of very successful

VLSI placement systems based on partitioning [6, 15, 19].

References

[1] M. A. Breuer. Min-cut placement. *Journal of Design Automation and Fault Tolerant Computing*, 1(4):343–362, October 1977.

[2] T. N. Bui, S. Chaudhuri, F. T. Leighton, and M. Sipser. Graph bisection algorithms with good average case behavior. *Combinatorica*, 7(2):171–191, 1987.

[3] T. Bultan and C. Aykanat. Circuit partitioning using parallel mean field annealing algorithms. *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, pages 534–541, December 1991.

[4] T. Bultan and C. Aykanat. Circuit partitioning using mean field annealing. *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, 8:to appear, 1995.

[5] P. K. Chan, D. F. Schlag, and J. Y. Zien. Spectral K-way ratio-cut partitioning and clustering. In *Proceedings of the 30th Design Automation Conference*, pages 749–754. ACM/IEEE, 1993.

[6] A. E. Dunlop and B. W. Kernighan. A procedure for placement of standard-cell VLSI circuits. *IEEE Transactions on Computer-Aided Design*, 4(1):92–98, January 1985.

[7] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th ACM/IEEE Design Automation Conference*, pages 175–181, 1982.

[8] M. R. Garey and D. S. Johnson. *Computers and Intractability.* W.H. Freeman and Co., New York, New York, 1979.

[9] L. Hagen, F. J. Kurdahi A. B. Kahng, and C. Ramachandran. On the intrinsic Rent parameter and spectra-based partitioning methodologies. *IEEE Transactions on Computer-Aided Design*, 13(1):27–37, January 1994.

[10] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by simulated annealing: An experimental evaluation; part I, graph partitioning. *Operations Research*, 37(6):865–892, November 1989.

[11] 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. *Operations Research*, 39(3):378–406, May 1991.

[12] N. Karmarkar and R. M. Karp. The differencing method of set partitioning. Technical Report UCB/CSD 82/113, University of California, Berkeley, December 1982.

[13] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, February 1970.

[14] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[15] J. M. Kleinhans, G. Sigl, F. M. Johannes, and K. J. Antreich. GORDION: VLSI placement by quadratic programming and slicing optimization. *IEEE Transactions on Computer-Aided Design*, 10(3):356–365, March 1991.

[16] B. Krishnamurthy. An improved min-cut algorithm for partitioning VLSI networks. *IEEE Transactions on Computers*, 33(5):438–446, May 1984.

[17] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout.* John Wiley and Sons, Inc., Chichester, West Sussex, England, 1990.

[18] C. Park and Y. Park. An efficient algorithm for VLSI network partitioning problem using a cost function with balancing factor. *IEEE Transactions on Computer-Aided Design*, 12(11):1686–1694, November 1993.

[19] R. L. Rivest. The "PI" (placement and interconnect) system. In *Proceedings of the 19th Design Automation Conference*, pages 475–481. IEEE, 1982.

[20] Y. G. Saab and V. B. Rao. An evolution-based approach to partitioning ASIC systems. In *Proceedings of the 26th ACM/IEEE Design Automation Conference*, pages 767–770, 1989.

[21] Y. G. Saab and V. B. Rao. Combinatorial optimization by stochastic evolution. *IEEE Transactions on Computer-Aided Design*, 10(4):525–535, April 1991.

[22] L. A. Sanchis. Multiple-way network partitioning. *IEEE Transactions on Computers*, 38(1):62–81, January 1989.

[23] D. G. Schweikert and B. W. Kernighan. A proper model for the partitioning of electrical circuits. In *Proceedings of the 9th ACM/IEEE Design Automation Conference*, pages 57–62, 1972.

[24] C. Sechen and D. Chen. An improved objective function for mincut circuit partitioning. In *Proceedings of the International Conference on Computer-Aided Design*, pages 502–505. IEEE, 1988.

[25] K. Shahookar and P. Mazumder. VLSI cell placement techniques. *ACM Computing Surveys*, 23(2):144–220, June 1991.

[26] H. Shin and C. Kim. A simple yet effective technique for partitioning. *IEEE Transactions on VLSI Systems*, 1(3):380–386, September 1993.

[27] Y.-C. Wei and C.-K. Cheng. Ratio cut partitioning for hierarchical designs. *IEEE Transactions on Computer-Aided Design*, 10(7):911–921, July 1991.

Appendix

Proposition 2.1: *A net e_j is critical if and only if either there exists a part P_s such that $\delta_j(s) = |e_j|$, or there exist two different parts P_s and P_t such that $\delta_j(s) = 1$ and $\delta_j(t) = |e_j| - 1$.*
Proof: We first prove that if a net is critical then either there exists a part P_s such that $\delta_j(s) = |e_j|$, or there exist two different parts P_s and P_t such that $\delta_j(s) = 1$ and $\delta_j(t) = |e_j| - 1$. Let Δ_j denote the number of parts that e_j connects. Note that if $\Delta_j \geq 2$, then e_j is in the cutset. If $\Delta_j \geq 3$, then any move of cells on e_j can decrease Δ_j by at most one, and so cannot remove e_j from the cutset, i.e., e_j is not critical. Thus, we must have $\Delta_j \leq 2$. If e_j is critical, then it has a cell that either would remove e_j from the cutset or insert it into the cutset if it is moved. For a cell to insert e_j into the cutset, we must have $\Delta_j = 1$, and so $\delta_j(s) = |e_j|$ for some part P_s. For a cell to remove e_j from the cutset, we must have $\Delta_j = 2$, and so $\delta_j(s) = 1$ and $\delta_j(t) = |e_j| - 1$ for two different parts P_s and P_t so that the move of the cell from P_s to P_t can remove e_j from the cutset.

Now, let us prove that a net e_j is critical if either there exists a part P_s such that $\delta_j(s) = |e_j|$, or there exist two different parts P_s and P_t such that $\delta_j(s) = 1$ and $\delta_j(t) = |e_j| - 1$. If $\delta_j(s) = |e_j|$ for some part P_s, then the move of a cell on e_j from P_s to another part in the partition will insert e_j into the cutset, and so e_j is critical. If $\delta_j(s) = 1$ and $\delta_j(t) = |e_j| - 1$ for two different parts P_s and P_t, then the move of a cell on e_j from P_s to P_t will remove e_j from the cutset, and so e_j is critical.
Proposition 2.3: *Consider the move of cell v_i from P_s to P_t. Let the cutsize before and after the move be denoted by $\chi(\Pi)$ and $\chi'(\Pi)$, respectively. Then, $\chi'(\Pi) = \chi(\Pi) - G_i(s,t)$ where $G_i(s,t)$ is the move gain of v_i before the move.*
Proof: By Propositions 2.1, 2.2, and the definition of costs of a cell, the move of cell v_i from P_s to P_t can only change the cutstates of the nets in $E_i(s,t)$ and $I_i(s)$, which consist of critical nets. For each net e_j in $E_i(s,t)$, the move yields $\delta_j(t) = |e_j|$, and, for each net e_j in $I_i(s)$, it yields $\delta_j(s) = |e_j| - 1$ and $\delta_j(t) = 1$. That is, the move removes the nets in $E_i(s,t)$ from the cutset and inserts those in $E_i(s,t)$ to the cutset. By Equation 5, the decrease in the cutsize due to the nets in $E_i(s,t)$ is equal to the $C_i(s,t)$, and, by Equation 6, the increase in the cutsize due to the nets in $I_i(s)$ is equal to $C_i(s,s)$. Therefore, we have, $\chi'(\Pi) = \chi(\Pi) - C_i(s,t) + C_i(s,s) = \chi(\Pi) - (C_i(s,t) - C_i(s,s)) = \chi(\Pi) - G_i(s,t)$.
Proposition 2.4: *The initial cost computation algorithm given in Figure 3 computes costs of each cell as defined in Equations 5 and 6. Furthermore, it has a time complexity of $O(pk)$.*
Proof: The **for** loop of lines 1-10 performs the same operations for each cell; hence, let us consider a cell v_i in P_s. The **for** loop of lines 2-3 initializes each cost of v_i to zero. By Proposition 2.2, the costs of v_i depend only on critical nets, and these costs are computed as in Equation 5 and Equation 6. Thus, for each net e_j on v_i, we first check whether e_j is critical or not using Proposition 2.1. If $\delta_j(s) = 1$ and there exists a part P_t such that $\delta_j(t) = |e_j| - 1$, then e_j is critical and line 8 increments $C_i(s,t)$ by c_j as in Equation 5. If $\delta_j(s) = |e_j|$, then line 10 increments $C_i(s,s)$ by c_j as in Equation 6.

We now determine the time complexity of this algorithm. The **for** loop of lines 2-3 is executed $\Theta(k)$ times, as each cell has k costs. Lines 7-8 and line 10 take constant time. Line 6 requires $O(k)$ time because we should check every part in the partition in the worst case. The **for** loop of lines 4-10 iterates d_i times. Since $\sum_{i=1}^{n} d_i = p$ by Equation 1, the algorithm takes $O(pk)$ time.
Proposition 2.5: *The cost update algorithm given in Figure 4 updates the costs of the cell moved and those of its neighbors as defined in Equations 5 and 6. Furthermore, if the locking mechanism is used, then this algorithm takes $O(pkG_{max})$ time for n cell moves. If not, then it takes $O(kG_{max}D_{v,max}D_{e,max})$ time per move where $D_{v,max}$ is the maximum cell degree, and $D_{e,max}$ is the maximum net degree.*
Proof: For each net e_j on v_i, we must have $\delta_j(s) \geq 1$ before the move, and $\delta_j(t) \geq 1$ after the move since v_i is moved from P_s to P_t. Moreover, by Proposition 2.2, we should only consider the critical nets on v_i for the cost updates. Hence, by Proposition 2.1, the remaining cases are as follows:

1. A net e_j on v_i is critical before the move if and only if one of the following cases holds:
   1.a) $\delta_j(s) = 1$ and there exists a part P_l such that $\delta_j(l) = |e_j| - 1$,
   1.b) $\delta_j(s) = |e_j| - 1$ and there exists a part P_l such that $\delta_j(l) = 1$, and
   1.c) $\delta_j(s) = |e_j|$.
2. A net e_j on v_i is critical after the move if and only if one of the following cases holds:
   2.a) $\delta_j(t) = 1$ and there exists a part P_l such that $\delta_j(l) = |e_j| - 1$,
   2.b) $\delta_j(t) = |e_j| - 1$ and there exists a part P_l such that $\delta_j(l) = 1$, and

2.c) $\delta_j(t) = |e_j|$.

Now, case (1.a) is equivalent to case (2.c) for $l = t$, whereas it can be handled by cases (1.b) and (2.b) for $l \neq t$ when $|e_j| = 2$. Similarly, case (2.a) is equivalent to case (1.c) for $l = s$, whereas it can be handled by cases (1.b) and (2.b) for $l \neq s$ when $|e_j| = 2$. Note that, cases (1.a) and (2.a) for $l \neq t$ and for $l \neq s$, respectively, do not involve any updates when $|e_j| > 2$. Hence, it suffices to consider only cases (1.b), (1.c), (2.b), and (2.c). In the algorithm, lines 3-6, lines 7-10, lines 13-16, and lines 17-20 handle cases (1.c), (1.b), (2.c), and (2.b), respectively.

In case (1.c), since $\delta_j(s) = |e_j|$ before the move, c_j is already included in the internal costs $C_r(s,s)$ of each cell v_r on e_j. After the move, e_j is a cut net and does not contribute to the internal costs of these cells; hence, its weight, c_j, should be subtracted from these internals costs, which is done in line 5. In case (1.b), since $\delta_j(s) = |e_j| - 1$ before the move, c_j is already included in the external cost $C_r(l,s)$ of a cell v_r in P_l where $l \neq s$. After the move, e_j is not critical any more, and so its contribution to this external cost should be eliminated, which is done in lines 9. Lines 11-12 update $\delta_j(s)$ and $\delta_j(t)$, as the number of terminals of e_j in P_s should be decremented by 1, and that in P_t should be incremented by 1 with the move of v_i, which is on e_j, from P_s to P_t. In case (2.c), since $\delta_j(t) = |e_j|$ after the move, e_j should contribute to the internal costs $C_r(t,t)$ of each cell v_r on e_j, which is done in lines 15. In case (2.b), since $\delta_j(t) = |e_j| - 1$ after the move, e_j becomes a critical net. Then, e_j should contribute to the external cost $C_r(l,t)$ of a cell v_r in P_l where $l \neq t$, which is done in lines 19.

Since the mobility values of a cell depend on its gains, which in turn depend on its costs, any cost change should be forwarded to the corresponding gains both for the PLM and the FMS algorithms, and to the corresponding gains and mobility values for the PFM algorithm. The gain (and mobility) updates of neighbor cells are performed in lines 6, 10, 16, and 20 using Equations 7 and 11.

For the cell moved, which is v_i, there are no changes in its costs, but the interpretation of its costs changes, and so its gains (and mobility values) should be updated. To see this, note that v_i is moved from P_s to P_t. If $\delta_j(s) = |e_j|$ for a net e_j on v_i before the move, then c_j is already included in $C_i(s,s)$. After the move, we have $\delta_j(s) = |e_j| - 1$ and $\delta_j(t) = 1$, and so c_j should be included in $C_i(t,s)$, but, since $C_i(t,s) = C_i(s,s)$, it is already included. If $\delta_j(s) = 1$ and $\delta_j(t) = |e_j| - 1$ for a net e_j on v_i before the move, then c_j is already included in $C_i(s,t)$. After the move, we have $\delta_j(t) = |e_j|$, and so c_j should be included in $C_i(t,t)$, but, since $C_i(t,t) = C_i(s,t)$, it is already included. On the other hand, if $\delta_j(s) = 1$ and $\delta_j(l) = |e_j| - 1$ for a net e_j on v_i where $l \neq t$ before the move, then c_j is already included in $C_i(s,l)$. After the move, we have $\delta_j(t) = 1$ and $\delta_j(l) = |e_j| - 1$, and so c_j should be included in $C_i(t,l)$, but, since $C_i(t,l) = C_i(s,l)$, it is already included. Hence, no changes are necessary in the costs of v_i.

Next, we determine the time complexity of the cost update algorithm. Assume that the locking mechanism is used, and so each cell is locked immediately after it is moved and it is not moved any more. We will prove that, for each net, we need to do a constant number of update operations. Recall that the computation of a gain or mobility of a cell takes constant time provided that the associated costs are computed already, and also that each cell has k costs, and P_s and P_t represents the source and target parts in a move, respectively.

For a net e_j, lines 4-6 are executed at most once because all the unlocked cells on e_j can only be in P_s, and, after a move involving a cell on e_j from P_s, we can no longer have $\delta_j(s) = |e_j|$. For a net e_j, lines 8-10 are executed at most three times. To see this, if $\delta_j(s) = |e_j| - 1$ and $\delta_j(l) = 1$ for some part P_l before a move, then only P_s and P_l can again be a source part for a move involving a cell on e_j. Now, since P_l can have only one unlocked cell that is a terminal of e_j, then we can have $\delta_j(l) = |e_j| - 1$ at most once. Also, if $\delta_j(s) = |e_j| - 1$ before a move of a cell on e_j from P_s, then, later, we can have $\delta_j(s) = |e_j| - 1$ at most once again only if the only terminal of e_j in P_l has been moved to P_s.

For a net e_j, lines 14-16 are executed at most once because all the terminals of e_j are already in P_t, and P_t cannot be a target part again satisfying $\delta_j(t) = |e_j|$. For a net e_j, lines 18-20 are executed at most three times. To see this, if we already have $\delta_j(t) = |e_j| - 1$ after a move to P_t, then e_j has at least one locked cell in P_t. Assume that all the other cells on e_j are unlocked and the only cell of e_j which is not in P_t is in P_l after the move. If all the unlocked cells of e_j in P_t are moved to P_l then we will have $\delta_j(l) = |e_j| - 1$ after the last move. Final execution of these lines may occur only if $|e_j| = 3$, and the last unlocked cell of e_j in P_l is moved to P_t. Therefore, it follows that for a net e_j, lines 4-6 and lines 14-16 are executed at most once, respectively, and lines 8-10 and lines 17-20 are executed at most three times, respectively.

The **for** loop of lines 4-6 takes $O(kG_{max}|e_j|)$ time since we update $(k-1)$ gains for each cell on e_j and these gain updates can involve $(k-1)$ deletions from bucket lists. Line 8 requires $O(k)$ time, line 9 takes constant time, and line 10 takes $O(G_{max})$ time since it can involve only one deletion from a bucket list. The **for** loop of lines 14-16 takes $O(kG_{max}|e_j|)$ time since we update $(k-1)$ gains for each cell on e_j and these

gain updates can involve $(k-1)$ deletions from bucket lists. Finally, line 18 requires $O(k)$ time, line 19 takes constant time, and line 20 takes $O(G_{max})$ time since it can involve only one deletion from a bucket list. Moreover, line 1 requires $O(kG_{max})$ time since it can involve $(k-1)$ deletions from bucket lists. In total, the cost update algorithm runs in time $O(kG_{max} + \sum_{j=1}^{m} 2(kG_{max}|e_j| + 3k + 3G_{max})) = O(pkG_{max})$ for n cell moves.

If the locking mechanism is not used, we cannot predict how many times each cell will be moved, but can do the following worst-case analysis for a single cell move. The **for** loop of lines 2-20 can iterate at most $D_{v,max}$ times where $D_{v,max}$ is the maximum cell degree. The **for** loops of lines 4-6 and 14-16, respectively, can iterate at most $D_{e,max}$ times where $D_{e,max}$ is the maximum net degree. Since, from the preceding time complexity analysis, we know how much time each of the other operations takes, the cost update algorithm runs in time $O(kG_{max} + 2D_{v,max}(kG_{max}D_{e,max} + k + G_{max})) = O(kG_{max}D_{v,max}D_{e,max})$ for a cell move. If we assume that the mobility concept is used, and so the size of each bucket array is equal to the scale factor S, the time complexity becomes $O(kSD_{v,max}D_{e,max})$.

Proposition 3.2: *The time complexity of PLM algorithm (Figure 5) for a k-way partitioning of a circuit with p pins is $O(N_{out}pk(k + G_{max}))$ where N_{out} is the number of phases per pass. Its space complexity is $O(pk + k^2G_{max} + pN_{out})$.*

Proof: Line 1 initializes bucket list pointers in $k(k-1)$ bucket arrays to null, and each bucket array has $2G_{max}+1$ buckets. Thus, line 1 takes $\Theta(k^2G_{max})$ time. Line 4 uses the initial cost computation algorithm in Figure 3, and thus requires $O(pk)$ time by Proposition 2.4. In line 5, the move gains of each cell are computed, and each cell is inserted into the bucket lists in $\Theta(nk)$ time since the computation of a move gain and insertion to a bucket list require constant time. Since there are $k(k-1)$ different move directions, a search for the move with the maximum gain involves only one move at the front of each bucket list indexed by the maximum gain pointer. Thus, line 7 can be done in $\Theta(k^2)$ time. A deletion of a cell from k bucket lists in line 8 requires $O(kG_{max})$ time. Line 10 takes constant time since it only involves locking the cell moved and changing the part in which the cell moved lies. In line 11, the cost update algorithm in Figure 4 is used, and this algorithm takes $O(pkG_{max})$ time for n cell moves by Proposition 2.5. For the next move-and-lock phase, the moves of the unlocked cells remaining in the bucket lists should be removed. Line 13 does this if there are unlocked cells, i.e., if $N_{in} < n$. Since there are $(n - N_{in})$ cells that are not moved yet, and each of them has $(k-1)$ moves in the bucket lists, line 13 takes $O(nkG_{max})$. Line 13 can also be done in $O(k^2G_{max}+kn)$ time by searching each slot in all the bucket arrays and deleting each bucket node in the non-empty bucket lists. We will use the first expression for simplicity. Hence, the **repeat** loop of lines 6-11 takes $N_{in}(\Theta(k^2) + O(kG_{max}) + O(1)) + O(pkG_{max}) = O(pk^2 + pkG_{max})$ time since $N_{in} = \Theta(n) = O(p)$, and the **for** loop of lines 3-13 takes $N_{out}(O(pk)+O(nk)+O(pk^2+pkG_{max})+O(nkG_{max})) = O(N_{out}(pk^2+pkG_{max}))$ time.

Constructing the maximum prefix subsequence in line 14 involves a visit to every move performed and saved in a sequence. Thus, line 14 takes $O(N) = O(N_{out}N_{in}) = O(N_{out}p)$ time. In line 16, every move included in the maximum prefix subsequence is made permanent, and the distribution of nets on the cell moved is updated; hence, line 16 requires $O(N)O(N_{out}p)$ time. Also, lines 15 and 17 take constant time. Therefore, the time complexity of the PLM algorithm per pass (lines 2-18) is $O(N_{out}(pk^2 + pkG_{max})) + O(N_{out}p) + O(N_{out}p) = O(N_{out}(pk^2 + pkG_{max}))$ time.

For the space complexity analysis, consider a hypergraph $H = (V, E)$ with n cells, m nets, and p terminals to be partitioned into k parts. The cells require $O(n)$ space, the distributions of the nets to the parts require $O(mk)$ space, and the nets need $O(p)$ space. There are $k(k-1)$ bucket arrays each with a size of $2G_{max}+1$, and there are $n(k-1)$ bucket list nodes, as each cell is inserted into $(k-1)$ bucket lists. Hence, the bucket list data structure takes $O(k^2G_{max} + nk)$ space. Since we also save each tentative move performed during a pass, we need a space of size N where $N = O(N_{out}p)$. Hence, the total amount of space required is $O(pk + k^2G_{max} + N_{out}p)$.

Proposition 3.3: *The time complexity of PFM algorithm (Figure 6) for a k-way partitioning of a circuit with p pins is $O(k(p + kS + N(k + D_{v,max}D_{e,max}S)))$ where $D_{v,max}$ and $D_{e,max}$ are the maximum cell and net degrees, respectively, and N is the number of moves per pass. Its space complexity is $O(pk + k^2S + N)$.*

Proof: Line 1 initializes bucket list pointers in $k(k-1)$ bucket arrays to null, and each bucket array has S buckets. Thus, line 1 takes $\Theta(k^2S)$ time. Line 3 uses the initial cost computation algorithm in Figure 3, and thus requires $O(pk)$ time by Proposition 2.4. In line 4, the move gains of each cell are computed, and each cell is inserted into the bucket lists on the basis of their mobility values in $\Theta(nk)$ time since the computation of a move gain as well as mobility and insertion to a bucket list require constant time. Since there are $k(k-1)$ different move directions, a search for the move with the maximum mobility involves only one move at the front of each bucket list indexed by the maximum mobility pointer. Thus, line 6 can be

done in $\Theta(k^2)$ time. Line 8 takes constant time. In line 9, the cost update algorithm in Figure 4 is used, and this algorithm takes $O(kSD_{v,max}D_{e,max})$ time for a single cell move by Proposition 2.5. Hence, the loop of lines 5-10 takes $N(\Theta(k^2) + O(1) + O(1) + O(kSD_{v,max}D_{e,max})) = O(N(k^2 + kSD_{v,max}D_{e,max}))$ time for each move.

Constructing the maximum prefix subsequence and finding the maximum prefix sum G_T are exactly the same as in the the proof of Proposition 3.2, and so line 11 requires $O(N)$ time. Line 12 and line 14 take constant time. Line 13 is similar to the one in the PLM algorithm, but takes $O(ND_{v,max})$ time. Since the PFM algorithm does not remove moves from the bucket lists at each cell move, all the moves for each cell in the bucket lists should be deleted for the next pass. Thus, line 15 takes $O(k^2S + kn)$ time. Therefore, the time complexity of the PFM algorithm per pass (lines 2-16) is $O(pk) + \Theta(nk) + O(N(k^2 + kSD_{v,max}D_{e,max}) + O(N) + O(ND_{v,max}) + O(k^2S + kn) = O(pk + k^2S + N(k^2 + kSD_{v,max}D_{e,max})$ time.

The space complexity analysis is exactly the same as the one in the proof of Proposition 3.2 except for the size of a bucket array, which is equal to S for the PFM algorithm. Hence, the total amount of space required is $O(pk + k^2S + N_{out}p)$.

Table 2: Cutsize averages for benchmark circuits. The number of parts is denoted by k. Each value in parentheses give the ratio of the average cutsize found by the respective algorithm to that of FMS algorithm. Bold values are the best values in each row.

PROBLEM		*CUTSIZE AVERAGES*							
Name	k	*FMS*	*PLM1*	*PLM2*	*PLM3*	*PFM1*	*PFM2*	*PFM3*	*SA*
	2	56.8	62.3 (1.10)	**54.5** (0.96)	54.7 (0.96)	99.9 (1.76)	60.2 (1.06)	59.2 (1.04)	67.2 (1.18)
	4	300.4	259.3 (0.86)	230.2 (0.77)	211.4 (0.70)	166.6 (0.55)	126.3 (0.42)	**111.2** (0.37)	130.0 (0.43)
struct	6	408.4	321.3 (0.79)	289.5 (0.71)	267.1 (0.65)	260.1 (0.64)	214.0 (0.52)	183.3 (0.45)	**160.6** (0.39)
	8	496.8	432.2 (0.87)	394.0 (0.79)	371.6 (0.75)	369.5 (0.74)	303.7 (0.61)	295.1 (0.59)	**180.0** (0.36)
	2	278.2	367.7 (1.32)	312.5 (1.12)	268.9 (0.97)	284.2 (1.02)	259.0 (0.93)	239.7 (0.86)	**226.0** (0.81)
	4	828.7	771.8 (0.93)	718.2 (0.87)	671.6 (0.81)	657.1 (0.79)	454.1 (0.55)	**416.4** (0.50)	424.2 (0.51)
prim2	6	968.5	873.2 (0.90)	833.8 (0.86)	822.8 (0.85)	839.4 (0.87)	592.6 (0.61)	537.4 (0.55)	**508.0** (0.52)
	8	1043.2	982.6 (0.94)	941.0 (0.90)	901.5 (0.86)	942.3 (0.90)	666.5 (0.64)	626.5 (0.60)	**565.8** (0.54)
	2	**48.6**	61.4 (1.26)	56.4 (1.16)	52.6 (1.08)	107.6 (2.21)	90.8 (1.87)	76.9 (1.58)	83.6 (1.72)
	4	388.2	340.8 (0.88)	350.5 (0.90)	330.4 (0.85)	303.7 (0.78)	184.8 (0.48)	**162.7** (0.42)	171.4 (0.44)
c7552	6	498.2	435.9 (0.88)	399.6 (0.80)	372.8 (0.75)	420.7 (0.84)	262.6 (0.53)	**218.1** (0.44)	219.2 (0.44)
	8	550.7	517.1 (0.94)	484.0 (0.88)	444.0 (0.81)	478.4 (0.87)	306.1 (0.56)	**257.8** (0.47)	259.4 (0.47)
	2	38.7	41.0 (1.06)	40.7 (1.05)	36.4 (0.94)	48.7 (1.26)	35.3 (0.91)	**30.6** (0.79)	34.6 (0.89)
	4	96.9	90.9 (0.94)	81.1 (0.84)	75.7 (0.78)	87.5 (0.90)	62.0 (0.64)	**57.1** (0.59)	68.4 (0.71)
c1355	6	112.1	102.1 (0.91)	94.3 (0.84)	88.1 (0.79)	102.0 (0.91)	72.1 (0.64)	**68.4** (0.61)	77.6 (0.69)
	8	122.2	112.4 (0.92)	101.1 (0.83)	93.1 (0.76)	110.2 (0.90)	78.1 (0.64)	**74.1** (0.61)	80.4 (0.66)
	2	83.8	101.4 (1.21)	92.3 (1.10)	80.3 (0.96)	110.5 (1.32)	92.6 (1.11)	**72.9** (0.87)	77.2 (0.92)
	4	249.1	230.6 (0.93)	222.1 (0.89)	216.4 (0.87)	225.2 (0.90)	168.7 (0.68)	**129.6** (0.52)	144.8 (0.58)
c3540	6	290.6	273.2 (0.94)	261.0 (0.90)	246.8 (0.85)	271.5 (0.93)	201.6 (0.69)	**165.5** (0.57)	183.0 (0.63)
	8	317.9	303.1 (0.95)	281.1 (0.88)	263.0 (0.83)	294.4 (0.93)	218.4 (0.69)	**181.5** (0.57)	197.4 (0.62)
	2	51.4	61.0 (1.19)	57.2 (1.11)	53.7 (1.05)	70.3 (1.37)	54.0 (1.05)	**43.8** (0.85)	44.8 (0.87)
	4	131.1	119.6 (0.91)	114.3 (0.87)	108.2 (0.83)	119.4 (0.91)	81.2 (0.62)	**70.0** (0.53)	77.4 (0.59)
c2670	6	162.8	142.1 (0.87)	131.4 (0.81)	120.8 (0.74)	139.6 (0.86)	91.9 (0.56)	**78.7** (0.48)	85.8 (0.53)
	8	185.2	168.7 (0.91)	149.3 (0.81)	127.4 (0.69)	157.1 (0.85)	101.4 (0.55)	87.8 (0.47)	**85.8** (0.46)
	2	76.4	82.6 (1.08)	74.3 (0.97)	**65.9** (0.86)	84.7 (1.11)	72.4 (0.95)	72.8 (0.95)	73.8 (0.97)
	4	205.8	181.5 (0.88)	161.0 (0.78)	144.6 (0.70)	152.6 (0.74)	123.6 (0.60)	**111.8** (0.54)	113.6 (0.55)
prim1	6	244.7	217.4 (0.89)	192.7 (0.79)	172.6 (0.71)	199.6 (0.82)	145.4 (0.59)	**130.6** (0.53)	131.0 (0.54)
	8	274.0	253.7 (0.93)	218.3 (0.80)	191.5 (0.70)	227.8 (0.83)	159.0 (0.58)	148.7 (0.54)	**144.2** (0.53)
	2	26.6	30.0 (1.13)	27.9 (1.05)	24.0 (0.90)	31.7 (1.19)	27.3 (1.03)	23.2 (0.87)	**22.2** (0.83)
	4	85.3	80.2 (0.94)	75.7 (0.89)	76.6 (0.90)	79.4 (0.93)	50.4 (0.59)	**40.6** (0.48)	46.8 (0.55)
s838	6	107.4	99.3 (0.92)	94.4 (0.88)	83.2 (0.77)	100.2 (0.93)	65.3 (0.61)	**54.3** (0.51)	61.0 (0.57)
	8	121.3	116.1 (0.96)	103.8 (0.86)	94.3 (0.78)	111.3 (0.92)	75.6 (0.62)	**62.6** (0.52)	68.4 (0.56)
	2	57.9	61.5 (1.06)	62.5 (1.08)	**53.8** (0.93)	93.9 (1.62)	82.9 (1.43)	77.6 (1.34)	71.2 (1.23)
	4	438.2	374.1 (0.85)	320.0 (0.73)	295.7 (0.67)	339.9 (0.78)	190.7 (0.44)	**148.2** (0.34)	184.8 (0.42)
ind1	6	530.6	475.0 (0.90)	445.2 (0.84)	413.0 (0.78)	461.4 (0.87)	305.3 (0.58)	266.4 (0.50)	**263.4** (0.50)
	8	579.6	550.6 (0.95)	521.6 (0.90)	474.3 (0.82)	529.5 (0.91)	381.4 (0.66)	339.0 (0.58)	**293.2** (0.51)
	2	113.4	122.9 (1.08)	105.3 (0.93)	100.9 (0.89)	174.5 (1.54)	157.9 (1.39)	157.9 (1.39)	**89.8** (0.79)
	4	336.0	314.1 (0.93)	268.1 (0.80)	240.3 (0.72)	362.4 (1.08)	245.6 (0.73)	245.6 (0.73)	**157.4** (0.47)
test03	6	400.2	371.3 (0.93)	344.3 (0.86)	322.6 (0.81)	424.7 (1.06)	320.1 (0.80)	320.1 (0.80)	**226.8** (0.57)
	8	437.1	413.4 (0.95)	384.2 (0.88)	364.6 (0.83)	457.1 (1.05)	348.2 (0.80)	348.2 (0.80)	**250.8** (0.57)
	2	36.0	40.5 (1.13)	38.3 (1.07)	33.8 (0.94)	51.4 (1.43)	46.7 (1.30)	43.2 (1.20)	**33.2** (0.92)
	4	169.7	156.2 (0.92)	134.6 (0.79)	118.0 (0.70)	146.4 (0.86)	100.4 (0.59)	**74.6** (0.44)	76.2 (0.45)
balu	6	206.0	193.7 (0.94)	181.1 (0.88)	168.3 (0.82)	182.9 (0.89)	155.5 (0.75)	**123.9** (0.60)	125.6 (0.61)
	8	224.9	213.2 (0.95)	201.5 (0.90)	190.8 (0.85)	204.7 (0.91)	170.8 (0.76)	158.9 (0.71)	**146.2** (0.65)
	2	89.5	85.6 (0.96)	84.1 (0.94)	**80.8** (0.90)	140.2 (1.57)	93.3 (1.04)	91.8 (1.03)	81.8 (0.91)
	4	289.7	273.0 (0.94)	262.5 (0.91)	248.6 (0.86)	264.6 (0.91)	153.8 (0.53)	**137.8** (0.48)	151.2 (0.52)
test06	6	356.5	328.4 (0.92)	303.6 (0.85)	286.0 (0.80)	321.4 (0.90)	204.5 (0.57)	175.1 (0.49)	**173.0** (0.49)
	8	394.0	374.4 (0.95)	349.6 (0.89)	319.0 (0.81)	361.2 (0.92)	233.1 (0.59)	203.2 (0.52)	**191.8** (0.49)

Table 3: Minimum Cutsizes for benchmark circuits. The number of parts is denoted by k. Each value in parentheses give the ratio of the minimum cutsize found by the respective algorithm to that of FMS algorithm. Bold values are the best values in each row.

PROBLEM		*MINIMUM CUTSIZES*							
Name	k	*FMS*	*PLM1*	*PLM2*	*PLM3*	*PFM1*	*PFM2*	*PFM3*	*SA*
	2	40	43 (1.07)	43 (1.07)	47 (1.18)	58 (1.45)	**33** (0.82)	**33** (0.82)	36 (0.90)
	4	202	206 (1.02)	175 (0.87)	173 (0.86)	136 (0.67)	83 (0.41)	**76** (0.38)	121 (0.60)
struct	6	302	268 (0.89)	231 (0.76)	220 (0.73)	212 (0.70)	159 (0.53)	**122** (0.40)	145 (0.48)
	8	406	391 (0.96)	310 (0.76)	280 (0.69)	317 (0.78)	235 (0.58)	257 (0.63)	**163** (0.40)
	2	**154**	238 (1.55)	190 (1.23)	205 (1.33)	218 (1.42)	215 (1.40)	182 (1.18)	182 (1.18)
	4	731	713 (0.98)	635 (0.87)	602 (0.82)	527 (0.72)	409 (0.56)	**351** (0.48)	388 (0.53)
prim2	6	883	821 (0.93)	785 (0.89)	765 (0.87)	774 (0.88)	519 (0.59)	**479** (0.54)	487 (0.55)
	8	991	936 (0.94)	883 (0.89)	866 (0.87)	815 (0.82)	598 (0.60)	576 (0.58)	**535** (0.54)
	2	**21**	38 (1.81)	37 (1.76)	32 (1.52)	59 (2.81)	56 (2.67)	37 (1.76)	76 (3.62)
	4	295	298 (1.01)	287 (0.97)	256 (0.87)	243 (0.82)	157 (0.53)	**120** (0.41)	159 (0.54)
c7552	6	445	400 (0.90)	359 (0.81)	315 (0.71)	369 (0.83)	211 (0.47)	**164** (0.37)	208 (0.47)
	8	501	455 (0.91)	442 (0.88)	409 (0.82)	403 (0.80)	263 (0.52)	**200** (0.40)	243 (0.49)
	2	20	30 (1.50)	27 (1.35)	22 (1.10)	40 (2.00)	21 (1.05)	**19** (0.95)	22 (1.10)
	4	78	81 (1.04)	70 (0.90)	66 (0.85)	77 (0.99)	56 (0.72)	**53** (0.68)	64 (0.82)
c1355	6	96	93 (0.97)	85 (0.89)	81 (0.84)	86 (0.90)	**61** (0.64)	64 (0.67)	73 (0.76)
	8	107	104 (0.97)	89 (0.83)	83 (0.78)	102 (0.95)	71 (0.66)	**68** (0.64)	78 (0.73)
	2	**56**	72 (1.29)	64 (1.14)	66 (1.18)	72 (1.29)	**56** (1.00)	**56** (1.00)	69 (1.23)
	4	210	190 (0.90)	195 (0.93)	183 (0.87)	200 (0.95)	151 (0.72)	**84** (0.40)	141 (0.67)
c3540	6	260	250 (0.96)	227 (0.87)	230 (0.88)	255 (0.98)	175 (0.67)	**148** (0.57)	176 (0.68)
	8	284	276 (0.97)	261 (0.92)	246 (0.87)	281 (0.99)	191 (0.67)	**161** (0.57)	184 (0.65)
	2	**22**	44 (2.00)	46 (2.09)	43 (1.95)	50 (2.27)	36 (1.64)	27 (1.23)	39 (1.77)
	4	92	104 (1.13)	100 (1.09)	90 (0.98)	94 (1.02)	**62** (0.67)	63 (0.68)	73 (0.79)
c2670	6	134	115 (0.86)	111 (0.83)	104 (0.78)	124 (0.93)	79 (0.59)	**68** (0.51)	80 (0.60)
	8	155	146 (0.94)	126 (0.81)	111 (0.72)	134 (0.86)	87 (0.56)	**75** (0.48)	84 (0.54)
	2	48	61 (1.27)	51 (1.06)	49 (1.02)	65 (1.35)	51 (1.06)	**47** (0.98)	67 (1.40)
	4	160	159 (0.99)	138 (0.86)	127 (0.79)	125 (0.78)	111 (0.69)	**97** (0.61)	109 (0.68)
prim1	6	201	181 (0.90)	170 (0.85)	151 (0.75)	172 (0.86)	123 (0.61)	**114** (0.57)	126 (0.63)
	8	224	222 (0.99)	182 (0.81)	161 (0.72)	196 (0.88)	137 (0.61)	**126** (0.56)	131 (0.58)
	2	**16**	22 (1.38)	21 (1.31)	17 (1.06)	23 (1.44)	**16** (1.00)	**16** (1.00)	**16** (1.00)
	4	65	62 (0.95)	65 (1.00)	63 (0.97)	61 (0.94)	36 (0.55)	**32** (0.49)	43 (0.66)
s838	6	84	86 (1.02)	83 (0.99)	74 (0.88)	88 (1.05)	54 (0.64)	**45** (0.54)	58 (0.69)
	8	102	101 (0.99)	90 (0.88)	82 (0.80)	102 (1.00)	63 (0.62)	**53** (0.52)	64 (0.63)
	2	**20**	30 (1.50)	27 (1.35)	25 (1.25)	30 (1.50)	37 (1.85)	39 (1.95)	48 (2.40)
	4	341	269 (0.79)	235 (0.69)	198 (0.58)	212 (0.62)	118 (0.35)	**93** (0.27)	153 (0.45)
ind1	6	438	414 (0.95)	391 (0.89)	348 (0.79)	378 (0.86)	235 (0.54)	**200** (0.46)	219 (0.50)
	8	522	506 (0.97)	489 (0.94)	424 (0.81)	455 (0.87)	284 (0.54)	**249** (0.48)	278 (0.53)
	2	**60**	88 (1.47)	86 (1.43)	82 (1.37)	124 (2.07)	110 (1.83)	110 (1.83)	83 (1.38)
	4	262	260 (0.99)	206 (0.79)	191 (0.73)	282 (1.08)	194 (0.74)	194 (0.74)	**142** (0.54)
test03	6	339	335 (0.99)	307 (0.91)	285 (0.84)	388 (1.14)	283 (0.83)	283 (0.83)	**212** (0.63)
	8	371	378 (1.02)	353 (0.95)	331 (0.89)	416 (1.12)	313 (0.84)	313 (0.84)	**239** (0.64)
	2	**27**	**27** (1.00)	**27** (1.00)	**27** (1.00)	**27** (1.00)	**27** (1.00)	**27** (1.00)	**27** (1.00)
	4	132	130 (0.98)	115 (0.87)	76 (0.58)	117 (0.89)	69 (0.52)	**46** (0.35)	54 (0.41)
balu	6	171	173 (1.01)	164 (0.96)	134 (0.78)	154 (0.90)	131 (0.77)	**99** (0.58)	106 (0.62)
	8	200	198 (0.99)	182 (0.91)	174 (0.87)	183 (0.92)	135 (0.68)	**129** (0.65)	140 (0.70)
	2	**62**	73 (1.18)	70 (1.13)	65 (1.05)	91 (1.47)	72 (1.16)	74 (1.19)	75 (1.21)
	4	187	237 (1.27)	219 (1.17)	217 (1.16)	190 (1.02)	**104** (0.56)	107 (0.57)	137 (0.73)
test06	6	297	289 (0.97)	265 (0.89)	248 (0.84)	264 (0.89)	158 (0.53)	**144** (0.48)	153 (0.52)
	8	344	345 (1.00)	319 (0.93)	275 (0.80)	299 (0.87)	183 (0.53)	**157** (0.46)	170 (0.49)

Table 4: Execution time averages for benchmark circuits. The number of parts is denoted by k. 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*
	2	1.99	**1.76**	2.83	4.45	2.01	3.15	9.08	155.28
	4	**3.32**	5.40	17.12	60.99	6.62	13.43	67.52	415.06
struct	6	**4.88**	8.21	35.03	172.36	11.16	31.93	220.90	697.88
	8	**6.91**	9.42	58.17	409.30	13.63	53.71	383.23	950.30
	2	5.56	**4.08**	7.38	14.25	4.31	6.52	22.13	913.96
	4	**7.26**	8.94	42.51	168.66	11.55	37.03	172.54	3565.84
prim2	6	**9.41**	17.24	76.33	337.94	15.78	79.55	453.64	18406.74
	8	**13.18**	21.44	127.35	860.38	20.74	123.10	743.87	37304.18
	2	2.89	2.74	4.07	6.85	**2.34**	3.26	8.75	565.02
	4	**5.01**	8.05	22.38	84.94	6.22	17.55	82.04	3742.70
c7552	6	**6.77**	14.14	53.45	250.42	9.70	43.68	258.13	10692.46
	8	**9.66**	18.09	84.43	646.58	14.51	79.65	494.32	17971.80
	2	0.55	0.45	0.86	1.49	**0.41**	0.77	2.43	161.12
	4	**0.63**	1.02	3.31	13.53	0.90	2.92	13.55	565.60
c1355	6	**0.88**	1.60	6.99	43.89	1.22	6.07	33.73	1058.70
	8	**1.20**	2.08	13.39	94.95	1.77	10.42	70.24	1395.08
	2	1.26	**0.83**	1.74	2.94	0.94	1.40	2.93	584.76
	4	**1.53**	2.30	6.58	25.42	1.61	6.57	29.00	1507.28
c3540	6	**2.16**	3.02	15.60	79.89	2.48	13.61	76.69	2491.46
	8	**2.87**	3.89	29.63	188.76	3.38	25.80	174.31	3497.48
	2	0.90	0.67	1.11	2.05	**0.56**	1.25	3.15	240.12
	4	**1.17**	1.40	5.85	21.33	1.32	4.29	16.54	1046.72
c2670	6	**1.74**	2.65	11.80	68.54	2.25	9.64	58.37	1873.30
	8	**2.51**	3.48	25.20	177.20	3.03	17.35	122.16	3049.84
	2	0.91	0.90	1.35	2.59	**0.64**	1.01	3.42	205.12
	4	**1.31**	2.24	6.77	22.05	2.07	4.63	24.98	1235.46
prim1	6	**2.07**	3.62	14.13	70.12	3.38	10.64	84.67	2788.42
	8	**2.84**	5.05	22.35	180.91	4.41	20.63	151.04	2747.78
	2	0.34	0.34	0.52	1.03	**0.22**	0.37	1.10	87.58
	4	**0.54**	0.90	2.89	8.28	0.55	2.35	11.06	352.32
s838	6	0.79	1.51	5.66	30.29	**0.78**	5.48	33.16	604.66
	8	**1.06**	2.04	11.29	74.31	1.14	9.09	74.54	830.38
	2	3.57	3.35	4.57	7.87	**2.63**	4.08	11.02	526.50
	4	**4.84**	8.69	31.65	106.00	6.91	22.34	121.60	4932.18
ind1	6	**6.45**	13.22	51.71	269.21	9.71	48.60	287.05	11504.72
	8	**8.61**	15.29	76.97	604.14	11.21	65.75	479.60	19825.56
	2	2.22	2.00	3.44	4.66	2.81	7.37	**1.45**	864.48
	4	3.46	5.30	17.44	63.90	3.64	19.71	**0.27**	2947.66
test03	6	4.80	8.02	29.99	163.29	5.20	32.98	**0.38**	6693.38
	8	6.76	11.42	58.52	368.39	6.85	42.98	**0.45**	11201.86
	2	0.71	0.77	1.15	2.22	**0.64**	0.79	2.50	61.22
	4	**0.92**	1.49	5.16	19.76	1.23	5.34	26.18	587.58
balu	6	**1.28**	2.32	9.93	53.89	1.85	7.58	73.85	1200.98
	8	**1.72**	2.83	13.95	114.94	1.94	15.14	135.98	1584.56
	2	2.11	2.06	3.62	6.36	**1.79**	2.68	7.46	145.24
	4	**3.27**	4.36	14.45	59.78	3.73	10.73	63.89	3237.82
test06	6	**4.18**	6.49	30.22	156.73	5.57	25.32	167.44	8134.66
	8	**5.44**	8.25	50.57	375.36	7.10	44.10	390.10	13701.64

Table 5: Average (Avr) results for performances of algorithms. Averages were taken over all our test instances. k is the number of parts. Bold values are the best values in each row.

k	*FMS*	*PLM1*	*PLM2*	*PLM3*	*PFM1*	*PFM2*	*PFM3*	*SA*
A) *Avr % Improvement in Avr Cutsizes wrt FMS Algorithm*								
2	0.0	-13.1	-4.5	**5.1**	-45.0	-17.2	-6.5	-0.5
4	0.0	9.0	16.4	21.8	15.4	42.8	**50.5**	48.1
6	0.0	10.1	16.5	22.4	12.3	37.7	45.5	**46.1**
8	0.0	6.6	14.1	21.0	10.6	35.9	41.9	**46.5**
B) *Avr % Improvement in Minimum Cutsizes wrt FMS Algorithm*								
2	**0.0**	-41.7	-32.8	-25.1	-67.2	-37.3	-24.2	-51.6
4	0.0	-0.5	8.3	16.2	12.5	41.4	**49.5**	38.1
6	0.0	5.4	12.2	19.2	9.1	38.2	**45.7**	40.7
8	0.0	2.8	12.3	19.7	9.4	38.1	**43.3**	42.3
C) *Avr % Improvement in Maximum Cutsizes wrt FMS Algorithm*								
2	0.0	10.4	20.3	30.6	-11.4	6.6	16.7	**32.4**
4	0.0	13.8	18.5	23.1	15.1	40.0	48.0	**52.0**
6	0.0	11.8	17.1	24.0	13.5	36.0	42.9	**49.0**
8	0.0	9.7	16.1	23.9	13.7	34.7	41.3	**50.5**
D) *Stability: Avr % Ratio of Standard Deviation to Avr Cutsize*								
2	22.6	17.3	15.6	**14.8**	17.9	19.7	19.0	15.2
4	**6.6**	6.6	7.8	8.5	9.3	11.5	11.7	7.2
6	**4.8**	5.6	5.9	5.6	6.5	8.5	8.9	5.5
8	4.2	4.7	5.2	5.1	5.0	7.5	7.3	**4.1**
E) *Avr Ratio of Avr Running Times per pass to those of FMS Algorithm*								
2	**1.0**	1.4	2.5	4.7	**1.0**	1.9	5.6	-
4	**1.0**	1.6	5.8	22.6	1.2	4.3	24.3	-
6	**1.0**	1.7	9.1	51.7	1.3	6.9	53.3	-
8	**1.0**	1.7	11.8	95.2	1.3	8.7	83.8	-
F) *Avr Ratio of Avr Running Times to those of FMS Algorithm*								
2	1.0	0.9	1.5	2.6	**0.8**	1.4	3.3	219.7
4	**1.0**	1.5	5.2	18.9	1.3	4.4	19.0	769.5
6	**1.0**	1.8	7.4	38.5	1.4	6.6	38.7	1273.7
8	**1.0**	1.6	9.3	66.8	1.4	8.1	55.1	1464.9
G) *Avr of Avr Number of Passes*								
2	8.7	5.6	4.9	**4.5**	7.4	6.4	5.1	-
4	7.7	7.5	7.1	6.7	9.2	8.0	**6.0**	-
6	7.9	8.2	6.5	**5.7**	9.3	7.9	**5.7**	-
8	8.1	7.9	6.3	5.6	8.9	7.6	**5.0**	-

Table 6: Cutsize averages by PFM3 for different values of the scale factor S. G_{max} is the 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**
balu	2	48.0	51.8	48.2	52.4	43.2	**38.4**	45.8	43.8	43.8
	4	102.6	83.8	**66.0**	70.8	78.6	**66.0**	73.4	84.6	79.2
	6	163.0	144.8	137.0	129.6	**121.6**	127.6	134.8	128.2	130.4
	8	186.0	180.2	168.6	162.4	159.6	163.0	163.8	**158.4**	161.4

Table 7: Cutsize averages by PLM for different values of N_{in} and N where N is the total number of cell moves per pass, N_{in} is the number of move-and-lock phases per pass, n is the number of cells, and k is the 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
test03	2	98.0	113.6	114.4	121.2	100.0	103.4	113.0	95.6	105.6	**94.2**	97.6	95.6
	4	303.2	308.6	234.2	328.2	276.6	273.8	244.0	259.6	263.0	226.8	220.0	**219.8**
	6	379.8	347.6	332.0	389.0	359.4	332.6	311.4	315.6	364.4	317.0	**292.0**	294.2
	8	421.2	418.6	413.4	439.0	418.4	380.0	376.0	378.8	416.6	**355.0**	367.8	364.4