Chapter 10. Bipartitioning

Introduction

As detailed in Chapter 9, automatic-mapping tools for multi-FPGA systems are important to their success. A major part of this process is a fast partitioning algorithm that effectively minimizes inter-partition communication. In this chapter we present such a tool. Specifically, we investigate how best to quickly split logic into two parts. Along with the techniques discussed in Chapter 11, this forms a multi-way partitioning algorithm capable of optimized to an arbitrary, fixed multi-FPGA system topology.

Logic partitioning is also a critical issue for digital logic CAD in general. Effective algorithms for partitioning circuits enable us to apply divide-and-conquer techniques to simplify most of the steps in the mapping process. For example, standard cell designs can be broken up so that a placement tool need only consider a portion of the overall design at any one time, yielding higher quality results, in a shorter period of time. Also, large designs must be broken up into pieces small enough to fit into multiple devices.

For all of these tasks, the goal is to minimize the communication between partitions while ensuring that each partition is no larger than the capacity of the target device. While it is possible to solve the case of unbounded partition sizes exactly [Cheng88], the case of balanced partition sizes is NP-complete [Garey79]. As a result, numerous heuristic algorithms have been proposed.

In a 1988 survey of partitioning algorithms [Donath88] Donath stated “there is a disappointing lack of data comparing partitioning algorithms”, and “unfortunately, comparisons of the available algorithms have not kept pace with their development, so we cannot always judge the cost-effectiveness of the different methods”. This statement still holds true, with many approaches but few overall comparisons. This chapter addresses the bipartitioning problem by comparing many of the existing techniques, along with some new optimizations. It focuses primarily on those approaches that build on the Kernighan-Lin, Fiduccia-Mattheyses (KLFM) algorithm [Kernighan70, Fiduccia82].

One of the surprising results to emerge from this study is that by appropriately applying existing techniques an algorithm based upon KLFM can produce results better than the current state-of-the-art. In Table 1 we present the results of our algorithm (Optimized KLFM), along with results of three of the best current methods (Paraboli [Riess94], EIG1 [Hagen92], and Network Flow [Yang94]), on a set of standard benchmarks [MCNC93]. Note that the EIG1 algorithm is meant to be used for ratio-cut partitioning, not mincut partitioning as presented here.

Table 1. Quality comparison of partitioning methods. Values for KLFM and Optimized KLFM[1] are the best of ten trials. The EIG1 and Paraboli results are from [Riess94] (though EIG1 was proposed in [Hagen92]), and the Network Flow results are from [Yang94]. All tests require partition sizes to be between 45% and 55% of the total circuit sizes.

Mapping	Basic KLFM	**Optimized KLFM**	EIG1	Paraboli	Network Flow
s38584	243	**52**	76	55	47
s35932	136	**46**	105	62	49
s15850	105	**46**	215	91	67
s13207	105	**62**	241	91	74
s9234	65	**45**	227	74	70
Geom. Mean	118.8	**49.8**	156.5	73.1	60.3
Normalized	2.386	**1.000**	3.143	1.468	1.211

The results show that our algorithm produces significantly better solutions than the current state-of-the-art bipartitioning algorithms, with the nearest competitor producing results 21% worse than ours (thus, our algorithm is 17% better). Our algorithm is also fast, taking at most 7 minutes on the largest examples. Note that bipartitioning with replication has shown some promising results (all of the algorithms in the table do not use replication). [Kuznar94a, Kuznar94b] has reported results only 7-10% worse than ours. However, these results have no cap on the maximum partition size, while all other trials have a maximum partition size of 55% of the logic. In fact, some of Kuznar et al's runs have partitions of size 60% or larger. As will be demonstrated, allowing a partitioning algorithm to use a larger maximum partition size can greatly reduce the cutset size. Also, their work does not include primary inputs connected to both partitions as part of the cutset, while the cutsizes reported for the other approaches, including ours, do include such primary inputs in the cutset.

In the rest of this chapter we discuss the basic KLFM algorithm and compare numerous optimizations to the basic algorithm. This includes methods for clustering and unclustering circuits, initial partition creation, extensions to the standard KLFM inner loop, and the effects of increasing the maximum allowed partition size and the number of runs per trial.

---

1 Optimized KLFM includes recursive connectivity clustering, presweeping, per-run clustering on gate-level netlists, iterative unclustering, random initialization, and fixed 3rd-level gains. Each of these techniques is described later in this chapter.

Although the work described in this chapter is applicable to many situations, it has been biased by the fact that we are targeting multi-FPGA systems. One part of this is that the time it takes to perform the partitioning is important, and is thus a primary concern in this work. In tasks such as ASIC design, we can afford to allow the partitioner to run for hours or days, since it will take weeks to create the final implementation. In contrast, a multi-FPGA system is ready to use seconds after the mapping has been completed, and users demand the highest turnaround time possible. If the mapping software can complete a mapping in minutes instead of hours, the user can use the multi-FPGA system more interactively and thus more efficiently. Thus, there is significant interest in using an efficient partitioning method, such as KLFM partitioning, as opposed to more brute force approaches such as simulated annealing, which can take multiple hours to complete. Targeting our partitioning work towards multi-FPGA systems has several other impacts, which will be discussed later in this paper.

Methodology

In our work we have integrated numerous concepts from the bipartitioning literature, along with some novel techniques, to determine what features make sense to include in an overall system. We are primarily interested in Kernighan-Lin, Fiduccia-Mattheyses based algorithms, though we do include some of the spectral partitioning approaches as well. Note that there is one major omission from this study: the use of logic replication (i.e., the duplication of nodes to reduce the cutset). This is primarily because of uncertainty in how to limit the amount of replication allowed in the multi-FPGA partitioning problem. We leave this aspect to future work.

Table 2. Sizes of example circuits.

Mapping	s38584	s35932	s15850	s13207	s9234	s5378
Nodes (gates, latches, I/Os)	22451	19880	11071	9445	6098	3225

The best way to perform this comparison would be to try every combination of techniques on a fixed set of circuits, and determine the overall best algorithm. Unfortunately, we consider such a large number of techniques that the possible combinations reach into the thousands, even ignoring the ranges of numerical parameter settings relevant to some of these algorithms. Instead, we use our experience with these algorithms to try and choose the best possible set of techniques, and then try inserting into this mix each technique that was not chosen. In some cases, where it seemed likely that there would be some benefit of examining multiple techniques together and exploiting synergistic effects, we also tested those sets of techniques. In the comparisons that follow we always use all the features of the best mix of techniques found except where specifically stated otherwise.

The 6 largest circuits from the MCNC partitioning benchmark suite [MCNC93] are used as test cases for this work (Table 2). One of the largest, s38417, was not used because it was found to be corrupted at the storage site. While these circuits have the advantage of allowing us to compare with other existing algorithms, the examples are a bit small for today's partitioning tasks (the largest is less than 25,000 gates) and it is unclear how representative they are for bipartitioning. We hope that in the future a standard benchmark suite of real end-user circuits, with sizes ranging up to the hundreds of thousands of gates, will be available to the community.

```
Create initial partitioning;
While cutsize is reduced {
   While valid moves exist {
      Use bucket data structures to find unlocked node in each
         partition that most improves/least degrades cutsize when
         moved to other partition;
      Move whichever of the two nodes most improves/least degrades
         cutsize while not exceeding partition size bounds;
      Lock moved node;
      Update nets connected to moved nodes, and nodes connected to
         these nets;
   } endwhile;
   Backtrack to the point with minimum cutsize in move series just
      completed;
   Unlock all nodes;
} endwhile;
```

Figure 70. The Fiduccia-Mattheyses variant of the Kernighan-Lin algorithm.

Basic Kernighan-Lin, Fiduccia-Mattheyses Bipartitioning

One of the best known, and most widely extended, bipartitioning algorithms is that of Kernighan and Lin [Kernighan70], especially the variant developed by Fiduccia and Mattheyses [Fiduccia82]. Pseudo-code for the algorithm is given in Figure 70. It is an iterative-improvement algorithm, in that it begins with an initial partition and iteratively modifies it to improve the cutsize. The *cutsize* is the number of nets connected to nodes in both partitions, and is the value to be minimized. The algorithm moves a node at a time, moving the node that causes the greatest improvement, or the least degradation, in the cutsize. If we allowed the algorithm to move any arbitrary node, it could decide to move the node just moved in the previous iteration, returning to the previous state. Thus, the algorithm would be caught in an infinite loop, making no progress. To deal with this, we lock down a node after it is moved, and never move a locked node. The algorithm continues moving nodes until no unlocked node can be moved without violating the size constraints. It is only after the algorithm has exhausted all possible nodes that it checks whether it has improved the cutset. It looks back across all the intermediate states since the last check, finding the minimum cutsize. This allows it to climb out of local minima, since it is allowed to try out bad intermediate moves, hopefully finding a better later state. After it moves back to the best intermediate state, it unlocks all nodes and continues. Once the algorithm fails to find a better intermediate state between checks it finishes with the last chosen state.

One important feature of the algorithm is the bucket data structure used to find the best node to move. The data structure has an array of lists, where each list contains nodes in the same partition that cause the same change to the cutset when moved. Thus, all nodes in partition 1 that increase the cutset by 5 when moved would be in the same list. When a node is moved, all nets connected to it are updated. There are four situations to look for: 1) A net that was not in the cutset that now is. 2) A net that was in the cutset that now is not. 3) A net that was firmly in the cutset that is now removable from the cutset. 4) A net that was removable from the cutset that is now firmly in the cutset. A net is "firmly in the cutset" when it is connected to two nodes, or a locked node, in each partition. All other nets in the cutset are "removable from the cutset", since they are connected to only one node in one of the partitions, and that node is unlocked. Thus, the net can be removed from the cutset by moving that node. Each of these four situations means that moving a node connected to that net may have a different effect on the cutsize now than it would have had if it was moved in the previous step. All nodes connected to one of these four types of nets are examined and moved to a new list in the bucket data structure if necessary.

The basic KLFM algorithm can be extended in many ways. We can choose to partition before or after technology-mapping. We can cluster circuit nodes together before partitioning, both to speed up the algorithm's run-time, and to give some better local optimization properties to the KLFM's primarily global viewpoint. We also have a choice of initial partition creation methods, from completely random to more intelligent methods. The main search loop can be augmented with more complex cost metrics, possibly adding more lookahead to the choice of nodes to move. We can uncluster the circuit and reapply partitioning, using the previous cut as the initial partitioning of the subsequent runs. Finally, we can consider how these features are improved or degraded by larger or smaller maximum partition sizes, and by multiple runs. In this chapter, we will consider each of these issues in turn, examining not only how the different approaches to each step compare with one another, but also how they combine together to form a complete partitioning solution.

Clustering and Technology-Mapping

One of the most common optimizations to the KLFM algorithm is clustering, the grouping together of nodes in the circuit being partitioned. Nodes grouped together are removed from the circuit, and the clusters take their place. Nets that were connected to a grouped node are instead connected to the cluster containing that node. Clustering algorithms are applied to the partitioning problem both to boost performance, and also to improve quality. The performance gain is due to the fact that since many nodes are replaced by a single cluster, the circuit to be partitioned now has fewer nodes, and thus the problem is simpler. Note that the clustering time can be significant, so we usually cluster the circuit only once, and if several independent runs of the KLFM algorithm are performed we use the same clustering for all runs. The ways in which clustering improves quality are twofold. First, the KLFM algorithm is a global algorithm, optimizing for macroscopic properties of the circuit. It may overlook more local, microscopic concerns. An intelligent clustering algorithm will often focus on local information, grouping together a few nodes based on local properties. Thus, a smart clustering algorithm can perform good local optimization, complementing the global optimization properties of the KLFM algorithm. Second, it has been shown that the KLFM algorithm performs much better when the nodes in the circuit are connected to at least an average of 6 nets, while nodes in circuits are typically connected to between 2.8 and 3.5 nets [Goldberg83]. Clustering should in general increase the number of nets connected to each node, and thus improve the KLFM algorithm. Note that most algorithms (including the best KLFM version we found) will partition the clustered circuit, and then use this as an initial split for another run of partitioning, this time on the unclustered circuit. Several variations on this theme will be discussed in a later section.

The simplest clustering method is to randomly combine connected nodes. The idea here is not to add any local optimization to the KLFM algorithm, but instead to simply exploit KLFM's better results when the nodes in the circuit have greater connectivity. A maximum random matching of the circuit graph [Bui89] can be formed by randomly picking pairs of connected nodes to cluster, and then reclustering as necessary to form the maximum number of disjoint pairs. Unfortunately, this is complex and time-consuming, possibly requiring $O(n^3)$ time [Galil86]. We chose to test a simpler algorithm (referred to here as *random clustering*), that should generate similar results while being more efficient and easier to implement. Each node is examined in random order and clustered with one of its neighbors (note that a node connected to a neighbor by N nets is N times as likely to be clustered with that neighbor). A node that was previously the target of a clustering is not used as a source of a clustering, but an unclustered node can choose to join a grouping with an already clustered node. Note that with random clustering a separate clustering is always generated for each run of the KLFM algorithm.

Numerous more intelligent clustering algorithms exist. *K-L clustering* [Garbers90] (not to be confused with KL, the Kernighan-Lin algorithm) is a method that looks for multiple independent short paths between nodes, expecting that these nodes should be placed into the same partition. Otherwise, each of these paths will have a net in the cutset, degrading the partition quality. In its most general form, the algorithm requires that two nodes be connected by k independent paths (i.e., paths cannot share any nets), of lengths at most $l_1...l_k$ respectively, to be clustered together. Checking for K-L connectedness can be very time-consuming, especially for longer paths. The biggest problem is high fanout nets, which are quite common in digital circuits. Specifically, if we are looking for potential nodes to cluster, and the source node of the search is connected to a clock or reset line, most of the nodes in the system are potential candidates, and a huge number of paths need to be checked. However, since huge fanout nets are the most likely to be cut in any partitioning, we can accelerate the algorithm by ignoring all nets with fanout greater than some constant. Also, if $l_1 = 1$, then the potential cluster-mates are limited to the direct neighbors of a node (though transitive clustering is possible, with A and C clustered together with B because both A and C are K-L connected with node B, while A and C are not directly K-L connected). In our study of K-L clustering we ignored all nets with fanout greater than 10, and used $k = 2$, $l_1 = 1$, $l_2 = 3$. The values of maximum considered fanout and l_1 were chosen to give reasonable computation times. While [Garbers90] recommends $k = 3$, $l_1 = 1$, $l_2 = 3$, $l_3 = 3$, we have found that this yielded few clustering opportunities (this will be discussed later), and the parameters we chose gave the greatest clustering opportunities with reasonable run time. Using $l_2 = 4$ would increase the clustering opportunities, but would also greatly increase run time.

A much more efficient clustering algorithm, related to K-L clustering, has been proposed [Roy93] (referred to here as *bandwidth clustering*). In this method, each net e in the circuit provides a bandwidth of $1/(|e|-1)$ between all nodes connected to it, where $|e|$ is the number of nodes or clusters connected to that net. All pairs of nodes that have a total bandwidth between them of more than 1.0 are clustered. Thus, nodes must be directly connected by at least two 2-terminal nets to be clustered, or a larger number of higher fanout nets. This clustering is similar to K-L clustering with $k = 2$, $l_1 = 1$, $l_2 = 1$, though it requires greater connectivity if the connecting nets have more than two terminals. Transitive clustering is allowed, so if the bandwidth between A&C is zero, they may still be clustered together if A&B and B&C each have a bandwidth of greater than 1.0 between them. There is an additional phase (carried out after all passes of recursive clustering, discussed below) that attempts to balance cluster sizes.

A clustering algorithm similar to bandwidth clustering, but which does not put an absolute lower bound on the necessary amount of bandwidth between the nodes, and which also considers the fanout of the nodes involved, has also been tested. It is based upon work done by Schuler and Ulrich [Schuler72], with several modifications. We will refer to it as *connectivity clustering*. Like random clustering, each node is examined in a random order and clustered with one of its neighbors. If a node has already been clustered it will not be the source of a new clustering attempt, though more than two nodes can choose to cluster with the same node. Nodes are combined with the neighbor with which they have the greatest connectivity.

Connectivity is defined in Equation 2. $Bandwidth_{ij}$ is the total bandwidth between the nodes (as defined in bandwidth clustering), where each *n*-terminal net contributes 1/(*n*-1) bandwidth between each pair of nodes to which it is connected. In this method nodes are more likely to be clustered if they are connected by many nets (the $bandwidth_{ij}$ in the numerator), if the nodes are small (the $size_i$ & $size_j$ in the denominator), and if most of the nodes' bandwidth is only between those two nodes (the $fanout_i$ - $bandwidth_{ij}$ & $fanout_j$ - $bandwidth_{ij}$ terms in the denominator). While most of these goals seem intuitively correct for clustering, the reason for the size limits is to avoid large nodes (or subsequent large clusters in recursive clustering, defined below) attracting all neighbors into a single huge cluster. Allowing larger nodes to form huge clusters early in the clustering will adversely affect the circuit partitioning.

$$connectivity_{ij} = \frac{bandwidth_{ij}}{size_i * size_j * \left(fanout_i - bandwidth_{ij}\right) * \left(fanout_j - bandwidth_{ij}\right)}$$

Equation 2. Connectivity metric for connectivity clustering.

While all the clustering techniques described so far have been bottom-up, using local characteristics to determine which nodes should be clustered together, it is possible to perform top-down clustering as well. A method proposed by Yeh, Cheng, and Lin [Yeh92] (referred to here as *shortest-path clustering*) iteratively applies a partitioning method to the circuit until all pieces are small enough to be considered clusters. At each step it considers an individual group at a time, where a group contains all nodes that have always been on the same side of the cuts in all prior partitionings. The algorithm then iteratively chooses a random source and sink node, finds the shortest path between those nodes, and increases the flow on these edges by 0.1. The flow is a number used in computing net lengths, where the current net length is $exp(10*flow)$. Before each partitioning, all flows are set to zero. When the flow on a net reaches 1.0, the net is part of the cutset. Once there is no uncut path between the random pairs of nodes chosen in the current iteration, the algorithm is finished with the current partitioning. In this way, the algorithm proceeds by performing a large number of 2-terminal net routings on the circuit graph, with random source and sink for each route, until it exhausts the resources in the system. Note that the original algorithm limits the number of subpartitions of any one group. Since this is not an important issue for our purposes, it was not included in our implementation. Once the algorithm splits up a group into subpartitions, the sizes of the new groups are checked to determine if they should be further subdivided. For our purposes, the maximum allowable cluster size is equal to (total circuit size)/100, which is half the maximum partition size variation. There are several alterations that can be made to this algorithm to boost performance, details of which can be found in Chapter 11.

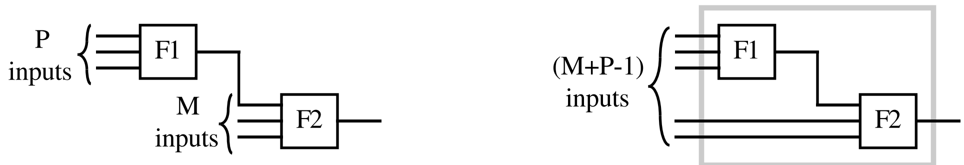

Figure 71. Example for the discussion of the size of logic functions. The *P*-input and *M*-input functions cascaded together (left) are fit into a *(M+P-1)* input LUT (right).

Before describing the last clustering method, it is necessary to discuss how to calculate the size of a logic node in the circuit being clustered. For our application (multi-FPGA systems), we are targeting FPGAs such as the Xilinx 3000 series [Xilinx94], where all logic is implemented by lookup-tables (LUTs). A LUT is a logic block that can implement any function of N variables, where N is typically 4 or 5. Since we will be partitioning circuits before technology-mapping (the reasons for this will be discussed later), we cannot simply count the number of LUTs used, since several of the gates in the circuit may be combined into a single LUT. An important aspect of a LUT-based implementation is that we can combine an M-input function with a P-input function that generates one of the M inputs into an (M+P-1)-input function (see Figure 71). The reason that it is an (M+P-1)-input function, and not an (M+P)-input function, is that the output of the P-input function no longer needs to be an input of the function since it is computed inside the LUT. A 1-input function (inverter or buffer) requires no extra inputs on a LUT. We can therefore say a logic node of P inputs uses up P-1 inputs of a LUT, and thus the size of a P-input function is (P-1), with a minimum size of 0. Any I/O nodes (i.e., external inputs and outputs) have a cost of 0. This is because if size keeps an I/O node out of a partition in which it has neighbors (i.e., nodes connected to the same net as the I/O node), a new I/O must be added to each partition to communicate the signal across the cut. Thus, moving an I/O node to a partition in which it has a neighbor never uses extra logic capacity. Although latches should also have a size of 0, since most FPGAs have more than sufficient latch resources, for simplicity we treat them identically to combinational logic nodes.

The last clustering technique we explored is not a complete clustering solution, but instead a preprocessor (called *presweeping*) that can be used before any other clustering approach. The idea is that there are some nodes that should always be in the same partition. Specifically, one of these nodes has a size of zero, and that node can always be moved to the other node's partition without increasing the cut size. The most obvious case is an I/O node from the original circuit which is connected to some other node N. This I/O node will have a size of zero, will be connected to one net, and moving the I/O node to node N's partition can only decrease the cut size (the cut size may not actually decrease, since another node connected to the net between N and the I/O node may still be in that other partition). Another situation is a node R, which is connected to exactly two nets, and one of these two nets is a 2-terminal net going to node *S*. Again, node *R* will have a size of zero, and can be moved to *S*'s partition without increasing the cutsize. The presweeping algorithm goes through the circuit looking for such situations, and clusters together the involved nodes (*R* with *S*, or *N* with the I/O node). Note that presweeping can be very beneficial to some clustering algorithms, such as K-L and Bandwidth, since such algorithms may be unable to cluster together the pairs found by presweeping. For example, an I/O node connected to only one net will never be clustered by the K-L clustering algorithm. Since the presweeping clustering should never hurt a partitioning (except due to random variation), presweeping will always be performed in this study unless otherwise stated.

Table 3. Quality comparison of clustering methods. Values are minimum cutsize for ten runs using the specified clustering algorithm, plus the best KLFM partitioning and unclustering techniques. Source mappings are not technology-mapped. The "No Presweep" column is connectivity clustering applied without first presweeping. All other columns include presweeping.

Mapping	Random	K-L	Bandwidth	Connectivity	Shortest-Path	No Presweep
s38584	177	88	112	57	50	59
s35932	73	86	277	47	45	70
s15850	70	90	124	60	59	65
s13207	109	94	87	73	72	79
s9234	63	79	56	52	51	65
s5378	84	78	88	68	67	66
Geom. Mean	89.7	85.6	108.7	58.8	56.5	67.1

Results for the various clustering algorithms are presented in Table 3 and Table 4. The shortest-path clustering algorithm generates the best results, with connectivity clustering performing only about 4% worse. In terms of performance, the shortest-path algorithm takes more than twice as long as the connectivity clustering algorithm. This is because clustering with the shortest-path algorithm takes more than 15 times as long as the connectivity approach. Shortest-path clustering would thus be even worse compared to connectivity clustering if the partitioner does not share clustering between runs. As we will show later, it is sometimes a good idea not to share clusterings. Because of the significant increase in run time because of shortest-path clustering, with only a small increase in quality, we use the connectivity algorithm for all of our other comparisons. We can also see that presweeping is a good idea, since connectivity clustering without presweeping does about 14% worse in terms of cutsize, while taking about 13% longer.

Table 4. Performance comparison of clustering methods. Values are total CPU seconds on a SPARC-IPX for ten runs using the specified algorithm, plus the best KLFM partitioning and unclustering techniques. The "No Presweep" column is connectivity clustering applied without first presweeping. All other columns include presweeping.

Mapping	Random	K-L	Bandwidth	Connectivity	Shortest-Path	No Presweep
s38584	2157	2041	2631	1981	4715	2183
s35932	3014	1247	2123	2100	3252	2114
s15850	780	500	871	643	1354	713
s13207	648	428	629	549	1279	696
s9234	326	266	460	333	669	416
s5378	120	147	223	181	447	189
Geom. Mean	710.4	526.5	824.4	667.6	1412.5	751.5

One surprising result is that K-L clustering only does slightly better than random clustering, and Bandwidth clustering actually does considerably worse. The reason for this is that these clustering algorithms seem to require technology-mapping, and the comparisons in the tables are for non-technology-mapped circuits. Technology-mapping for Xilinx FPGAs is the process of grouping together logic nodes to best fill a CLB (an element capable of implementing any 5-input function, or two 4-input functions). Thus, it combines several basic gates into a single CLB. The reason that K-L and Bandwidth clustering perform poorly on non-technology-mapped (gate-level) circuits is that there are very few clustering opportunities for these algorithms. Imagine a sum-of-products implementation of a circuit. In general, any specific AND gate in the circuit will be connected to two or three input signals and some OR gates. Any AND gates connected to several of the same inputs will in general be replaced by a single AND gate. The OR gates are connected to other AND gates, but will almost never be connected to the same AND gate twice. The one possibility, an OR gate connected to an AND gate's output as well as producing one of that AND gate's inputs, is a combinational cycle, and usually not allowed. Thus, there will be almost no possibility of finding clusters with Bandwidth clustering, and few K-L clustering opportunities. While many gate-level circuits will not be simple sum-of-products circuits, we have found that there are still very few clustering opportunities for the K-L and Bandwidth algorithms.

Unfortunately, technology-mapping before partitioning is an extremely poor idea. In Table 5, columns 2 through 4 shows results for applying the various clustering algorithms to the Xilinx 3000 technology-mapped versions of the circuits (note that only four of the examples are used, because the other examples were small enough that the size of a single CLB was larger than the allowed partition size variation). Column 5 ("No Tech Map") has the results for connectivity clustering on gate-level (non-technology-mapped) circuits. The results show that technology-mapping before partitioning almost doubles the cutsize. The K-L and Bandwidth clustering algorithms do perform almost as well as the connectivity clustering for these circuits, but we are much better off simply partitioning the gate-level circuits. This has an added benefit of speeding up technology-mapping as well, since we can technology-map each of the partitions in parallel. Note that we may increase the logic size by partitioning before technology-mapping, because there are fewer groupings for the technology-mapper to consider. However, in many technologies (especially multi-FPGA systems) the amount of logic that can fit on the chip is constrained much more by the number of I/O pins than on the logic size, and thus decreasing the cutsize by a factor of two is worth a small increase in logic size. This increase in logic size is likely to be small since the gates that technology-mapping will group into a CLB share signals, and are thus likely to be placed into the same partition.

Table 5. Quality comparison of clustering methods on technology-mapped circuits. Values are minimum cutsize for ten runs using the specified algorithm. The values in the column marked "Unclusterable" are the results of applying Connectivity clustering to technology-mapped files, but allowing the algorithm to uncluster the groupings formed by the technology-mapping. Note that only the four largest circuits are used, because technology-mapping for the others causes clusters to exceed allowed partition size variation.

Mapping	K-L	Bandwidth	Connectivity	No Tech Map	Unclusterable
s38584	169	159	120	57	60
s35932	155	157	143	47	53
s15850	86	90	87	60	60
s13207	118	119	116	73	72
Geom. Mean	127.7	127.9	114.7	58.5	60.9

It is fairly surprising that technology-mapping has such a negative effect on partitioning. There are two possible explanations: 1) technology-mapping produces circuits that are somehow hard for the KLFM algorithm to partition or 2) technology-mapping creates circuits with inherently much higher minimum cutsizes. There is evidence that the second reason is the underlying cause, that technology-mapped circuits simply cannot be partitioned as well as gate-level circuits, and that it is not simply due to a poor partitioning algorithm. To demonstrate this, we use the fact that the technology-mapped circuits for the Xilinx 3000 series contain information on what gates are clustered together to form a CLB. This lets us to consider the technology-mapping not as a permanent restructuring of the circuit, but instead simply as another clustering preprocessor. We allowed our algorithm to partition the circuit with the technology-mapped files, with connectivity clustering applied on top, then uncluster to basic gates and partition again. The results are shown in the final column of Table 5. Although the results for this technique are slightly worse than pure connectivity clustering, they are is still much better than the permanently technology-mapped versions. The small example circuit (Figure 72) demonstrates the problems technology-mapping can cause. There is a balanced partitioning of the circuit with a cutsize of 2, as shown in gray at left. However, after technology-mapping (CLBs are shown by gray loops), the only balanced partitioning puts the smaller CLBs in one partition, the larger CLB on the other. This split has a cutsize of 5.

The effects of technology mapping on cutsize have been examined previously by Weinmann [Weinmann94], who determined that technology-mapping before partitioning is actually a good idea, primarily for performance reasons. However, in his study he used only a basic implementation of Kernighan-Lin (apparently not even the Fiduccia-Mattheyses optimizations were applied), thus generating cutsizes significantly larger than what our algorithm produces, with much slower performance. Thus, the benefits of any form of clustering would help the algorithm, making the clustering provided by technology-mapping competitive. However, even these results report a 6% improvement in arithmetic mean cutsize for partitioning before technology-mapping, and the difference in geometric mean is actually 19% [2].

Figure 72. Example of the impact of technology-mapping on partitioning quality. The circuit s27 is shown (clock, reset lines, and I/O pins are omitted). At left is a balanced partition of the unmapped logic, which has a cutsize of 2. Gray loops at right indicate logic grouped together during technology-mapping. The only balanced partitioning has the largest group in one partition, the other two in the other partition, yielding a cutsize of 5.

---

[2] Throughout this chapter we use geometric instead of arithmetic means because we believe improvements to partitioning algorithms will result in some percentage decrease in each cutsize, not a decrease of some constant number of nets in all examples. That is, it is likely that an improved algorithm would reduce cutsizes for all circuits by 10%, and would not reduce cutsizes by 10 nets in both large and small examples. Thus, the geometric mean is more appropriate.

Unclustering

When we use clustering to improve partitioning, we will usually partition the circuit, uncluster it, and partition again. There are several ways to uncluster. Most obviously, we can either choose not to uncluster at all (*no unclustering*), or we can completely remove all clustering in one step (*complete unclustering*). However, there are better alternatives. The important observation is that while clustering we can build a hierarchy of clusters by recursively applying a clustering method, and then uncluster it in a way that exploits this hierarchy. In *recursive clustering*, after the circuit is initially clustered we reapply the clustering algorithm again upon the already clustered circuit. Clusters are never allowed to grow larger than half the allowed partition size variation. Recursive clustering continues until no more clusters can be formed. While we are clustering we remember what clusters are formed at each step, with clusters formed in the *ith* pass forming the *ith* level of a clustering hierarchy.

There are two ways to take advantage of the clustering hierarchy formed during recursive clustering. The most obvious method is that after partitioning completes (that is, when a complete pass of moving nodes fails to find any state better than the results of the previous pass) we remove the highest level of the clustering hierarchy, leaving all clusterings at the lower levels alone, and continue partitioning. That is, subclusters of clusters at the highest level, as well as those clusters that were not reclustered in the highest level, will remain clustered for the next pass. This process repeats until all levels of the clustering have been removed (note that clustering performed by presweeping is never removed, since there is nothing to be gained by doing so). In this way, the algorithm performs very coarse-grain optimization during early passes, very fine grain optimization during late passes, as well as medium-grain optimization during the middle passes. This algorithm, which we will refer to here as *iterative unclustering*, is based on work by Cong and Smith [Cong93].

An alternative to iterative unclustering is *edge unclustering*. This technique is based on the observation that at any given point in the partitioning there is likely to be some fine-grained, localized optimization, and some coarse-grained, global optimization that should be done. Specifically, those nodes that are very close to the current cut should be very carefully optimized, while nodes far from the cut need much less detailed examination. The edge unclustering algorithm is similar to iterative unclustering in that it keeps unclustering the highest levels of clustering remaining in between runs of the KLFM partitioning algorithm. However, instead of removing all clusters at a given level, it only removes clusters that are adjacent to the cut (i.e., those clusters connected to edges that are in the cutset). In this way, we will end up eventually unclustering all clusters next to the cut, while other clusters may remain together. When there are no more clusters left adjacent to the cut, we completely uncluster the circuit and partition one final time with KLFM.

Table 6. Quality comparison of unclustering methods. Values are minimum cutsize for ten runs using the specified algorithm. Source mappings are not technology-mapped, and are clustered by presweeping and connectivity clustering.

Mapping	Single-level Clustering		Recursive Clustering			
	No Uncluster	Complete Uncluster	No Uncluster	Complete Uncluster	Iterative Uncluster	Edge Uncluster
s38584	95	77	167	88	57	56
s35932	157	156	90	75	47	46
s15850	77	67	123	84	60	62
s13207	101	79	119	89	73	72
s9234	68	61	105	54	52	58
s5378	79	68	125	70	68	68
Geom. Mean	92.4	80.1	119.3	75.6	58.8	59.7

Table 7. Performance comparison of unclustering methods. Values are run times on a SPARC-IPX for ten runs using the specified algorithm. Source mappings are not technology-mapped, and are clustered by presweeping and connectivity clustering.

Mapping	Single Clustering		Recursive Clustering			
	No Uncluster	Complete Uncluster	No Uncluster	Complete Uncluster	Iterative Uncluster	Edge Uncluster
s38584	1220	1709	1104	1784	1981	2023
s35932	1224	1664	1359	1798	2100	2127
s15850	380	491	301	485	643	646
s13207	375	525	282	429	549	572
s9234	219	283	145	262	333	335
s5378	104	144	82	132	181	162
Geom. Mean	411.4	557	338.9	533.6	667.6	664.8

As the results in Table 6 and Table 7 show, using recursive clustering and a hierarchical unclustering method (iterative or edge unclustering) has a significant advantage. The methods that do not uncluster are significantly worse than all other approaches, by up to more than a factor of two. Using only a single clustering pass plus complete unclustering yields a cutsize 36% larger than the best unclustering (iterative), and even complete unclustering of a recursively clustered mapping yields a 29% larger cutsize. The difference between the two hierarchical unclustering methods is only 1.5%, with three mappings having smaller cutsizes with edge unclustering, and two mappings having smaller cutsizes with iterative unclustering. Thus, it appears that the difference between the two approaches is slight enough to be well within the margins of error of this survey, with no conclusive winner. In this survey, we use iterative unclustering except where explicitly stated otherwise.

Initial Partition Creation

KLFM is an iterative-improvement algorithm that gives no guidance on how to construct the initial partitioning that is to be improved. As one might expect, there are many ways to construct this initial partitioning, and the method chosen has an impact on the results.

The simplest method for generating an initial partition is to just randomly create one (*random initialization*) by randomly ordering the clusters in the circuit (initial partition creation takes place after clustering), and then finding the point in this ordering that best balances the total cluster sizes before and after this point. All nodes before this point are in one partition, and all nodes after this point are in the other partition.

Table 8. Quality comparison of initial partition creation methods. Values are minimum cutsize for ten runs using the specified algorithm.

Mapping	Random	Seeded	Breadth-first	Depth-first
s38584	57	57	57	56
s35932	47	47	47	47
s15850	60	60	60	60
s13207	73	75	80	74
s9234	52	68	52	52
s5378	68	79	80	78
Geom. Mean	58.8	63.4	61.4	60.2

An alternative to this is *seeded initialization*, which is based on work by Wei and Cheng [Wei89]. The idea is to allow the KLFM algorithm to do all the work of finding the initial partitioning. It randomly chooses one cluster to put into one partition, and all other clusters are placed into the other partition. The standard KLFM algorithm is then run with the following alterations: 1) partitions are allowed to be outside the required size bounds, though clusters can not be moved to a partition that is too large, and 2) at the end of the pass, it accepts any partition within size bounds instead of a partition outside of the size bounds. Thus, the KLFM algorithm should move clusters related to the initial "seed" cluster over to the small partition, thus making all nodes that end up in the initially 1-cluster partition much more related to one-another than a randomly generated partitioning.

Table 9. Performance comparison of initial partition creation methods. Values are total CPU seconds on a SPARC-IPX for ten runs using the specified algorithm.

Mapping	Random	Seeded	Breadth-first	Depth-first
s38584	1981	1876	1902	2033
s35932	2100	2053	2090	2071
s15850	643	604	613	584
s13207	549	531	561	533
s9234	333	302	319	325
s5378	181	186	177	173
Geom. Mean	667.6	641.0	652.5	647.5

We can also generate an initial partitioning that has one tightly connected partition by *breadth-first initialization*. This algorithm again starts with a single node in one of the partitions, but then performs a breadth-first search from the initial node, inserting all nodes found into the seed node's partition. Once the seed partition grows to contain as close to half the overall circuit size as possible the rest of the nodes are placed into the other partition. To avoid searching huge fanout nets such as clocks and reset lines, which would create a very unrelated partition, nets connected to more that 10 clusters are not searched. *Depth-first initialization* can be defined similarly, but should produce much less related partitions.

Results for these initial partition construction techniques are shown in Table 8 and Table 9. The data shows that random is actually the best initialization technique, followed by depth-first search. The "more intelligent" approaches of seeded and breadth-first do 7% and 4% worse than random, respectively, and the differences occur only for the three smaller mappings. There are three reasons for this. First, recursive clustering and iterative unclustering seem to be able to handle the larger circuits well, regardless of how the circuit is initialized. With larger circuits there are more levels of hierarchy and the algorithms consistently get the same results. For smaller mappings there are fewer levels and much greater variance in results. Since there are thus many potential cuts that might be found when partitioning these smaller circuits, getting the greatest variance in the starting point will allow greater variety in results, and better values will be found (as will worse, but we only accept the best value of ten runs). Thus, the more random starting points perform better (random and depth-first initialization). Also, the more random the initial partitioning, the easier it is for the partitioner to move away from the initial partitioning. Thus, the partitioner is not trapped in a potentially poor partitioning, and can generate better results. Some of these effects can be seen in Figure 73 , which contains the distribution of results for ten runs of both random (black bars) and seeded (gray bars) initialization. As can be seen, there is little variation in the larger circuits. There is greater variation for smaller circuits, and for the random algorithm in general. Also, the random algorithm seems to be finding very different results from the seeded initialization, since few of the same cuts are found by the two algorithms for smaller circuits.

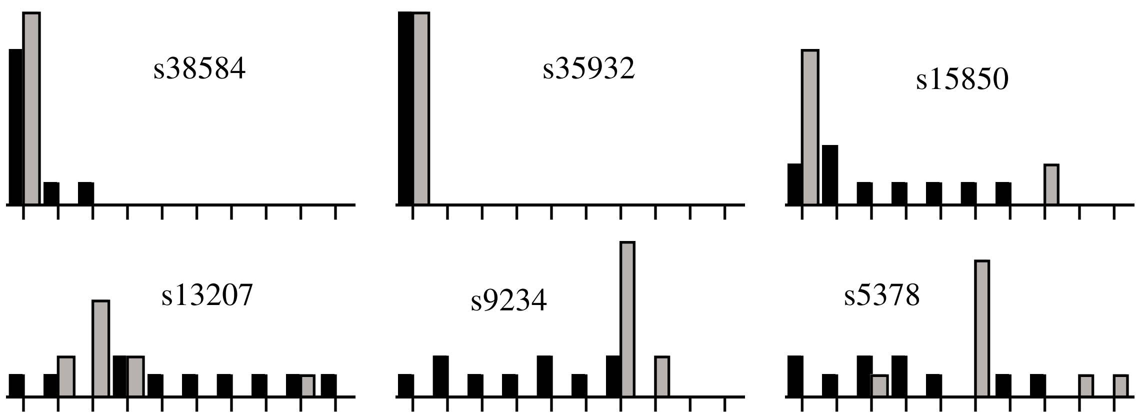

Figure 73. Distribution of results from partitioning with random (black bars) and seeded (gray bars) initialization. The *ith* bar from the left represents the *ith* best cutsize found by either algorithm, and the height indicates how many different runs of the algorithm (out of ten) achieved that result. There is little variation in the larger circuits. There is more variation in smaller circuits, and more variation in random initialization results in general.

While the previous discussion of initial partition generation has focused on simple algorithms, we can in fact use more complex, complete partitioning algorithms to find initial partitions. Specifically, there exists a large amount of work on "spectral" partitioning methods (as well as others) that constructs a partitioning from scratch. We will consider here the IG-Match [Cong92], EIG1 and EIG-IG [Hagen92] spectral partitioning algorithms. One important note is that these algorithms are designed to optimize for the ratio-cut objective [Wei89], which does not necessarily generate balanced partitions. However, we obtained the programs from the authors and altered them to generate only partitions with sizes between 49% and 51% of the complete circuit size, the same allowed partition size variation used throughout this chapter. These algorithms were applied to clustered circuits to generate initial partitionings. These initial partitionings were then used by our KLFM partitioning algorithm.

Table 10. Quality comparison of spectral initial partition creation methods. IG-Match [Cong92], EIG1 and EIG-IG [Hagen92] are spectral partitioning algorithms, used here to generate initial partitions. Entries labeled "n/a" are situations where the algorithm failed to find a partitioning within the required partition size bounds. Some of the spectral algorithms may move several clusters from one side of the cut to the other at once, missing the required size bounds (required only for our purposes, not for the ratio-cut metric for which they were designed). "All Spectral" is the best results from all three spectral algorithms.

Mapping	Random	EIG1	EIG1-IG	IG-Match	All Spectral
s38584	57	57	57	57	57
s35932	47	47	47	47	47
s15850	60	60	96	96	60
s13207	73	111	82	82	82
s9234	52	54	54	n/a	54
s5378	68	78	78	n/a	78
Geom. Mean	58.8	65.0	66.8	n/a	61.8

Table 11. Performance comparison of spectral initial partition creation methods. Values are total CPU seconds on a SPARC-IPX for the clustering, initialization, and partitioning algorithms combined. Values marked with "*" do not include the time for the failed IG-Match runs. "All Spectral" is the combined times for all three spectral partitionings.

Mapping	Random	EIG1	EIG1-IG	IG-Match	All Spectral
s38584	1981	336	445	1207	1988
s35932	2100	444	463	540	1447
s15850	643	89	102	206	397
s13207	549	79	95	152	326
s9234	333	42	56	n/a	98*
s5378	181	26	39	n/a	65*
Geom. Mean	667.6	102.3	127.8	n/a	365.2*

As the results show (Table 10 and Table 11), the algorithms (when taken as a group, under "All Spectral") produce fairly good results, but are still 5% worse than random initialization. They do have the advantage of faster run times (including the time to perform spectral initialization on the clustered circuits), since they do not require, and cannot use, multiple partitioning runs. Also, as we will show in the next section the spectral approaches are competitive when our KLFM algorithm is restricted to the same runtimes as the spectral algorithm. However, the extra complexity of the spectral algorithm, along with their inability to take advantage of multiple runs, make us conclude that spectral initialization is not worthwhile.

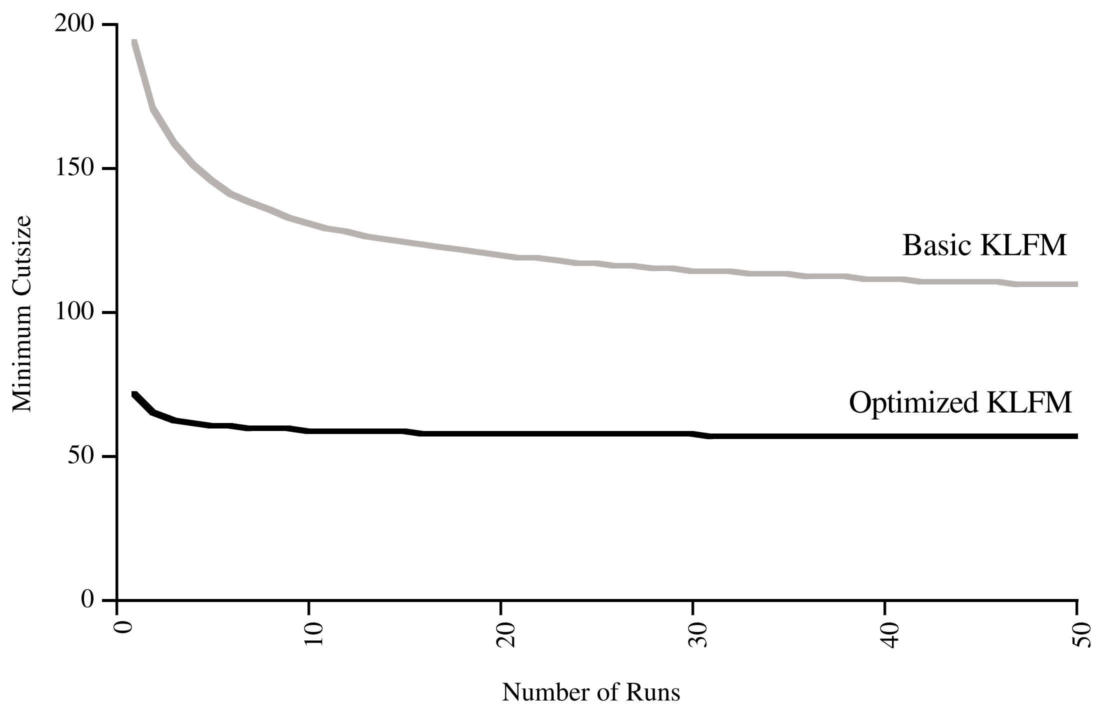

Figure 74. Graphs of cutsizes for different numbers of runs of both basic KLFM, and our optimized version of KLFM. Values shown are the geometric means of the results for all 6 test circuits.

Multiple Runs

While all of our tests have involved ten separate runs of the algorithm under consideration, and we retain the best result of these ten runs, we can consider using more or less runs per test. Basic KLFM is notoriously variable from run to run, and using multiple runs (up to even a hundred or more) is essential for achieving good results. To test how our algorithm responded to multiple runs, we ran one hundred runs of our best algorithm. We then used these datasets to determine the expected best cutsize found by an arbitrary number of runs of the algorithm. The results are shown in Figure 74, as well as similar results for the basic KLFM algorithm. As can be seen, not only does our optimized algorithm generate better results than the base KLFM algorithm, but it also has much less variability than the original algorithm, thus requiring fewer runs to be performed in general. Multiple runs are still valuable, since running the algorithm twice produces results 10% better on average than only a single run, and ten runs produces results 18% better than a single run. However, there are significantly diminished returns from further runs. Twenty runs produce results only 2% better than ten runs, and the best values found from all one hundred runs are only 5% better than those produced from ten runs. It is unclear exactly how many runs should be used in general, since for some situations a 2% improvement in cutsize is critical, while for others it is performance that is the primary concern. We have chosen to use ten runs for all of the tests presented in this chapter unless stated otherwise.

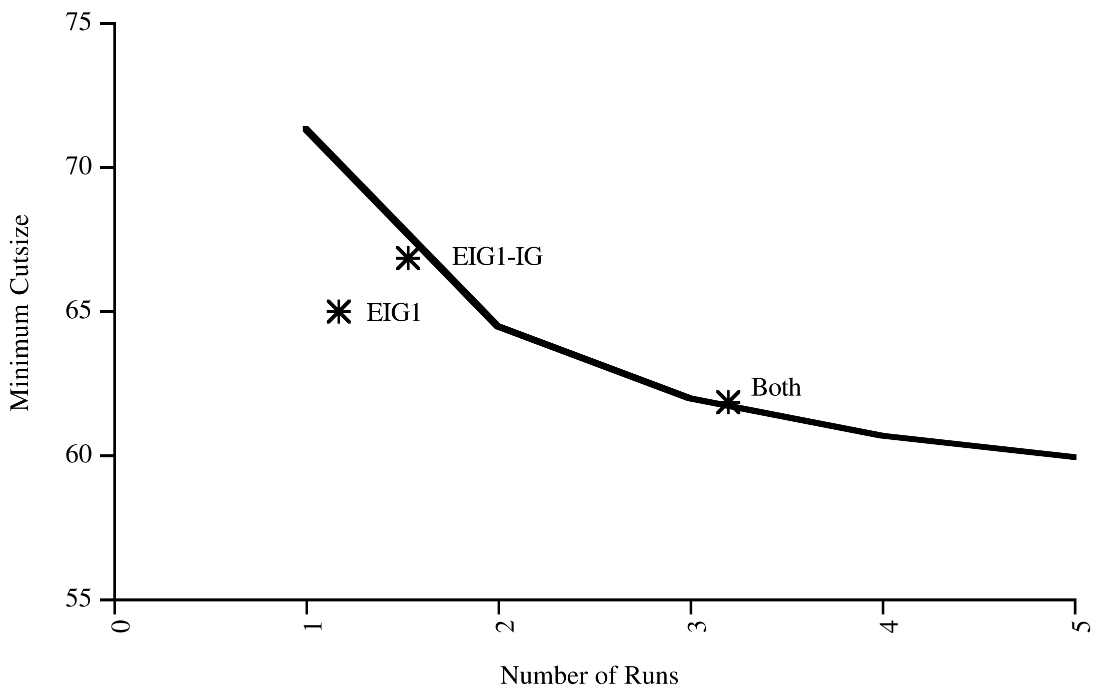

Figure 75. Graphs of cutsizes for different numbers of runs of our optimized version of KLFM versus the spectral initialization approaches. The spectral approaches are placed horizontally so that they line up to the equivalent amount of time spent on the optimized KLFM algorithm. Values shown are the geometric means of the results for all 6 test circuits.

Figure 75 shows details of the optimized KLFM runs from Figure 74, as well as the results from the spectral initialization approaches discussed earlier. The spectral results are placed horizontally so that they line up to the equivalent amount of time spent on the optimized KLFM algorithm. That is, in the time it takes to run both EIG1 and EIG1-IG initialized runs of our optimized KLFM algorithm approximately 3.2 runs of the optimized KLFM algorithm with random initializations could be performed. Note that the time to perform clustering is factored into the time to perform the first run of the optimized KLFM algorithm. As can be seen, the spectral approaches are somewhat more time efficient than our optimized algorithm, and thus might be useful in extremely time critical situations. However, multiple runs of our optimized algorithm can be run in parallel, performing better than the spectral approaches, and with slightly more time sequential runs of our optimized algorithm on a single processor produce better quality than the spectral approaches. Because of this, and also because the spectral approaches are much more complex than the optimized algorithm (since the spectral approaches perform spectral initialization, a complex process, and then run our entire optimized algorithm as well), we will use random initialization for our optimized algorithm.

Higher-Level Gains

The basic KLFM algorithm evaluates node moves purely on how much the move immediately affects the cutsize. However, there are often several possible moves that have the same effect on the cutsize, but these moves may have very different ramifications for later moves. Take for example the circuit in Figure 76 left. If we move either **B** or **E** to the other partition, the cutsize remains the same. However, by choosing to move **B**, we can reduce the cutsize by one by then moving **A** to the other partition. If we move **E**, it will take two further moves (**C** and **D**) to remove the newly cut three-terminal net from the cutset, and this would still keep the cutsize at 2 because of the edge from **C** to the rest of the logic.

To deal with this problem, and give the KLFM algorithm some lookahead ability, Krishnamurthy proposed *higher-level gains* [Krishnamurthy84]. If a net has n unlocked nodes in a partition, and no locked nodes in that partition, it contributes an *nth*-level gain of 1 to moving a node from that partition, and an *(n+1)th*-level gain of -1 to moving a node to that partition. The first-level gains are identical to the standard KLFM gains, with a net currently uncut giving a first-level gain of -1 to its nodes, and a net that can be uncut by moving a node **A** gives a first-level gain of 1 to node **A**. The idea behind this formulation is that an *nth*-level gain of 1 indicates that by moving N nodes, including the node under consideration, we can remove a net from the cutset. An *(n+1)th*-level gain of -1 means that by moving this node, we can no longer remove this net by moving the n nodes connected to the net in the other partition. Moves are compared based on the lowest-order gain in which they differ. So a node with gains (-1, 1, 0) (1st-level gain of -1, 2nd-level of 1, 3rd-level of 0) would be better to move than a node of (-1, 0, 2), but worse to move than a node of (0, 0, 0). To illustrate the gain computation better, we give the examples in Figure 76 right. Net **123** has one node in the left partition, giving a 1st-level gain of 1 for moving a node out of this partition, and a 2nd-level gain of -1 for moving a node to this partition. It has two nodes in the right partition, giving a 2nd-level gain of 1 for moving a node from this partition, and a 3rd-level gain of -1 for moving a node to this partition. Thus, node **1** has a gain vector of (1,0,-1), and nodes **2** and **3** have gains of (0,0,0), since the 2nd-level gains of 1 & -1 cancel each other. This makes sense, because after moving either node **2** or **3** you have almost the same situation for net **123** as the current state. Note that if node **3** were locked, node **2** would have a gain vector of (0,-1,0), and node **1** would have a gain vector of (1,0,0), since there is no longer any contribution to the gain vector of net **123** from the state of the right partition. For net **45** there is a 2nd-order gain of 1 for moving nodes out of the left partition, and a 1st-order gain of -1 for moving nodes into the right partition, giving nodes **4** and **5** a gain vector of (-1,1,0). If node **4** was locked, then node 5 would have a gain vector of (-1,0,0), since there is no longer any contribution to the gain vector of net **45** from the state of the left partition. Net **678** is similar to **45**, except that it has a 3rd-order, not a 2nd-order, gain of 1. So, we can rank the nodes (from best to move to worst) as **1**, **23**, **45**, **678**, where nodes grouped together have the same gains. If we do move **1** first, **1** would now be locked into the other partition, and nodes **2** and **3** would have a 1st-level gain of -1, and no other gains. Thus, they would become the worst nodes to move, and node **4** or **5** would be the next candidate.

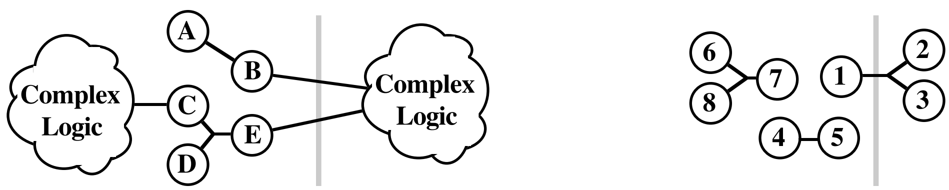

Figure 76. Examples for the higher-level gains discussion.

Note that the definition of *nth*-level gains given above is slightly different than Krishnamurthy's. Specifically, in Krishnamurthy's definition the rule that gives an *nth*-level gain to a net with *n* unlocked nodes in a partition is restricted to nets that are currently in the cutset. Thus, nets **678** and **45** would both have gains (-1, 0, 0). However, as we have seen, allowing nth-level gains for nets not in the cutset allows us to see that moving a node on **45** is better than moving a node on **678**, since it is easier to then remove **45** from the cutset than it is **678**. Also, this definition handles 1-terminal nets naturally, while Krishnamurthy requires no 1-terminal nets to be present in the circuit. A 1-terminal net with our definitions would have a 1st-level gain of 1 for having only one node in the starting partition, but a 1st-level gain of -1 because there are no nodes in the other partition, yielding an overall 1st-level gain of 0. Note that 1-terminal nets are common in clustered circuits, occurring when all nodes connected to a net are clustered together.

Table 12. Quality comparison of higher-level gains. Numbers in column headings are the highest higher-level gains considered. Note that a fixed gain-level of 1 is identical to KLFM without higher-level gains. Values are minimum cutsize for ten runs using the specified algorithm.

Mapping	Dynamic	Fixed				
		1	2	3	4	20
s38584	57	57	57	57	57	57
s35932	49	47	49	47	47	47
s15850	60	64	62	60	60	60
s13207	75	77	77	73	73	73
s9234	52	56	52	52	52	52
s5378	66	71	70	68	68	68
Geom. Mean	59.2	61.2	60.4	58.8	58.8	58.8

Table 13. Performance comparison of higher-level gains. Numbers in column headings are the highest higher-level gains considered. Values are total CPU seconds on a SPARC-IPX for ten runs using the specified algorithm.

Mapping	Dynamic	Fixed				
		1	2	3	4	20
s38584	1904	1606	1652	1981	2078	3910
s35932	2321	1830	1862	2100	2297	2766
s15850	630	509	518	643	678	956
s13207	551	425	446	549	572	815
s9234	338	252	250	333	355	466
s5378	186	130	134	181	185	241
Geom. Mean	677.2	524.5	536.4	667.6	703.8	990.8

There is an additional problem with using higher-level gains on clustered circuits: huge runtimes. The KLFM partitioning algorithm maintains a bucket for all nodes with the same gains in each partition. Thus, if the highest fanout node has a fanout of N, in KLFM without higher-level gains there must be $2*N+1$ buckets per partition (the N-fanout node can have a total gain between $+N$ and $-N$). If we use M-level gains (i.e., consider higher-level gains between 1st-level and Mth-level inclusive), we would require $(2*N+1)^M$

different buckets. In unclustered circuits this is fine, since nodes will have a fanout of at most 5 or 6. Unfortunately, clustered circuits can have nodes with fanout on the order of hundreds. This causes not only a storage problem, but also a performance problem, since the KLFM algorithm will often have to perform a linear search of all buckets of gains between occupied buckets, and buckets will tend to be sparsely filled. We have found two different techniques for handling these problems. First, the runtimes are acceptable as long as the number of buckets is reasonable (perhaps a few thousand). So, given a specific bound *N* on the largest fanout node (which is fixed after every clustering and unclustering step), we can set *M* to the largest value that requires less than a thousand buckets be maintained. This value is recalculated after every unclustering step, allowing us to use a greater number of higher-level gains as the remaining cluster sizes get smaller. We call this technique *dynamic gain-levels*. An alternative to this is to exploit the sparse nature of the occupied gain buckets. That is, among nodes with the same 1st- and 2nd-level gains, there will be few different occupied gain buckets. What we can do is perform the dynamic gain-level computation to determine the number of array locations to use, but each of these array locations is actually a sorted list of occupied buckets. That is, once the dynamic computation yields a given *M*, all occupied gain buckets with the same first *M* gains will be placed in the list in the same array location. In this way, circuits with large clusters, and thus very sparse usage of the possible gain levels, have only 2 or 3 gain-levels determining the array location, while circuits with small or no clusters, and thus more dense usage of the smaller possible gain locations, have more of their gain orders determining the array locations. In this latter technique, called *fixed gain-levels*, the user can specify how many gain-levels the algorithm should consider, and the algorithm automatically adapts its data structures to the current cluster sizes.

As shown in Table 12 and Table 13, using more gain levels improves the results, but only up to a point. Once we consider gains up to the 3rd level, we get all the benefits of up to 20 gain levels. Thus, extra gain levels beyond the 3rd level only serve to slow down the algorithm, up to a factor of 50% or more. Dynamic gain-levels produces results between those of 2-level and 3-level fixed gains. This is to expected, since at high clustering levels the dynamic algorithm uses only 2 gain levels, though once the circuit is almost totally unclustered it expands to use several more gain-levels. In this survey we use fixed, 3-level gains.

Dual Partitioning

During partitioning, the goal is to minimize the number of nets in the cutset. Because of this, it seems odd that we move nodes from partition to partition instead of moving nets. As suggested by Yeh, Cheng, and Lin [Yeh91], we can combine both approaches in a single partitioning algorithm. The algorithm consists of Primal passes, which are the basic KLFM outer loop, and Dual passes, which are the KLFM outer loop, except that nets are moved instead of nodes. In this way, the Dual pass usually removes a net from the cutset at each step, though this may be more than balanced by the addition of other nets into the cutset. Just as in the KLFM algorithm, a single Primal or Dual pass moves each node or net once, and when no more objects can be moved the state with the lowest cutsize is restored. Primal and Dual passes are alternated, and the algorithm ends when two consecutive passes (one Primal, one Dual, in either order) produce no improvement. When performing unclustering, we start with a Primal pass after each unclustering.

While the concept of moving nets may seem straightforward, there are some details to consider. First, when we move a net, we actually move all nodes connected to that net to the destination partition. Nodes already in that partition remain unlocked, while moved nodes are locked. Because we are moving nets and not nodes, the bucket data structure holds nets sorted by their impact in the cutsize, not nodes. An odd situation occurs when a net is currently in the cutset. Since it has nodes in each partition, it is a candidate to be moved to either partition. Also, because we are moving nets and not nodes, it is unclear how to apply higher-level gains to this problem, so higher-level gains are only considered in the Primal passes.

One of the problems with the Dual partitioning passes is that they are excessively slow. When we move a net, it not only affects the potential gain/loss of moving neighboring nets (where two nets are neighbors if they both connect to a shared node), it can affect the neighbor's neighbors as well. The gain of moving a net is the sum of the gain of removing the net from the cutset (1 if the net is currently cut, 0 otherwise), plus gains or losses from adding or removing neighboring nets from the cutset (by moving a node connected to a neighboring net, we may add or remove that net from the cutset). Thus, when we move a net, we may add or remove a neighboring net to or from the cutset. That neighbor's neighbors may have already expected to add or remove the neighbor from the cutset, and their gains may need to be recalculated. In a recursively clustered circuit, or even in a circuit with very high fanout nets (such as clocks and reset lines), most of the nets in the system will be neighbors or neighbors of neighbors. Thus, each move in a Dual pass will need to recalculate the gains of most of the nets in the system, taking a significant amount of time.

The solution we adopted is to ignore high fanout nets in the Dual pass. In our study, we do not consider moving high fanout nets (those nets connected to more than 10 nodes) since it is unlikely that moving a high fanout net will have a positive effect on the cutsize. We also do not consider the impact of cutting these high fanout nets when we decide what nets to move. Thus, when a neighbor of this net is moved, we do not have to recalculate the gains of all neighbors of this high fanout net, since these nets do not have to worry about cutting or uncutting the high fanout net. Note that this makes the optimization inexact, and at the end of a Dual pass we may return to what we feel is the best intermediate state, but which is actually worse than other states, including the starting point for this pass. To handle this, we re-evaluate the cutsize at this state, and only accept it if it is in fact better than the original starting point. Otherwise, we backtrack to the starting point. In our experience the cutsize calculation is almost always correct.

Table 14. Quality comparison of Dual partitioning. Values are minimum cutsize for ten runs of the specified algorithm. The data does not include the largest circuit due to excessive runtimes.

Mapping	No Dual Passes	Dual Passes
s35932	47	47
s15850	60	62
s13207	73	75
s9234	52	51
s5378	68	67
Geom. Mean	59.2	59.5

Table 15. Performance comparison of Dual partitioning. Values are total CPU seconds on a SPARC-IPX for ten runs of the specified algorithm. The data does not include the largest circuit due to excessive runtimes.

Mapping	No Dual Passes	Dual Passes
s35932	2100	10695
s15850	643	19368
s13207	549	8415
s9234	333	8293
s5378	181	5445
Geom. Mean	667.6	9532.5

Data from testing the Dual partitioning passes within our best algorithm is shown in Table 14 and Table 15. As can be seen, there is little difference in the quality of the two solutions, and in fact using the Dual passes actually degrades the quality slightly. The Dual passes also slow overall algorithm runtimes by a factor of over 14 times, even with the performance enhancements discussed previously. Obviously, without any signs of improvement in partitioning results, there is no reason to suffer such a large performance degradation.

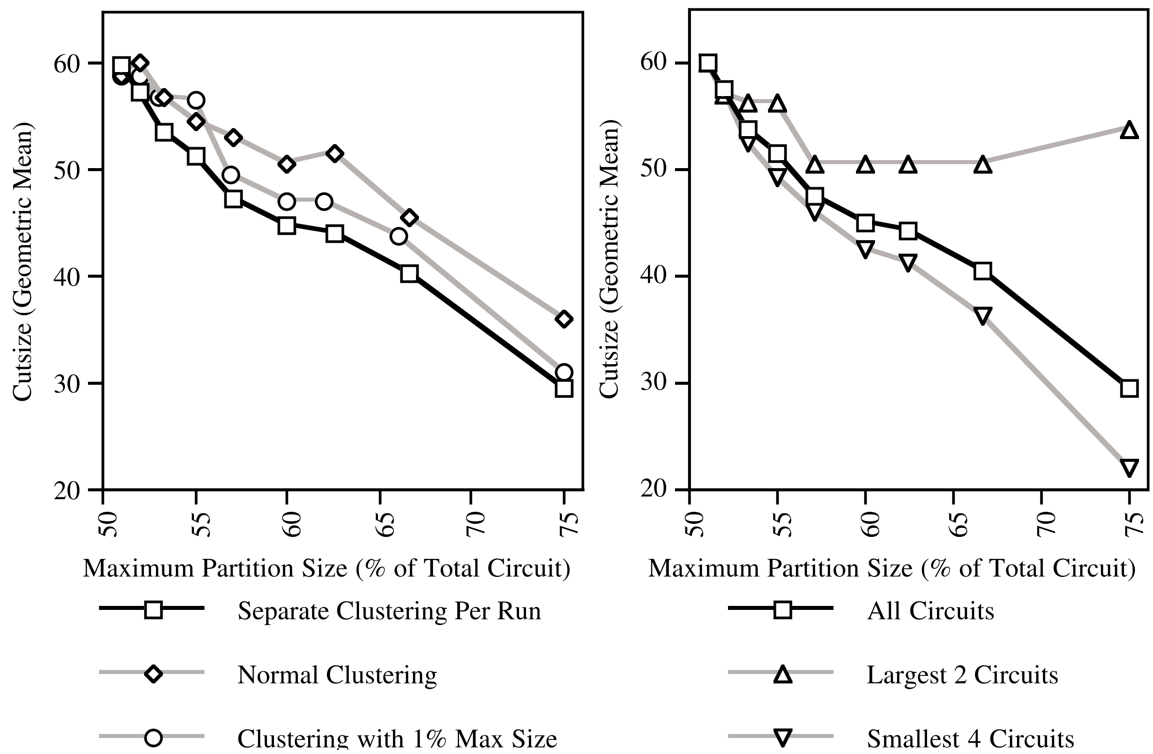

Figure 77. Graphs of partitioning results as the maximum allowed partition size is increased. At left are the results for separate clustering calculations for each run of the algorithm ("Separate Clustering Per Run"), one clustering for each partition size ("Normal Clustering"), and one clustering for each partition size, plus a maximum cluster size of 1% of the total circuit ("Clustering with 1% Max Size"). At right we have more detail on the "Separate Clustering Per Run", with the results for all circuits, plus one line for just the largest 2 circuits, and one for the smallest 4. Both "Largest 2" and "Smallest 4" lines are scaled to have the same value as "All Circuits" for the leftmost data point.

Partition Maximum Size Variation

Variation in the allowed partition size can have a significant impact on partitioning quality. In partitioning, we put limits on the sizes of the partitions so that the partitioner cannot place most of the nodes into a single partition. Allowing all nodes into a single partition obviously defeats the purpose of partitioning in most cases, since we are usually trying to divide the problem into manageable pieces. The variance in partition size defines the range of sizes allowed, such as between 45% and 55% of the entire circuit. There are two incentives to allow as much variance in the partition sizes as possible. First, the larger the allowable variation, the greater the number of possible partitionings. With more possible partitionings, it is likely that there will be better partitionings available, and hopefully the partitioner will generate smaller cutsizes. The second issue is that there needs to be enough variance in partition sizes to let each node move between partitions. If the minimum partition size plus the size of a large node is greater than the maximum partition size then this node can never be moved. This will artificially constrain the placement of this node to the node's initial partition assignment, which is often a poor choice. While we might expect that the size of the nodes in the graph being partitioned will be small, and thus not require a large variation in partition sizes, we will usually cluster together nodes before partitioning, greatly increasing the maximum node size. A smaller partition variation will limit the maximum cluster size, limiting the effectiveness of clustering optimizations. In general, we will require that the maximum cluster size be at most half the size of the allowable variation in partition sizes. In this way, if we have maximum-sized clusters as move candidates from both partitions, at least one of them will be able to move.

Conflicting with the desire to allow as much variation in partition sizes as possible is the fact that the larger the variation, the greater the wastage of logic resources in a multi-chip implementation, particularly a multi-FPGA system. Specifically, when we partition to a system of 32 FPGAs, we iteratively apply our bipartitioning algorithm. We split the overall circuit in half, then split each of these partitions in half, and so on until we generate a total of 32 subpartitions. Now, consider allowing partition sizes to vary between 40% and 60% of the logic being split. On average, it is likely that better partitions exist at points where the partition sizes are most unbalanced, since with the least amount of logic in one partition there is the least chance that a net is connected to one of those nodes, and thus the cutsize is likely to be smaller. This means that many of the cuts performed may yield one partition containing nearly 60% of the nodes, and the other containing close to 40%. Thus, after 5 levels of partitioning, there will probably be one partition containing $.6^5 = .078$ of the logic. Now, an FPGA has a fixed amount of logic capacity, and since we need to ensure that each partition fits into an individual FPGA, all FPGAs must be able to hold that amount of logic. Thus, for a mapping of size N, we need a total FPGA logic capacity of $32*(.078*N) = 2.488*N$, yielding a wastage of about 60%. In contrast, if we restrict each partition to between 49% and 51%, the maximum subpartition size is $.51^5 = .035$, the required total FPGA logic capacity is $1.104*N$, and the wastage is about 10%. This is a much more reasonable overhead and we will thus restrict the partition sizes considered in this chapter to between 49%-51% of the total logic size. Note that by a similar argument we can show that partitioning algorithms that lack strong control over partition sizes, such as ratio-cut algorithms [Wei89], are unsuitable for our purposes.

As we just discussed, the greater the allowed variation in partition sizes, the better the expected partitioning results. To test this out, we applied our partitioning algorithm with various allowed size variations. The results are shown in Figure 77, and contain all of the optimizations discussed in this chapter, except: the "Clustering with 1% Max Size" only allows clusters to grow to 1% of the total circuit size, while the others allow clusters to be as large as half the allowed partition size variation (that is, maximum partition size - 50%). The "Separate clustering" line does not share clusterings, while the other lines share one clustering among all runs with the same partition size bound. As is shown, the achieved geometric means of the six circuits decreases steadily as we increase the maximum partition size. However, how we perform clustering has an impact on achieved quality, and the difference is greater for larger allowed partition sizes. Specifically, when the maximum allowed partition size is 51%, using the same clustering for all runs of the algorithm produces results as good as using separate clusterings for each run. Using a shared clustering is also faster than separate clustering, at least when all runs are performed sequentially. However, as the allowed partition size gets larger, it becomes important to use multiple different clusterings. Note that while each of these runs is performed with the connectivity clustering algorithm, the algorithm randomly chooses nodes as starting points of clusters, and thus different runs will produce somewhat different clusterings.

The reason why a single clustering does poorly for larger partition sizes is that it reduces the value of multiple runs, with almost all runs producing identical results. Specifically, as the allowed partition size grows, the allowed cluster size grows as well. When a partition is only allowed to be at most 51% of the total circuit size, no cluster can contain more than 1% of the circuit, and there will be at least 100 clusters. When the maximum partition size is 75%, a cluster can be 25% of the circuit size, and there will be relatively few top-level clusters. Thus, when partitioning is performed with this few clusters, all of the different runs will get the same results before the first unclustering, even though we create the initial partitionings randomly. Since the algorithm is totally deterministic, all of these runs will produce the same values. In fact, for partition sizes greater than 60% all ten runs of the algorithm for each circuit with shared clusterings produced the same results, and only s15850 had more than one result for a maximum partition size of 60%. To deal with this, we also ran the algorithm with a maximum cluster size of 1% of the total circuit size regardless of the maximum partition size. This technique is successful not only in better using multiple partition runs, with many different results being generated for a circuit with a specific maximum partition size, but also produces results that are up to 14% lower than the normal clustering results. However, this is still not as good as separate clusterings per run, which produces results up to 9% lower than clustering with a fixed maximum size. Because of this, for partition maximum sizes larger than 51% we use separate clusterings for each run of the partitioner.

While increasing the maximum partition size can produce lower cutsizes, most of this gain is due to improvement on the smaller circuits, while the larger circuits sometimes actually have worse results as the size variation increases. The line "Largest 2 Circuits" in Figure 77 right is the geometric mean of only the largest two circuits, s38584 and s35932, while "Smallest 4 Circuits" represents the other test cases. These lines have been scaled to be identical to the "All Circuits" value at the leftmost data-point, so that if the benefit of increasing the maximum partition size was uniform across all circuits, the three lines should line up perfectly. However, our algorithm does worse on the larger circuits as the maximum partition size increases, with the geometric mean actually increasing at the rightmost trials. The true optimum cutsize cannot get larger with larger maximum partition sizes, since when increasing the allowed partition size, the algorithm could still return a partitioning that satisfies the smaller partition bounds. Thus, the results should never increase as the maximum partition size increases, and should in general decrease. We are forced to conclude that our algorithm is unable to exploit the larger partition size bounds for the larger circuits, and in fact gets sidetracked by this extra flexibility.

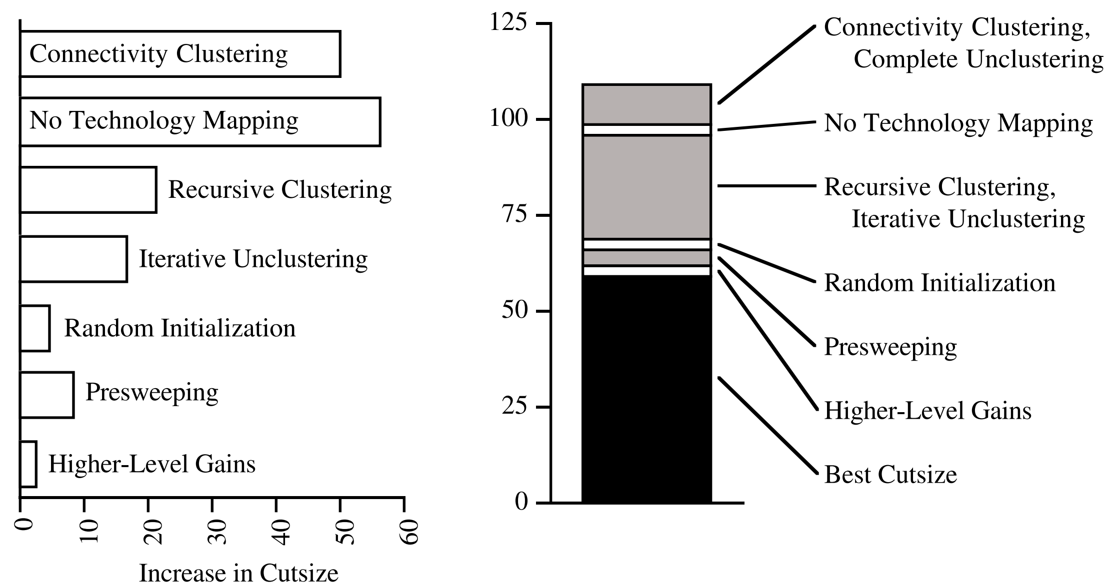

Figure 78. Two methods of determining the contribution of individual partitioning techniques to the overall results. At left are the results of comparing our best algorithm vs. taking the specified technique and replacing it with the worst alternative in this chapter. At right are the resulting cutsizes after starting with the worst algorithm, then iteratively adding the technique that gives the greatest improvement at that point.

Overall Comparison

While throughout this chapter we have discussed how individual techniques impact an overall partitioning algorithm, it is natural to wonder which of these techniques is the most important, and how much of the cutsize improvement is due to any specific technique. We have attempted to answer this question in two ways. First, we can take the comparisons we have made throughout this chapter, and bring them together into a single graph (Figure 78 left). Here we show the difference between the cutsizes generated by our best algorithm and the cutsizes generated with the same algorithm, except the specified technique has been replaced with the worst alternative considered in this chapter. For example, the "Connectivity Clustering" line is the difference between our best algorithm, which uses Connectivity clustering, and the best algorithm with Bandwidth clustering used instead. Note that the alternative used for iterative unclustering is complete clustering, not no unclustering, since complete unclustering is a very commonly used technique when any clustering is applied.

Our second comparison was made by starting with an algorithm using the worst choice for each of the techniques, and then iteratively adding whichever of the best techniques gives the greatest improvement in cutsize. Specifically, we ran the worst algorithm, and then ran it several more times, this time with one of the best techniques substituted into the mix. Whichever technique reduced the overall cutsize the most was inserted into the algorithm. We then tried running this algorithm several more times again, this time with both that best technique inserted, as well as each of the other techniques inserted one at a time. This process was repeated until all techniques were inserted. The resulting cutsizes, and the technique that was added to achieve each of these improvements, are shown in Figure 78 right. The initial, worst algorithm used was basic KLFM with seeded initialization and technology-mapped files.

As shown in the graphs in Figure 78, the results are mixed. Both of the comparisons show that connectivity clustering, recursive clustering, and iterative unclustering have a significant impact, presweeping has a modest impact, and both random initialization and higher-level gains cause only a small improvement. The results are somewhat mixed on technology-mapping, with the right comparison indicating only a small improvement, while the left comparison indicates a decrease in cutsize of almost a factor of two.

The graphs in Figure 78 give the illusion that we can pinpoint which individual techniques are responsible for what portion of the improvements in cutsizes. However, it appears that cutsize decreases are most likely due more to synergy between multiple techniques than to the sum of individual techniques. In Figure 79 we present all of the data used to generate Figure 78 right. The striped bar at left is the cutsize of the worst algorithm. The other groups of bars represent the cutsizes generated by adding each possible unused technique to the best algorithm found in the prior group of bars. Thus, the leftmost group of 6 bars represent 6 possible techniques to add to the worst algorithm, and the group of 5 bars just to the right represent the 5 possible additions to the best algorithm from the leftmost group of bars.

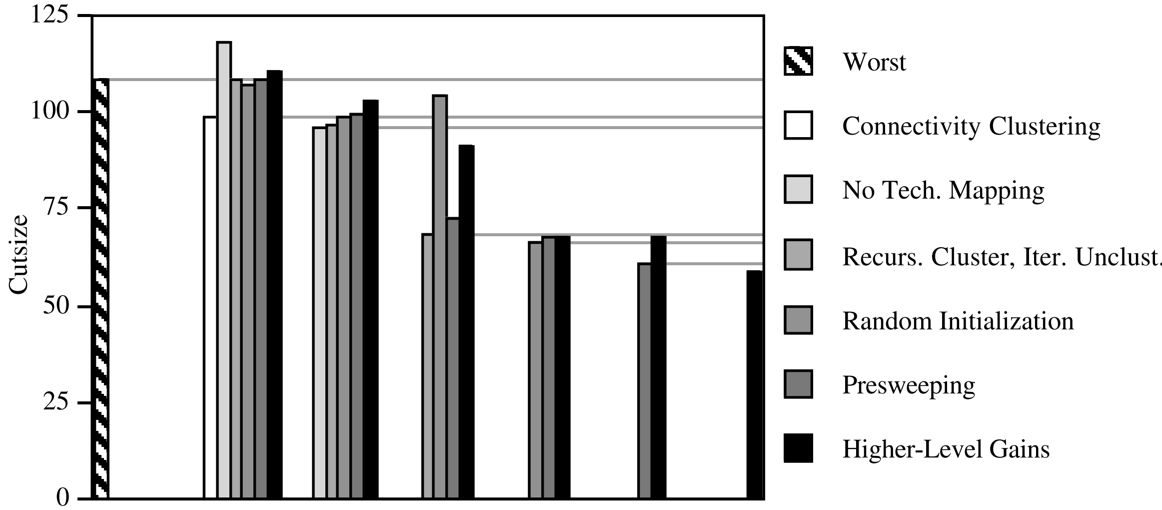

Figure 79. Details of the comparison of individual features. The bar at left is the cutsize of the worst algorithm. Each group of bars is the set of all possible improvements to the algorithm. Gray horizontal lines show the cutsize of the best choice in a given group of bars.

The important observation to be made from Figure 79 is that any specific technique can have a radically different impact on the overall cutsize depending on what other techniques are used. For example, if we take the worst algorithm and apply it to non-technology mapped files, the resulting cutsizes increase (degrade) by about 9%; once we add connectivity clustering to the worst algorithm we then see an improvement of 3% by working on non-technology mapped files. In fact, Figure 79 shows cases where we degrade the cutsize by applying random initialization, presweeping, or higher-level gains, even though all of these techniques are used in our best algorithm, and the cutsizes would increase if we removed any of these techniques. The conclusion to be reached seems to be that it is not just individual techniques that generate the best cutsizes, but it is the intelligent combination of multiple techniques, and the interactions between them, that is responsible for the strong partitioning results we achieve.

Pseudocode and Computational Complexity

In this section we present pseudocode for our algorithm, including the individual optimizations, as well as a discussion of the computational complexity of the algorithm. The overall algorithm flow is given in Figure

80. The first steps of the algorithm, reading in the input file and building the data structures, can be done in linear time. This will include constructing the pointers between nodes and signals in the circuit. Presweeping (see Figure 81) can also be done in linear time.

```
Read in gate-level netlist;
Build internal data structures;
                     // Presweeping and recursive clustering
Presweep circuit;
do {
   connectivity clustering;
} until no new clusters formed;
                     // Initialize circuit
Create random initial partitioning;
                     // Perform partitioning with 3 gain levels
repeat 10 times {
   do {
      perform KLFM inner loop;
      remove highest level of clusters;
   } until no clusters remain;
   perform final KLFM inner loop on totally unclustered circuit;
}
return best of 10 results;
```

Figure 80. Optimized Kernighan-Lin, Fiduccia-Mattheyses algorithm.

```
For each node N {
   if N is an I/O node {
      cluster N with any neighboring node;
   } else if N is a 1-input function {
      if any neighbor M is connected to N by a 2-terminal net {
         cluster N and M;
      }
   }
}
```

Figure 81. Presweeping.

Recursive connectivity-clustering (Figure 82) is more complex. In each clustering pass each node is examined, and for each highest-level cluster every signal connected to it is examined, as well as every terminal on that signal (where a terminal exists at each connection between a node and a signal). Since every signal has at least one terminal, one pass takes O(C*T) time, where T is the number of terminals, and C is the number of clusters in the highest level of the hierarchy at the beginning of the pass (where in the first pass all nodes are at the same level, including clusters formed by presweeping). Note that we do not need to examine nodes and clusters at lower levels, since such elements were not clustered during the last pass of the algorithm. Any element not clustered in one pass will never be clustered in subsequent passes, since the only reason not to cluster an element is that it has no neighbors small enough to cluster with, and this will not change in subsequent passes. Thus, if there are C candidates for clustering in one pass, there can be at most C/2 candidates in the next pass, and will probably be much less due both to clusters formed by three or more elements in one pass, as well as elements which cannot be clustered. Thus, in recursive connectivity clustering, with N = the number of nodes in the circuit, the first pass will take O(N*T), the second O(N/2*T), the third O(N/4*T), and so on. Since N*T + N/2*T + N/4*T + N/8*T + ... = 2N*T, the entire recursive connectivity clustering algorithm is O(N*T). Gatelevel netlists have only a few terminals per gate (usually two or three inputs and one output), and thus T is only a constant factor larger than N. Thus, the entire clustering process is $O(N^2)$.

```
For each cluster N at the highest level {
   If N has not yet been clustered during this pass {
      for each node or cluster M in circuit {
         bandwidth(M) = 0;
      }
      for each signal S connected to N {
         for each node or cluster X connected to S {
            bandwidth(X) = bandwidth(X) + bandwidth due to S;
         }
      }
      find neighbor Y with highest connectivity(Y, N) where
         size(Y) + size(N) <= maximum cluster size;
      if Y exists {
         cluster N and Y;
      }
   }
}
```

Figure 82. Connectivity clustering.

The random initialization algorithm is shown in Figure 83. The random assignment of numbers to elements can be done in linear time: place the elements in any order in an array. Then, randomly pick any element (including the last) and swap it with the element in the last position. Ignore this element and repeat with the second to last position. Continuing this process, a random, non-duplicated ordering can be determined in linear time. Since the other steps in the initialization simply make linear passes through this array, the time for the process is O(E), where E is the number of elements in the clustered circuit. This is obviously at worst O(N).

The KLFM algorithm, with 3rd-level gains, is shown in Figure 84. In [Fiduccia82] it was shown that the basic KLFM algorithm, without iterative unclustering and higher-level gains, is O(N). Addition of iterative unclustering increase the runtime to at most O(N log N) since there can be at most log N levels in the clustering hierarchy, and the KLFM algorithm is run once per clustering level. Note that the algorithm may actually do better than this, because at higher levels of the hierarchy there is less work to do, and an argument such as that applied for the clustering algorithm may improve the runtimes to possibly even O(N). It is unclear what the addition of our optimized higher-level gains structure does to the runtimes. A naïve implementation of the bucket structure could require a linear search through a 3-dimensional table, where each dimension can be as large as the number of nodes in the input circuit. This would yield an $O(N^3 \log N)$ algorithm. However, in our experience the optimized data structures presented here avoid much of this performance degradation, and the algorithm operates nearly as fast as the version without higher-level gains. If the higher-level gain bucket data structure does prove to be a limiting factor, higher-level gains can be removed from the algorithm with only a 4% quality degradation.

```
Max = total number of elements in clustered circuit;
Without duplication, randomly assign numbers 1..Max to elements;
below = 0;
above = total circuit size;
For (I = 1 to Max-1) {
   below = below + size(element(I));
   above = above - size(element(I));
   balance(I) = absolute value(above - below);
}
J = value of I that minimizes balance(I);
For (K = 1 to Max) {
   if K > I {
      assign element(K) to partition 1;
   } else {
      assign element(K) to partition 0;
   }
}
```

Figure 83. Random initialization.

All the other steps in the algorithm are trivial with the proper data structures. Thus, the dominant part of asymptotic complexity of this algorithm is the recursive clustering step. Ignoring higher-level gains, the overall algorithm is $O(N^2)$. It is likely that under reasonable conditions the algorithm with 3rd-level gains is also $O(N^2)$. If the algorithm is run multiple times, such as when the algorithm is recursively applied, the runtime is $O(\max(N^2, T{*}N \log N))$, where T is the number of partitioning runs performed.

Conclusions

There are numerous approaches to augmenting the basic Kernighan-Lin, Fiduccia-Mattheyses partitioning algorithm, and the proper combination is far from obvious. We have demonstrated that technology-mapping before partitioning is a poor choice, significantly impacting mapping quality. Clustering is very important, and we found that Connectivity clustering performs well, though Shortest-path clustering is a reasonable alternative. Recursive clustering and a hierarchical unclustering technique help take advantage of the full power of the clustering algorithm, with iterative unclustering being slightly preferred to edge unclustering. Augmenting the basic KLFM inner loop with at least 2nd- and 3rd-level gains improves the final results, while Dual passes are not worthwhile, and greatly increase run times. Finally, when the allowed maximum partition size is greater than 51% of the total circuit size, creating a clustering on a per-run basis produces better results than shared clustering. The table in the chapter introduction shows that applying all of these techniques generates results at least 17% better than the state-of-the-art in partitioning research.

```
Initialize bucket data structures (using 3 levels of gains)
While cutsize is reduced {
   While valid moves exist {
      Use buckets to find unlocked node in each partition that has
         the best gain (using 3 levels of gains);
      Move whichever of the two nodes has the best gain while not
         exceeding partition size bounds;
      Lock moved node;
      Update nets connected to moved nodes, and nodes connected to
         these nets;
   } endwhile;
   Backtrack to the point with minimum cutsize in move series just
      completed;
   Unlock all nodes;
} endwhile;
```

Figure 84. The KLFM inner loop, augmented with 3rd-level gains.

This chapter has included several novel techniques, or efficient implementations of existing work. We have started from the base work of Schuler and Ulrich [Schuler72] to develop an efficient, effective clustering method. We have also created the presweeping clustering pre-processor to help most algorithms handle small fanout gates. We have shown how shortest-path clustering can be implemented efficiently. We developed the edge unclustering method, which is competitive with iterative unclustering. Finally, we have extended the work of Krishnamurthy [Krishnamurthy84], both to allow higher-order gains to be applied to nets not in the cutset, and also to give an efficient implementation, even when the circuit is clustered.

Beyond the details of how exactly to construct the best partitioner, there are several important lessons to be learned. As we have seen, the only way to determine whether a given optimization to a partitioning algorithm makes sense is to actually try it out, and to consider how it interacts with other optimizations. We have shown that many of the optimizations had greater difficulty working on clustered circuits than on unclustered circuits, yet clustering seems to be important to achieve the best results. Also, many of the clustering algorithms seem to assume the circuit will be technology-mapped before partitioning, yet technology-mapping the circuit will greatly increase the cutsize of the resulting partitionings. However, it is quite possible to reach a different conclusion if we use only the basic KLFM algorithm, and not any of the numerous enhancements proposed since then. By using the basic KLFM algorithm, cutsizes are huge, and subtle effects can be ignored. While a decrease of 10 in the cutsize is not significant when cutsizes are in the hundreds, it is critical when cutsizes are in the tens. Thus, it is important that as we continue research in partitioning we properly place new concepts and optimizations in the context of what has already been discovered.