IEEE Trans. on CAD, Vol. 16, No. 8, pp. 849-866, August 1997.

An Evaluation of Bipartitioning Techniques

Scott Hauck
Department of EECS
Northwestern University
Evanston, IL 60208 USA
hauck@eecs.nwu.edu

Gaetano Borriello
Department of CSE
University of Washington
Seattle, WA 98195 USA
gaetano@cs.washington.edu

Abstract

Logic partitioning is an important issue in VLSI CAD, and has been an area of active research for at least the last 25 years. Numerous approaches have been developed and many different techniques have been combined for a wide range of applications. In this paper, we examine many of the existing techniques for logic bipartitioning and present a methodology for determining the best mix of approaches. The result is a novel bipartitioning algorithm that includes both new and pre-existing techniques. Our algorithm produces results that are at least 16% better than the state-of-the-art while also being efficient in run-time.

Introduction

Logic partitioning is one of the critical issues in CAD for digital logic. Effective algorithms for partitioning circuits enable one 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. Traditionally, this problem was important for breaking up a complex system into several custom ASICs. Now, with the increasing use of FPGA-based emulators and prototyping systems, partitioning is becoming even more critical.

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 [Alpert95a].

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 paper 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 Fiduccia-Mattheyses [Fiduccia82] variant of the Kernighan-Lin [Kernighan70] algorithm (hereafter referred to as KLFM).

In the rest of this paper 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 and alterations to the standard KLFM inner loop, and the effects of increasing the maximum allowed partition size and the number of runs per trial. As we will demonstrate in the conclusions, this has helped build a bipartitioning algorithm that is significantly better than the current state-of-the-art.

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.

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 stated otherwise. Specifically, the default algorithm includes the following optimizations (each of these techniques is described later in this paper):

- recursive connectivity clustering
- presweeping
- node sizes of (inputs-1), minimum of 0
- non-technology mapped (gate-level) netlists
- iterative unclustering
- random initialization
- LIFO gainarray buckets
- fixed 3rd-level gains
- maximum partition size of 51% of the total circuit size
- maximum cluster size of 1% of the total circuit size

This algorithm is run 10 times, and the best of these ten runs is returned. The clustering is calculated once, and shared among these different runs.

Table 1. Sizes of example circuits.

Circuit	Nodes (gates, latches, IOs)	Nets	Pins (node-net connections)
s38417	25589	23953	64299
s38584	22451	20999	61309
s35932	19880	18152	55420
industry3	15406	21923	65791
industry2	12637	13419	48158
s15850	11071	10474	27209
s13207	9445	8776	23442
biomed	6514	5742	21040
s9234	6098	5870	15026
s5378	3225	3046	8241

The 7 largest circuits from the MCNC partitioning benchmark suite [MCNC93], as well as three commonly used partitioning benchmarks (industry2, industry3, biomed) are used as test cases for this work (Table 1). 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 only about 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.

In the sections that follow, we consider each of the possible optimizations to the basic KLFM algorithm. The results presented are cutsizes and runtimes. The cutsize is the number of nets connected to nodes in both partitions, and thus is the quantity we seek to minimize. Runtimes are CPU seconds on a SPARCstation 5 model 70, and include the time to cluster the circuit as well as the total time to perform all of the multiple runs of the partitioning algorithm. Note that when we are making overall comparisons, we use the geometric mean $\left(\prod_{i=1}^{n} A_i\right)^{1/n}$ instead of the more common arithmetic mean $\left(\sum_{i=1}^{n} A_i\right)/\,n$. This is 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. Also, an arithmetic mean tends to give more weight to percentage increases and decreases of larger values. Thus, since our benchmark suite ranges from 3,046 to 23,953 nets, and cutsizes should also vary over a fairly large range, the arithmetic mean would be dominated by an algorithm's results on the larger circuits. Finally, we are primarily concerned with comparing the results of different algorithms on the set of benchmarks, and will consider the ratio of their performances. Unfortunately, with an arithmetic mean the average of the ratios for each example is different from the ratio of the averages. However, the geometric mean of the ratios is identical to the ratio of the geometric means.

```
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 1. 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 1. 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 cutsize 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 paper, 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 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*) inspired by Bui et al [Bui89], 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 another 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 $l_1 \ldots l_k$ 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 1. $Bandwidth_{ij}$ is the total bandwidth between the nodes (as defined in bandwidth clustering), where each net contributes $1/(|e|-1)$ 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.

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

Equation 1. 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. 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 1% of the total circuit size. There are several alterations that can be made to this algorithm to boost performance, details of which can be found in [Hauck95].

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. One possibility is to simply assume that all logic functions are the same size, and assign an area of 1 to all nodes. However, in many cases the input circuit can have simple and complex functions mixed together. Since the goal of bipartitioning is to equalize the logic in the two partitions, thus allowing the two partitions to each fit into the same chip area, this model can be quite inaccurate. For example, this would assume that a 5-input AND gate and an inverter would occupy the same space. In most circumstances, the AND gate would be much larger than the inverter, and in fact in many technologies this inverter would consume no area at all, since it could be combined with the previous gate. For example, in a CMOS circuit a 5-input AND gate followed by an inverter would take up less space than the AND gate alone, since the gates in a CMOS technology automatically invert their output signals. If the circuit being partitioned has already been technology mapped (restructured into physically realizable gates in a given technology), then direct measurements of the gate areas could be obtained. Unfortunately, as we will show later in this section, it is better to partition before technology mapping, and thus some estimations of the final area of the logic must be made.

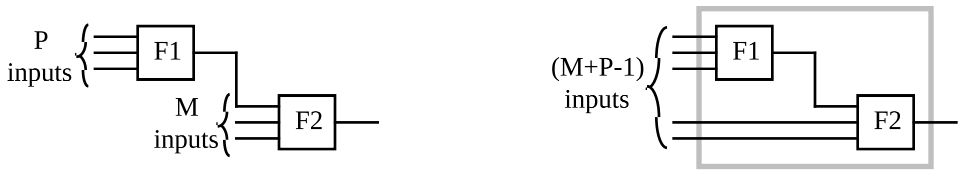

Figure 2. Example for the discussion of the size of logic functions. The *P*-input and *M*-input functions cascaded together (left) are combined into a *(M+P-1)* input LUT (right).

In this work we have chosen to target FPGAs such as the Xilinx 3000 series [Xilinx92], 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 2). 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. Although it may seem strange to have nodes with a 0 logic size, as we have shown 1-input functions (inverters or buffers) are often free in a given technology, or in fact may yield even smaller logic area than if the function were not on that chip. Any I/O nodes (i.e., external inputs and outputs) have a cost of 0 as well. 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. Also, most technologies have a fixed region for handling all I/O nodes, and thus the I/O nodes do not consume space otherwise useable by other logic functions. 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.

Even though the (P-1) size metric has been developed specifically for FPGA technologies, similar metrics can be developed for other target technologies. For example, a reasonable estimate for the size of a CMOS gate is it's number of inputs. Even in this technology, inverters are often free, and I/O pads should not be counted as part of the logic size of a partition. Thus, to retarget this partitioner to a CMOS technology simply requires increasing the size of all non-0 size nodes by one (a P size metric with 0 sized inverters and I/O pads).

Table 2. 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. The "Time" row values are geometric mean times for running the specified algorithm on each of the example circuits.

Mapping	Random	K-L	Bandwidth	**Connectivity**	Shortest-Path	No Presweep
s38417	134	297	168	**57**	69	64
s38584	166	88	69	**54**	50	59
s35932	73	86	277	**47**	45	70
industry3	404	517	436	**427**	462	467
industry2	210	194	239	**181**	184	269
s15850	70	90	124	**60**	59	65
s13207	113	94	87	**73**	72	79
biomed	96	176	158	**83**	141	83
s9234	63	79	56	**52**	51	65
s5378	84	78	88	**68**	68	66
Geom. Mean	118.7	135.4	139.6	**82.4**	87.9	95.1
Time	604.6	386.6	706.1	**499.5**	1498.6	600.7

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. Even in technologies where inverters and I/O pads are assigned a size greater than 0, presweeping inverters on 2-terminal nets and all I/O pads is still a reasonable heuristic, though it is no longer guaranteed not to degrade the possible partitionings.

Results for the various clustering algorithms are presented in Table 2. The connectivity clustering algorithm generates the best results, with shortest-path clustering performing only about 7% worse. In terms of performance, partitioning with the shortest-path clustering algorithm takes more than three times as long as with the connectivity clustering algorithm. This is because clustering with the shortest-path algorithm usually takes more than 20 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, as well as a slight decrease 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 15% worse in terms of cutsize, while taking about 20% longer.

One surprising result is that both K-L and Bandwidth clustering do considerably worse than random clustering. 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 never be connected to the same AND gate twice. 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.

Table 3. 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 five largest MCNC circuits are used, because the non-MCNC circuits only had connectivity information, not the actual logic functions, and technology-mapping for the smaller MCNC examples causes clusters to exceed 1% of the total circuit size. Because of limitations of our current implementation, all entries except the “No Tech Map” column use separate clusterings for each of the 10 runs of the algorithm

Mapping	K-L	Bandwidth	Connectivity	**No Tech Map**	Unclusterable
s38417	133	116	102	**57**	43
s38584	169	159	120	**54**	60
s35932	155	157	143	**47**	53
s15850	97	95	86	**60**	60
s13207	118	119	116	**73**	72
Geom. Mean	131.9	126.8	111.8	**57.6**	56.8

Unfortunately, technology-mapping before partitioning is an extremely poor idea. In Table 3, 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 the five largest MCNC benchmarks are used, because the other MCNC benchmarks were small enough that the size of a single CLB was larger than the allowed partition size variation, and the three other circuits only had connectivity information without the actual logic functions). 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 generate results closer to connectivity clustering's for these circuits than the non-technology mapped examples, 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 after partitioning 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 the amount of logic that can fit on the chip is constrained at least as much by the number of I/O pins as by 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.

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 grouped together to form a CLB. This lets us consider 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 3. As can be seen, once the technology mapping is allowed to be removed from the circuit, the partitioner can produce results just as good as the version operating on non-technology-mapped circuits. However, since technology-mapping is a complex, time-consuming process, and many technology-mappers would not retain information about mapped gates (which is what allowed us to undo the technology-mapping for the "Unclusterable" case), partitioning on non-technology-mapped files is preferred to technology-mapping, partitioning, and then re-technology-mapping.

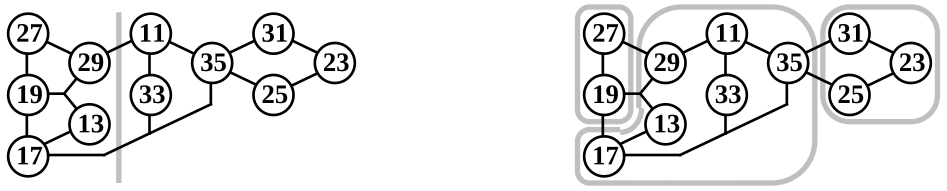

Figure 3. Example of the impact of technology-mapping on partitioning quality. The circuit s27 is shown (clock, reset lines, and I/O pins are omitted). At left is a balanced partition of the unmapped logic, which has a cutsize of 2. Gray loops at right indicate logic grouped together during technology-mapping. The only balanced partitioning has the largest group in one partition, the other two in the other partition, yielding a cutsize of 5.

The small example circuit (Figure 3) 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%.

Unclustering

When we use clustering to improve partitioning, we will usually partition the circuit, uncluster it, and partition again. The results of partitioning the clustered circuit is used as an initial partitioning for the subsequent partitioning of the unclustered circuit. 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 during 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. That is, if the maximum partition size is 51% of the logic, and thus the minimum is 49%, the maximum partition size variation is 2%, and no cluster can be formed that would include more than 1% of the total circuit size. This guarantees that, ignoring locked nodes, a node from one of the two partitions can always be moved without violating the partition size constraints. 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 coarse-grain optimization during early passes, medium-grain optimization during the middle passes, and fine grain optimization during late passes. This algorithm, which we will refer to here as *iterative unclustering*, is based on work by Cong and Smith [Cong93].

Table 4. 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. The "Time" row values are geometric mean times for running the specified algorithm on each of the example circuits.

Mapping	Single-level Clustering		Recursive Clustering			
	No Uncluster	Complete Uncluster	No Uncluster	Complete Uncluster	**Iterative Uncluster**	Edge Uncluster
s38417	178	150	140	87	**57**	55
s38584	90	69	213	131	**54**	57
s35932	157	156	90	75	**47**	45
industry3	580	413	1145	539	**427**	411
industry2	268	216	498	245	**181**	205
s15850	77	67	123	84	**60**	62
s13207	101	79	119	89	**73**	72
biomed	212	196	162	109	**83**	83
s9234	68	61	105	54	**52**	56
s5378	79	68	125	70	**68**	68
Geom. Mean	142.5	120.0	185.3	113.4	**82.4**	83.6
Time	329.0	436.4	237.0	383.9	**499.5**	540.8

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. When there are no more clusters left adjacent to the cut, we completely uncluster the circuit and partition one final time with KLFM.

As the results in Table 4 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 46% larger than the best unclustering (iterative), and even complete unclustering of a recursively clustered mapping yields a 38% larger cutsize. The difference between the two hierarchical unclustering methods is only 1.5%, with four mappings having smaller cutsizes with edge unclustering, and four 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 5. Quality comparison of initial partition creation methods. Values are minimum cutsize for ten runs using the specified algorithm. The "Time" row values are geometric mean times for running the specified algorithm on each of the example circuits.

Mapping	**Random**	Seeded	Breadth-first	Depth-first
s38417	**57**	69	57	65
s38584	**54**	87	56	54
s35932	**47**	47	47	47
industry3	**427**	477	427	427
industry2	**181**	181	181	181
s15850	**60**	60	60	60
s13207	**73**	75	80	74
biomed	**83**	105	104	102
s9234	**52**	68	52	52
s5378	**68**	79	80	78
Geom. Mean	**82.4**	95.3	86.7	86.5
Time	**499.5**	511.4	498.6	502.1

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 cannot 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.

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.

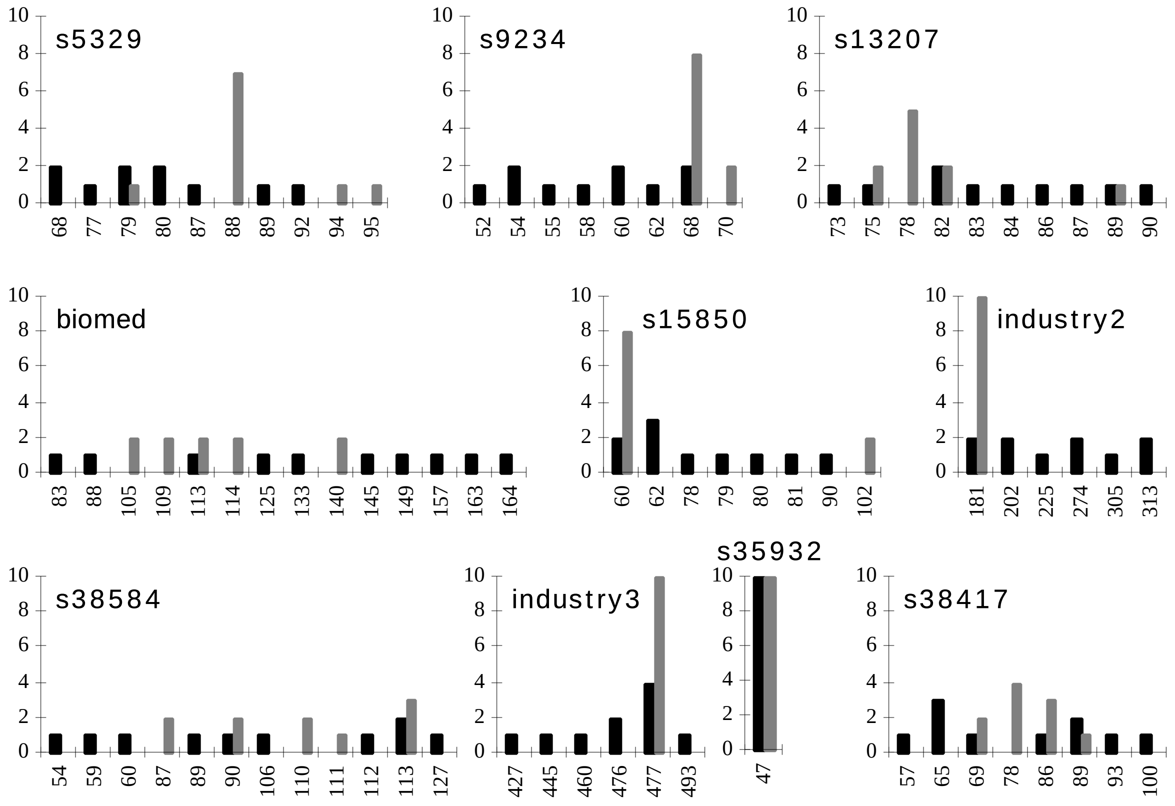

Figure 4. Distribution of results from partitioning with random (black bars) and seeded (gray bars) initialization. The *ith* bar from the left represents the *ith* best cutsize found by either algorithm, and the height indicates how many different runs of the algorithm (out of ten) achieved that result.

Results for these initial partition construction techniques are shown in Table 5. 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 16% and 5% worse than random, respectively. There are two reasons for this. First, 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. Second, the "more intelligent" approaches tend to produce less variation in the initial partitionings, which produces less variation in the results. Since we pick the best of multiple runs, by having greater variation in each run we get better overall results. These effects can be seen in Figure 4, 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 greater variation for the random algorithm in general. Also, the two algorithms seem to be finding somewhat different results, since often the seeded algorithm finds cutsizes the random did not, and vice-versa.

Table 6. Quality comparison of spectral initial partition creation methods. EIG1 and EIG-IG [Hagen92] are spectral partitioning algorithms, used here to generate initial partitions. "All Spectral" is the best results from the two spectral algorithms. The "Time" row values are geometric mean times for running the specified algorithm on each of the example circuits. The benchmark industry3 was not included because of space limitations in the spectral partitioners.

Mapping	**Random**	EIG1	EIG1-IG	All Spectral
s38417	**57**	65	65	65
s38584	**54**	56	56	56
s35932	**47**	47	47	47
industry2	**181**	315	202	202
s15850	**60**	60	96	60
s13207	**73**	111	82	82
biomed	**83**	87	87	87
s9234	**52**	54	54	54
s5378	**68**	78	78	78
Geom. Mean	**68.6**	79.8	77.0	73.5
Time	**448.1**	77.8	107.8	188.2

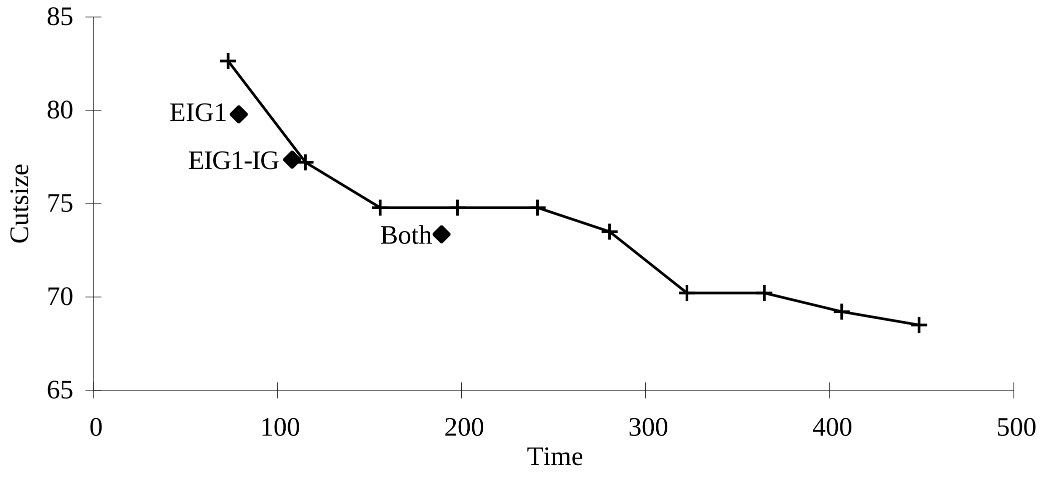

Figure 5. Graphs of cutsizes for different numbers of runs of our optimized version of KLFM versus the spectral initialization approaches. Values shown are the geometric means of the results for the 9 test circuits (all but industry3).

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 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 paper. These algorithms were applied to clustered circuits to generate initial partitionings. These initial partitionings were then used by our KLFM partitioning algorithm.

As the results show (Table 6), the algorithms (when taken as a group, under "All Spectral") produce fairly good results, but are still 7% 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. A more detailed quality/performance comparison is contained in Figure 5. This graph shows the quality produced by the different algorithms vs. the amount of time needed to produce the results. The line is the results of the 10 runs of our optimized algorithm, with data point i representing the best of the first i runs, and the time needed to complete those runs. 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 KLFM 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.

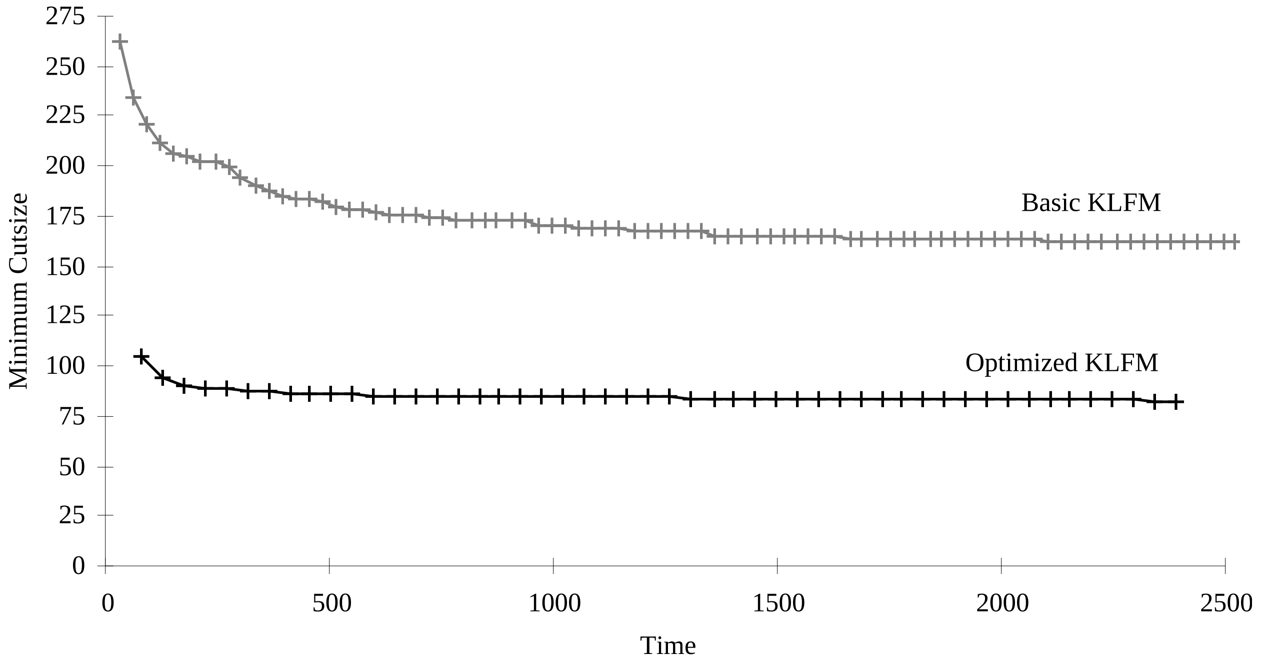

Figure 6. Graphs of cutsizes for different numbers of runs of both basic KLFM, and our optimized version of KLFM. Values shown are the geometric means of the results for all 10 test circuits.

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 our best algorithm for 5 sets of 50 runs each. That is, for each test circuit we ran our algorithm 250 times, though these 250 times consisted of one clustering shared among each 50 runs, and the value of the ith run within a set is the best value from runs 1..i from that set. For comparison, we also ran 5 sets of 100 runs of the basic KLFM algorithm, for a total of 500 runs (we ran more runs of the basic algorithm because it is faster, and we wish to compare results produced with the same amount of computation time). The geometric mean of the results across all benchmarks and sets of runs for a given benchmark are shown in Figure 6. As can be seen, not only does our optimized algorithm generate better results than the basic 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 fifty runs are on average only 4% better than those produced from ten runs. For comparison, basic KLFM produces a 10% improvement from one run to two, 26% from one to ten, 9% from ten to twenty, and 15% from ten to fifty.

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 paper unless stated otherwise.

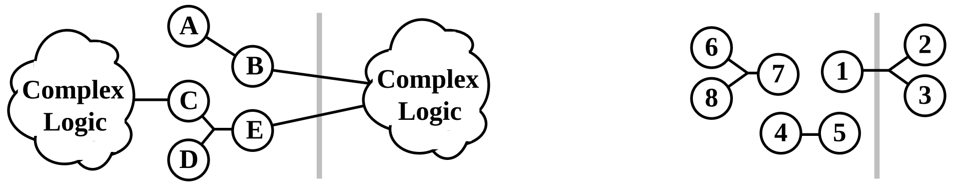

Figure 7. Examples for the higher-level gains discussion.

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 7 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 example in Figure 7 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.

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.

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 okay, 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 7, using more gain levels improves the results, but only up to a point. Once we consider gains up to the 4th level, we get all the benefits of up to 20 gain levels, and in fact the values for 4th-level gains are better than for 20th-level gains. Thus, extra gain levels beyond the 4th level only serve to slow down the algorithm, up to a factor of 50% or more. Note that the only circuit with an improvement from 3rd-level to 4th-level gains is industry2, and it then degrades significantly when 5th-level gains are added. We thus feel that this circuit should be ignored when considering what gain-levels to apply, and conclude that 3rd-level gains are the best tradeoff between quality and runtimes. 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.

Table 7. 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 optimized KLFM without higher-level gains. Values are minimum cutsize for ten runs using the specified algorithm. The "Time" row values are geometric mean times for running the specified algorithm on each of the example circuits.

Mapping	Dynamic	1	2	**3**	4	5	20
s38417	57	56	58	**57**	57	57	57
s38584	56	57	55	**54**	54	53	53
s35932	49	47	49	**47**	47	47	47
industry3	426	426	426	**427**	427	427	427
industry2	180	208	180	**181**	173	196	196
s15850	60	64	62	**60**	60	60	60
s13207	75	77	77	**73**	73	73	73
biomed	83	98	83	**83**	83	83	83
s9234	52	56	52	**52**	52	52	52
s5378	66	71	70	**68**	68	68	68
Geom. Mean	82.9	87.2	83.9	**82.4**	82.0	82.9	82.9
Time	532.1	388.6	404.4	**499.5**	601.4	607.7	936.3

Table 8. Quality comparison of bucket management policies. The "Time" row values are geometric mean times for running the specified algorithm on each of the example circuits.

Mapping	**LIFO**	FIFO
s38417	**57**	57
s38584	**54**	53
s35932	**47**	47
industry3	**427**	426
industry2	**181**	196
s15850	**60**	60
s13207	**73**	72
biomed	**83**	83
s9234	**52**	52
s5378	**68**	68
Geom. Mean	**82.4**	82.7
Time	**499.5**	489.3

Bucket Management

Even with the higher-level gains discussed in the previous section, there will be many times where multiple nodes have the same gain values, and thus end up in the same gain bucket. Depending on how nodes are inserted and removed from these buckets, either a Last-In-First-Out (LIFO), First-In-First-Out (FIFO) or random policy can be implemented. Although this issue has in general been ignored in the literature, it has been shown [Hagen95] that the policy can have an effect on the results produced. Specifically, consider what happens when a node is moved. This move can alter the gain values of its neighbors, and any neighbor whose gain values are altered will be inserted into a new gain bucket. Under a LIFO policy, one of these moved neighbors is more likely to be moved next than in a FIFO policy, since under a LIFO policy it would be considered before any other nodes with the same gain values. Thus, a LIFO policy tends to continue optimizing in the same area of the circuit over time, while a FIFO policy will tend to optimize more smoothly over the entire circuit. Intuitively, it would seem that a LIFO ordering would be preferred over a FIFO ordering, and that is in fact what has been found previously [Hagen95].

In Table 8 we compare LIFO and FIFO bucket management policies within the context of our optimized algorithm. As can be seen, although LIFO buckets do perform better than FIFO buckets, it is only by about 0.4%. In fact, FIFO buckets actually produce results at least as good as LIFO buckets for nine of the ten benchmarks, and produce better results for three of them. Thus, we conclude that the order of insertion and deletion from the gain buckets in an optimized KLFM algorithm has little or no effect on the quality of the results produced. In this survey we use LIFO bucket management except where specifically stated otherwise.

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, Yeh95], 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 9. Quality comparison of Dual partitioning. Values are minimum cutsize for ten runs of the specified algorithm. The data does not include the larger circuits due to excessive runtimes. The "Time" row values are geometric mean times for running the specified algorithm on each of the example circuits.

Mapping	**No Dual Passes**	Dual Passes
s15850	**60**	62
s13207	**73**	75
biomed	**83**	83
s9234	**52**	51
s5378	**68**	67
Geom. Mean	**66.3**	66.7
Time	**227.7**	4729.6

Data from testing the Dual partitioning passes within our best algorithm is shown in Table 9. Only the smaller examples are included, since for larger examples the partitioner was unable to finish even a single run with Dual passes within an hour's time. 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 20 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.

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. Specifically, when we partition to a system of 32 chips, 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, we usually assume a chip has a fixed amount of logic capacity, and since we need to ensure that each partition fits into an individual chip, all chips must be able to hold that amount of logic. Thus, for a mapping of size *N*, we need a total chip 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 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 paper to between 49%-51% of the total logic size, except where stated otherwise.

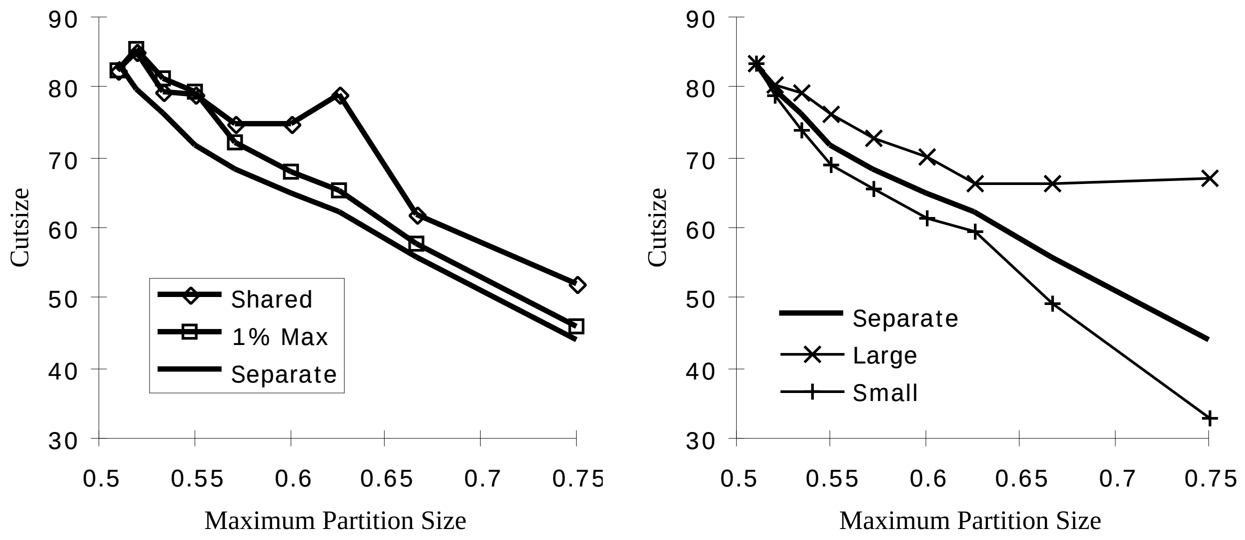

Figure 8. Graphs of partitioning results as the maximum allowed partition size is increased. At left are the results for separate clustering calculations for each run of the algorithm ("Separate"), one clustering for each partition size ("Shared"), and one clustering for each partition size, plus a maximum cluster size of 1% of the total circuit ("1% Max"). At right we have more detail on the "Separate" runs, with the results for all circuits, plus one line for the 4 largest circuits (excluding industry3), and one for the other 6. Both "Large" and "Small" lines are scaled to have the same value as "All Circuits" for the leftmost data point.

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 8, and contain all of the optimizations discussed in this paper, 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 - minimum partition size)/2). 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 ten 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 with shared clustering produced the same results for each circuit, and only s15850 and s35932 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 17% 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 “Large” in Figure 8 right is the geometric mean of four of the largest circuits (industry2, s38584, s35932, s38417) while “Small” represents the other test cases. These lines have been scaled to be identical to the “Separate” 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 trial. 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.

Overall Comparison

Throughout this paper we have discussed how individual techniques impact an overall partitioning algorithm. It is natural to also 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 Figure 9. The figure contains most of the comparisons presented in the paper, which represent removing a given technique from our optimized algorithm, as well as similar comparisons which use the basic KLFM algorithm as a baseline. Basic KLFM is assumed to operate on gate-level (non-technology-mapped) circuits, uses random initialization and FIFO buckets, and has no clustering, higher-level gains, nor dual passes. Note that some techniques are used by both the basic and optimized KLFM algorithms (specifically random initialization, gate-level netlists, and FIFO buckets). The height of the bars represents the difference between including the specified technique and using the worst alternative technique. Thus, the height of the “Random Initialization” bars represents the difference between random and seeded initialization, with the right bar using all of the other techniques from the optimized KLFM algorithm, while the left bar uses only the techniques from the basic KLFM algorithm. A gray bar indicates that including the specified technique actually degrades the results. Thus, using technology-mapped files within the optimized KLFM algorithm is a bad idea, as was discussed earlier.

As shown in the graphs in Figure 9, the results are mixed. It appears that for the optimized algorithm, the most important optimizations relate to circuit clustering, with recursive connectivity clustering, iterative unclustering, and non-technology-mapped circuits having large impacts. Presweeping, random initialization, and higher-level gains also have a significant impact, while dual passes and FIFO bucket management do not seem to make a difference.

The discussion above gives the illusion that we can pick specifically which optimization gives us what benefit independent of what other optimizations are used. However, if we compare the results of optimizations within the optimized algorithm with those within the basic algorithm, we see that this is not true. Specifically, the only optimizations that have a consistent impact on the cutsize is recursive connectivity clustering, iterative unclustering, and presweeping. For FIFO bucket management, which has only a slight impact on the optimized algorithm, it has a huge impact on the basic KLFM algorithm, greater than any other considered (in fact, it's bar stretches to six times the length shown, and was chopped off to make the impact of the other optimizations visible). For technology mapping, random vs. seeded initialization, higher-level gains[1], and dual passes, these optimizations have opposite impacts on the cutsize, where the technique that is best for our optimized algorithm actually degrades the quality produced by a basic KLFM algorithm. Thus, it appears that we cannot simply consider optimizations in isolation, because the interaction between techniques can have a significant impact on the results, and some optimizations may negate, or even be hindered by, the results of others.

We believe these results have an impact on how partitioning algorithms should be developed and evaluated. Specifically, Figure 9 clearly demonstrates that we cannot simply consider optimizations in isolation, since the performance of an optimization within the framework of the basic KLFM algorithm can be radically different from its performance within a high-quality partitioning algorithm. Thus, we believe that future optimizations must be evaluated not solely upon whether they produce better results that the basic KLFM algorithm, but must instead be shown to contribute to a complete partitioning algorithm, one capable of producing results better than those of the best current algorithms. As we will show in the conclusions, the partitioning algorithm developed in this paper produces results significantly better than the current state-of-the-art in logic bipartitioning.

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. 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. Bucket insertion and removal ordering also does not seem to significantly impact the quality of an optimized KLFM algorithm. 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.

By appropriately applying the techniques discussed in this paper, an algorithm based upon KLFM can produce results better than the current state-of-the-art. In Table 10 we present the results of our algorithm (Strawman), along with results of four of the best current methods (Paraboli [Riess94], EIG1 [Hagen92], MELO [Alpert95b] and FBB [Yang94]), on a set of standard benchmarks. Those benchmarks beginning with "s" are the XNF versions of the MCNC partitioning benchmark suite [MCNC93], while the rest were obtained from Charles Alpert's benchmark set [Alpert96] in NET format and translated into XNF. All partitioners were restricted to finding partitions containing at most 55% of the logic, and all non-I/O nodes have a size of 1. 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 19% worse than ours (thus, our algorithm is 16% better). Our algorithm is also fast, taking at most 5 minutes per run on the largest examples, resulting in a total sequential runtime of at most 46 minutes.

---

[1] While it is quite surprising that in our tests higher-level gains actually degrade the quality of the basic KLFM algorithm, it appears that the effect is more of producing *different* results than worse results. Specifically, the cutsizes produced by the basic KLFM algorithm with higher-level gains vary greatly from the results produced by the basic KLFM without higher-level gains, with some much better and some much worse, while other optimizations tend to produce a more uniform increase or decrease in cutsize across all the benchmarks.

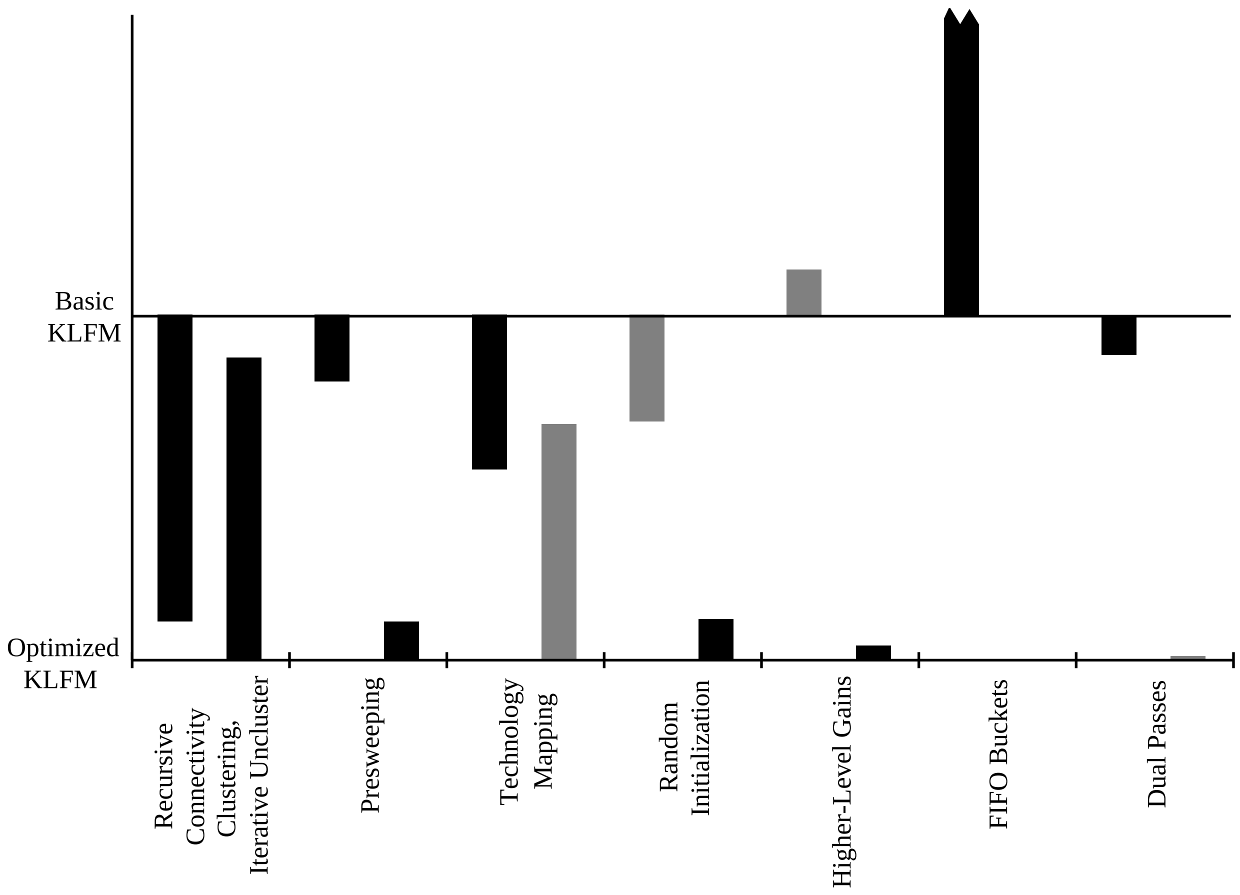

Figure 9. Graph comparing the impact of various partitioning techniques on both basic and optimized KLFM. The vertical scale is geometric mean cutsize, with the baseline being the results of optimized KLFM, and the upper line the results of basic KLFM (gate-level netlists, no clustering, random initialization, FIFO buckets, and no higher-level gains nor dual passes). The distance between these two lines is proportional to the change in cutsize between basic and optimized KLFM. The height of the bars represents the change in cutsize when the technique is included or removed, with a black bar indicating the specified technique causes an improvement, while a gray bar indicates the technique degrades the results. The left "FIFO Buckets" is cut off for readability, and is actually about 6 times as tall as shown.

This paper 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. 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.

Table 10. Quality comparison of partitioning methods. Values for basic FM and Strawman[2] are the best of ten trials. The EIG1 and MELO results are from [Alpert95b] (though EIG1 was proposed in [Hagen92]), the Paraboli results are from [Riess94], and the FBB results are from [Yang94]. All tests require partition sizes to be between 45% and 55% of the total circuit sizes, and assume that all non-I/O nodes have unit area.

Example	Nodes	Nets	Pins	FM	EIG1	Paraboli	MELO	FBB	**Strawman**	**Time**
balu	801	735	2697	30	110	41	28		**27**	**36**
s1423	831	757	2317	19	23	16		13	**14**	**37**
prim1	833	902	2908	56	75	53	64		**49**	**44**
bm1	882	903	2910	58	75		48		**49**	**42**
test04	1515	1658	5975	124	207		61		**48**	**100**
test03	1607	1618	5807	87	85		60		**55**	**84**
test02	1663	1720	6134	134	196		106		**93**	**102**
test06	1752	1641	6638	77	295		90		**60**	**119**
struct	1952	1920	5471	45	49	40	38		**33**	**58**
test05	2595	2750	10076	124	167		102		**72**	**189**
19ks	2844	3282	10547	130	179		119		**112**	**191**
prim2	3014	3029	11219	249	254	146	169		**143**	**193**
s5378	3225	3046	8241	102					**59**	**143**
s9234	6098	5870	15026	70	166	74	79	70	**42**	**250**
biomed	6514	5742	21040	135	286	135	115		**83**	**923**
s13207	9445	8776	23442	108	110	91	104	74	**57**	**594**
s15850	11071	10474	27209	170	125	91	52	67	**44**	**640**
indust2	12637	13419	48158	705	525	193	319		**188**	**1084**
indust3	15406	21923	65791	377	399	267			**256**	**1268**
s35932	19880	18152	55420	162	105	62		49	**47**	**2762**
avq.sm	21918	22124	76231	499	598	224			**131**	**1056**
s38584	22451	20999	61309	168	76	55		47	**49**	**2265**
avq.lrg	25178	25384	82751	431	571	139			**140**	**1095**
s38417	25589	23953	64299	419	121	49		58	**53**	**2661**
Norm. Geom. Mean				1.99	2.39	1.28	1.29	1.19	**1.00**	

Acknowledgments

This paper has benefited from the help of several people, including Lars Hagen, Andrew B. Kahng, D. F. Wong, and Honghua Yang. This research was funded in part by the Advanced Research Projects Agency under Contract N00014-J-91-4041. Scott Hauck was supported by an AT&T Fellowship.

2 Strawman, the optimized KLFM algorithm developed in this paper, includes recursive connectivity clustering, presweeping, per-run clustering on gate-level netlists, iterative unclustering, random initialization, FIFO bucket management, and fixed 3rd-level gains. All non-I/O nodes have unit area. Presweeping is allowed to cluster inverters (which also have a size of 1), and the clusters thus formed are the total size of all nodes clustered together.

List of References

[Alpert95a] C. J. Alpert, A. B. Kahng, "Recent Directions in Netlist Partitioning: A Survey", Integration: the VLSI Journal, Vol. 19, No. 1-2, pp. 1-81, 1995.

[Alpert95b] C. J. Alpert, S.-Z. Yao, “Spectral Partitioning: The More Eigenvectors, The Better”, Dsign Automation Conference, pp. 195-200, 1995.

[Alpert96] C. Alpert, http://ballade.cs.ucla.edu:8080/~cheese/benchmarks.html, 1996.

[Bui89] T. Bui, C. Heigham, C. Jones, T. Leighton, “Improving the Performance of the Kernighan-Lin and Simulated Annealing Graph Bisection Algorithms”, *Design Automation Conference*, pp. 775-778, 1989.

[Cheng88] C. K. Cheng, T. C. Hu, “Maximum Concurrent Flow and Minimum Ratio-cut”, *Technical Report CS88-141*, University of California, San Diego, December, 1988.

[Cong93] J. Cong, M. Smith, “A Parallel Bottom-up Clustering Algorithm with Applications to Circuit Partitioning in VLSI Design”, *Design Automation Conference*, pp. 755-760, 1993.

[Donath88] W. E. Donath, "Logic Partitioning", in *Physical Design Automation of VLSI Systems*, B. Preas, M. Lorenzetti, Editors, Menlo Park, CA: Benjamin/Cummings, pp. 65-86, 1988.

[Fiduccia82] C. M. Fiduccia, R. M. Mattheyses, "A Linear-Time Heuristic for Improved Network Partitions", *Design Automation Conference*, pp. 241-247, 1982.

[Galil86] Z. Galil, “Efficient Algorithms for Finding Maximum Matching in Graphs”, *ACM Computing Surveys*, Vol. 18, No. 1, pp. 23-38, March, 1986.

[Garbers90] J. Garbers, H. J. Prömel, A. Steger, “Finding Clusters in VLSI Circuits”, *International Conference on Computer-Aided Design*, pp. 520-523, 1990.

[Garey79] M. Garey, D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, San Francisco, CA: Freeman, 1979.

[Goldberg83] M. K. Goldberg, M. Burstein, “Heuristic Improvement Technique for Bisection of VLSI Networks”, *International Conference on Computer Design*, pp. 122-125, 1983.

[Hagen92] L. Hagen, A. B. Kahng, “New Spectral Methods for Ratio Cut Partitioning and Clustering”, *IEEE Transactions on Computer-Aided Design*, Vol. 11, No. 9, pp. 1074-1085, September, 1992.

[Hagen95] L. Hagen, J. H. Huang, A. B. Kahng, “On Implementation Choices for Iterative Improvement Partitioning Algorithms”, *European Design Automation Conference*, pp. 144-149, 1995.

[Hauck95] S. Hauck, G. Borriello, “Logic Partition Orderings for Multi-FPGA Systems”, *International Symposium on Field-Programmable Gate Arrays*, pp. 32-38, 1995.

[Kernighan70] B. W. Kernighan, S. Lin, "An Efficient Heuristic Procedure for Partitioning of Electrical Circuits", *Bell Systems Technical Journal*, Vol. 49, No. 2, pp. 291- 307, February 1970.

[Krishnamurthy84] B. Krishnamurthy, "An Improved Min-Cut Algorithm for Partitioning VLSI Networks", *IEEE Transactions on Computers*, Vol. C-33, No. 5, pp. 438-446, May 1984.

[MCNC93] MCNC Partitioning93 benchmark suite. E-mail benchmarks@mcnc.org for ftp access.

[Riess94] B. M. Riess, K. Doll, F. M. Johannes, “Partitioning Very Large Circuits Using Analytical Placement Techniques”, *Design Automation Conference*, pp. 646-651, 1994.

[Roy93] K. Roy, C. Sechen, “A Timing Driven N-Way Chip and Multi-Chip Partitioner”, *International Conference on Computer-Aided Design*, pp. 240-247, 1993.

[Schuler72] D. M. Schuler, E. G. Ulrich, “Clustering and Linear Placement”, *DAC* pp. 50-56, 1972.

[Wei89] Y.-C. Wei, C.-K. Cheng, “Towards Efficient Hierarchical Designs by Ratio Cut Partitioning”, *International Conference on Computer-Aided Design*, pp. 298-301, 1989.

[Weinmann94] U. Weinmann, “FPGA Partitioning under Timing Constraints”, in W. R. Moore, W. Luk, Eds., *More FPGAs*, Oxford: Abingdon EE&CS Books, pp. 120-128, 1994.

[Xilinx92] Xilinx, Inc., *The Programmable Gate Array Data Book*, 1992.

[Yang94] H. Yang, D. F. Wong, “Efficient Network Flow Based Min-Cut Balanced Partitioning”, *International Conference on Computer-Aided Design*, pp. 50-55, 1994.

[Yeh91] C.-W. Yeh, C.-K. Cheng, T.-T. Y. Lin, “A General Purpose Multiple Way Partitioning Algorithm”, *Design Automation Conference*, pp. 421-426, 1991.

[Yeh92] C.-W. Yeh, C.-K. Cheng, T.-T. Lin, "A Probabilistic Multicommodity-Flow Solution to Circuit Clustering Problems", *International Conference on Computer-Aided Design*, pp. 428-431, 1992.

[Yeh95] C.-W. Yeh, C.-K. Cheng, T.-T. Y. Lin, “Optimization by Iterative Improvement: An Experimental Evaluation on Two-Way Partitioning”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 14, No. 2, pp. 145-153, February 1995.