DeFer: Deferred Decision Making Enabled Fixed-Outline Floorplanner*

Jackey Z. Yan Chris Chu
Department of Electrical and Computer Engineering
Iowa State University, Ames, IA 50010
{zijunyan, cnchu}@iastate.edu

ABSTRACT

In this paper, we present *DeFer* — a fast, high-quality and non-stochastic fixed-outline floorplanning algorithm. *DeFer* generates a non-slicing floorplan by compacting a slicing floorplan. To find a good slicing floorplan, instead of searching through numerous slicing trees by simulated annealing as in traditional approaches, *DeFer* considers only one *single* slicing tree. However, we generalize the notion of slicing tree based on the principle of *Deferred Decision Making (DDM)*. When two subfloorplans are combined at each node of the generalized slicing tree, *DeFer* does not specify their orientations, the left-right/top-bottom order between them, and the slice line direction. *DeFer* even does not specify the slicing tree structures for small subfloorplans. In other words, we are deferring the decisions on these factors, which are specified arbitrarily at an early step in traditional approaches. Because of *DDM*, one slicing tree actually corresponds to a huge number of slicing floorplan solutions, all of which are efficiently kept in one *single* shape curve. With the final shape curve, it is straightforward to choose a good floorplan fitting into the fixed outline. Several techniques are also proposed to further optimize the wirelength. Experimental results on benchmarks with only hard blocks and with both hard and soft blocks show that *DeFer* achieves the *best* success rate, the *best* wirelength and the *best* runtime on average compared with other state-of-the-art floorplanners.

Categories and Subject Descriptors

B.7.2 [**Hardware, Integrated Circuits, Design Aids**]: Layout

General Terms

Algorithms, Design, Performance

Keywords

Fixed Outline, Floorplanning, Deferred Decision Making

1. INTRODUCTION

Floorplanning has become a very crucial step in modern VLSI designs. As the start of physical design flow, floorplanning not only determines the top-level spatial structure of a chip, but also initially optimizes the interconnections. Thus a good floorplan solution among circuit modules definitely has a positive impact on the placement, routing and even manufacturing. In the nanometer scale era, the ever-increasing complexity of ICs promotes the prevalence of hierarchical design. However, as pointed out by Kahng [1], classical outline-free floorplanning [2] can not satisfy such requirements of modern designs. In contrast, fixed-outline floorplanning enabling the hierarchical framework is preferred by modern ASIC designs. Nevertheless, fixed-outline floorplanning has been shown to be much more difficult, compared with classical outline-free floorplanning, even without considering the wirelength optimization [3].

1.1 Previous Work

Simulated annealing has been the most popular method of exploring good solutions on the fixed-outline floorplanning problem. Using sequence pairs representation, Adya et al. [4] modified the objective function, and proposed a few new moves based on slacks computation to guide a better local search. In [5], Chen et al. adopted the B*-tree [6] structure representing the geometric relationships among modules, and performed a novel 3-stage cooling schedule to speed up the annealing process. To improve the floorplanning scalability, in [7] a multilevel partitioning step was performed beforehand on the original circuit. Different from traditional multilevel frameworks, a top-down uncoarsing followed by a bottom-up coarsening approach were adopted. Most recently, by enumerating the positions in sequence pairs during the searching process, Chen et al. [8] applied Insertion after Remove (IAR) to accelerate the original local searching. As a result, both the runtime and success rate[1] have been enhanced dramatically. All of the above techniques are based on simulated annealing. Generally the authors tried various approaches to improve the algorithm efficiency. However, one common drawback is that these techniques become quite slow when the size of circuits grows large, e.g., 100 modules. Additionally the annealing-based techniques always have a hard time handling circuits with soft modules, because they need to search an extremely large solution space, which takes a long time to finish.

Some researchers have adopted non-stochastic methods. Sassone et al. [9] proposed a row-oriented block packing technique which organizes the modules by rows based on their dimensions. However, the technique cannot handle soft modules. In [10], Zhan et al. applied a quadratic analytical approach similar to those used for placement problems. To generate a non-overlapping floorplan, the quadratic approach relies on a legalization process. However, this legalization is very difficult for circuits with big hard macros. Cong et al. [11] presented an area-driven look-ahead floorplanner in a hi-

*This work was supported by NSF under grant CCF-0540998.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2008, June 8–13, 2008, Anaheim, California, USA
Copyright 2008 ACM 978-1-60558-115-6/08/0006…5.00

[1]The *success rate* is defined as the ratio of the number of runs resulting a layout within fixed-die, to the total number of runs.

erarchical framework. Two main techniques were used in their algorithm: the row-oriented block packing (ROB) and zero-dead space (ZDS). To handle both hard and soft modules, ROB was extended from [9]. ZDS was used to pack soft modules. Nevertheless, ROB may generate a layout with large whitespace when the module sizes within a subfloorplan are quite different from each other, e.g., a design with big hard macros.

1.2 Our Contributions

This paper presents a fast, high-quality, and non-stochastic fixed-outline floorplanner called *DeFer*. It can handle both hard and soft modules. Experimental results show that, compared with other state-of-the-art floorplanners, *DeFer* achieves the *best* success rate, the *best* wirelength and the *best* runtime on average for benchmarks with only hard blocks and with both hard and soft blocks.

DeFer generates a final non-slicing floorplan by compacting a slicing floorplan. It has been proved in [12] that any compact non-slicing floorplan can be generated by the compaction. In traditional annealing-based approaches, obtaining a good slicing floorplan usually takes a long time. Because the algorithms have to search as many slicing trees as possible, such that the "local minimum" can be possibly avoided. By comparison, *DeFer* considers only one *single* slicing tree. However, to guarantee that a large solution space is explored, we generalize the notion of slicing tree based on the principle of *Deferred Decision Making (DDM)*. When two subfloorplans are combined at each node of the generalized slicing tree, *DeFer* does not specify their orientations, the left-right/top-bottom order between them, and the slice line direction. For small subfloorplans, *DeFer* even does not specify the slicing tree structures. In other words, we are deferring the decisions on these four factors correspondingly: (1) Subfloorplan Orientation; (2) Subfloorplan Order; (3) Slice Line Direction; (4) Slicing Tree Structure. Note that traditional annealing-based approaches specify these factors arbitrarily at an early step. Because of *DDM*, one slicing tree actually represents a huge number of slicing floorplan solutions. Moreover, all of these solutions are efficiently kept by only one *single* shape curve. With the final shape curve, it is straightforward to choose a good slicing floorplan fitting into the fixed outline. To realize the *DDM* idea, we propose the following techniques:

- ***Generalized Slicing Tree*** — To defer the decisions on these three factors: (1) Subfloorplan Orientation; (2) Subfloorplan Order; (3) Slice Line Direction, we generalize the original slicing tree [2]. In the generalized slicing tree, one tree node can represent both orientations of its two child nodes, both orders between them and both horizontal and vertical slice lines. In order to carry out the according combination of such new slicing trees, original shape curve [2] operations are extended to curve *flipping* and curve *merging*. In this paper all *slicing trees* and *shape curve operations* mean the generalized version by default.

- ***Enumerative Packing*** — To defer the decision on the slicing tree structure among a set of modules, we develop the *Enumerative Packing (EP)* technique. It can enumerate all possible slicing tree structures and build up one shape curve capturing all slicing layouts among the modules. This computation could be extremely expensive in terms of CPU time and memory usage. But using the technique of dynamic programming, *EP* can be efficiently applied to up to 10 modules.

- ***Block Swapping and Mirroring*** — To make the decision on the subfloorplan order (left-right or top-bottom), we propose three techniques: *Rough Swapping*, *Detailed Swapping*, and *Mirroring*. The motivation is to greedily optimize the wirelength among the modules.

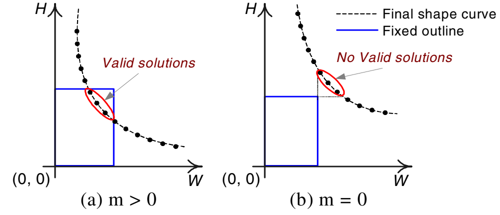


Figure 1: Final shape curve with fixed outline and candidate points.

The rest of this paper is organized as follows. Section 2 describes the overview of the algorithm. Section 3 introduces the generalized slicing tree. Section 4 describes the *Enumerative Packing* technique. Section 5 illustrates the *Block Swapping and Mirroring*. Experimental results are presented in Section 6. Finally, this paper ends with a conclusion and the direction of future work.

2. OVERVIEW OF THE ALGORITHM

Essentially, *DeFer* has five steps. By initially *deferring* the decisions in Steps 1 and 2, *DeFer* explores a huge collection of slicing layouts, all of which are efficiently kept in one final shape curve at the top; By finally *making* the decisions in Steps 3 and 4, *DeFer* chooses good slicing layouts fitting into the fixed outline. The algorithm flow is as follows.

1. **Partitioning Step:** As the number of modules in one design becomes large, exploring all slicing layout solutions among them is very expensive. Thus, the purpose of this step is to divide one original circuit into several small subcircuits, and initially minimize the interconnections among them. *hMetis* [13], the state-of-the-art hypergraph partitioner, is called to perform a recursive bi-sectioning on the circuit, until every subcircuit contains less than or equal to *maxN* modules ($maxN = 10$ by default). During this process, a high-level slicing tree structure is built up where each leaf node represents a subcircuit. Due to the generalized notion of slicing tree, the whole slicing tree not only sets up a hierarchical framework, but also represents many possible packing solutions among the subcircuits.

2. **Combining Step:** In this step, we first defer the decision on the slicing tree structure of each subcircuit, by applying the *Enumerative Packing* technique to explore all slicing packing layouts within the subcircuit. After that, an associated shape curve representing these possible layouts for each subcircuit is produced. Then, based on the hierarchical framework in Step 1, *DeFer* traverses from bottom-up constructing a shape curve for every tree node. The final shape curve at the top will hold all explored slicing floorplan layouts of the whole circuit.

3. **Back-tracing Step:** Once the final shape curve is available, it is fairly straightforward to choose the points fitting into the fixed outline (see Fig. 1). However, we have three cases here. Let m be the number of points enclosed into the fixed outline, and to make a trade-off between runtime and solution quality, *DeFer* chooses K points at most ($K = 11$ by default). (1) If $m > K$, based on the geometric observation between aspect ratio and wirelength in [8], only K points are chosen such that the aspect ratio is nearest to 1; (2) If $0 < m \leq K$, all m points are chosen; (3) If $m = 0$, *DeFer* still chooses at most K points near the upper-right corner of the fixed outline (see Fig. 1 (b)), in that we will try to compact them into

the fixed outline in Step 5. For each of the points we choose, a back-tracing process is applied. Since every point in the parent curve is generated by adding two points from two child curves, the back-tracing can be propagated from the top to the bottom level. During this process, the decisions on every subfloorplan orientation, slice line direction and slicing tree structure of each subcircuit are also made.

4. **Swapping Step:** The fourth step is to make decisions on the subfloorplan order. As pointed out in [11], in slicing structures switching the left-right or top-bottom order of two child subfloorplans would not change the dimension of their parent floorplan outline, but it may actually improve the interconnections. Different from the previous work, we execute three rounds of various wirelength refinement processes through the hierarchical framework. In the first round, we apply *Rough Swapping* technique from top-down, followed by a second round with *Detailed Swapping*. Finally, *Mirroring* is applied to further improve the wirelength and fix the order between every pair of child subfloorplans.

5. **Compacting Step:** After fixing the slicing floorplan structure, the last step is that of compacting all modules to the center of the fixed outline. The compaction can put modules nearer to each other, such that the wirelength is further reduced. The candidate floorplan with the best wirelength is the final output solution. If previous floorplan is outside of the fixed outline, instead of compacting modules to the center, *DeFer* compacts them to the lower-left corner, so that potentially there is a higher chance to find a valid layout within the fixed outline. If it still fails, then *DeFer* would restart from Step 1, and try another run. By default *DeFer* attempts 5 runs at most.

The main techniques are discussed in detail in Sections 3-5.

3. GENERALIZED SLICING TREE

In this section, we introduce the generalized slicing tree, which enables the deferred decisions on the first three factors.

In an ordinary slicing tree, the parent tree node of two child subfloorplans A and B can be labeled 'H' ('V') to specify that A and B are separated by a horizontal (vertical) slice line. In addition, the order between the two child nodes in the slicing tree specifies the top-bottom (left-right) order of A and B in the layout. For example, if A is on the left of B, then in the ordinary slicing tree, the left child is A, the right child is B, and the parent node is labeled 'V'. However, if we want to switch to other layouts between A and B, then the previous slicing tree has to be changed as well. Considering the huge amount of layout possibilities, the constraint of the ordinary representation becomes obvious.

Here we introduce a new operator — '$\oplus$' to incorporate both 'H' and 'V' slice line directions. Additionally we do not differentiate the 'top-bottom' or 'left-right' order between the two child subfloorplans any more, which means even though we put A at the left child, it can be switched to the right later on. In [14], only the orientation for each *module* has been generalized. Here, we even do not specify the orientation for each *subfloorplan*. As a result, the decisions on *Subfloorplan Orientation*, *Subfloorplan Order* and *Slice Line Direction* are deferred. Now each parent node in the slicing tree represents all *sixteen* slicing layouts between two child subfloorplans (see Fig. 2).

Since the slicing tree is only a structural representation of different layouts between two subfloorplans, to actualize the combination we use the corresponding shape curve operations, and each subfloorplan layout property is captured by its associated shape curve. In order to derive such compatible operations for the new operator —'$\oplus$', we

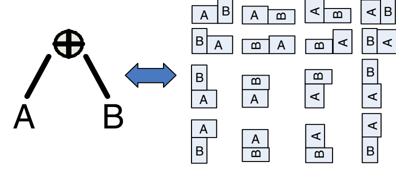
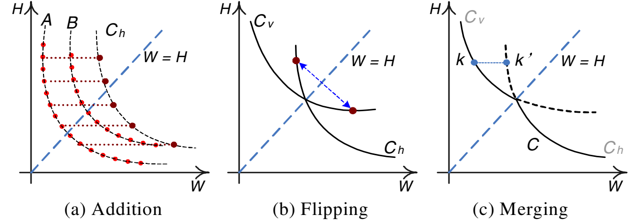


Figure 2: Generalized slicing tree and sixteen different layouts.



(a) Addition (b) Flipping (c) Merging

Figure 3: Extended shape curve operations.

develop three steps to combine two child curves A and B into one parent curve C (see Fig. 3).

1. **Addition:** Firstly, we add two curves A and B horizontally to get curve C_h, on which each point corresponds to a horizontal combination of two subfloorplan layouts from A and B.

2. **Flipping:** Next, we flip curve C_h symmetrically based on the $W = H$ line to derive curve C_v. The purpose of doing this is to generate the curve that contains the corresponding vertical combination cases from the two subfloorplan layouts.

3. **Merging:** The final step is to merge C_h and C_v into the parent curve C. In the merging, for a given height, the point with a smaller width out of C_h and C_v will be taken (see Fig. 3 (c)).

As a result, we have derived three shape curve operations which correspond to the '$\oplus$' operation in the associated slicing tree combination. Now given two child shape curves corresponding to two child subfloorplans in the slicing tree, the new operations can be applied to combine these two curves into one parent curve, in which the entire slicing layouts between the two subfloorplans are captured.

Sometimes it is not necessary to hold a huge number of points on the curve, especially at the later stage. Considering the trade-off between runtime and solution quality, in the current implementation *DeFer* keeps at most 1000 points for each shape curve. This pruning strategy makes our algorithm even faster. In spite of this, however, *DeFer* still reaches the *highest* success rate among all floorplanners.

4. ENUMERATIVE PACKING

In order to defer the decision on the slicing tree structure, we propose the *Enumerative Packing (EP)* technique that can efficiently enumerate all possible slicing structures among a set of modules, and finally keep all slicing layouts into one shape curve.

4.1 A Naive Approach of Enumeration

In this subsection, we plot out a naive way to enumerate all slicing packing solutions among n modules. We first enumerate all slicing tree structures and then enumerate all permutations of the modules. The complete slicing tree structures for 3 to 6 modules are listed in Figure 4. *Note that we are using the generalized slicing tree which does not differentiate the left-right order between two child subtrees.* From Figure 4, we notice that the number of different slicing tree structures is actually very limited.

To completely explore all slicing packing solutions among n modules, for each slicing tree structure, different permutations of the

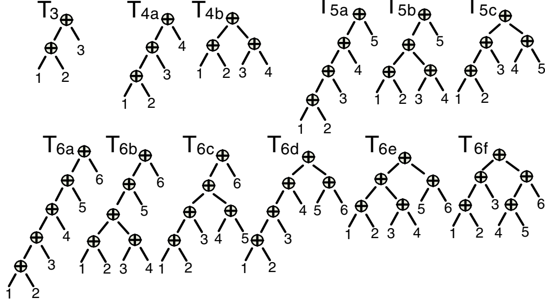


Figure 4: List of different slicing tree structures.

modules should also be considered. For example in Figure 4, in tree T_{4a} four modules A, B, C and D can be mapped to leaves "$1-2-3-4$" by the order "$A-B-C-D$" or "$A-C-B-D$". Obviously these two orders derive two different layouts. However, again because the generalized slicing tree does not differentiate the left-right order between two child subtrees which share the same parent node, for example, orders "$A-B-C-D$" and "$B-A-C-D$" are exactly the same in T_{4a}. After pruning such redundancy, we have $\frac{4!}{2} = 12$ non-redundant permutations for mapping four modules to the four leaves in T_{4a}. Therefore, for each slicing tree structure of n modules, we first enumerate all non-redundant permutations, for each one of which a shape curve is produced, and then merge these curves into one curve associated with each slicing tree structure. Finally, these curves from all slicing tree structures are merged into one curve that captures all possible slicing layouts among these n modules. To show the amount of computations in this process, we list the number of '$\oplus$' operations for different numbers of modules in the second column of Table 1.

4.2 Enumeration by Dynamic Programming

Table 1 shows that the naive approach can be extremely expensive in both runtime and memory usage. Alternatively, we notice that the shape curve for a set of modules (M) can be defined recursively by Equation 1 below.

$$S(M) = \underset{A \subset M, B = M-A}{\text{MERGE}} (S(A) \oplus S(B)) \tag{1}$$

$S(M)$ is a shape curve capturing all slicing layouts among modules in M, MERGE() is similar to the *Merging* in Figure 3 (c), but operates on shape curves from different sets. Based on Equation 1, we can use *Dynamical Programming (DP)* to implement the shape curve generation. First of all, we generate the shape curve representing the outline(s) of each module. For hard modules, there are two points[2] in each curve. For soft modules, only several points from each original curve are sampled. And then starting from the smallest subset of modules, we proceed to build up the shape curves for the larger subsets step by step, until the shape curve $S(M)$ is generated. Since in this process the previously generated curves can be reused for building up the curves of larger subsets of modules, many redundant computations are eliminated. After applying *DP*, the resulted numbers of '$\oplus$' operations are listed in the third column of Table 1.

4.3 Impact of EP on Packing

To control the quality of packing in *EP*, we can adjust the number of modules in the set. Consequently the impact on packing is: *the more modules a set contains, the more different slicing tree structures we explore, the more slicing layout possibilities we have, and thus the better quality of packing we will gain at the top level.*

[2]One point if the hard module is a square.

n	# of $\oplus$ by naive approach	# of $\oplus$ with *DP*
2	1	1
3	6	6
4	45	25
5	400	90
6	4,155	301
7	49,686	966
8	674,877	3,025
9	10,295,316	9,330
10	174,729,015	28,501

Table 1: Comparison on # of '$\oplus$' operation.

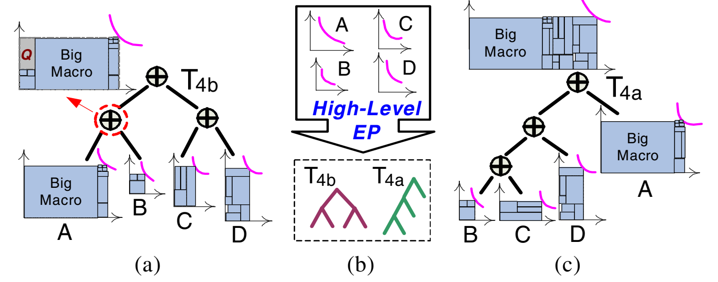


Figure 5: Illustration of high-level *EP*.

However, if the set contains too many modules, two problems appear in *EP*: 1) The memory to store results from previous sets can be expensive; 2) Since the interconnections among the modules are not considered, the wirelength may be increased. Due to these two concerns, in the first step of *DeFer*, we apply *hMetis* to recursively cut the original circuit into a bunch of smaller subcircuits. This process not only helps us to cut down the number of modules in each subcircuit, but initially optimizes the wirelength as well. Later on as applying *EP* within each subcircuit, the wirelength would not become a big concern, because this is only a locally packing exploration among a small number of modules. In other words, in the spirit of *DDM*, instead of deferring the decision on the slicing tree structure among all modules in the circuit, first we fix the high-level slicing tree structure among the subcircuits by partitioning, and then defer the decision on the slicing tree structure among the modules within each subcircuit.

4.4 Extension of EP at High-Level

In the modern SoCs design, the usage of *Intellectual Property (IP)* becomes more and more popular, which makes a circuit usually contain numbers of big hard macros. Due to the large size differences from other small or medium modules, they may produce some large deadspaces. For example in Figure 5 (a), after the partitioning step, an original circuit has been cut into four subcircuits A, B, C and D. Subcircuit A contains a big hard macro. Respecting the slicing tree structure of T_{4b}, you may find that no matter how hard *EP* explores various packing layouts within subcircuits A or B, there is always a large deadspace, such as Q, in the parent subfloorplan. This is because the high-level slicing tree structure among subcircuits has been fixed by partitioning, so that some small subcircuit is forced to combine with some large subcircuit. Thus, to solve this problem, we need to explore other slicing tree structures among the subcircuits.

To do so, in addition to applying *EP* on a set of *modules*, we use it on a set of *subfloorplans*. In Figure 5 (b), *EP* is applied on the four shape curves coming from subfloorplans A, B, C and D, respectively. Hence, all slicing tree structures (T_{4a} and T_{4b}) and permutations among these subfloorplans can be completely explored. Eventually one tightly-packed slicing layout can be chosen during the back-tracing step (see Fig. 5 (c)). For the current implementation, in the high-level slicing tree if the total area of big hard macros in one subfloorplan is more than 55% of this subfloorplan area, *DeFer*

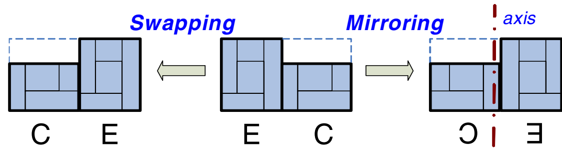


Figure 6: Swapping and mirroring.

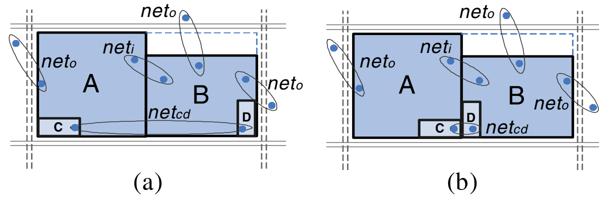


Figure 7: Illustration of motivation on *Rough Swapping*.

would apply *EP* to further explore the various slicing tree structures of that subfloorplan.

5. BLOCK SWAPPING AND MIRRORING

After the back-tracing step, the decision on subfloorplan order (left-right or top-bottom) has not been made yet. Making use of this property, this section focuses on optimizing the wirelength.

Basically, we develop three techniques here: (1) *Rough Swapping*; (2) *Detailed Swapping*; (3) *Mirroring*. Each of them is trying to switch the positions of two subfloorplans to improve the Half-Perimeter Wirelength (HPWL). Figure 6 illustrates the differences between *Swapping* and *Mirroring*. In *Mirroring*, instead of simply swapping two subfloorplans, we first figure out the symmetrical axis of the outline at their parent floorplan, and then attempt to mirror them based on this axis. When calculating the HPWL, in *Rough Swapping* we treat all internal modules to be at the center of their subfloorplan outline. In contrast, in *Detailed Swapping* we use the actual center coordinates of each module in calculating the HPWL.

Now we want to address the importance of *Rough Swapping*. For example in Figure 7, when we try to swap two subfloorplans A and B, two types of nets need to be considered: internal nets net_i between A and B, and external nets net_o between the modules inside A or B and other outside modules or fixed pads. Let C and D be two modules inside A and B, respectively. Modules C and D are highly connected by net_{cd}. After the back-tracing step, the coordinates of C and D are still unknown. If we randomly specify the positions of C and D as shown in Figure 7 (a), then we may swap A and B to gain better wirelength. Alternatively, if C and D are specified in the positions in Figure 7 (b), then we may not swap them. As we can see, the randomly specified module position may mislead us to make the wrong decision. To avoid such "noise" generated by net_i in the swapping process, the best thing to do is to assume C, D and all modules inside subfloorplans A and B are at the centers of A and B, such that the right decision can be made based on net_o.

Essentially, we first apply *Rough Swapping* to the floorplan from top-down, followed by a second round with *Detailed Swapping*. Finally, *Mirroring* is applied to further optimize the wirelength.

6. EXPERIMENTAL RESULTS

In this section, we present the experimental results. We compare *DeFer* with all the best publicly available state-of-the-art fixed-outline floorplanners. All experiments were performed on a Linux machine with Intel Core Duo[3] 1.86 GHz CPU and 2GB memory. The wirelength is calculated using HPWL. For each circuit, we choose 3 different fixed-outline aspect ratios: 1, 2 and 3 with the same maximum percentage of white space $\gamma = 10\%$. Every floorplanner runs 100 times for each test case, and the results are averaged over all *successful* runs. *Note that because PATOMA has fixed the seed of hMetis internally, and produces the same result no matter how many times it runs for the same test case, we run it only once.* For *IMF* and *DeFer*, this seed is the same as the index of each run. For each type of benchmark, we finally normalize all results to *DeFer's* results.

The first set of experiment performs on *GSRC Hard-Block Benchmarks* [15] with 100, 200 and 300 hard modules. *DeFer* compares with four floorplanners: *Parquet 4.5* [4], *IMF* [7], *IARFP* [8] and *PATOMA* [11]. All I/O pads are scaled to the boundary. *Parquet 4.5* runs in wirelength minimization mode, and the parameters for other floorplanners are defaulted. The results are summarized in Table 2. For every test case *DeFer* reaches a 100% success rate. At the same time, *DeFer* generates 39%, 16% and 17% better wirelength in $99\times$, $87\times$ and $16\times$ faster runtime than *Parquet 4.5*, *IMF* and *IARFP*, respectively. From the authors of [8], we also get a second version of *IARFP* in which the parameters are tuned specifically to individual circuits. For this version, the average success rate is close to 100%, but the total runtime and wirelength are typically worse than the first version. Compared with *PATOMA*, *DeFer* is $1.8\times$ slower. However, considering the total runtime is so *short* and *DeFer* achieves $2.25\times$ higher success rate with even 45% better wirelength, this slowdown is acceptable.

Second, we compare *DeFer* with *PATOMA* on the *HB Benchmarks* [16]. These circuits are generated from the IBM/ISPD98 suite containing both hard and soft modules ranging from 500 to 2000, some of which are big hard macros. Detailed statistics are listed in the second column of Table 3. The positions of I/O pads are the same as what the benchmark specifies. From Table 3, we can see that *DeFer* does not achieve 100% success rate for only three test cases, and the success rate is $2.22\times$ higher than *PATOMA*. In terms of the wirelength, *DeFer* is also 26% better on average, while $3.85\times$ faster than *PATOMA*. We also run *Parquet 4.5* on this benchmark. However, it is so slow that even running one test case *once* takes thousands of seconds. So for each test case, we only run it once instead of 100 times, but none of the results fit into the fixed outline. This suite of benchmarks is considered to be extremely hard to handle, because it not only contains both hard and soft modules, but also big hard macros. As far as we know, only the above three floorplanners can handle soft modules. Obviously, *DeFer* reaches the best results.

7. CONCLUSION

As the earliest stage of VLSI physical design, floorplanning has numerous impacts on the final performance of ICs. In this work, we have proposed a fast, high-quality and non-stochastic fixed-outline floorplanner *DeFer*. Based on the principle of *Deferred Decision Making*, *DeFer* over-performs all other state-of-the-art floorplanners in every aspect. Such a high-quality and efficient floorplanner is expected to handle the increasing complexity of modern ASIC designs. In the future, we will further improve the algorithm quality by refining the pruning and high-level *EP* strategies. We are also considering integrating *DeFer* into placement tools to handle large-scale mixed-size designs.

Acknowledgments

The authors would like to thank the UCLA CAD group, Song Chen, and Tung-Chieh Chen for the help with *PATOMA*, *IARFP* and *IMF*, respectively.

[3] In the experiments, only one core was used.

Circuit	Aspect Ratio	Parquet 4.5 [4]			IMF [7]			IARFP [8]			PATOMA [11]			DeFer		
		Suc%	HPWL	Time(s)	Suc%	HPWL	Time(s)	Suc%	HPWL	Time(s)	Suc%	HPWL	Time(s)	Suc%	HPWL	Time(s)
n100	1	36%	251552	10.10	100%	250680	7.62	100%	219644	3.98	0%	—	—	100%	208134	0.26
	2	37%	300782	10.45	100%	275867	9.83	97%	273423	4.24	0%	—	—	100%	228001	0.26
	3	23%	352371	11.09	100%	303861	11.23	91%	339150	4.50	0%	—	—	100%	249813	0.26
n200	1	30%	469482	42.40	100%	438467	41.17	97%	393508	6.86	0%	—	—	100%	377126	0.42
	2	19%	556552	42.01	98%	457053	43.96	71%	464706	7.26	100%	550060	0.22	100%	405436	0.39
	3	12%	647905	49.91	99%	489045	46.87	28%	562239	7.52	0%	—	—	100%	435186	0.40
n300	1	25%	683770	96.08	100%	584578	74.34	71%	549530	8.49	100%	653711	0.35	100%	501348	0.62
	2	14%	797195	91.04	100%	604471	68.35	10%	636372	8.96	100%	796725	0.32	100%	537460	0.61
	3	13%	942869	79.69	100%	638384	70.15	0%	—	—	100%	949580	0.34	100%	578879	0.60
Norm		**0.232**	**1.39**	**98.99**	**0.997**	**1.16**	**87.35**	**0.628**	**1.17**	**16.40**	**0.444**	**1.45**	**0.55**	**1**	**1**	**1**

Table 2: Comparison with other floorplanners on GSRC Hard-Block Benchmarks.

Cir ibm	#S./#H. /#N.	A.R.	PATOMA			DeFer		
			Suc%	WL(e^6)	Time(s)	Suc%	WL(e^6)	Time(s)
01	665	1	100%	2.60	8.52	100%	2.69	1.52
	/246	2	0%	—	—	100%	2.78	1.53
	/4236	3	100%	5.60	1.79	100%	2.85	1.51
02	1200	1	0%	—	—	100%	6.22	5.25
	/271	2	0%	—	—	97%	6.59	7.17
	/7652	3	0%	—	—	100%	6.39	4.73
03	999	1	100%	12.83	5.83	100%	8.86	3.40
	/290	2	100%	12.77	4.88	100%	8.68	3.43
	/7956	3	0%	—	—	100%	8.88	3.43
04	1289	1	0%	—	—	100%	9.22	4.14
	/295	2	0%	—	—	86%	9.41	8.45
	/10055	3	0%	—	—	100%	9.49	3.90
05	564	1	100%	12.49	14.00	100%	12.82	2.93
	/0	2	100%	12.97	13.45	100%	12.94	2.93
	/7887	3	100%	13.48	13.17	100%	13.62	2.93
06	571	1	0%	—	—	100%	7.80	3.07
	/178	2	0%	—	—	100%	7.69	2.96
	/7211	3	0%	—	—	100%	7.89	3.07
07	829	1	0%	—	—	100%	14.37	5.38
	/291	2	100%	25.09	7.77	100%	14.51	4.92
	/11109	3	100%	24.68	7.33	100%	14.94	4.64
08	968	1	0%	—	—	100%	14.24	6.39
	/301	2	0%	—	—	100%	14.22	6.27
	/11536	3	0%	—	—	100%	14.44	6.39
09	860	1	0%	—	—	100%	13.14	4.72
	/253	2	0%	—	—	100%	13.12	4.73
	/11008	3	100%	12.58	19.6	100%	13.33	4.74
10	809	1	100%	48.35	17.44	100%	34.34	5.51
	/786	2	0%	—	—	100%	33.80	5.68
	/16334	3	0%	—	—	100%	35.29	5.39
11	1124	1	100%	20.46	31.44	100%	22.44	7.42
	/373	2	0%	—	—	100%	22.60	7.42
	/16985	3	0%	—	—	100%	23.04	7.43
12	582	1	0%	—	—	100%	30.72	3.42
	/651	2	0%	—	—	54%	32.60	10.38
	/11873	3	0%	—	—	100%	31.74	2.65
13	530	1	0%	—	—	100%	26.86	5.43
	/424	2	100%	43.63	10.05	100%	26.93	5.18
	/14202	3	0%	—	—	100%	27.41	5.18
14	1021	1	100%	78.82	38.42	100%	51.87	8.46
	/614	2	100%	58.57	36.94	100%	52.42	8.45
	/26675	3	100%	63.48	33.22	100%	54.13	8.45
15	1019	1	0%	—	—	100%	64.39	10.22
	/393	2	0%	—	—	100%	63.98	10.23
	/28270	3	0%	—	—	100%	65.18	10.21
16	633	1	0%	—	—	100%	55.98	6.16
	/458	2	100%	88.57	15.57	100%	57.43	6.66
	/21013	3	100%	97.47	21.97	100%	58.05	6.23
17	682	1	100%	99.38	43.90	100%	96.63	8.90
	/760	2	100%	94.26	55.32	100%	96.91	8.90
	/30556	3	100%	101.23	50.79	100%	99.33	8.90
18	658	1	100%	53.78	33.00	100%	49.46	6.73
	/285	2	100%	49.04	39.10	100%	49.54	6.73
	/21191	3	100%	51.86	39.09	100%	51.25	6.73
Norm			**0.450**	**1.26**	**3.85**	**1**	**1**	**1**

Table 3: Comparison with PATOMA on HB Benchmarks.

8. REFERENCES

[1] A. B. Kahng. Classical Floorplanning Harmful? In *Proc. ISPD*, pp. 207–213, 2000.

[2] R. H. J. M. Otten. Efficient Floorplan Optimization. In *Proc. ICCD*, pp. 499–502, 1983.

[3] S. N. Adya and I. L. Markov. Fixed-outline Floorplanning Through Better Local Search. In *Proc. ICCD*, pp. 328–334, 2001.

[4] S. N. Adya and I. L. Markov. Fixed-outline Floorplanning: Enabling Hierarchical Design. In *IEEE Transactions on VLSI Systems*, vol. 11, no. 6, pp. 1120–1135, 2003.

[5] T.- C. Chen and Y.- W. Chang. Modern Floorplanning Based on Fast Simulated Annealing. In *Proc. ISPD*, pp. 104–112, 2005.

[6] Y. C. Wang, Y. W. Chang, G. M. Wu and S. W. Wu. B*-Tree: A New Representation for Non-Slicing Floorplans. In *Proc. DAC*, pp. 458–463, 2000.

[7] T.- C. Chen, Y.- W. Chang and S.- C. Lin. IMF: Interconnect-Driven Multilevel Floorplanning for Large-Scale Building-Module Designs. In *Proc. ICCAD*, pp. 159–164, 2005.

[8] S. Chen and T. Yosihmura. A Stable Fixed-Outline Floorplanning Method. In *Proc. ISPD*, pp. 119–126, 2007.

[9] P. G. Sassone and S. K. Lim. A Novel Geometric Algorithm for Fast Wire-Optimized Floorplanning. In *Proc. ICCAD*, pp. 74–80, 2003.

[10] Y. Zhan, Y. Feng and S. S. Sapatnekar. A Fixed-Die Floorplanning Algorithm Using an Analytical Approach. In *Proc. ASP-DAC*, pp. 771–776, 2006.

[11] J. Cong, M. Romesis and J. R. Shinnerl. Fast Floorplanning by Look-Ahead Enabled Recursive Bipartitioning. In *Proc. ASP-DAC*, pp. 1119–1122, 2005.

[12] M. Lai, and D. F. Wong. Slicing Tree Is a Complete Floorplan Representation. In *Proc. DATE*, pp. 228–232, 2001.

[13] G. Karypis and V. Kumar. Multilevel K-way Hypergraph Partitioning. In *Proc. DAC*, pp. 343–348, 1999.

[14] L. Stockmeyer. Optimal orientations of cells in slicing floorplan designs. In *Information and Control* , vol. 57, pp. 91–101, 1983.

[15] GSRC Floorplan Benchmarks. http://vlsicad.eecs.umich.edu/BK/GSRCbench/

[16] HB Floorplan Benchmarks. http://cadlab.cs.ucla.edu/cpmo/HBsuite.html