Fast Template Placement for Reconfigurable Computing Systems [*]

K. Bazargan R. Kastner M. Sarrafzadeh

Department of Electrical and Computer Engineering
Northwestern University
Evanston, IL 60208-3118

{kiarash,kastner,majid}@ece.nwu.edu

September 16, 1999

Abstract

The advances in the programmable hardware have lead to new architectures, where the hardware can be dynamically adapted to the application to gain better performance. One of many challenging problems in realizing a general-purpose reconfigurable system is the placement of the modules on the reconfigurable functional unit (RFU). In reconfigurable systems, we are interested both in online template placement, where arrival time of tasks is not known until runtime and offline in which the operations are scheduled at compile time. In this paper, we present online and offline heuristic template placement methods for both hard and soft templates, which can be developed internally or obtained externally (Ips). The proposed online algorithm is faster by a linear factor (about 15-30 times in practice) than the best known online algorithms, but its placement quality is comparable or slightly worse. For offline placement, we present algorithms based on simulated annealing and greedy methods and show the superiority of their placements over the ones generated by an online algorithm.

1 Introduction

As the FPGAs get larger and faster, both the number and complexity of the modules to load on them increases, hence better speedups can be achieved by exploiting FPGAs in hardware systems. Gokhale *et. al.* report speedups of 200x in [10] for the string matching problem. Adario *et. al.* [1] achieve 3 times the pipelined implementation of image processing applications

[*]This work was supported by DARPA under contract number DABT63-97-C-0035.

by exploiting dynamic reconfiguration of the hardware. Furthermore, the ability to reconfigure the chip as it is running enables the implementation of dynamically reconfigurable hardware systems which adapt themselves to the application for better performance [10, 16, 28]. Hauck has reported many applications in reconfigurable systems in [12]. Such systems usually consist of a host processor and an FPGA "co-processor" called Reconfigurable Functional Unit (RFU). The RFU can be programmed *in the course of the running time of the program* with varying configurations in different stages of the program.

An example is shown in Figure 1. As shown in Figure 1-a, three parts of the code are mapped to RFU operations (RFUOPs, also called modules). When the program is running the loop containing RFUOP2 (time *t1*), two RFUOPs are loaded on the chip. Later, when the program is about to enter the loop at time t2, there is no space on the RFU to place RFUOP3. Hence, RFUOP2 is swapped out of the chip and RFUOP3 is loaded. RFUOP1 is still on the chip and may be reused later in the program.

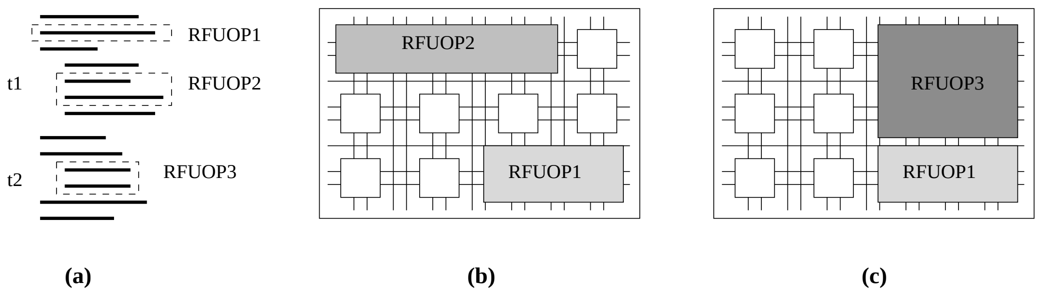


Figure 1: **(a)** The running code **(b)** RFU configuration at time t1 **(c)** RFU configuration at a later time t2

Unfortunately, rather long delays in reprogramming reconfigurable functional units (RFUs) keep us from achieving very high speedups for general purpose computing [9]. Wirthlin and Hutchings [28] report an overall speedup of 23x, while the speedup could be 80x if configuration time was zero (the configuration time is 16% to 71% of the total running time).

A number of methods have been proposed to overcome the delays in reconfiguring the RFUs. Among them are:

1. *Compiler techniques* such as prefetching [11] and configuration compression [14]. The prefetching method overlaps RFU configuration with computation. The configuration compression reduces the configuration time by transferring fewer bits to the RFU and hence decreasing the configuration time.

2. *Hardware Caching techniques* keep most frequently used operations on the RFU and hence eliminate the need for reprogramming it when such operations are called. The authors

know of no concrete results published on such methods.

Although these algorithms are necessary for a practical reconfigurable system, we still need fast and powerful physical design CAD tools to do configuration management of the RFUs both offline and online. In the offline version, the flow of the program is known in advance (e.g., in DSP applications, or loops containing basic blocks) and hence the scheduler and configuration management component can do various optimizations in the configuration of the RFU before the system starts running. On the contrary, in the online version the decision on what operations should be launched is not known beforehand. The flow of the program is not known in advance and hence the RFU configuration management should be done on the fly.

Both online and offline versions of the template placement algorithms are important for reconfigurable computing systems. The online is important since it is intrinsically hard to accurately predict the run time behavior of a general program at compile time; hence one needs online placement methods for at least parts of the RFU manager kernels. The offline algorithm can be exploited to generate compact placements for a group of RFU operations, which will execute in sequence (e.g., part of the code in a basic block). The compact placement of the group of RFU modules can be seen as one atomic module when the online placement method is running. Furthermore, placements generated by an offline method can serve as baseline solutions for the online versions, and help us devise better online algorithms. Hence, the most important feature of an offline placement algorithm is the quality of placement it generates, even though it might be a slow method.

To this date the place and route algorithms proposed for FPGAs, which are mostly modifications of the traditional algorithms for ASIC designs, are generally very slow or do not generate high quality placements. Examples are [20, 25, 22]. The only fast placement algorithm reported in the literature is a work by Callahan *et. al.* [6] which is a linear time algorithm for mapping and placement of data flow graphs on FPGAs, but limited to datapaths only. In fact, the only way to gain major speed-ups in reconfigurable computing systems is to use template placement/routing (the traditional on-the-fly synthesis/placement/routing of individual units would make the system several orders of magnitude slower).

For the online version, our goal is to devise efficient methods for placing RFU operations on the chip in a fast manner to be used in a reconfigurable computing system. In addition to being fast, such methods should be able to pack the modules on the RFU tightly to use the chip area efficiently. The effect of placing more modules on the RFU is similar to having a large data cache on a computer: it is more likely that a requested RFUOP is already on the chip and hence there is no need for reloading it.

In the case of offline placement, our goal is to find methods for placing RFU operations on the chip as compactly as possible. The offline methods can be used both as a subroutine by the online algorithm (e.g., in pre-placing operations in a basic block as a single online module) and as a baseline for assessing the quality of online methods. We propose simulated annealing as well as greedy offline algorithms for the placement of the modules on RFU, and show the effectiveness of the proposed methods by comparing their placements with those of our online version.

The rest of the paper is organized as follows: In Section 2 we have described our model of the reconfigurable system. We have also defined measures to compare effectiveness of different RFUOP placement algorithms. Section 3 deals with the online placement. The offline algorithm is presented in Section 4. Section 5 is the conclusion and suggestions for further research on the subject.

2 Our Model of a Reconfigurable Computing System

Brebner [4] suggests an environment in which the runtime system dynamically chooses between hardware (RFU operation) and software (main host CPU instructions) implementations of the same function based on profile data or other criteria. We use the same paradigm in our model. An RFUOP r_i can be either *accepted* or *rejected* based on availability of RFU real estate. If an RFUOP is rejected, the same function should be performed by the host CPU and hence a running time penalty is incurred. We use set $\mathcal{ACC}$ to represent RFUOPs which are accepted (See Equation 1).

In our model, we assume there is no communication between RFUOPs. The data to be processed by an RFUOP is loaded on the RFU before the RFUOP starts execution, and after it is done, the result is read into CPU registers (as an example for this communication scheme, see Chimaera [13] architecture). Furthermore, the RFUOP can be hard or soft module (template) either developed internally or obtained externally (IPs).

Our model which deals with the placement engine of the RFU configuration management interface, assumes that the RFUOPs have been scheduled during compile time. Furthermore, it does not consider any caching of the modules on the chip during the run-time.

The set

$$\mathcal{RFUOPS} = \{r_1, r_2, \cdots, r_n \mid r_i = (w_i, h_i, s_i, e_i)\}$$

represents all the RFU operations defined in the system, where w_i, h_i, s_i and e_i are all positive integers with the additional constraint that $s_i < e_i$. w_i and h_i are the width and height of the implementation of the RFUOP r_i in the library respectively. s_i is the time the operation r_i is invoked and $e_i - s_i$ is the time-span it is resident in the system.

The placement engine can be invoked in only two ways: *insert* a module which is not currently on the chip (at time s_i) and *delete* a currently placed module from the chip (at time e_i). If there is a cache manager in the system (See Figure 2), it will issue insertion/deletion requests to the placement engine only when such operations should actually take place. For example if an RFUOP is invoked and the cache manager detects that the module is already on the chip, it will issue no requests to the placement engine. On the other hand, if a module which was previously swapped out (placement engine had received a *delete* command on that RFUOP) and is invoked again, the cache manager will request the placement engine to insert the RFUOP as if it was the first time this RFUOP is being invoked.

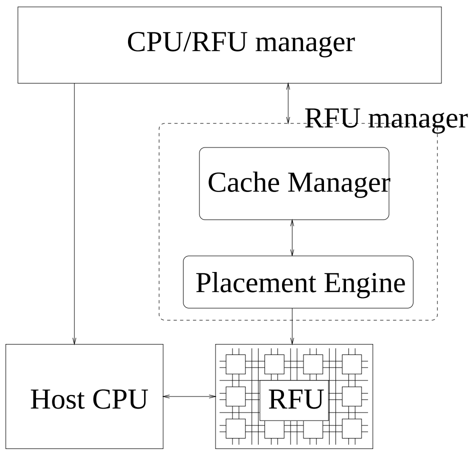


Figure 2: A sample model of a reconfigurable computing system

At any given time, there might be a number of modules on the RFU which can perform different operations concurrently. If in such a case a new RFUOP is invoked (cache manager sends an *insert* request to the placement engine) and there is no space and no idle RFUOP on the chip, then the request is *rejected*. Since the RFU cannot perform the operation, the main CPU should execute instructions to perform the same function, incurring some penalty to the running time. Otherwise (if the RFUOP is *accepted*), it is loaded onto RFU and executed. We assume that higher levels of the RFU configuration management will block insertion requests for RFUOPs which have not shown performance gains, i.e., the application profile data shows that the time to load the RFUOP plus its execution on the RFU is more than the time to perform the same function on the host CPU on the average.

The set $\mathcal{ACC}$ represents all the RFUOPs which are *accepted*, in addition to their locations on the chip. Given $\mathcal{RFUOPS}$ and RFU dimensions W and H, the placement engine decides where to place RFUOPs.

$$\mathcal{ACC} = \{(r_i, x_i, y_i) \mid \quad r_i \in \mathcal{RFUOPS},$$

$$x_i \geq 0, x_i + w_i < W$$
$$y_i \geq 0, y_i + h_i < H\} \tag{1}$$

where (x_i, y_i) is the coordinate on the RFU where RFUOP r_i is placed. Obviously, the conditions

$$W \geq w_i, \quad \forall i = 1 \cdots n$$
$$H \geq h_i, \quad \forall i = 1 \cdots n$$

must be met for all the RFUOPs. Note that the cardinality of $\mathcal{ACC}$ set could be equal to that of $\mathcal{RFUOPS}$. Also, it is important to note that the placements in $\mathcal{ACC}$ do not allow modules to be placed out of chip boundary (See Equation 1), but some RFUOP boxes might overlap. We will deal with this issue in sections 3.2 and 4.1.

The placement of RFUOPs on the RFU can be modeled as a three dimensional template placement problem. In a 3-D placement, we have a box whose base is a rectangle with the same dimensions as the RFU ($W \times H$) and its height is the time axis (See Figure 3-a). RFUOPs are also modeled as 3-D boxes (we use $box(r_i)$ to refer to the corresponding box of the RFUOP r_i). The base of the box corresponding to RFUOP r_i is a $w_i \times h_i$ rectangle and its height is the time-span the RFUOP resides on the RFU, i.e., $(e_i - s_i)$. So, the end points of the diagonal of $box(r_i)$ have coordinates (x_i, y_i, s_i) and $(x_i + w_i - 1, y_i + h_i - 1, e_i - 1)$.

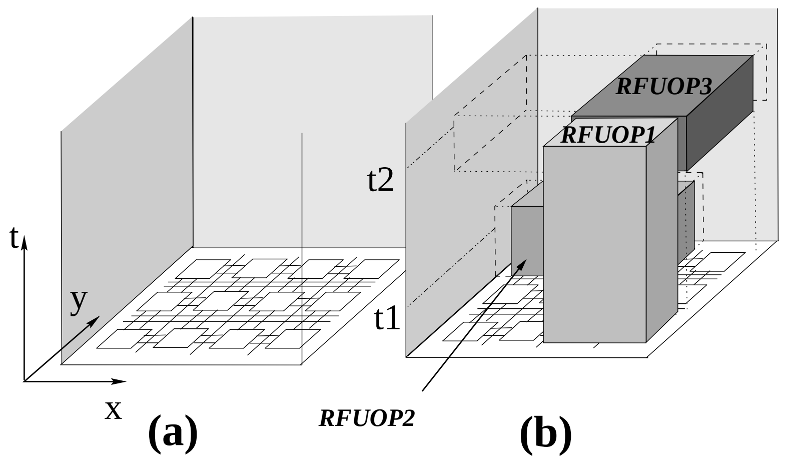


Figure 3: **(a)** The placement box **(b)** A 3-D placement

Horizontal cuts with the placement box correspond to RFU configurations at different points in time. For example, the cut $t = t1$ in Figure 3 corresponds to Figure 1-b and the cut $t = t2$ corresponds to Figure 1-c. Boxes corresponding to RFUOPs cannot be placed at any arbitrary point in the RFU box. The base of the RFUOP should be placed on the cut plane corresponding to $t = s_i$. However, the base can slide on the cut plane as long as it does not cross the chip boundary.

The penalty in rejecting an RFU operation depends both on the complexity of the operation (we assume the complexity to be linearly proportional to the size of the module implementing the RFUOP) times number of cycles the RFUOP was supposed to take on the RFU. The number of RFU cycles could be an indication of how many times (for example in a loop) the RFUOP is supposed to be executed. We can formulate the penalty of rejecting an RFUOP r_i as $penalty(r_i)$ defined as:

$$\begin{aligned} penalty(r_i) &= w_i \times h_i \times (e_i - s_i) \\ &= volume(box(t_i)) \end{aligned} \tag{2}$$

The penalty of a placement $P \in calACC$ is defined as the sum of penalties of the rejected modules:

$$Penalty(P) = \sum_{r_i \in \mathcal{RFUOPS} \text{ and } \nexists (r_i,x,y) \in P} penalty(r_i) \tag{3}$$

The overlap of a placement $P \in calACC$ is defined as the total overlapping volume of all the RFUOP boxes:

$$Overlap(P) = \sum_{(r_i,x_i,y_i),(r_j,x_j,y_j) \in P} box(r_i) \cap box(r_j) \tag{4}$$

Offline placement or 3-D template placement is the problem of finding the placement $P \in \mathcal{ACC}$ with minimum $Penalty(P)$ and the additional constraint that no two RFUOP boxes overlap, i.e.,$Overlap(P) = 0$. Online placement is similar to the offline version, but differs in the fact that at any given time, the decisions are made only on the horizontal cut at that time. In other words, the modules are processed in the time they start and the algorithm only looks at the current cut plane when deciding on whether there is room for a new module or where to place it.

3 Online Placement

This section deals with the online (two-dimensional) placement problem [3]. In Subsection 3.1, we summarize the results of the previous works on two-dimensional bin-packing and investigate their applications to the problem of placing modules on RFUs. Our method is described in Section 3.2. Experimental results for the online version are shown in Section 3.5.

3.1 Previous Work on Two-Dimensional Bin-Packing

The problem of packing RFUOPs on a chip is similar to the well-studied problem of two-dimensional bin-packing. The latter is an extension of the classical one-dimensional bin-packing (for a survey on bin-packing algorithms, refer to [18, 17]).

The one-dimensional bin-packing problem is similar to placing modules in rows of configurable logic, as done in the standard cell architecture. The two-dimensional bin-packing problem can be used when the operations to be loaded on the RFU are rectangles which can be placed anywhere on the chip.

The algorithms for the off-line version of the problem can be used when the flow of a program is known in advance (e.g., in a loop with a rather simple data flow graph). We can run an efficient off-line bin-packing algorithm to find a compact placement for a given set of RFUOPs.

Although our interest mostly lies in the two-dimensional version of the problem, we have studied the one-dimensional form as well. The reason is that the 1D form, due to its simpler nature, has been extensively studied in the past and many results have been published on the quality of the packing generated by different algorithms. Furthermore, the results are extendible to two-dimensions.

Two well known online algorithms for the one-dimensional bin-packing problem are the First Fit (FF) and Best Fit (BF) [17]. The FF algorithm puts the arriving module in the lowest indexed bin which can accommodate the module. The BF algorithm chooses the bin which has the smallest room to accommodate the module. Since both FF and BF consider all the currently used bins for placing the new module, they require $O(n)$ time for each insertion operation in the worst case, n being number of bins. In practice, FF is faster than BF. It has been shown that the quality of BF and FF is fairly close to the lower bound for online bin-packing algorithms [8, 17, 23, 26].

There are different evolutions of the BF and FF for the two-dimensional version of bin-packing or the strip packing problem reported in [8]. Among them are Next-Fit-Level, Next-Fit-Shelf, Harmonic-Shelf and Best-Fit Aligned. These algorithms have asymptotically small wasted space, but for small number of modules and bins waste considerable amount of space in order to reserve room for the coming modules. They reserve some bins at the beginning for different classes of module sizes. The interested reader is referred to the same reference for more details.

Another fast successful algorithm for strip packing is the bottom-left (BL) heuristic implemented in total quadratic time ($O(n)$ time for each insertion, where n is the number of modules currently placed) by Chazelle [7]. The algorithm generates a placement which has at most three times the optimal area in the worst case, however, the author claims much better quality in practice. The new items are placed in the lowest possible location that they fit and are slided to the left as much as they can. Chazelle's implementation preserves a property of the placement called *bottom-left stability*, which cannot be met when items can leave the system as well as arrive, hence Chazelle's implementation cannot be used in reconfigurable computing. Healy and Creavin present an algorithm in [15] with time complexity $O(n \log n)$ for each insertion.

In the next subsection we present our online method which is a generalization of the BF and FF heuristics for the two-dimensional bin-packing and has time complexity $O(\log n)$, but does not consider all candidate empty rectangles when placing a new item. In Subsection 3.5, we show how much quality loss is caused by this simplification.

3.2 Our Online Placement Method

As we mentioned in earlier sections, our goal is to devise a fast, but not necessarily optimal placement method to work as the placement engine of the architecture described in Section 2. The important question is, how much quality loss one can tolerate to get a faster method. The answer lies with the application requirements. If there are not so many modules needed on the chip simultaneously and hence we can afford wasting more space on the chip, then a fast placement algorithm is preferred.

The methods we have proposed are two-dimensional extensions of the FF, BF and BL algorithms. The generic algorithm is consisted of two parts: (a) An empty space partitioning manager both for insertion and deletion and (b) A search engine and bin-packing rule. The partitioning part divides the empty regions (sometimes referred to as "holes" in the literature) on the chip into not necessarily disjoint rectangles called "empty rectangles". Part (b) is responsible for selecting an empty rectangle to accommodate a module whose insertion is requested. All empty rectangles that the module fits in, are candidates for accommodating that module. The bin-packing rule is used to favor one over the others. Finally, the module will be placed at the lower-left corner of the selected empty rectangle.

For example the criteria could be to choose the empty rectangle with minimum area (Best Fit), or to pick the one with the lowest bottom side, breaking the tie by choosing the one with the leftmost left edge (bottom-left heuristic).

In Subsections 3.3 – 3.3.5 we describe different parts of the algorithm in more detail. Subsection 3.4 discusses the time complexity of the method.

3.3 Handling Empty Rectangles

An important part of the algorithm is the way it handles empty spaces. An *empty rectangle* is a rectangle, which does not overlap any of the modules on the chip. A *maximal empty rectangle (MER)* is an empty rectangle, which is not contained by other empty rectangles. Four maximal empty rectangles are shown in Figure 4. The top-right corner of all four is point 'A', and their bottom-left corners are 'B', 'C', 'D' and 'E'. The intersection of rectangles (B,A) and (E,A) is an example of an empty rectangle.

In the worst case, number of MERs could be quadratic in terms of number of modules. An example of such a case is shown in Figure 5. In this figure, there are $\frac{n}{2}$ MERs with 'A' as their bottom-right corners (the other corners are 'B', 'C', 'D', 'E' and 'F'), $\frac{n}{2} - 1$ with 'G' as their bottom-right corner, $\frac{n}{2} - 2$ with 'H', ... So on the whole, there are $O(n^2)$ MERs. If a placement algorithm stores all the MERs explicitly, the maximum required space would be quadratic in terms of number of modules.

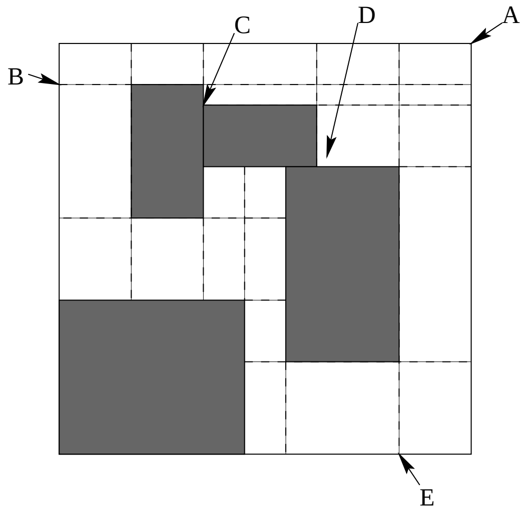


Figure 4: A placement and maximal empty rectangles (MERs). Four sample maximal empty rectangles (out of nine) are shown. The top-right corner of all four is point 'A' and the bottom-left corners of them are points 'B', 'C', 'D' and 'E'

Both [7] and [15] use doubly connected edge list (DCEL) data structure [24] (Section 1.2.3.2) to store the empty space as a set of "holes" which take linear space in terms of number of modules. The reason for the linear space complexity (versus quadratic) is that the DCEL data structure keeps the maximal empty rectangles implicitly. To obtain the list of MERs from DCEL, one has to spend linear time. Hence to find a location for a newly arrived module, one can search the DCEL list in linear time (provided a good implementation like [7]) to report all possible candidates for a bottom-left placement.

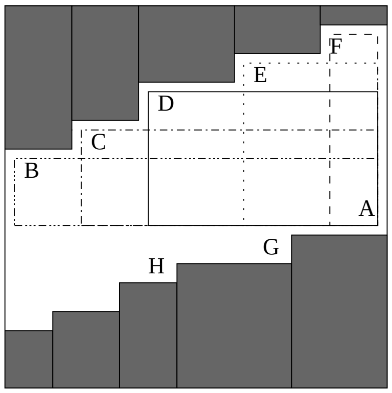


Figure 5: A placement in which number of MERs is quadratic in terms of number of modules.

We, on the contrary, keep the empty rectangles explicitly in a list. We have implemented two categories of methods: 1. Keeping all the MERs (only one implementation) and 2. Keeping disjoint empty rectangles (different implementations, each using a different heuristic). As stated earlier, the first approach takes quadratic space in terms of number of modules on the chip, while the second one needs only linear space. Since the first method keeps all the MERs and hence checks all of them for placing an arriving module, the quality of its placement is better than any method of the second category, provided that the same bin-packing rule is used (see Subsection 3.3.4). However, methods of the second category are faster. In Subsections 3.3.1 and 3.3.2 the implementations of these two categories are explained in more detail.

3.3.1 Keeping All Maximal Empty Rectangles (KAMER)

As we stated earlier, keeping all the MERs (KAMER) increases the space requirement of the algorithm by a linear factor and also slows down the insertion and deletion operations. When implementing a placement method of this type, we are obviously not looking for the fastest placement algorithm, but rather are interested in finding a baseline for comparison against faster algorithms, or use it where running time of the placement algorithm is not important. Since the KAMER algorithm considers more candidates for placing a newly arrived module, the placement it generates is superior in quality than the methods that keep only linear number of empty rectangles (provided that a good bin-packing rule is used).

Lets use an example to show how we implement the insertion operation in KAMER. Suppose we have chosen the MER with corners (A,D) to place the lightly shaded module and the module is going to be inserted at the bottom-left corner of the MER (Figure 6). Before the module is inserted, there are 15 MERs in the placement. The newly arrived module (partially) overlaps with 11 of the 15 (e.g., (A,E), (A,D), (H,C), ...). Each MER, which has some intersection with the module, should be split into smaller MERs. For example, Figure 6-(b) shows how MER (G,B) is split into four smaller MERs. In this example, the total number of MERs after insertion of the module will be 36. As seen in this example, many MERs should be checked for overlapping the module and more than one could be split after its insertion.

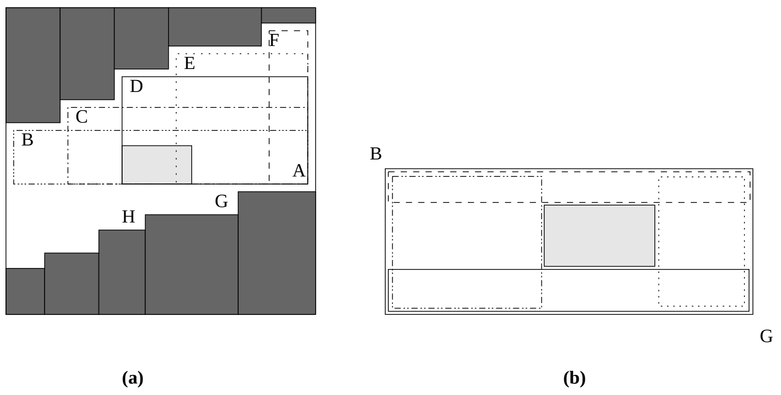


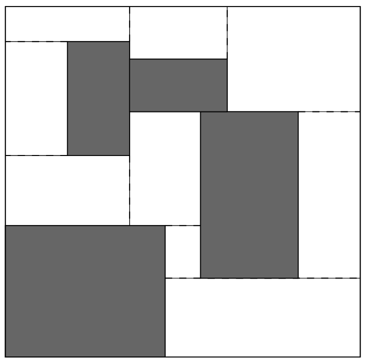
Figure 6: **(a)** The lightly shaded rectangle is going to be placed on the lower-left corner of the MER (A,D). **(b)** The way MER (G,B) will be split after insertion.

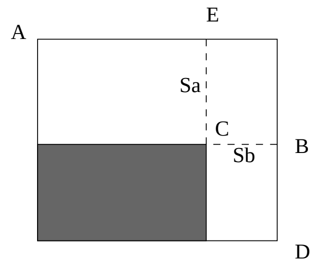
Note that if just after insertion of the new module we delete it, the deletion operation should merge 21 empty rectangles into 11 (the reverse of what we did for insertion). Since we are not trying to find a running time efficient algorithm when keeping all MERs, we can simplify the deletion operation by starting with an empty chip and inserting all the modules one by one except for the recently deleted one.

3.3.2 Keeping Non-overlapping Empty Rectangles

In order to avoid quadratic space requirement and increased running time, we can keep only linear number of empty rectangles in terms of number of modules. These empty rectangles are not necessarily maximal and hence we might lose some quality in the placement, i.e., there might actually be some room for placing a module, but the partitioning is done in a way that none of the empty rectangles has room for it. An example of non-overlapping partitioning of the empty region is shown in Figure 7-a. As an example of quality loss, suppose we have partitioned the empty space in Figure 7-b using segment S_b (assume S_a does not exist for the moment). Now, if a module whose dimensions are slightly less than those of (E,D), it can in fact be placed on the chip, but since it doesn't fit in neither of (A,B) and (C,D), our placement method rejects it.

When a new module arrives, our method searches in the list of empty rectangles for all empty rectangles which can accommodate the module. Then uses a bin- packing rule to choose


(a)



(b)

Figure 7: A placement and $O(n)$ partitioning of the empty space.

one. More details can be found in Subsection 3.3.4). Finally, the module is placed on the lower-left corner of this empty rectangle.

Since the empty rectangles are non-overlapping, only the selected empty rectangle should be split into two smaller ones. Figure 7-b shows an example. The empty rectangle can be split into two smaller ones by choosing either of the segments S_a and S_b (but not both). If S_a is selected, then the "L"-shaped region is split into empty rectangles (A,B) and (C,D), and if S_b is selected, it is split into rectangles (A,C) and (E,D). Using this scheme, one can guarantee that number of empty rectangles considered for placing each module is linear in terms of number of modules on the chip [15].

We have tried different heuristics for how to choose between the two segments. Let the two rectangles formed by choosing S_a be a_1 and a_2 and the rectangles formed by choosing S_b be b_1 and b_2. Let $W(r)$ and $H(r)$ be the width and the height of rectangle r respectively. The heuristics we have tried can be defined in terms of these values as follows:

1. *Shorter Segment (SSEG):* Choose the shorter segment of the two.

2. *Longer Segment (LSEG):* Choose the longer segment of the two.

3. *Square empty rectangles (SQR):* Let $A(r)$ be the "normalized" aspect ratio of rectangle r defined as: $A(r) = \frac{\max\{W(r),H(r)\}}{\min\{W(r),H(r)\}}$. Let $f(s)$ be the maximum normalized aspect ratio of the two rectangles formed by segment s. For example, $f(S_a) = \max\{A(a_1), A(a_2)\}$. The heuristic returns the segment with minimum $f(s)$. Intuitively, this heuristic tries to form empty rectangles which are close to squares. The reason behind favoring square empty rectangles is that for most of the modules, the square implementation has the least area

among all different rectangular shapes and hence it is more likely that the library contains more square modules than those with very high/low aspect ratios.

4. *Large square empty rectangles (LSQR):* This heuristic is similar to the previous one, except that $f(s)$ is set to the normalized aspect ratio of the larger rectangle of the two formed by segment s. Intuitively, this heuristic tries to make the larger rectangles to be close to squares. This might result in bad aspect ratios for the smaller empty rectangle.

5. *Large empty rectangles (LER):* Let $f(S_x)$ be defined as $f(S_x) = |W(x_1)H(x_1) - W(x_2)H(x_2)|$, where x_1 and x_2 are the two rectangles formed by choosing S_x. The heuristic chooses the segment which has greater $f(s)$. Intuitively, this heuristic tries to form larger empty rectangles.

6. *Balanced Empty Rectangles (BER):* Similar to the above, but chooses the segment with smaller $f(s)$.

to

3.3.3 Searching the Empty Rectangles List for Candidates

Searching the empty rectangles to find those which can accommodate the module can be done in logarithmic time using a two-dimensional layered range tree [24] (Theorem 2.12 of the same reference) which takes $O(n \log n)$ space to store the empty rectangles and $O(\log n + K)$ time to check which empty rectangles can accommodate a module, K being number of reported candidates. The interested reader is referred to [24] for more details.

3.3.4 Bin-Packing Rules

We use a cost function to choose from candidate empty rectangles for placing a new module. The lower the cost, the more favorable to put the module in that empty rectangle. Using this generic cost function, one can implement BL, FF, BF, etc. For example, by setting the cost to:

$$\text{Cost}(\mathit{emptyRect}, \mathit{module}) = \\ |\mathit{emptyRect.area} - \mathit{module.area}|$$

we can implement a 2-D extension of the Best Fit algorithm. To implement the BL algorithm:

$$\text{Cost}(\mathit{emptyRect}, \mathit{module}) = \\ \mathit{emptyRect.bottom} * \mathit{CHIP\_WIDTH} + \mathit{emptyRect.left}$$

where $CHIP\_WIDTH$ is an upper-bound for the width of the width of the empty rectangles.

Implementing the First-Fit algorithm is easier: we don't need to define the cost function. The module will be placed in the first empty rectangle which has room for it.

3.3.5 Deletions Operations

When a module is deleted, it introduces an empty rectangle on the chip. We might be able to merge this empty rectangle with neighboring empty rectangles to get larger empty rectangles. An example of this case is shown in Figure 8. Number of empty rectangles to merge is amortized constant in terms of number of modules, because number of empty rectangles at any time is linear in terms of number of modules on the chip.

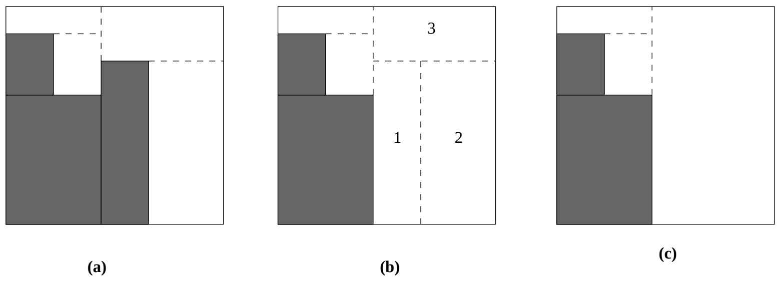


Figure 8: Merging empty rectangles after deleting a module. **(a)** The right-most module is to be deleted. **(b)** Empty rectangle 1 was inserted in place of the deleted module. **(c)** Empty rectangles 1, 2 and 3 where merged to form a larger empty rectangle.

If we do not merge the empty rectangles, the chip will be partitioned into smaller and smaller empty rectangles (because we are just splitting, not merging) which eventually will not be able to accommodate any new modules.

In order to find the empty rectangles adjacent to a given empty rectangle, we use an "adjacency graph". Each empty rectangle or module corresponds to a vertex in this graph. There is an edge between two vertices of the adjacency graph iff the corresponding empty rectangles are adjacent (part of a side of one of them overlaps with part of a side of the other one). The degree of each vertex in this graph is amortized constant since the graph is planar, which means adding or deleting a vertex in this graph takes amortized constant time.

When adding a new empty rectangle to the list of empty rectangles, we add a vertex to the adjacency graph and add the ($O(1)$ amortized) edges as well. The adjacency list of the newly created vertex is a subset of the adjacency list of its parent (the empty rectangle which was split and formed the new empty rectangle). When deleting an empty rectangle, the corresponding vertex and its edges can be deleted in amortized constant time.

If the only operation we did with the empty rectangles was the merging operation, then at the end of the process there would still be some empty rectangles left which could not be merged using the above technique. An example of such a case is shown in Figure 9-a. If we replace empty rectangles 1 and 2 with 3 and 4, as shown in Figure 9-b (i.e., use the other segment to partition the L-shaped region formed by 1 and 2), then all the empty rectangles can be merged into one which, in this example, spans the whole empty space. When adding a new empty rectangle (introduced by deleting a module), our algorithm checks the neighbors of the empty rectangle and merges it with them as many times as it can. Then it will look for neighboring empty rectangles which cannot be merged, but form an L-shaped region. Using the heuristics described in Subsection 3.3.2, the algorithm might favor to do the operation shown in the above example.

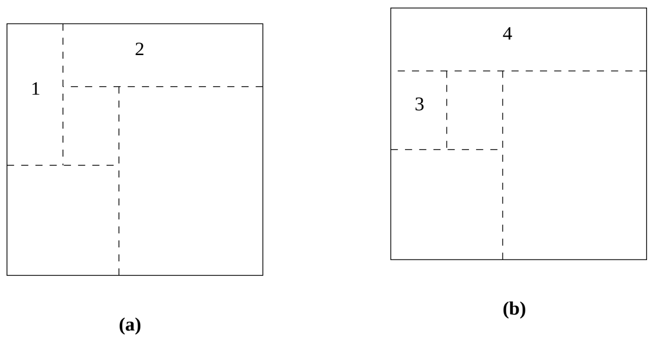


Figure 9: **(a)** Empty rectangles which cannot be merged when using the "merge adjacent rectangles" heuristic. **(b)** Rectangles 1 and 2 were switched with 3 and 4 allowing for all the empty rectangles to merge into one spanning the whole chip.

3.4 Time Complexity of the Algorithm

Here, we discuss the time complexity of the online algorithm which takes only linear (in terms of number of modules currently on the chip) number of empty rectangles. The operations done when insertion a module are looking for empty rectangles which have room for the new module, choosing one, splitting the empty rectangle and finally updating the adjacency graph. As discussed in Subsection 3.3.3, searching for empty rectangles to accommodate a new module takes logarithmic time. Since the adjacency graph of the placement is planar, updating the graph after inserting the new module takes constant time. So, on the average, it takes logarithmic time (in terms of number of modules currently on the chip) to insert a module on the RFU.

When deleting a module, we should update the adjacency graph and also do the merge operation (merge two neighboring empty rectangles to form a larger one) and possibly switch the segment used to split an L-shaped empty region into two rectangles (see Figure 9). All

Data class	Min len	Max len	Avg len	Distribution
A	3	30	16.5	Uniform
B	14	19	16.5	Uniform
C	2	40	21	Uniform
D	2	64	21	Only powers of 2

Table 1: Description of different data classes.

these operations take amortized constant time due to the fact that there are always only linear number of empty rectangles on the chip at any time. However, the insertion of the newly formed empty rectangle (after all the merging and switching operations) in the "range tree data structure" (see Subsection 3.3.3) takes logarithmic time.

So, on the whole, both insertion and deletion operations take amortized logarithmic time for each incident. On the whole, we have n modules and hence the overall time complexity of the algorithm is $O(n \log n)$.

3.5 Experimental Results for Online Placement

We have used the model described in Section 2 for our insert/delete events. We have generated different data sets containing the invocation of the RFUOPs. Each data set is a sequence of insertion and deletion of RFUOPs sorted by the time they occur. The events are uniformly distributed on the timeline with average density of 30 RFUOPs on the chip at any given time. We have simulated the running of a program on the reconfigurable computing system for different combinations of empty space partitioning (KAMER - Keep all MERs, SSEG, LSEG, SQR, LSQR, LER, BER. See Subsection 3.3.2) and bin-packing heuristics (FF, BF and BL).

With our current implementation of the placement engine, it takes about 120μsec. to place an RFUOP using SSEG-FF method, and about 2.16msec. using KAMER-BF on the average. We ran the code on a Pentium-II 130.

The data files are called Cnnnn where 'C' is the class of RFUOP module width/height distributions (one of A, B, C and D) and 'nnnn' is number of insertion events (we have done experiments with 'nnnn' being 2048, 4096, 8192 and 16384). Table 1 describes distribution of module dimensions for different classes of events. Please note that the average width/height of data classes 'A' and 'B' are the same, so are the average dimensions of 'C' and 'D' modules.

The penalty reported in the graphs is the same as what was described in Equation 2. The tables show the percentage of the accepted events as well. Subsections 3.5.1–3.5.3 present the results of different experiments.

Bin-Pack	Data set	KAMER	SSEG	BER	LSQR	LSEG	LER	SQR
FF	A2048	79.25	74.2600	61.52	70.3600	52.83	73.8700	70.36
FF	A4096	84.59	79.1000	66.84	74.3900	58.37	79.4900	74.73
FF	A8192	79.71	73.3900	63.23	69.8700	55.87	74.8800	68.11
FF	A16384	81.35	75.0800	63.59	70.4200	55.73	76.1300	69.38
BF	A2048	82.52	77.49	67.18	75.05	58.93	76.46	74.66
BF	A4096	87.06	81.76	73.22	80.32	64.57	81.66	79.78
BF	A8192	82.28	77.57	67.85	73.91	59.04	76.12	73.77
BF	A16384	84.04	78.81	68.5	75.36	60.92	78.25	75.44
BL	A2048	81.84	76.22	61.72	73.29	55.57	76.07	71.83
BL	A4096	86.18	81.93	70.29	78.56	62.33	81.42	78.54
BL	A8192	81.17	75.71	65.04	72.9	59.71	76.54	72.18
BL	A16384	83.46	77.39	64.97	74.53	58.23	78.29	73.25

Table 2: Percentage of accepted modules for different data of class 'A' for different partitioning heuristics.

3.5.1 Empty Rectangle Management Heuristics

In this subsection we report the percentage of accepted insertion events (i.e., $|\mathcal{ACC}|/|\mathcal{RFUOPS}|$) as well as penalties (i.e., $Penalty(P)$ as defined in Equation 3) for data set 'Annn' when different empty space partitioning heuristics are used. The chip size is set to 100x100 (the average total area of the modules on the chip is $30 \times 1/4 \times (3+30)^2 = 8167$).

Table 2 shows the percentage of acceptance in insertion events for different combinations of the bin-packing rules and partitioning heuristics. One can notice that the rate of acceptance is fairly constant for different sizes of the data set when a particular combination of bin-packing and partitioning heuristics is used. Figure 10 shows the penalties only for BF. Although might not be easily seen in the graph, SSEG and LSQR are the best among partitioning heuristics which keep only $O(n)$ empty rectangles. It can be seen that BER and LSEG heuristics generate very bad placements. The reason is that their placements partition the empty space into narrow strips which cannot accommodate most of the modules.

3.5.2 Different Chip Sizes

The effect of changing the chip size (equivalently, changing the average module area) on the acceptance rate is addressed in this subsection. We have simulated chip sizes 80x80, 100x100, 151x66 and 120x120. Data set A16384 is used in which the average area of the modules on the chip at any given time is 8167. Note that 100x100 and 151x66 have approximately the same area but are different in shape.

Table 3 shows the percentage of insertion events accepted. Figure 11 shows the penalties

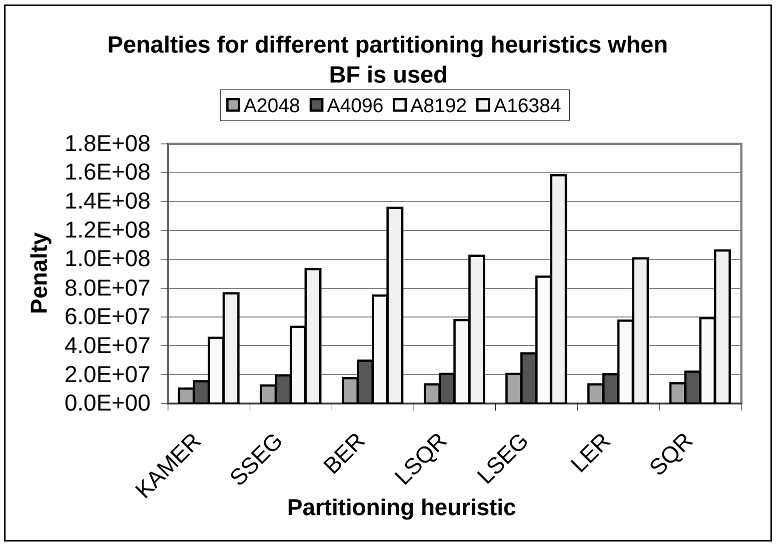


Figure 10: Penalties for different class 'A' data sets when each of the partitioning heuristics are used. The chip size is 100x100.

	FF				BF				BL			
Chip Sizes	KAMER	SSEG	LER	SQR	KAMER	SSEG	LER	SQR	KAMER	SSEG	LER	SQR
80x80	66.36	60.3	62.08	56.43	68.14	63.27	63.97	60.18	67.55	61.96	63.21	58.93
100x100	81.35	75.08	76.13	69.38	84.04	78.81	78.25	75.44	83.46	83.46	78.29	73.25
151x66	81.23	74.47	72.68	68.84	83.85	77.95	72.73	75.25	82.47	76.48	74.44	73.15
120x120	92.6	87.63	87.86	81.2	95.43	91.65	90.04	88.52	94.82	90.47	89.89	86.77

Table 3: Percentage of accepted modules for different chip sizes.

for different chip sizes when partitioning heuristics are applied to A16384 data set.

3.5.3 Different Module Sizes

The experiments in this subsection deal with the effect of different module size distributions on the acceptance rate of the insertion events. Table 4 shows the percentage of accepted insertion events for different data sets. Note that we can only compare sets A16384 to B16384 or compare sets C16384 to D16384 because their average module dimensions as well as chip sizes are the same.

		FF				BF				BL			
Data set	Chip size	KAMER	SSEG	LER	LSQR	KAMER	SSEG	LER	LSQR	KAMER	SSEG	LER	LSQR
ra16384	100x100	81.35	75.08	76.13	70.22	84.04	78.81	78.25	75.37	83.46	83.46	78.29	74.53
rb16384	100x100	81.65	78.43	73.77	73.67	82.76	80.35	73.95	76.64	82.9	79.39	74.48	74.14
rc16384	128x128	88.84	82.25	84.12	76.2	91.66	85.74	86.34	81.97	91.27	84.95	86.51	80.89
rd16384	128x128	89.61	79.7	85.42	76.45	92.08	85.5	88.75	82.76	91.78	86.54	87.62	85.38

Table 4: Percentage of insertion events accepted for different data sets.

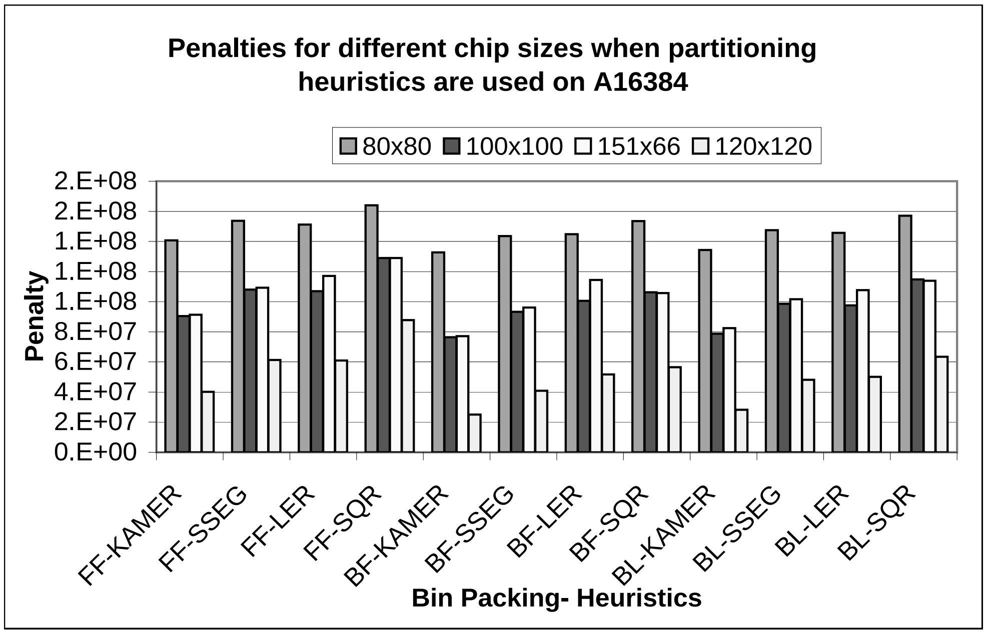


Figure 11: Penalties for different chip sizes when partitioning heuristics are applied to data set A16384.

The difference between sets A16384 and B16384 is in the variance of RFUOP dimensions. The width/height of RFUOPs of A16384 are in the range 3–30 while those of B16384 are in range 14–19. The modules in set B16384 are close to squares. Surprisingly, LSQR heuristic performs the worst (although not shown LSQR performs better than SQR) in most cases. Another counter-intuitive result is that the acceptance rate does not increase as the variance of RFUOP dimensions decrease. One would expect such a decrease because since the modules are very similar in shape, the empty rectangles are supposed to accommodate arriving modules well.

The fact that module dimensions of data set D16384 are powers of two, helped in generating better placements. However, the increase in acceptance rate is not as high as one might expect.

4 Offline Placement

This section deals with the offline placement problem [2]. Subsection 4.1 describes our 3d-placement method. Experimental results on offline placement are shown in Section 4.2.

4.1 3-D Template Placer

We implemented four different offline algorithms for the 3-D placement problem. The four methods are listed below.

1. *KAMER-BF Decreasing:* In this method, we first sort the RFUOPs based on their box volumes, and eliminate $(100 - X)\%$ smallest RFUOP boxes (X being a parameter. We tried $X = 5, 10, \cdots$). Then keeping the same temporal order as the original input, give the remaining RFUOPs (largest $X\%$ modules) as the input to our best online algorithm (i.e., *KAMER-BF*). Intuitively, we are willing to eliminate small modules to open some space for the larger ones. The reason behind eliminating the small RFUOP boxes is that, intuitively, small modules fragment the 3-D placement and block larger ones (with higher volume and hence larger penalties of rejection) from being placed.

2. *Simulated Annealing (SA):* Starting from an empty 3-D placement, use a simulated annealing method to accept or reject RFUOPs, trying to minimize the penalty of the 3-D placement.

3. *Low-temperature Annealing (LTSA):* Starting from the placement generated by *KAMER-BF Decreasing-X%* use low-temperature annealing to add/remove RFUOPs to/from $\mathcal{ACC}$ list. All RFUOPs are considered for placement (not only the $X\%$ largest placed by the online method). An RFUOP accepted by the online method might be rejected or Displaced based on the annealing decisions.

4. *Zero-temperature Annealing (ZTSA):* Starting from the placement generated by *KAMER-BF Decreasing-X%* use zero-temperature annealing to add as many $(100 - X)\%$ smallest RFUOP boxes to the $\mathcal{ACC}$ list as you can, trying to monotonically decrease the penalty of the placement. In contrast to LTSA method, the RFUOP boxes placed by the online algorithm are not removed or displaced. This method is greedy and much faster than LTSA.

The annealing core is the same for SA, LTSA and ZTSA methods. Their only difference is in the starting temperature of the annealing process. The annealing core starts with an element $P_0 \in \mathcal{ACC}$ and applies three different moves to generate other placements $P_1, P_2, \cdots$ where $P_i \in \mathcal{ACC}$ trying to minimize $Penalty(P)$. The moves are:

1. *Op1: Accept RFUOP*
   - Precondition: $r_i \in \mathcal{RFUOPS}$, $(r_i, x, y) \notin P$
   - Operation:

$$P \leftarrow P \cup \Big\{(r_i, rand(0, W - w_i),\\ rand(0, H - h_i))\Big\}$$

   where $rand(a, b)$ generates a random integer number in the range $(a, b - 1)$.

- Post-condition: $Overlap(P) = 0$

In fact, we use a more mature way than generating random coordinates to choose the location of the RFUOP box. We look for all possible empty boxes which can accommodate $box(r_i)$ (similar to what we did in the online version) and choose one randomly.

2. *Op2: Reject RFUOP*
   - Precondition: $(r_i, x_i, y_i) \in P$
   - Operation: $P \leftarrow P \setminus \{(r_i, x_i, y_i)\}$
3. *Op3: Displace RFUOP*
   - Precondition: $(r_i, x_i, y_i) \in P$
   - Operation:

$$\begin{aligned} x_{inew} &\leftarrow x_i + rand(-\delta, \delta) \\ y_{inew} &\leftarrow y_i + rand(-\delta, \delta) \\ P &\leftarrow \big(P \setminus \{(r_i, x_i, y_i)\}\big) \cup \\ &\quad\ \big\{(r_i, x_{inew}, y_{inew}))\big\} \end{aligned}$$

   - Post-condition:

$$\begin{aligned} x_{inew} \geq 0 \text{ and } x_{inew} < W - w_i \\ y_{inew} \geq 0 \text{ and } y_{inew} < H - h_i \\ Overlap(P) = 0 \end{aligned}$$

The selection of the RFUOPs to add to the $\mathcal{ACC}$ list (i.e., accept) or remove from the $\mathcal{ACC}$ list (i.e., reject) or displace is done randomly. In Section 5 we will discuss the effect of choosing RFUOPs with different probabilities. We have also discussed the effect of choosing the annealing operations with different probabilities.

$Penalty(P_i)$ is used as the cost for each placement. Note that we could have allowed overlaps between RFUOP boxes and try to resolve it towards the end of the annealing process. In that case, the cost would have been $Penalty(P) + \lambda(T) \times Overlap(P)$, where λ is an increasing function of annealing temperature T, to ensure that the overlap cost converges to zero at the end of the annealing process. We did perform experiments with this method, but the method which allows no overlaps to occur is faster. (The authors in [27] report that 2-D placement methods which allow/prevent overlaps generate placements of fairly equal qualities).

Data class	Min len	Max len	Avg len	D	Chip Size	Distribution
Tiny	3	30	16.5	5	50×50	Uniform
Small	3	30	16.5	10	70×70	Uniform
A	3	30	16.5	30	100×100	Uniform

Table 5: Description of different data classes for the offline experiemnts. D is the density (average number of RFUOPs in the system at any time-slice).

4.2 Experimental Results for Offline Placement

We use the model described in Section 2 for our insert/delete events. We generated different data sets containing the invocation of the RFUOPs. Each data set is a sequence of insertion and deletion of RFUOPs sorted by the time they occur. The events are uniformly distributed on the timeline with average density of D RFUOPs on the chip at any given time, D being a parameter of the input file. We have simulated the running of a program on the reconfigurable computing system by placing as many RFUOP boxes on the 3-D placement as we can. The modules, which we cannot place on the RFU-time volume are rejected.

The data files are called Cnnnn (see Table 5), where 'C' is the class of RFUOP module width/height distributions (one of Tiny, Small and A) and 'nnnn' is number of insertion events (we have done experiments with 'nnnn' being 50, 100, 200, 1024 and 2048). The reason we have not used the same data files as in the online case is that those files were so large for the annealing process that the program took several hours to finish.

The penalty reported in the following tables is the same as Equation 3 (sum of box volumes of rejected RFUOPs). The tables show the ratio of accepted RFUOPs to the total number of RFUOPs (i.e., $|\mathcal{ACC}|/|\mathcal{RFUOPS}|$) as well.

The experiments with different values of X for KAMER-BF Decreasing method showed that using $X < 93$ result in higher penalties than $X = 100$. In the cases where $X \geq 93$, slight improvements in the penalty of the placement was seen, and hence we did not report the results of these experiments. Also, pure annealing took long times (e.g., hours for Small100 data set) and hence we did not report the results of SA either. However, LTSA and ZTSA methods yielded good results.

Table 6 shows the ratio of accepted RFUOPs when the output of KAMER-BF Decreasing with X=100% is used as input to the low-temperature annealing method. The results of LTSA are compared to the online algorithm (KAMER-BFD with X=100). In the same table, the penalties of the two methods are also shown. As can be seen, the acceptance rate decreases in some cases but the penalty always improves. The reason is that smaller RFUOP boxes

Data Set	LTSA-100 acc. rate	Online acc. rate	Ratio
Tiny50	70	84	83.33%
Tiny100	72	83	86.75%
Small100	86	84	102.38%
Small200	81	89.5	90.50%
Small1024	84.47	84.57	99.88%
A100	87	89	97.75%

Data Set	LTSA-100 penalty	Online penalty	Ratio
Tiny50	147287	213153	69.10%
Tiny100	253566	307879	82.36%
Small100	464049	508923	91.18%
Small200	539435	612623	88.05%
Small1024	4468662	4643786	96.23%
A100	427761	456627	93.68%

Table 6: The LTSA-100 columns show the acceptance rates and penalties for different data sets when the result of KAMER-BFD with X=100% is used as input to LTSA. The online column shows the acceptance rates and penalties for KAMER-BFD with X=100

are replaced with larger ones, hence increasing number of rejected modules but decreasing the penalty. Table 7 is similar to Table 6, but X is set to 20, instead of 100. As can be seen, the LTSA method is able to improve the online results substantially.

Table 8 shows the acceptance rate and penalties for the case where KAMER-BF Decreasing with $X = 20\%$ is run first, and its placement is used as starting point for the ZTSA method. The ZTSA only accepts the RFUOPs which are not placed by the online algorithm, and hence is very fast. It can be seen that although it is a greedy method, it still can improve the results of the online method.

5 Conclusion and Future Work

We summarized the results of previous work on floorplanning for reconfigurable systems and showed why it is important to deal with both online and offline placement algorithms. For the online problem, we presented a class of fast, but not optimal and a slow but high-quality placement algorithm. For the offline problem, We devised simulated annealing and greedy placement methods for the 3-D placement of the RFUOPs and showed their effectiveness.

For the online problem, we showed that by giving up slightly on the quality (SSEG-BF in

Data Set	LTSA-20 acc. rate	Online acc. rate	Ratio
Tiny50	76	84	90.48%
Tiny100	82	83	98.79%
Small100	81	84	96.43%
Small200	85.5	89.5	95.53%
A100	81	89	91.01%

Data Set	LTSA-20 penalty	Online penalty	Ratio
Tiny50	148975	213153	69.89%
Tiny100	225603	307879	73.28%
Small100	287153	508923	56.42%
Small200	359980	612623	58.76%
A100	213036	456627	46.65%

Table 7: The LTSA-20 columns correspond to KAMER-BFD with X=20% followed by LTSA. Online columns correspond to KAMER-BFD with X=100.

Data Set	ZTSA-20 acc. rate	Online acc. rate	Ratio
Tiny50	74	84	88.09%
Tiny100	79	83	95.18%
Small100	74	84	88.09%
Small200	77	89.5	86.03%
A100	73	89	82.02%

Data Set	ZTSA-20 penalty	Online penalty	Ratio
Tiny50	149194	213153	69.99%
Tiny100	261549	307879	84.95%
Small100	486376	508923	95.57%
Small200	571716	612623	93.32%
A100	282587	456627	61.88%

Table 8: The ZTSA-20 columns correspond to KAMER-BFD with X=20% followed by ZTSA. Online columns correspond to KAMER-BFD with X=100.

comparison to KAMER-BF), one can gain about 16x speedup (138μsec. vs. 2.16msec.). Since we normally have extra RFU resources available, this trade-off would not degrade the performance. In fact, most reconfigurable computing systems utilize about 60-70% of the RFU resources.

We also showed that the variance of the module shapes affects the quality of the placement one can get. We have also done experiments in which an RFUOP has different representations in the library both in the online and the offline cases and gained quality improvements of up to 60% in penalties and 5-10% in acceptance rate. (For brevity, details of this experiment is not included. If suggested by the reviewers, it can be included in the final version).

Another important issue to be addressed is the effect of weighting different modules when choosing them for insertion/deleting into the active tasks. It would be interesting to observe how the result of our method changes if modules with smaller volumes are more likely to be removed from the active task list. The small modules probably fragment the placement box and cause rejection of larger modules and hence increase the overall penalty. Also, the effect of selecting the four annealing moves in the offline algorithm (See Section 4.1) with different probabilities should be examined.

We intend to combine our offline placement method with a scheduling algorithm to see how we can gain from the flexibility of the modules on the time axis. The offline algorithm can give estimates on the availability of RFU area, and the scheduler can use this information as the available RFU resources to schedule the operations.

References

[1] A. M. S. Adario, E. L. Roehe and S. Bampi " Dynamically Reconfigurable Architecture for Image Processing Applications". In *Design Automation Conference*, page 623, 1999.

[2] K. Bazargan, R. Kastner and M. Sarrafzadeh. "3-D Floorplanning: Simulated Annealing and Greedy Placement Methods for Reconfigurable Computing Systems ". To appear in *Rapid System Prototyping*, 1999.

[3] K. Bazargan and M. Sarrafzadeh. "Fast Online Placement for Reconfigurable Computing Systems". To appear in *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, 1999.

[4] G. Brebner. "The Swappable Logic Unit: a Paradigm for Virtual Hardware". In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, pages 77–86, 1997.

[5] J. Burns, A. Donlin, J. Hogg, S. Singh, and M. Wit. "A Dynamic Reconfiguration Run-Time System". In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, pages 66–75, 1998.

[6] T. J. Callahan, P. Chong, A. DeHon, and J. Wawrzynek. "Fast Module Mapping and Placement for Datapaths in FPGAs". In *International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, pages –, February 1998.

[7] B. Chazelle. "The Bottom-Left Bin-Packing Heuristic: An Efficient Implementation". *IEEE Transactions on Computers*, C-32(8):697–707, August 1983.

[8] J. D. Cho and M. Sarrafzadeh. "A Buffer Distribution Algorithm for High-Speed Clock Routing". In *Design Automation Conference*, pages 537–543, 1993.

[9] A. DeHon and J. Wawrzynek. "Embedded Tutorial: Reconfigurable Computing: What, Why, and Implications for Design Automation ". In *Design Automation Conference*, page 610, 1999.

[10] M. Gokhale, B. Holmes, A. Kopser, D. Kunze, D. Lopresti, S. Lucas, R. Minnich, and P. Olsen. "Splash: A Reconfigurable Linear Logic Array". In *International Conference on Parallel Processing*, pages 526–532, 1990.

[11] S. Hauck. "Configuration Prefetch for Single Context Reconfigurable Coprocessors". In *International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, pages 65–74, February 1998.

[12] S. Hauck. "The Roles of FPGAs in Reprogrammable Systems". *Proceedings of the IEEE*, 86(4):615–638, April 1998.

[13] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. "The Chimaera Reconfigurable Functional Unit". In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, pages 87–96, 1997.

[14] S. Hauck, Z. Li, and E. J. Schwabe. "Configuration Compression for the Xilinx XC6200 FPGA". In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, pages 138–146, 1998.

[15] P. Healy and M. Creavin. "An Optimal Algorithm for Rectangle Placement". In *Technical Report UL-CSIS-97-1*. Dept. of Computer Science and Information Systems, University of Limerick, Limerick, Ireland, 1997.

[16] C. Iseli and E. Sanchez. "Spyder: A Reconfigurable VLIW Processor using FPGAs". In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, pages 17–24, 1993.

[17] E. G. Coffman Jr, M. R. Garey, and D. S. Johnson. *"Chapter 2: Approximation Algorithms for Bin Packing: A Survey"*, pages 46–93. PWS Publishing Company, 20 Park Plaza, Boston, MA 02116-4324, 1997. Approximation Algorithms for NP-Hard Problems, ed. D. S. Hochbaum.

[18] E. G. Coffman Jr. and P. W. Shor. "Packings in Two Dimensions: Asymptotic Average-Case Analysis of Algorithms". *Algorithmica*, 9(3):253–277, March 1993.

[19] R. M. Karp. *Complexity of Computer Computations*, chapter "Reducibility Among Combinatorial Problems", pages 85–163. Plenum Press, 1972.

[20] H. Krupnova, C. Rabedaoro, and G. Saucier. "Synthesis and Floorplanning for Large Hierarchical FPGAs". In *Proceedings of ACM Symposium on Field-Programmable Gate Arrays (FPGA)*, pages –, February 1997.

[21] H. Liu and D.F. Wong. "Network Flow Based Circuit Partitioning for Time-Multiplexed FPGAs". In *International Conference on Computer-Aided Design*, pages –, 1998.

[22] H. Liu and D.F. Wong. "Circuit Partitioning for Dynamically Reconfigurable FPGAs". In *International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, pages –, 1999.

[23] C. Longway and R. Siferd. "A Doughnut Layout Style for Improved Switching Speed with CMOS VLSI Gates". *IEEE Journal of Solid-State Circuits*, 24(1):194–198, 1989.

[24] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, 1985.

[25] J. Shi and Dinesh Bhatia. "Performance Driven Floorplanning for FPGA Based Designs". In *Proceedings of ACM Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 112–118, February 1997.

[26] P. W. Shor. "The average-case analysis of some on-line algorithms for bin packing". *Combinatorica*, 6(2):179–200, 1986.

[27] W. J. Sun and C. Sechen. "Efficient and Effective Placement for Very Large Circuits". *IEEE Transactions on Computer Aided Design*, 14(3):349–359, March 1995.

[28] M. J. Wirthlin and B. L. Hutchings. "A Dynamic Instruction Set Computer". In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, pages 99–107, 1995.