Fast Template Placement for Reconfigurable Computing Systems

Kiarash Bazargan, Ryan Kastner, and Majid Sarrafzadeh
Northwestern University

This article presents fast online placement methods for dynamically reconfigurable systems, as well as offline 3D placement algorithms for statically reconfigurable architectures.

As FPGAs get larger and faster, both the number and complexity of the modules to load on them increase, hence better speedups can be achieved by exploiting FPGAs in hardware systems. Gokhale et al.[9] report speedups of 200 times for the string matching problem. Adario et al.[1] achieve three times the pipelined implementation of image processing applications 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 that adapt themselves to the application for better performance.[9,15,25] Hauck has reported many applications in reconfigurable systems.[11] Such systems usually consist of a host processor and an FPGA "coprocessor" called a 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 1a, 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 can be reused later in the program.

Unfortunately, rather long delays in reprogramming RFUs keep us from achieving very high speedups for general-purpose computing.[8] Wirthlin and Hutchings[25] report an overall speedup of 23 times, while the speedup could be 80 times if configuration time was zero (the configuration time is 16% to 71% of the total running time).

We 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); 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; hence, the RFU configuration management should be done on the fly. An example of such a case is multithreading, in which the flow of the code cannot be determined beforehand.

Both online and offline versions of the template placement algorithms are important for

0740-7475/00/$10.00 © 2000 IEEE

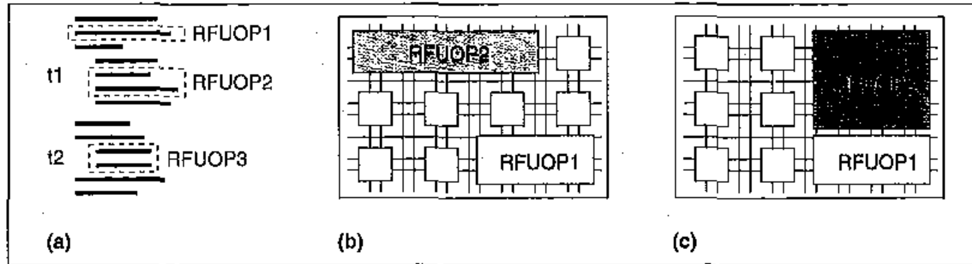

Figure 1. (a) The running code. (b) RFU configuration at time t1. (c) RFU configuration at a later time (t2).

reconfigurable computing systems. The online version is important, since it is intrinsically hard to accurately predict the runtime behavior of a general program at compile time; hence, one needs online placement methods for at least parts of the RFU manager kernels. The importance of the fast online placement methods lies in the difference between software caching and hardware caching. In software caching (the traditional data caching in programs), when a requested page is not in the cache, the penalty to load it is in terms of tens of CPU instructions. But in case of hardware caching (using a relatively small RFU to load and run many hardware modules), if an RFU operation is missing, loading configuration into a specific location on the RFU might require hundreds or thousands of cycles. The delay consists of two parts: loading the configuration bits onto the reconfigurable device and finding an appropriate location for the configuration.

The offline algorithm can be exploited to generate compact placements for a group of RFU operations that 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 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.[18,22,19,26] In fact, the only way to gain major speedups 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 runtime reconfigurable computing system. In addition to being fast, such methods should be able to tightly pack the modules on the RFU 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; hence, there is no need to reload 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 preplacing 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 article is organized as follows. First, we describe our model of the reconfigurable system. We also define measures to compare the effectiveness of different RFUOP placement algorithms. Then we discuss online placement. Next, we discuss the offline algo-

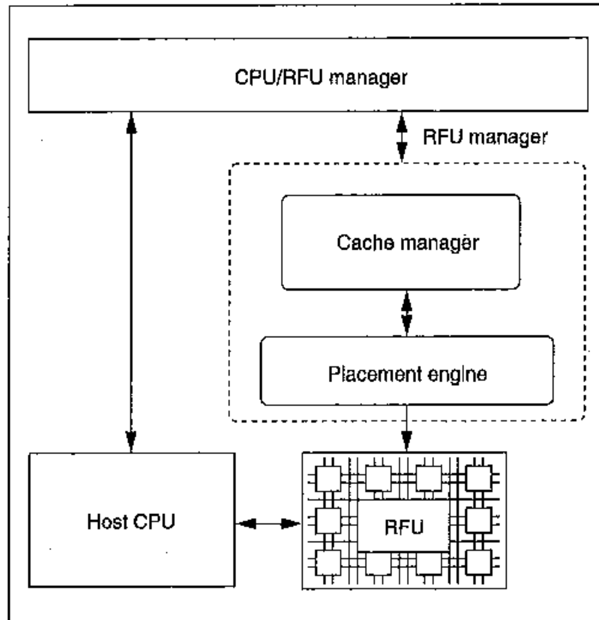

Figure 2. A sample model of a reconfigurable computing system.

rithm. Finally, we present our conclusion and suggestions for further research on the subject.

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 the availability of RFU real estate. If an RFUOP is rejected, the same function should be performed by the host CPU; hence, a running time penalty is incurred. We use set *ACC* to represent RFUOPs that are accepted (see Equation 1 below).

In our model, we assume there is no communication between RFUOPs. The data to be processed by an RFUOP are loaded on the RFU before the RFUOP starts execution. After it is done, the result is read into CPU registers (as an example of this communication scheme, see Chimaera[12] architecture). Assuming there are no significant connections between the modules, the placement problem can be solved much faster than when there are lots of wires between methods.

Furthermore, the RFUOP can be a hard or soft module (template) either developed internally or obtained externally (IPs). A hard module has fixed shape. On the other hand, a soft module has different implementations with approximately the same area, but different aspect ratios.

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 runtime.

The set

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

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 that is not currently on the chip (at time s_i) or 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 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 that was previously swapped out (placement engine had received a delete command on that RFUOP) is invoked again, the cache manager will request the placement engine to insert the RFUOP as if it were the first time this RFUOP was invoked.

At any given time, there might be a number of modules on the RFU that 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

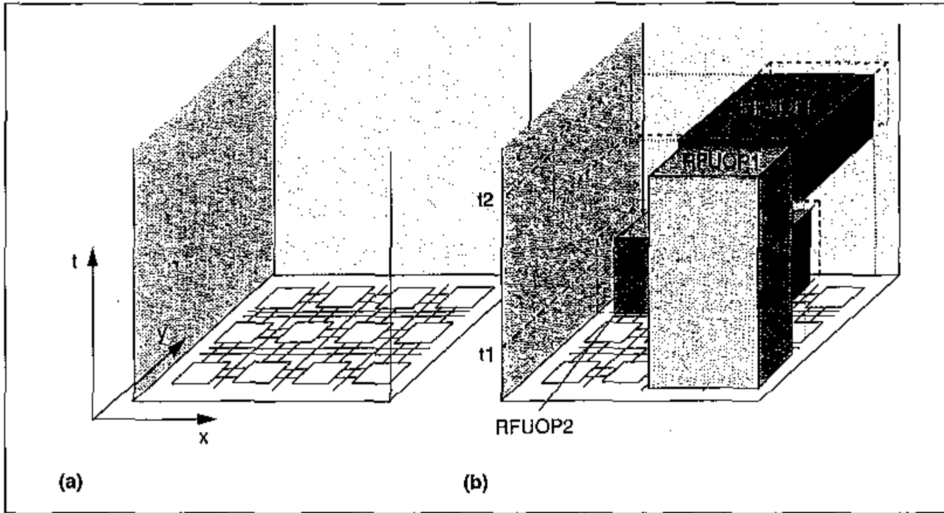

Figure 3. (a) The placement box. (b) A 3D placement.

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 that have not shown performance gains, i.e., the application profile data show 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 *ACC* represents all placements containing the RFUOPs that are accepted and their locations on the chip. Given *RFUOPS* and RFU dimensions W and H, the placement engine decides where to place RFUOPs.

$$ACC = \left\{\left\{(r_i, x_i, y_i) \mid \; r_i \in RFUOPS \atop {x_i \geq 0, x_i + w_i < W \atop y_i \geq 0, y_i + h_i < H}\right\}\right\} \quad (1)$$

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

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

must be met for all the RFUOPs. Furthermore, each placement $P \in ACC$ should contain only one entry for each RFUOP r_i. Note that the cardinality of $P \in ACC$ set could be equal to that of *RFUOPS*. Also, it is important to note that the placements in *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 below.

The placement of RFUOPs on the RFU can be modeled as a 3D template placement problem. In a 3D 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 3a). RFUOPs are also modeled as 3D 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 $(x_i + w_i - 1, y_i + h_i - 1, e_i - 1)$.

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 1b, and the cut $t = t2$ corresponds to Figure 1c. 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 for rejecting an RFU operation depends on the complexity of the operation (we assume the complexity to be linearly proportional to the size of the module implementing the RFUOP) times the 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:

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

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

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

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

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

Offline placement or 3D template placement is the problem of finding the placement $P \in ACC$ 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 looks at only the current cut plane when deciding on whether there is room for a new module or where to place it.

Online placement

This section deals with the online (2D) placement problem.[3] Below, we summarize the results of the previous works on 2D bin-packing and investigate their applications to the problem of placing modules on RFUs. Our method is also described below. Experimental results for the online version appear below as well.

Previous work on 2D bin-packing

The problem of packing RFUOPs on a chip is similar to the well-studied 2D bin-packing problem. The latter is an extension of the classical 1D bin-packing (for surveys on bin-packing algorithms, refer to Coffman and Shor[17] and Coffman et al.[16]). The 1D bin-packing problem is similar to placing modules in rows of configurable logic, as done in the standard cell VLSI architecture. The 2D bin-packing problem can be used when the operations to be loaded on the RFU are rectangles that can be placed anywhere on the chip.

Two well-known online algorithms for the 1D bin-packing problem are First Fit (FF) and Best Fit (BF).[16] The FF algorithm puts the arriving module in the lowest indexed bin that can accommodate the module. The BF algorithm chooses the bin that has the smallest room to accommodate the module (to minimize wasted space). 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 qualities of BF and FF are fairly close to the lower bound for online bin-packing algorithms.[7,16,20,23]

There are different evolutions of the BF and FF for the 2D version of bin-packing or the strip-packing problem.[7] These algorithms have asymptotically small wasted space, but for a small number of modules and bins, a considerable amount of space is wasted in order to reserve room for future modules.

Another fast successful algorithm for strip packing is Chazelle's[6] bottom-left (BL) heuristic implemented in total quadratic time: $O(n)$ time for each insertion, where n is the number of modules currently placed. 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 dynamically reconfigurable computing. Healy and Creavin[14] present an algorithm with time complexity $O(n \log n)$ for each insertion.

In the next subsection, we present our online method that is a generalization of the BF and FF heuristics for the 2D bin-packing and has time complexity $O(\log n)$, but does not consider all candidate empty rectangles when placing a new item. Below, we show how much quality loss is caused by this simplification.

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 we described above. The important question is, how much quality loss can we tolerate for a faster method? The answer lies with the application requirements. If there are not so many modules needed on the chip simultaneously (and *not* the total number of modules in the system), we can afford wasting more space on the chip. The reason is that there would probably be enough empty space on the chip for the new modules, and RFU area would not be very important. In such cases, a fast placement algorithm is preferred to a slow, but high-quality one.

The methods we have proposed are 2D extensions of the FF, BF, and BL algorithms. The generic algorithm consists of two parts: an empty space partitioning manager for insertion and deletion and a search engine and bin-packing rule. The partitioning part divides the empty region on the chip into not necessarily disjoint rectangles called "empty rectangles." The second part is responsible for selecting an empty rectangle to accommodate a module whose insertion is requested. All empty rectangles that can accommodate the module are candidates for the location of the module on the chip. 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).

Below, we describe different parts of the algorithm in more detail and the time complexity of the method.

Handling empty rectangles

An important part of the algorithm is the way it handles the empty space. An *empty rectangle* is a rectangle that does not overlap any of the modules on the chip. A *maximal empty rectangle* (MER) is an empty rectangle that is not contained by other empty rectangles. Four MERs are shown in Figure 4 (not all MERs are shown). 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.

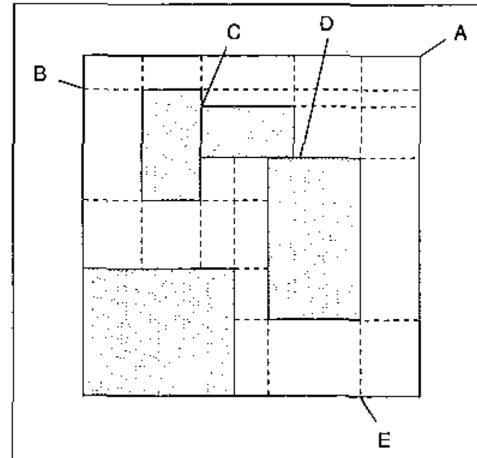

Figure 4. A placement and maximal empty rectangles (MERs).

Chazelle[6] and Healy and Creavin[14] use doubly connected edge list (DCEL) data structure[21] to store the empty space as a set of "holes" that take linear space in terms of the number of modules. The reason for the linear space complexity (versus quadratic) is that the DCEL data structure keeps the MERs 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 (using a good implementation as in Chazelle[6]) to report all possible candidates for a bottom-left placement.

On the contrary, we keep the empty rectangles explicitly in a list. We have implemented two categories of methods: keeping all the MERs (only one implementation) and keeping disjoint empty rectangles (different implementations, each using a different heuristic). As we stated earlier, the first approach takes quadratic space in terms of the 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. However, methods of the second category are

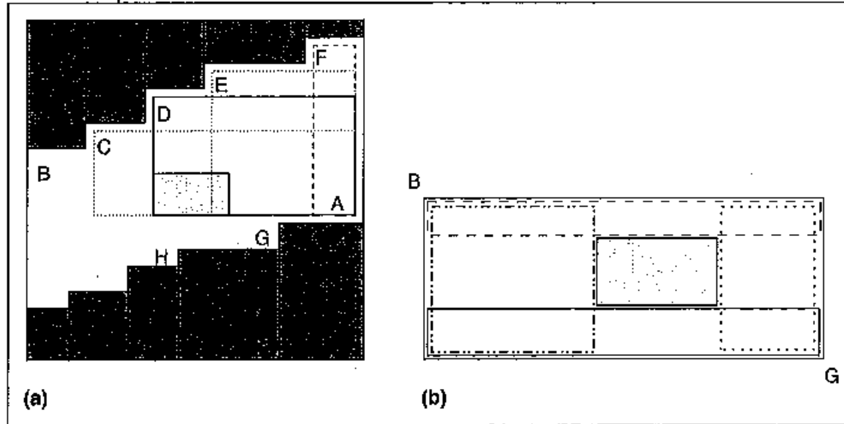

Figure 5. (a) The lightly shaded rectangle is going to be placed on the lower-left corner of the MER (A, D). (b) The way MER (G, B) will split after insertion.

faster. Next, we explain in more detail the implementations of these two categories.

Keeping all MERs. As we stated earlier, the keeping all MERs (KAMERs) algorithm 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, but rather the best-quality placement algorithm. The KAMER algorithm can be used as a baseline for comparison against faster algorithms. Since the KAMER algorithm considers more candidates for placing a newly arrived module, the placement it generates is superior in quality to the methods that keep only a linear number of empty rectangles (provided that a good bin-packing rule is used).

The following example shows how we have implemented 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 5). Before the module is inserted, there are 15 MERs in the placement. The newly arrived module overlaps, in some cases partially, with 11 of the 15 (e.g., (A, E), (A, D), (H, C), ...) MERs. Each MER, which has some intersection with the module, should split into smaller MERs. For example, Figure 5b shows how MER (G, B) splits into four smaller MERs. In this example, the total number of MERs after insertion of the module will be 36. A this example shows, many MER should be checked for overlap ping the module, and more thai one could split after insertion.

Note that if just after insertion o the new module we delete it, th deletion operation should merge 2 empty rectangles into 11 (th reverse of what we did for inse tion). Since we are not trying to fin a running time efficient algorithn when keeping all MERs, we can sir plify the deletion operation by star ing with an empty chip an inserting all the modules one by on except for the recently deleted one

Keeping nonoverlapping empty rectangles
In order to avoid the quadratic space require ment and increased running time, we can kee only linear number of empty rectangles i terms of the number of modules. These empt rectangles are not necessarily maximal, an hence we might lose some quality in the place ment. The reason for quality loss in comparison to KAMER is that each nonmaximal empty rec tangle in the linear partitioning is contained b at least one MER in the quadratic partitionin of the empty space. If a module can be place in the linear partitioning (i.e., there is at leas one non-MER large enough to contain the ne module), then it can be placed in the MER pa titioning. Obviously, the reverse is not true.

An example of nonoverlapping partitionin of the empty region is shown in Figure 6a. As a example of quality loss, suppose we have pa titioned the empty space in Figure 6b using seg ment S_b (assume S_a does not exist for th moment). Now, if there is a module whos dimensions are slightly less than those of (E, D) it can in fact be placed on the chip, but since i does not fit in (A, B) or (C, D), the placemen method rejects it.

When a new module arrives, the algorithm searches in the list of empty rectangles for al empty rectangles that can accommodate th module. Then the algorithm uses a bin-pack ing rule to choose one. More details can b found below. Finally, the module is placed o

the lower-left corner of the selected empty rectangle.

Since the empty rectangles are nonoverlapping, only the selected empty rectangle should split into two smaller ones. Figure 6b shows an example. Suppose the module (shown in the empty rectangle) is just inserted in empty rectangle (A, D). The empty rectangle can split into two smaller ones by splitting on either of the segments S_a or 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); if S_b is selected, it is split into rectangles (A, C) and (E, D). Using this scheme, one can guarantee that the number of empty rectangles considered for placing each module is linear in terms of the number of modules on the chip.[14]

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 are defined 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* (SQRs): 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 that are close to squares. The reason behind favoring SQRs is that for most of the modules, the square implementation has the least area among all different rectangular shapes, hence it is more likely that the library contains more square modules than those with very high/low aspect ratios.

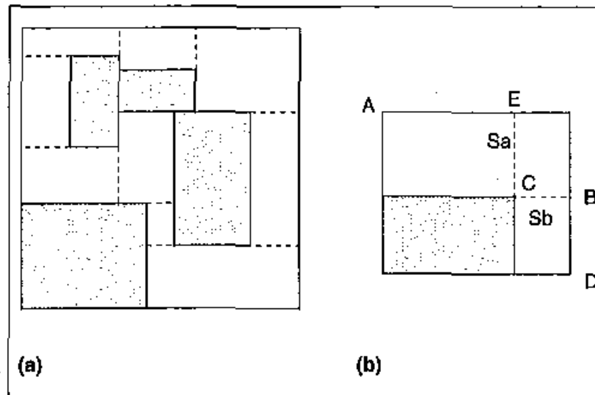

Figure 6. A placement and $O(n)$ partitioning of the empty space.

4. *Large square empty rectangles* (LSQRs): 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 a large aspect ratio for the smaller empty rectangle.

5. *Large empty rectangles* (LERs): 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 that has greater $f(s)$. Intuitively, this heuristic tries to form larger empty rectangles.

6. *Balanced empty rectangles* (BERs): Similar to the above, but chooses the segment with smaller $f(s)$.

Searching the empty rectangles list for candidates. Searching the empty rectangles to find those that can accommodate the module can be done in logarithmic time using a 2D layered range tree[21] that 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 the number of reported candidates. The interested reader is referred to Preparata and Shamos[21] for more details.

Bin-packing rules. We use a cost function to choose from candidate empty rectangles for

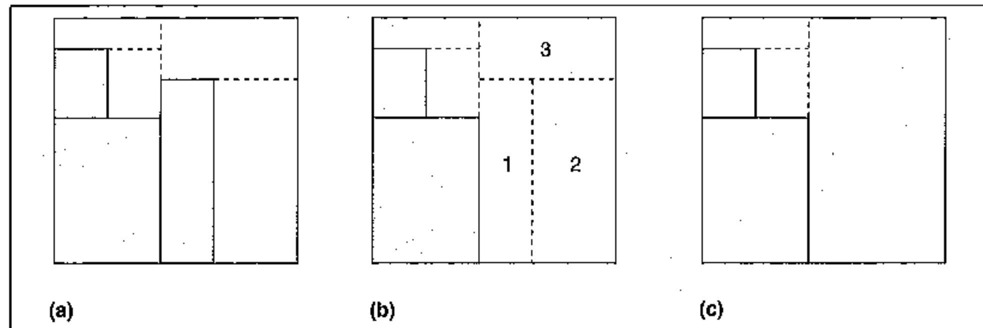

Figure 7. Merging empty rectangles after deleting a module. (a) The rightmost module is to be deleted. (b) Empty rectangle 1 was inserted in place of the deleted module. (c) Empty rectangles 1, 2, and 3 were merged to form a larger empty rectangle.

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:

Cost (*emptyRect*, *module*) =
|*emptyRect.area* – *module.area*|

we can implement a 2D extension of the BF algorithm. To implement the BL algorithm:

Cost(*emptyRect*, *module*) =
emptyRect.bottom *
CHIP_WIDTH + *emptyRect.left*

where *CHIP_WIDTH* is an upper bound for the width of the empty rectangles, *emptyRect.bottom* is the *y*-coordinate of the bottom of the empty rectangle being considered for accommodating the module, *emptyRect.left* is the *x*-coordinate of the left of the empty rectangle. The cost is, in fact, a lexicographical sort (first sort on *y*-coordinate, then on *x*-coordinate) of the bottom-left corner of the empty rectangles.

Implementing the FF algorithm is easier: We do not need to define the cost function. The module is placed in the first empty rectangle that has room for it.

Deletion 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 7. The number of empty rectangles to merge is amortized constant in terms of the number of modules, because the number of empty rectangles at any time is linear in terms of the number of modules on the chip.

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) that eventually will not be able to accommodate any new modules.

Time complexity of the algorithm

Here, we discuss the time complexity of the online algorithm that takes only linear (in terms of number of modules currently on the chip) number of empty rectangles. The operations done when inserting a module are looking for empty rectangles that have room for the new module, choosing one, splitting the empty rectangle, and finally updating the adjacency graph. As we just discussed, 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 the number of modules currently on the chip) to insert a module on the RFU.

When deleting a module, we should update the adjacency graph (used to find adjacent empty rectangles) and also do the merge operation (merge two neighboring empty rectangles to form a larger one). All these operations take amortized constant time due to the fact that there are always only a 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 above) 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, hence the overall time complexity of the algorithm is $O(n \log n)$.

Experimental results for online placement

We have used the model described above (in the section Our Model of a Reconfigurable Computing System) 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 an 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, SSEG, LSEG, SQR, LSQR, LER, and BER) and bin-packing heuristics (FF, BF, and BL). With our current implementation of the placement engine, it takes about 120 μsec to place an RFUOP using the SSEG-FF method and about 2.16 μsec using the 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 the number of insertion events (we have done experiments with nnnn being 2,048, 4,096, 8,192, and 16,384 events). Table 1 describes the 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 we described in Equation 2. The tables show the percentage of the accepted events as well.

Empty rectangle management heuristics. Here we report the percentage of accepted insertion events (i.e., $|ACC|/|RFUOPS|$) as well as penalties (i.e., $Penalty(P)$ as defined in Equation 3) for data set Annn when using different empty space partitioning heuristics. The chip size is set to 100×100 (the average total area of the modules on the chip is $30 \times 1/4 \times (3 + 30)^2 \approx 8{,}167$).

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 8 shows the penalties for only BF. Although not easily seen in the graph, SSEG and LSQR are the best among partitioning heuristics that keep only $O(n)$ empty rectangles. It is counterintuitive that BER and LSEG heuristics generate very bad placements. The reason is that their placements partition the empty space into narrow strips that cannot accommodate most of the modules.

Different chip sizes. We now address the effect of changing the chip size (equivalently, changing the average module area) on the acceptance rate. We have simulated chip sizes

Table 1. Description of different data classes.

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	Powers of 2

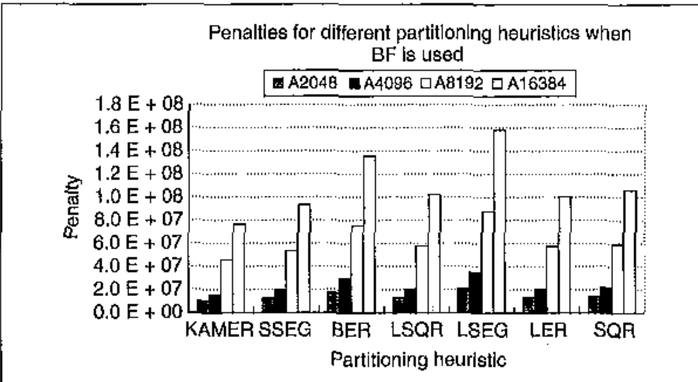

Figure 8. Penalties for different class A data sets when each of the partitioning heuristics is used. The chip size is 100 × 100.

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

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	72.95	9.71	76.54	72.18
BL	A16384	83.46	77.39	64.97	74.53	58.23	78.29	73.25

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

	FF				BF				BL			
Chip sizes	KAMER	SSEG	LER	SQR	KAMER	SSEG	LER	SQR	KAMER	SSEG	LER	SQR
80 × 80	66.36	60.3	62.08	56.43	68.14	63.27	63.97	60.18	67.55	61.96	63.21	58.93
100 × 100	81.35	75.08	76.13	69.38	84.04	78.81	78.25	75.44	83.46	83.46	78.29	73.25
151 × 66	81.23	74.47	72.68	68.84	83.85	77.95	72.73	75.25	82.47	76.48	74.44	73.15
120 × 120	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 4. Percentage of insertion events accepted for different data sets.

		FF				BF				BL			
Data set	Chip sizes	KAMER	SSEG	LER	SQR	KAMER	SSEG	LER	SQR	KAMER	SSEG	LER	SQR
ra16384	100 × 100	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	100 × 100	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	128 × 128	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	128 × 128	89.61	79.7	85.42	76.45	92.08	85.5	88.75	82.76	91.78	86.54	87.62	85.38

80 × 80, 100 × 100, 151 × 66, and 120 × 120. Data set A16384 is used in which the average area of the modules on the chip at any given time is 8,167. Note that 100 × 100 and 151 × 66 have approximately the same area but are different in shape.

Table 3 shows the percentage of insertion events accepted. Figure 9 shows the penalties for different chip sizes when partitioning heuristics are applied to A16384 data set. The shape of the RFU is not as important as area. Although the results for other data sets are not shown, they follow the same trend.

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 compare only set A16384 to B16384 or set C16384 to D16384 because their average module dimensions as well as chip sizes are the same.

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 the

range 14–19. The modules in set B16384 are close to squares. Surprisingly, the LSQR heuristic performs the worst (although not shown, LSQR performs better than SQR) in most cases. Another counterintuitive result is that the acceptance rate does not increase as the variance of RFUOP dimensions decreases. One would expect such a decrease; since the modules are very similar in shape, the empty rectangles are supposed to accommodate well the arriving modules.

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

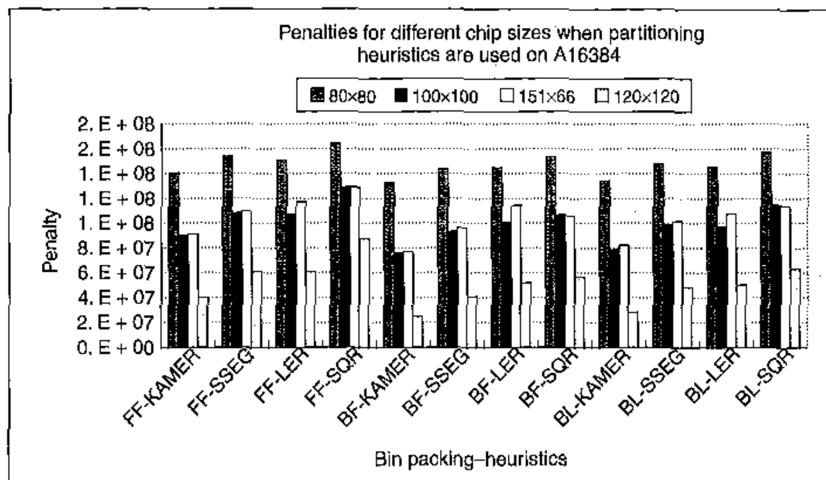

Figure 9. Penalties for different chip sizes when partitioning heuristics are applied to data set A16384.

Offline placement

This section deals with the offline placement problem.[2] First, we describe our 3D placement method. Then we give the experimental results for offline placement.

3D template placer

We implemented four different offline algorithms for the 3D 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 is a parameter. We tried $X = 5, 10, \ldots$) 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 3D 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 3D placement, use a simulated annealing method to accept or reject RFUOPs, trying to minimize the penalty of the 3D 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 the *ACC* list. All RFUOPs are considered for placement (not just 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 *ACC* list as you can, trying to monotonically decrease the penalty of the placement. In contrast to the 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 ele-

Table 5. Comparison between LTSA-100 and KAMER-BFD.

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%

ment $P_0 \in ACC$ and applies three different moves to generate other placements $P_1, P_2, \ldots$, where $P_i \in ACC$ trying to minimize $Penalty(P)$. The moves are:

1. *Op1: Accept RFUOP* moves an *RFUOPS*, r_i to the *P* set, avoiding overlaps. We look for all possible empty boxes that can accommodate $box(r_i)$ (similar to what we did in the online version) and choose one randomly.

2. *Op2: Reject RFUOP* deletes an *RFUOPS*, r_i from *P*.

3. *Op3: Displace RFUOP* changes the location of an *RFUOPS* randomly, avoiding overlaps.

$Penalty(P_i)$ is used as the cost for each placement. Note that we could have allowed overlaps between RFUOP boxes and try to resolve it toward 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 that allows no overlaps to occur is faster. (Sun and Sechen[24] report that 2D placement methods that allow/prevent overlaps generate placements of fairly equal qualities).

Experimental results for offline placement

We use the model we described earlier 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 an average density of D RFUOPs on the chip at any given time, D being between five and 30. We have simulated the running of a program on the reconfigurable computing system by placing as many RFUOP boxes on the 3D placement as we can. The modules we cannot place on the RFU-time volume are rejected.

For the experiments, we used smaller input files than for the online experiments. 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 in 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., $|P|/|RFUOPS|$) as well.

The experiments with different values of X for the KAMER-BF decreasing method showed that using $X < 93$ results 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 the Small 100 data set), and hence we did not report the results of SA either. However, LTSA and ZTSA methods yielded good results.

Table 5 shows the ratio of accepted RFUOPs when the output of KAMER-BF decreasing with $X = 100$ is used as the 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 are replaced with larger ones, hence increasing the number of rejected modules but decreasing the penalty. For more-detailed experiments, refer to Bazargan et al.[26]

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 3D 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 comparison to KAMER-BF), one can gain about 16 times the speedup (138 μsec versus 2.16 msec). 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% to 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, which is the case with soft modules, both in the online and the offline cases and gained quality improvements of up to 60% in penalties and 5% to 10% in acceptance rate. In these experiments, we modified the algorithm to try all shapes of a module on insertion and pick the one with minimum wasted area. Slightly worse results would be achieved if the algorithm uses the first shape that can be placed (and not go through all available shapes to pick the best). Details of these experiments are not included in the article for brevity.

Another important issue to be addressed is the effect of weighting different modules when choosing them for insertion into or deletion from the active tasks. Currently, our online method follows the temporal insertion/deletion requests from cache manager and adds modules to the active task list if there is room for them. A look-ahead scheme might avoid inserting a small module (even though it has room for it) to avoid fragmentation of the space. The small modules probably fragment the placement box and cause rejection of larger modules and hence increase the overall penalty. Our current implementation of the offline method treats all modules equally when choosing one for insertion into or deletion from the active task list.

It would be interesting to observe how the result of placement changes if modules with smaller volumes are more likely to be removed from the active task list. Also, the effect of selecting the four annealing moves in the offline algorithm 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 of the availability of RFU area, and the scheduler can use this information as the available RFU resources to schedule the operations. ■

Acknowledgment

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

References

1. A.M.S. Adario, E.L. Roehe, and S. Bampi, "Dynamically Reconfigurable Architecture for Image Processing Applications," *Design Automation Conf.*, pp. 623-628, 1999.
2. K. Bazargan, S. Kim, and M. Sarrafzadeh, "Nostradamus: A Floorplanner of Uncertain Designs," *IEEE Trans. Computer Aided Design*, pp. 389-397, 1999.
3. K. Bazargan and M. Sarrafzadeh, "Fast Online Placement for Reconfigurable Computing Systems," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, pp. 300-302, 1999.
4. G. Brebner, "The Swappable Logic Unit: A Paradigm for Virtual Hardware," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, pp. 77-86, 1997.
5. T.J. Callahan, P. Chong, A. DeHon, and J. Wawrzynek, "Fast Module Mapping and Placement for Datapaths in FPGAs," *Int'l ACM/SIGDA Symp. Field Programmable Gate*

Arrays, Feb. 1998.
6. B. Chazelle, "The Bottom-Left Bin-Packing Heuristic: An Efficient Implementation," *IEEE Trans. Computers*, vol. 32, no. 8, pp. 697-707, Aug. 1983.
7. J.D. Cho and M. Sarrafzadeh, "A Buffer Distribution Algorithm for High-Speed Clock Routing," *Design Automation Conf.*, pp. 537-543, 1993.
8. A. DeHon and J. Wawrzynek, "Embedded Tutorial: Reconfigurable Computing: What, Why, and Implications for Design Automation," *Design Automation Conf.*, pp. 610-615, 1999.
9. M. Gokhale, B. Holmes, A. Kopser, D. Kunze, D. Lopresti, S. Lucas, R. Minnich, and P. Olsen, "Splash: A Reconfigurable Linear Logic Array," *Int'l Conf. Parallel Processing*, pp. 526-532, 1990.
10. S. Hauck, "Configuration Prefetch for Single Context Reconfigurable Coprocessors," *Int'l ACM/SIGDA Symp. Field Programmable Gate Arrays*, pp. 65-74, Feb. 1998.
11. S. Hauck, "The Roles of FPGAs in Reprogrammable Systems," *Proc. IEEE*, vol. 86, no. 4, pp. 615-638, Apr. 1998.
12. S. Hauck, T.W. Fry, M.M. Hosler, and J.P. Kao, "The Chimaera Reconfigurable Functional Unit," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, pp. 87-96, 1997.
13. S. Hauck, Z. Li, and E.J. Schwabe, "Configuration Compression for the Xilinx XC6200 FPGA," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, pp. 138-146, 1998.
14. P. Healy and M. Creavin, "An Optimal Algorithm for Rectangle Placement," Technical Report UL-CSIS-97-1, Dept. of Computer Science and Information Systems, Univ. of Limerick, Limerick, Ireland, 1997.
15. C. Iseli and E. Sanchez, "Spyder: A Reconfigurable VLIW Processor Using FPGAs," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, pp. 17-24, 1993.
16. E.G. Coffman, Jr., M.R. Garey, and D.S. Johnson, "Approximation Algorithms for Bin Packing: A Survey," D.S. Hochbaum, ed., *Approximation Algorithms for NP-Hard Problems*. Boston: PWS Publishing Company, 1997, pp. 46-93.
17. E.G. Coffman, Jr., and P.W. Shor, "Packings in Two Dimensions: Asymptotic Average-Case Analysis of Algorithms," *Algorithmica*, vol. 9, no. 3, pp. 253-277, Mar. 1993.
18. H. Krupnova, C. Rabedaoro, and G. Saucier, "Synthesis and Floorplanning for Large Hierarchical FPGAs," *Proc. ACM Symp. Field-Programmable Gate Arrays*, Feb. 1997.
19. H. Liu and D.F. Wong, "Circuit Partitioning for Dynamically Reconfigurable FPGAs," *Int'l ACM/SIGDA Symp. Field Programmable Gate Arrays*, 1999.
20. C. Longway and R. Siferd, "A Doughnut Layout Style for Improved Switching Speed with CMOS VLSI Gates," *IEEE J. Solid-State Circuits*, vol. 24, no. 1, pp. 194-198, 1989.
21. F.P. Preparata and M.I. Shamos, *Computational Geometry: An Introduction*. New York: Springer-Verlag, 1985.
22. J. Shi and D. Bhatia, "Performance Driven Floorplanning for FPGA Based Designs," *Proc. ACM Symp. Field-Programmable Gate Arrays*, pp. 112-118, Feb. 1997.
23. P.W. Shor, "The Average-Case Analysis of Some On-Line Algorithms for Bin Packing," *Combinatorica*, vol. 6, no. 2, pp. 179-200, 1986.
24. W.J. Sun and C. Sechen, "Efficient and Effective Placement for Very Large Circuits," *IEEE Trans. Computer Aided Design*, vol. 14, no. 3, pp. 349-359, Mar. 1995.
25. M.J. Wirthlin and B.L. Hutchings, "A Dynamic Instruction Set Computer," *Proc. IEEE Symp. FPGAs for Custom Computing Machines*, pp. 99-107, 1995.
26. K. Bazargan, R. Kastner, and M. Sarrafzadeh, "3-D Floorplanning: Simulated Annealing and Greedy Placement Methods for Reconfigurable Computing Systems," *10th IEEE Int'l Workshop on Rapid System Prototyping*, pp. 38-43, 1999.

Kiarash Bazargan is a PhD student at Northwestern University's VLSI CAD Group. His research interests are in the areas of VLSI CAD and reconfigurable computing systems. He received his BS degree in computer science in 1996 from Sharif University of Technology (Tehran, Iran) and his MS in electrical and computer engineering in 1998 from Northwestern University.

Ryan Kastner is currently a MS/PhD student at Northwestern University working in VLSI CAD group under the guidance of Majid Sarrafzadeh. His research interests include reconfigurable computing and global routing. He received his BS degrees in both electrical and computer engineering from Northwestern University in 1999.

Majid Sarrafzadeh has been a professor of electrical engineering and computer science at Northwestern University since 1997. His research interests lie in the areas of VLSI CAD, design and analysis of algorithms, and VLSI architecture. He received his BS, MS, and PhD degrees in 1982, 1984, and 1987, respectively, from the University of Illinois at Urbana-Champaign in electrical and computer engineering. Dr. Sarrafzadeh is a Fellow of IEEE for his contribution to theory and practice of VLSI design. He received an NSF Engineering Initiation award, two distinguished paper awards in ICCAD, and the best paper award for physical design in DAC for his work in the area of physical design. He has served on the technical program committee of numerous conferences in VLSI design and CAD, including ICCAD, EDAC, and ISCAS. He has served as committee chairs of a number of these conferences, including the *International Conference on CAD* and the *International Symposium on Physical Design*. Professor Sarrafzadeh has published approximately 150 papers, is a coeditor of the book *Algorithmic Aspects of VLSI Layout* (1994 by World Scientific), coauthor of the book *An Introduction to VLSI Physical Design* (1996 by McGraw-Hill), and the author of an invited chapter in the *Encyclopedia of Electrical and Electronics Engineering* in the area of VLSI circuit layout (John Wiley & Sons, Inc.). Dr. Sarrafzadeh is on the editorial board of the *VLSI Design Journal*, co-editor-in-chief of the *International Journal of High-Speed Electronics*, and an associate editor of *IEEE Transactions on Computer-Aided Design*.

■ Send questions and comments about this article to the authors at Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208-3118; {kiarash, kastner, majid}@ece.nwu.edu.

CALL FOR PAPERS

Seventh International Conference on High-Performance Computing
http://www.hipc.org
Submission deadline: 19 April 2000

The seventh annual international HPC conference will meet in Bangalore, India, 17-20 December 2000. Cosponsors include the Indian Institute of Science, Bangalore; the IEEE Computer Society Technical Committee Parallel Processing, and ACM SIGARCH. General Cochairs are Viktor K. Prasanna, University of Southern California at prasanna@ganges.usc.edu and Sriram Vajapeyam, Indian Institute of Science, Bangalore, at sriram@csa.iisc.ernet.in.

Submit original unpublished manuscripts that demonstrate current research in all areas of high-performance computing including design and analysis of parallel and distributed systems and their applications in scientific, engineering, and commercial areas. Manuscripts must not exceed 15 double-spaced pages of text in 12-point type on 8.5 x 11-in. pages. Send papers to

Program Chair Mateo Valero
Dept. de Arquitectura de Computadores
Universidad Politecnica de Catalunya
c/ Jordi Girona 1-3, Modulo D6
08034 Barcelona, SPAIN
mateo@ac.upc.es