An Efficient Data Structure for Dynamic Two-Dimensional Reconfiguration

Sándor P. Fekete, Jan-Marc Reinhardt(✉), and Christian Scheffer

Department of Computer Science, TU Braunschweig, Braunschweig, Germany
{s.fekete,j-m.reinhardt}@tu-bs.de, scheffer@ibr.cs.tu-bs.de

Abstract. In the presence of dynamic insertions and deletions into a partially reconfigurable FPGA, fragmentation is unavoidable. This poses the challenge of developing efficient approaches to dynamic defragmentation and reallocation. One key aspect is to develop efficient algorithms and data structures that exploit the two-dimensional geometry of a chip, instead of just one. We propose a new method for this task, based on the fractal structure of a quadtree, which allows dynamic segmentation of the chip area, along with dynamically adjusting the necessary communication infrastructure. We describe a number of algorithmic aspects, and present different solutions. We also provide experimental data for various scenarios, indicating practical usefulness of our approach.

Keywords: FPGAs · Partial reconfiguration · Two-dimensional reallocation · Defragmentation · Dynamic data structures · Insertions and deletions

1 Introduction

In recent years, a wide range of methodological developments on FPGAs have made it possible to combine the performance of an ASIC implementation with the flexibility of software realizations. One important development is partial runtime reconfiguration, which allows overcoming significant area overhead, monetary cost, higher power consumption, or speed penalties (see e.g. [20]). As described in [13], the idea is to load a sequence of different modules by partial runtime reconfiguration. In a general setting, we are faced with a dynamically changing set of modules, which may be modified by deletions and insertions. Typically, there is no full a-priori knowledge of the arrival or departure of modules, i.e., we have to deal with an online situation. The challenge is to ensure that arriving modules can be allocated. Because previously deleted modules may have been located in different areas of the layout, free space may be fragmented, making it necessary to *relocate* existing modules in order to provide sufficient area. In principle, this can be achieved by completely *defragmenting* the layout when necessary; however, the lack of control over the module sequence makes

This work was supported by the DFG Research Group FOR-1800, "Controlling Concurrent Change", under contract number FE407/17-1.

F. Hannig et al. (Eds.): ARCS 2016, LNCS 9637, pp. 306–318, 2016.
DOI: 10.1007/978-3-319-30695-7_23

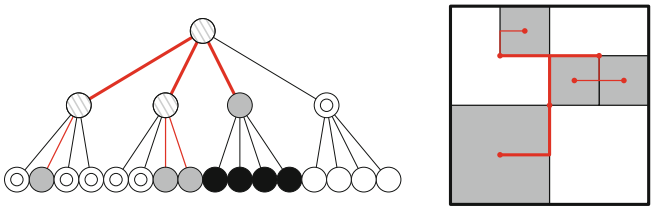

Fig. 1. A quadtree configuration (left) and the corresponding dynamically generated quadtree layout (right). Gray nodes are occupied, white ones with gray stripes fractional, black ones blocked, and white nodes without stripes empty. Maximally empty nodes have a circle inscribed. Red lines in the module layout indicate the dynamically produced communication infrastructure, induced by the quadtree structure (Color figure online).

it hard to avoid frequent full defragmentation, resulting in expensive operations for insertions if a naïve approach is used.

Dynamic insertion and deletion are classic problems of Computer Science. Many data structures (from simple to sophisticated) have been studied that result in low-cost operations and efficient maintenance of a changing set of objects. These data structures are mostly one-dimensional (or even dimensionless) by nature, making it hard to fully exploit the 2D nature of an FPGA. In this paper, we propose a 2D data structure based on a quadtree for maintaining the module layout under partial reconfiguration and reallocation. The key idea is to control the overall structure of the layout, such that future insertions can be performed with a limited amount of relocation, even when free space is limited.

Our main contribution is to introduce a 2D approach that is able to achieve provable constant-factor efficiency for different types of relocation cost. To this end, we give detailed mathematical proofs for a slightly simplified setting, along with sketches of extensions to the more general cases. We also provide experimental data for various scenarios, indicating practical usefulness of our approach.

The rest of this paper is organized as follows. The following Sect. 2 provides a survey of related work. For better accessiblity of the key ideas and due to limited space, our technical description in Sects. 3, 4, and 5 focuses on the case of discretized quadratic modules on a quadratic chip area. We discuss in Sect. 6 how general rectangles can be dealt with, with corresponding experimental data in Sect. 7. Along the same lines (and due to limited space), we do not explicitly elaborate on the dynamic maintenance of the communication infrastructure; see Fig. 1 for the basic idea.

2 Related Work

The problem considered in our paper has a resemblance to one-dimensional *dynamic storage allocation*, in which a sequence of storage requests of varying size have to be assigned to a block of memory cells, such that the length of

each block corresponds to the size of the request. In its classic form (without virtual memory), this block needs to be contiguous; in our setting, contiguity of two-dimensional allocation is a must, as reconfigurable devices do not provide techniques such as paging and virtual memory. Once the allocation has been performed, it is static in space: after a block has been occupied, it will remain fixed until the corresponding data is no longer needed and the block is released. As a consequence, a sequence of allocations and releases can result in fragmentation of the memory array, making it hard or even impossible to store new data.

On the practical side, classic buddy systems partition the one-dimensional storage into a number of standard block sizes and allocate a block in a smallest free standard interval to contain it. Differing only in the choice of the standard size, various systems have been proposed [9,15–17,22]. Newer approaches based on cache-oblivious structures in memory hierarchies include Bender et al. [2,6]. Theoretical work on one-dimensional contiguous allocation includes Bender and Hu [7], who consider maintaining n elements in sorted order, with not more than $O(n)$ space. Bender et al. [5] aim at reducing fragmentation when maintaining n objects that require contiguous space. Fekete et al. [13] study complexity results and consider practical applications on FPGAs. Reallocations have also been studied in the context of heap allocation. Bendersky and Petrank [8] observe that full compaction is prohibitively expensive and consider partial compaction. Cohen and Petrank [11] extend these to practical applications. Bender et al. [3] describe a strategy that achieves good amortized movement costs for reallocations. Another paper by the same authors [4] deals with reallocations in the context of scheduling.

From within the FPGA community, there is a huge amount of related work dealing with relocation: Becker et al. [1] present a method for enhancing the relocability of partial bitstreams for FPGA runtime configuration, with a special focus on heterogeneities. They study the underlying prerequisites and technical conditions for dynamic relocation. Gericota et al. [14] present a relocation procedure for Configurable Logic Blocks (CLBs) that is able to carry out online rearrangements, defragmenting the available FPGA resources without disturbing functions currently running. Another relevant approach was given by Compton et al. [12], who present a new reconfigurable architecture design extension based on the ideas of relocation and defragmentation. Koch et al. [18] introduce efficient hardware extensions to typical FPGA architectures in order to allow hardware task preemption. These papers do not consider the algorithmic implications and how the relocation capabilities can be exploited to optimize module layout in a fast, practical fashion, which is what we consider in this paper. Koester et al. [19] also address the problem of defragmentation. Different defragmentation algorithms that minimize different types of costs are analyzed.

The general concept of defragmentation is well known, and has been applied to many fields, e.g., it is typically employed for memory management. Our approach is significantly different from defragmentation techniques which have been conceived so far: these require a freeze of the system, followed by a computation of the new layout and a complete reconfiguration of all modules at once. Instead,

we just copy one module at a time, and simply switch the execution to the new module as soon as the move is complete. This leads to a *seamless, dynamic defragmentation of the module layout*, resulting in much better utilization of the available space for modules. All this makes our work a two-dimensional extension of the one-dimensional approach described in [13].

3 Preliminaries

We are faced with an (online) sequence of configuration requests that are to be carried out on a rectangular chip area. A request may consist of *deleting* an existing module, which simply means that the module may be terminated and its occupied area can be released to free space. On the other hand, a request may consist of *inserting* a new module, requiring an axis-aligned, rectangular module to be allocated to an unoccupied section of the chip; if necessary, this may require rearranging the allocated modules in order to create free space of the required dimensions, incurring some cost.

The rest of this section provides technical notation and descriptions. A square is called *aligned* if its size equals 2^{-r} for any $r \in \mathbb{N}_0$. It is called an r-square if its size is 2^{-r} for a specific $r \in \mathbb{N}_0$. A *quadtree* is a rooted tree in which every node has either four children or none. As a quadtree can be interpreted as the subdivision of the unit square into nested r-squares, we can use quadtrees to describe certain packings of aligned squares into the unit square.

Definition 1. *A* (quadtree) configuration T *assigns aligned squares to the nodes of a quadtree. The nodes with a distance j to the root of the quadtree form* layer j*. Nodes are also called* pixels *and pixels in layer j are called* j-pixels*. Thus, j-squares can only be assigned to j-pixels. A pixel p* contains *a square s if s is assigned to p or one of the children of p contains s. A j-pixel that has an assigned j-square is* occupied*. For a pixel p that is not occupied, with P the unique path from p to the root, we call p*

- blocked *if there is a $q \in P$ that is occupied,*
- free *if it is not blocked,*
- fractional *if it is free and contains a square,*
- empty *if it is free but not fractional,*
- maximally empty *if it is empty but its parent is not.*

The height h(T) *of a configuration T is defined as* 0 *if the root of T is empty. Otherwise, as the maximum $i + 1$ such that T contains an i-square.*

The *(remaining) capacity* $\mathrm{cap}(p)$ of a j-pixel p is defined as 0 if p is occupied or blocked and as 4^{-j} if p is empty. Otherwise, $\mathrm{cap}(p) := \sum_{p' \in C(p)} \mathrm{cap}(p')$, where $C(p)$ is the set of children of p. The *(remaining) capacity* of T, denoted $\mathrm{cap}(T)$, is the remaining capacity of the root of T.

See Fig. 1 for an example of a quadtree configuration and the corresponding packing of aligned squares in the unit square.

Quadtree configurations are transformed using *moves* (*reallocations*). A j-square s assigned to a j-pixel p can be *moved* (*reallocated*) to another j-pixel q by creating a new assignment from q to s and deleting the old assignment from p to s. q must have been empty for this to be allowed.

Definition 2. *A fractional pixel is* open *if at least one of its children is (maximally) empty. A configuration is called* compact *if there is at most one open j-pixel for every $j \in \mathbb{N}_0$.*

In (one-dimensional) storage allocation and scheduling, there are techniques that avoid reallocations by requiring more space than the sum of the sizes of the allocated pieces. See Bender et al. [4] for an example. From there we adopt the term *underallocation*. In particular, given two squares s_1 and s_2, s_2 is an x-underallocated copy of s_1, if $|s_2| = x \cdot |s_1|$ for $x > 1$.

Definition 3. *A* request *has one of the forms* $\textsc{Insert}(x)$ *or* $\textsc{Delete}(x)$*, where x is a unique identifier for a square. Let $v \in [0, 1]$ be the volume of the square x. The* volume *of a request σ is defined as*

$$\mathrm{vol}(\sigma) = \begin{cases} v & \text{if } r = \textsc{Insert}(x), \\ -v & \text{if } r = \textsc{Delete}(x). \end{cases}$$

Definition 4. *A sequence of requests $\sigma_1, \sigma_2, \ldots, \sigma_k$ is* valid *if $\sum_{i=1}^{j} \mathrm{vol}(\sigma_i) \leq 1$ holds for every $j = 1, 2, \ldots, k$. It is called* aligned*, if $|\mathrm{vol}(\sigma_j)| = 4^{-\ell_j}, \ell_j \in \mathbb{N}_0$, where $|.|$ denotes the absolute value, holds for every $j = 1, 2, \ldots, k$, i.e., if only aligned squares are packed.*

Our goal is to minimize the costs of reallocations. Costs can be measured in different ways, for example in the number of moves or the reallocated volume.

Definition 5. *Assume you fulfill a request σ and as a consequence reallocate a set of squares $\{s_1, s_2, \ldots, s_k\}$. The* movement cost *of σ is defined as $c_{\mathrm{move}}(\sigma) = k$, the* total volume cost *of σ is defined as $c_{\mathrm{total}}(\sigma) = \sum_{i=1}^{k} |s_i|$, and the* (relative) volume cost *of σ is defined as $c_{\mathrm{vol}}(\sigma) = \frac{c_{\mathrm{total}}(\sigma)}{|\mathrm{vol}(\sigma)|}$.*

4 Inserting into a Given Configuration

In this section we examine the problem of rearranging a given configuration in such a way that the insertion of a new square is possible.

4.1 Coping with Fragmented Allocations

Our strategy follows one general idea: larger empty pixels can be built from smaller ones; e.g., four empty i-pixels can be combined into one empty $(i-1)$-pixel. This can be iterated to build an empty pixel of suitable volume.

Lemma 6. *Let $p_1, p_2, \ldots, p_k$ be a sequence of empty pixels sorted by volume in descending order. Then $\sum_{i=1}^{k} \text{cap}(p_i) \geq 4^{-\ell} > \sum_{i=1}^{k-1} \text{cap}(p_i)$ implies the following properties:*

1. $k < 4 \Leftrightarrow k = 1$
2. $k \geq 4 \Rightarrow \sum_{i=1}^{k} \text{cap}(p_i) = 4^{-\ell}$
3. $k \geq 4 \Rightarrow \text{cap}(p_k) = \text{cap}(p_{k-1}) = \text{cap}(p_{k-2}) = \text{cap}(p_{k-3})$

Lemma 7. *Given a quadtree configuration T with four maximally empty j-pixels. Then T can be transformed (using a sequence of moves) into a configuration T^* with one more maximally empty $(j-1)$-pixel and four fewer maximally empty j-pixels than T while retaining all its maximally empty i-pixels for $i < j-1$.*

Theorem 8. *Given a quadtree configuration T with a remaining capacity of at least 4^{-j}, you can transform T into a quadtree configuration T^* with an empty j-pixel using a sequence of moves.*

Proof. Let $S = p_1, p_2, \ldots, p_n$ be the sequence containing all maximally empty pixels of T sorted by capacity in descending order. If the capacity of p_1 is at least 4^{-j}, then there already is an empty j-pixel in T and we can simply set $T^* = T$.

Assume $\text{cap}(p_1) < 4^{-j}$. In this case we inductively build an empty j-pixel. Let $S' = p_1, p_2, \ldots, p_k$ be the shortest prefix of S satisfying $\sum_{i=1}^{k} \text{cap}(p_i) \geq 4^{-j}$. Lemma 6 tells us $k \geq 4$ and the last four pixels in S', $p_{k-3}, p_{k-2}, p_{k-1}$ and p_k, are from the same layer, say layer ℓ. Thus, we can apply Lemma 7 to $p_{k-3}, p_{k-2}, p_{k-1}, p_k$ to get a new maximally empty $(\ell-1)$-pixel q. We remove $p_{k-3}, p_{k-2}, p_{k-1}, p_k$ from S' and insert q into S' according to its capacity.

We can repeat these steps until $k < 4$ holds. Then Lemma 6 implies that $k = 1$, i.e., the sequence contains only one pixel p_1, and because $\text{cap}(p_1) = 4^{-j}$, p_1 is an empty j-pixel.

4.2 Reallocation Cost

Reallocation cost is made non-trivial by *cascading moves*: Reallocated squares may cause further reallocations, when there is no empty pixel of the required size available.

Observation 9. *In the worst case, reallocating an ℓ-square is not cheaper than reallocating four $(\ell+1)$-squares – using any of the three defined cost types.*

Theorem 10. *The maximum total volume cost caused by the insertion of an i-square Q, $i \in \mathbb{N}_0$, into a quadtree configuration T with $\text{cap}(T) \geq 4^{-i}$ is bounded by*

$$c_{\text{total,max}} \leq \frac{3}{4} \cdot 4^{-i} \cdot \min\{(s-i), i\} \in O(|Q| \cdot h(T))$$

when the smallest previously inserted square is an s-square.

Proof. For $s \leq i$ there has to be an empty i-square in T, as $\text{cap}(T) \geq 4^{-i}$, and we can insert Q without any moves. In the following, we assume $s > i$. Let Q be the i-square to be inserted. We can assume that we do not choose an i-pixel with a remaining capacity of zero to pack Q – if there were no other pixels, $\text{cap}(T)$ would be zero as well. Therefore, the chosen pixel, say p, must have a remaining capacity of at least 4^{-s}. From Observation 9 follows that the worst case for p would be to be filled with 3 k-squares, for every $i < k \leq s$. Let v_i be the worst-case volume of a reallocated i-pixel. We get $v_i \leq \sum_{j=i+1}^{s} \frac{3}{4^j} = 4^{-i} - 4^{-s}$.

Now we have to consider cascading moves. Whenever we move an ℓ-square, $\ell > i$, to make room for Q, we might have to reallocate a volume of v_ℓ to make room for the ℓ-square. Let x_i be the total volume that is at most reallocated when inserting an i-square. Then we get the recurrence $x_i = v_i + \sum_{j=i+1}^{s} 3 \cdot x_j$ with $x_s = v_s = 0$. This resolves to $x_i = 3/4 \cdot 4^{-i} \cdot (s - i)$.

v_i cannot get arbitrarily large, as the remaining capacity must suffice to insert an i-square. Therefore, if all the possible i-pixels contain a volume of 4^{-s} (if some contained more, we would choose those and avoid the worst case), we can bound s by $4^i \cdot 4^{-s} \geq 4^{-i} \Leftrightarrow s \leq 2i$, which leads to $c_{\text{total,max}} \leq \frac{3}{4} \cdot 4^{-i} \cdot i$.

With $|Q| = 4^{-i}$ and $i < s < h(T)$ we get $c_{\text{total,max}} \in O(|Q| \cdot h(T))$.

Corollary 11. *Inserting a square into a quadtree configuration has a total volume cost of no more than* $3/16 = 0.1875$.

Proof. Looking at Theorem 10 it is easy to see that the worst case is attained for $i = 1$: $c_{\text{total}} = 3/4 \cdot 4^{-1} \cdot 1 = 3/16 = 0.1875$.

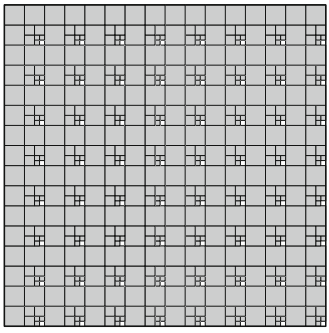

Fig. 2. The worst-case construction for volume cost for $s = 6$ and $i = 3$. Every 3-pixel contains three 4-, 5-, and 6-squares with only one remaining empty 6-pixel.

Theorem 12. *For every $i \in \mathbb{N}_0$ there are quadtree configurations T for which the insertion of an i-square Q causes a total volume cost of*

$$c_{\text{total,max}} \geq \frac{3}{4} \cdot 4^{-i} \cdot \min\{(s - i), i\} \in \Omega(|Q| \cdot h(T))$$

when the smallest previously inserted square is an s-square.

Proof. The worst case is attained for a quadtree configuration in which every i-pixel contains as many large squares as possible, while keeping the capacity of the configuration at 4^{-i}. This is achieved by a quadtree configuration containing three k-squares for every $i < k \leq 2i = s$. See Fig. 2 for an example with $i = 3$.

As a corollary we get an upper bound for the (relative) volume cost and a construction matching the bound: $c_{\text{vol,max}} \leq \frac{3}{4} \cdot \min\{(s-i), i\} \in \Theta(h(T))$.

The same methods we used to derive worst case bounds for volume cost can also be used to establish bounds for movement cost, which results in $c_{\text{move,max}} \leq 4^{\min\{s-i,i\}} - 1 \in O(4^{h(T)/2})$. A matching construction is the same as the one in the proof of Theorem 12.

5 Online Packing and Reallocation

We can avoid the worst cases presented in the previous section when we do not have to work with a given configuration and can handle all requests starting from the empty unit square.

5.1 First-Fit Packing

We present an algorithm that fulfills any valid, aligned sequence of requests and does not cause any reallocations on insertions. We call it *First Fit* in imitation of the well-known technique employed in one-dimensional allocation problems.

First Fit assigns items to be packed to the next available position in a certain order. For our 2D variant we use the z-order curve [21] to provide the order. We denote the position of a pixel p in z-order by $z(p)$, i.e., $z(p) < z(q)$ if and only if p comes before q in z-order.

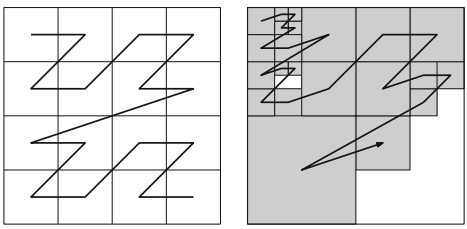

Fig. 3. The z-order for layer 2 pixels (left); a First Fit allocation and the z-order of the occupied pixels – which is not necessarily the insertion order (right).

First Fit proceeds as follows: A request to insert an i-square Q is handled by assigning Q to the first empty i-pixel in z-order; see Fig. 3. Deletions are more complicated. After unassigning a deleted square Q from a pixel p the following procedure handles reallocations (an example deletion can be seen in Fig. 4):

1: $S \leftarrow \{p'\}$, where p' is the maximally empty pixel containing p

2: **while** $S \neq \varnothing$ **do**
3: Let a be the element of S that is first in z-order.
4: $S \leftarrow S \setminus \{a\}$
5: Let b be the last occupied pixel in z-order.
6: **while** $z(b) > z(a)$ **do**
7: **if** the square assigned to b, B, can be packed into a **then**
8: Assign B to the first suitable descendant of a in z-order.
9: Unassign B from b.
10: Let b' be the maximally empty pixel containing b.
11: $S \leftarrow S \cup \{b'\}$
12: $S \leftarrow S \setminus \{b'' : b'' \text{ is child of } b'\}$
13: **end if**
14: Move the pointer z back in z-order to the next occupied pixel.
15: **end while**
16: **end while**

Invariant 13. *For every empty i-pixel p in a quadtree configuration T there is no occupied i-pixel q with $z(q) > z(p)$.*

Lemma 14. *Every quadtree configuration T satisfying Invariant 13 is compact.*

Lemma 15. *Given an ℓ-square s and a compact quadtree configuration T, then s can be assigned to an empty ℓ-pixel in T, if and only if* $\mathrm{cap}(T) \geq 4^{-l}$.

Theorem 16. *The strategy presented above is correct. In particular,*

1. *every valid insertion request is fulfilled at zero cost,*
2. *every deletion request is fulfilled,*
3. *after every request Invariant 13 holds.*

Proof. The first part follows from Lemmas 14 and 15 and point 3. Insertions maintain the invariant, because we assign it to the first suitable empty pixel in z-order. Deletions can obviously always be fulfilled. We still need to prove the important part, which is that the invariant holds after a deletion.

We show this by proving that whenever the procedure reaches line 3 and sets a, the invariant holds for all squares in z-order up to a. As we only move squares in negative z-order, the sequence of pixels a refers to is increasing in z-order. Since we have a finite number of squares, the procedure terminates after a finite number of steps when no suitable a is left. At that point the invariant holds throughout the configuration.

Assume we are at step 3 of the procedure and the invariant holds for all squares up to a. None of the squares considered to be moved to a fit anywhere before a in z-order – otherwise the invariant would not hold for pixels before a. Afterwards, no square that has not been moved to a fits into a, because it would have been moved there otherwise. Once we reach line 3 again, and set the new a, say a', consider the pixels between a and a' in z-order. If any square after a' would fit somewhere into a pixel between a and a', then the invariant would not have held before the deletion. Therefore, the invariant holds up to a'.

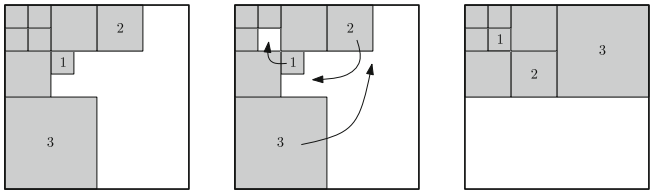

Fig. 4. Deleting a square causes several moves.

6 General Squares and Rectangles

Due to limited space and for clearer exposition, the description in the previous three sections considered aligned squares. We can adapt the technique to general squares and even rectangles at the expense of a constant factor.

Using 4-underallocation, we can pack any sequence of squares with total volume at most one into a 2×2 square, rounding the size of every square to the next power of two. An example is shown in Fig. 5. There, the solid gray areas are the packed squares and the shaded areas are space lost due to rounding. Note that even the best known result for merely packing squares – without considering deletions and reallocations – requires 5/2-underallocation [10].

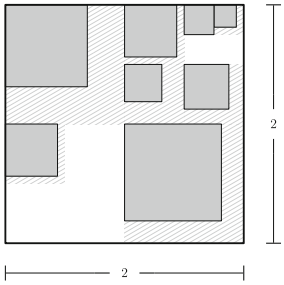

Fig. 5. Example of a dynamically generated quadtree layout.

Rectangles of bounded aspect ratio k are dealt with in the same way. Also accounting for intermodule communication, every rectangle is padded to the size of the next largest aligned square and assigned to the node of a quadtree, at a cost not exceeding a factor of $4k$ compared to the one we established for the worst case. As described in the following section, simulations show that the practical performance is even better.

7 Experimental Results

We carried out a number of experiments to evaluate the practical performance of our approach. For each test, we generated a random sequence of 1000 requests

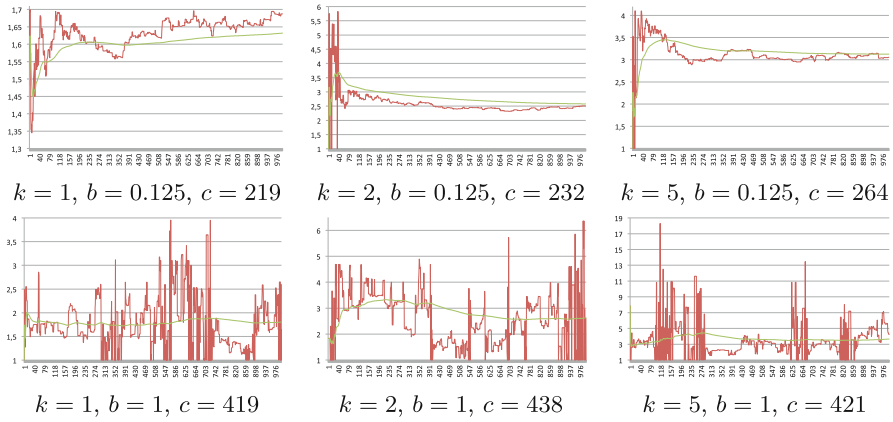

Fig. 6. Experimental evaluation of the First-Fit approach for different values of k and upper bounds of $b = 0.125$ and $b = 1$ for the side length of the considered squares: Each diagram illustrates the results of an experiment of 1000 requests that are randomly chosen as INSERT$(\cdot)$ (probability 0.7) or DELETE$(\cdot)$ (probability 0.3). We apply a larger probability for INSERT$(\cdot)$ to avoid the situation that repeatedly just a few rectangles are inserted and deleted. The red graph shows the total current underallocation after each request. The green graph shows the average of the total underallocation in the range between the first and the current request. We denote the number of collisions, i.e., the situations in that an INSERT$(\cdot)$ cannot be processed, by c (Color figure online).

that were chosen as INSERT$(\cdot)$ (probability 0.7) or DELETE$(\cdot)$ (probability 0.3). We apply a larger probability for INSERT$(\cdot)$ to avoid the (relatively simple) situation that repeatedly just a few rectangles are inserted and deleted, and in order to observe the effects of increasing congestion. The individual modules were generated by considering an upper bound $b \in [0, 1]$ for the side lengths of the considered squares. For $b = 0.125$, the value of the current underallocation seems to be stable except for the range of the first 50–150 requests. For $b = 1$, the current underallocation may be unstable, which could be caused by the following simple observation: A larger b allows larger rectangles that induce $4k$-underallocations.

Our experiments show that in practice, our approach achieves a much better underallocation than the theoretical worst-case bound of $1/4k$, see Fig. 6. Taking into account that a purely one-dimensional approach cannot provide an upper bound on the achievable underallocation, this indicates that our approach should be practically useful.

8 Conclusions

We have presented a data structure for exploiting the full dimensionality of dynamic geometric storage and reallocation tasks, such as online maintenance of the module layout for an FPGA. These first results indicate that our approach

is suitable for making progress over purely one-dimensional approaches. There are several possible refinements and extensions, including a more sophisticated way of handling rectangles inside of square pieces of the subdivision, explicit self-refining communication infrastructures, handling heterogeneous chip areas, and advanced algorithmic methods. These will be addressed in future work.

References

1. Becker, T., Luk, W., Cheung, P.Y.: Enhancing relocatability of partial bitstreams for run-time reconfiguration. In: Proceedings of the 15th Annual Symposium on Field-Programmable Custom Computing Machines, pp. 35–44 (2007)
2. Bender, M.A., Demaine, E.D., Farach-Colton, M.: Cache-oblivious B-trees. SIAM J. Comput. **35**, 341–358 (2005)
3. Bender, M.A., Farach-Colton, M., Fekete, S.P., Fineman, J.T., Gilbert, S.: Cost-oblivious storage reallocation. In: Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2014, pp. 278–288. ACM (2014)
4. Bender, M.A., Farach-Colton, M., Fekete, S.P., Fineman, J.T., Gilbert, S.: Reallocation problems in scheduling. Algorithmica **73**(2), 389–409 (2014)
5. Bender, M.A., Fekete, S.P., Kamphans, T., Schweer, N.: Maintaining arrays of contiguous objects. In: Kutyłowski, M., Charatonik, W., Gębala, M. (eds.) FCT 2009. LNCS, vol. 5699, pp. 14–25. Springer, Heidelberg (2009)
6. Bender, M.A., Fineman, J.T., Gilbert, S., Kuszmaul, B.C.: Concurrent cache-oblivious B-trees. In: Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures, pp. 228–237 (2005)
7. Bender, M.A., Hu, H.: An adaptive packed-memory array. ACM Trans. Database Syst. **32**(4), 26:1–26:43 (2007)
8. Bendersky, A., Petrank, E.: Space overhead bounds for dynamic memory management with partial compaction. ACM Trans. Program. Lang. Syst. **34**(3), 13:1–13:43 (2012)
9. Bromley, G.: Memory fragmentation in buddy methods for dynamic storage allocation. Acta lnformatica **14**, 107–117 (1980)
10. Brubach, B.: Improved bound for online square-into-square packing. In: Bampis, E., Svensson, O. (eds.) WAOA 2014. LNCS, vol. 8952, pp. 47–58. Springer, Heidelberg (2015)
11. Cohen, N., Petrank, E.: Limitations of partial compaction: towards practical bounds. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013, New York, NY, USA, pp. 309–320. ACM (2013)
12. Compton, K., Li, Z., Cooley, J., Knol, S., Hauck, S.: Configuration relocation and defragmentation for run-time reconfigurable systems. IEEE Trans. VLSI **10**, 209–220 (2002)
13. Fekete, S.P., Kamphans, T., Schweer, N., Tessars, C., van der Veen, J., Angermeier, J., Koch, D., Teich, J.: Dynamic defragmentation of reconfigurable devices. ACM Trans. Reconfigurable Technol. Syst. (TRETS) **5**(8), 8:1–8:20 (2012)
14. Gericota, M.G., Alves, G.R., Silva, M.L., Ferreira, J.M.: Run-time defragmentation for dynamically reconfigurable hardware. In: Lysaght, P., Rosenstiel, W. (eds.) New Algorithms, Architectures and Applications for Reconfigurable Computing, pp. 117–129. Springer, New York (2005)

15. Hinds, J.A.: An algorithm for locating adjacent storage blocks in the buddy system. Commun. ACM **18**, 221–222 (1975)
16. Hirschberg, D.S.: A class of dynamic memory allocation algorithms. Commun. ACM **16**, 615–618 (1973)
17. Knowlton, K.C.: A fast storage allocator. Commun. ACM **8**, 623–625 (1965)
18. Koch, D., Ahmadinia, A., Bobda, C., Kalte, H.: FPGA architecture extensions for preemptive multitasking and hardware defragmentation. In: Proceedings of the IEEE International Conference Field-Programmable Technology, Brisbane, Australia, pp. 433–436 (2004)
19. Koester, M., Kalte, H., Porrmann, M., Ruckert, U.: Defragmentation algorithms for partially reconfigurable hardware. In: Reis, R., Osseiran, A., Pfleiderer, H.-J. (eds.) Vlsi-Soc: From Systems To Silicon. IFIP International Federation for Information Proc, vol. 240, p. 41. Springer, Boston (2007)
20. Kuon, I., Rose, J.: Measuring the gap between FPGAs and ASICs. IEEE Trans. CAD Integr. Circ. Syst. **26**, 203–215 (2007)
21. Morton, G.: A computer oriented geodetic data base and a new technique in file-sequencing.Technical report, IBM Ltd., Ottawa, Ontario, March 1966
22. Shen, K.K., Peterson, J.L.: A weighted buddy method for dynamic storage allocation. Commun. ACM **17**, 558–562 (1974)