Journal of Systems Architecture 75 (2017) 15–25

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

An efficient data structure for dynamic two-dimensional reconfiguration☆,☆☆

Sándor P. Fekete*, Jan-Marc Reinhardt, Christian Scheffer

Department of Computer Science, TU Braunschweig, Germany

ARTICLE INFO

Article history:
Received 4 July 2016
Revised 30 January 2017
Accepted 17 February 2017
Available online 27 February 2017

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

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 a number of basic simulations that indicate that the theoretical worst-case bound may be pessimistic.

1. Introduction

In recent years, a wide range of methodological developments on FPGAs aim at combining 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. [2]). As described in [3], 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 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 basic simulation runs for various scenarios, indicating the quality of our approach.

The rest of this paper is organized as follows. The following Section 2 provides a survey of related work. For better accessiblity of the key ideas and due to limited space, our technical description in Section 3, Section 4, and Section 5 focuses on the case of discretized square modules on a quadratic chip area. We discuss in Section 6 how general rectangles can be dealt with, with corresponding simulations in Section 7. Along the same lines, we do not explicitly elaborate on the dynamic maintenance of the

☆ This work was supported by the DFG Research Group FOR-1800, "Controlling Concurrent Change", under contract number FE407/17-1 and 17-2.

☆☆ A preliminary extended abstract of this paper appears in ARCS2016 [1].

* Corresponding author.
E-mail addresses: s.fekete@tu-bs.de (S.P. Fekete), j-m.reinhardt@tu-bs.de (J.-M. Reinhardt), scheffer@ibr.cs.tu-bs.de (C. Scheffer).

http://dx.doi.org/10.1016/j.sysarc.2017.02.004
1383-7621/

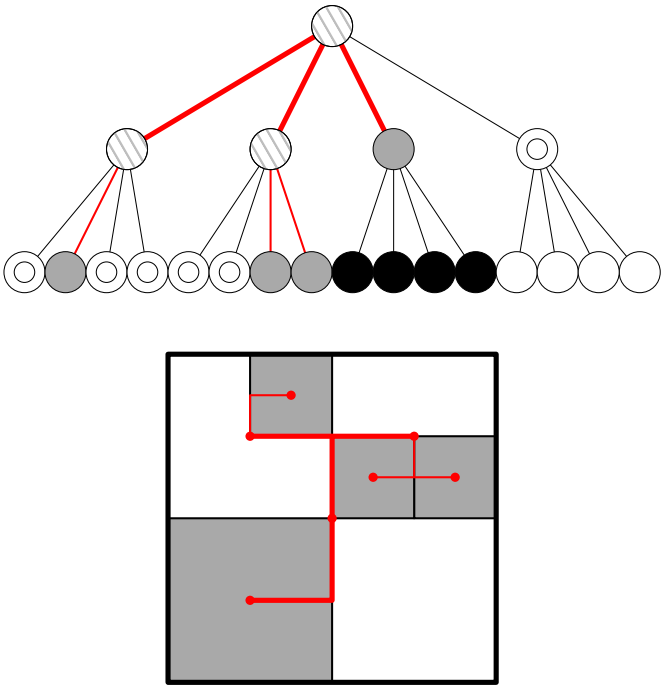

Fig. 1. A quadtree configuration (above) and the corresponding dynamically generated quadtree layout below). Gray nodes are occupied, white ones with gray stripes fractional, black ones blocked, and white nodes without stripes empty. Maximally empty nodes have a circle inscribed. Red lines in the module layout indicate the dynamically produced communication infrastructure, induced by the quadtree structure. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

communication infrastructure; see Fig. 1 for the basic idea. Further details are left to future work, with groundwork laid in [4].

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 [5–9]. Newer approaches based on cache-oblivious structures in memory hierarchies include Bender et al. [10,11]. Theoretical work on one-dimensional contiguous allocation includes Bender and Hu [12], who consider maintaining n elements in sorted order, with not more than $O(n)$ space. Bender et al. [13] aim at reducing fragmentation when maintaining n objects that require contiguous space. Fekete et al. [3] study complexity results and consider practical applications on FPGAs. Reallocations have also been studied in the context of heap allocation. Bendersky and Petrank [14] observe that full compaction, i.e., creating a contiguous block of free space on the heap, is prohibitively expensive and consider partial compaction. Cohen and Petrank [15] extend these to practical applications. Bender et al. [16] describe a strategy that achieves good amortized movement costs for reallocations, where allocated blocks can be moved at a cost to a new position that is disjoint from with the old position. Another paper by the same authors [17] deals with reallocations in the context of scheduling. Examples for packing problems in applied computer science come from allocating FPGAs. Fekete et al. [18] examined a problem dealing with the allocation of different types of resources on an FPGA that had to satisfy additional properties. For example, to achieve specified clock frequencies diameter restrictions had to be obeyed by the packing. The authors were able to solve the problem using integer linear programming techniques.

Over the years, a large variety of methods and results for allocating storage have been proposed. The classical sequential fit algorithms, First Fit, Best Fit, Next Fit and Worst Fit can be found in Knuth [19] and Wilson et al. [20]. These are closely related to problems of offline and online packing of two-dimensional objects. One of the earliest considered packing variants is the problem of finding a dense packing of a known set of squares for a rectangular container; see Moser [21], Moon and Moser [22] and Kleitman and Krieger [23], as well as more recent work by Novotný [24,25] and Hougardy [26]. There is also a considerable number of other related work on offline packing squares, cubes, or hypercubes; see [27–29] for prominent examples. The *online* version of square packing has been studied by Januszewski and Lassak [30] and Han et al. [31], with more recent progress due to Fekete and Hoffmann [32,33]. A different kind of online square packing was considered by Fekete et al. [34,35]. The container is an unbounded strip, into which objects enter from above in a Tetris-like fashion; any new object must come to rest on a previously placed object, and the path to its final destination must be collision-free.

There are various ways to generalize the online packing of squares; see Epstein and van Stee [36–38] for online bin packing variants in two and higher dimensions. In this context, also see parts of Zhang et al. [39]. A natural generalization of online packing of squares is online packing of rectangles, which have also received a serious amount of attention. Most notably, online strip packing has been considered; for prominent examples, see Azar and Epstein [40], who employ shelf packing, and Epstein and van Stee [36]. Offline packing of rectangles into a unit square or rectangle has also been considered in different variants; for examples, see [41], as well as [42]. Particularly interesting for methods for online packing into a single container may be the work by Bansal et al. [43], who show that for any complicated packing of rectangular items into a rectangular container, there is a simpler packing with almost the same value of items.

From within the FPGA community, there is a huge amount of related work dealing with problems related to relocation. Becker et al. [44] 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. [45] 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. [46], who present a new reconfigurable architecture design extension based on the ideas of relocation and defragmentation. Koch et al. [47] 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. [48] 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 concept aims at providing a seamless, dynamic defragmentation of the module layout, eventually 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 [3].

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.

Previous work on reallocation problems of this type has focused on one-dimensional approaches. Using these in a two-dimensional setting does not result in satisfactory performance. The main contribution of our paper is to demonstrate a two-dimensional approach that is able to achieve an efficiency that is probably within a constant factor of the optimum, even in the worst case, which requires a variety of mathematical details. For better accessiblity of the key ideas, our technical description in the rest of this Section 3, as well as in Section 4 and Section 5 focuses on the case of square modules on a quadratic chip area. Section 6 addresses how to deal with general rectangles.

The rest of this section provides technical notation and descriptions. A square is called *aligned* if its edge length 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 $r \in \mathbb{N}_0$. The *volume* of an r-square Q is $|Q| = 4^{-r}$. 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 a set of 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.

Observation 2. Let $p \neq q$ be two maximally empty pixels and P and Q be the paths from the root to p and q, respectively. Then $p \notin Q$ and $q \notin P$.

Proof. Without loss of generality, it is sufficient to show $p \notin Q$. Assume $p \in Q$. Let $r \in Q$ be the parent of q. As p is maximally empty and r is on the path from p to q, r must be empty. However, that would imply that q is not maximally empty, in contradiction to the assumption. □

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.

Lemma 3. *Let $p_1, p_2, \ldots, p_k$ be all maximally empty pixels of a quadtree configuration T. Then we have* $\mathrm{cap}(T) = \sum_{i=1}^{k} \mathrm{cap}(p_i)$.

Proof. The claim follows directly from the definition of the capacity, as the only positive capacities considered for $\mathrm{cap}(T)$ are exactly those of the maximally empty pixels. □

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.

We allow only one move at a time. For example, two squares cannot change places unless there is a sufficiently large pixel to temporarily store one of them. Furthermore, we do not put limitations on how to transfer a square from one place to another, i.e., we can always move a square even if there is no collision-free path between the origin and the destination.

Definition 4. 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. [17] 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 5. 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 6. 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 7. Assume we 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. Before we present our results in mathematical detail, including all necessary proofs, we give a short overview of the individual propositions and their significance: We first examine properties of quadtree configurations culminating in Theorem 10, which establishes that any configuration with sufficient capacity allows the insertion of a square. Creating the required contiguous space for the insertion comes at a cost due to required reallocations. This cost is analysed in detail in Section 4.2. There, we present matching upper and lower bounds on the reallocation cost for our three cost functions – total volume cost (Theorems 12 and 14), (relative) volume cost (Corollary 15), and movement cost (Theorems 16 and 17).

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

$$k < 4 \Leftrightarrow k = 1 \tag{1}$$

$$k \geq 4 \Rightarrow \sum_{i=1}^{k} \mathrm{cap}(p_i) = 4^{-\ell} \tag{2}$$

$$k \geq 4 \Rightarrow \mathrm{cap}(p_k) = \mathrm{cap}(p_{k-1}) = \mathrm{cap}(p_{k-2}) = \mathrm{cap}(p_{k-3}) \tag{3}$$

Proof. For $k \geq 2$, p_1 must be a pixel of smaller capacity than an ℓ-pixel, because otherwise we would not need p_2 for the sum to be greater than $4^{-\ell}$ – in contradiction to the assumption. Thus, we need to add up smaller capacities to at least $4^{-\ell}$. As we need at least four $(\ell+1)$-pixels for that, statement (1) holds.

In the following we assume $k \geq 4$. Let $x = \sum_{i=1}^{k-1} \mathrm{cap}(p_i)$. We know from the assumption that x is strictly less than $4^{-\ell}$, but $x + \mathrm{cap}(p_k)$ is at least $4^{-\ell}$. Consider the base-4 (quaternary) representation of $x/4^{-\ell}$: $x_4 = (x/4^{-\ell})_4$. It has a zero before the decimal point and a sequence of base-4 digits after. Let n be the rightmost non-zero digit of x_4. As the sequence is sorted in descending order and the capacities are all negative powers of four, adding the capacity of p_k can only increase n, or a digit right of n, by one. Since all digits right of n are zero, increasing one of them by one does not increase x to at least $4^{-\ell}$. Therefore, it must increase n. But if increasing n by one means increasing x to at least $4^{-\ell}$, then every digit of x_4 after the decimal point and up to n must have been three. Consequently, increasing n by one leads not only to $x + \mathrm{cap}(p_k) \geq 4^{-\ell}$ but also to $x + \mathrm{cap}(p_k) = 4^{-\ell}$, which is statement (2).

Furthermore, as n must have been three and the sequence is sorted, the previous three capacities added must have each increased n by exactly one as well. This proves statement (3). □

Lemma 9. *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 $i < j-1$.*

Proof. Let p_1, p_2, p_3 and p_4 be four maximally empty j-pixels and q_1, q_2, q_3 and q_4 be the parents of p_1, p_2, p_3 and p_4, respectively.

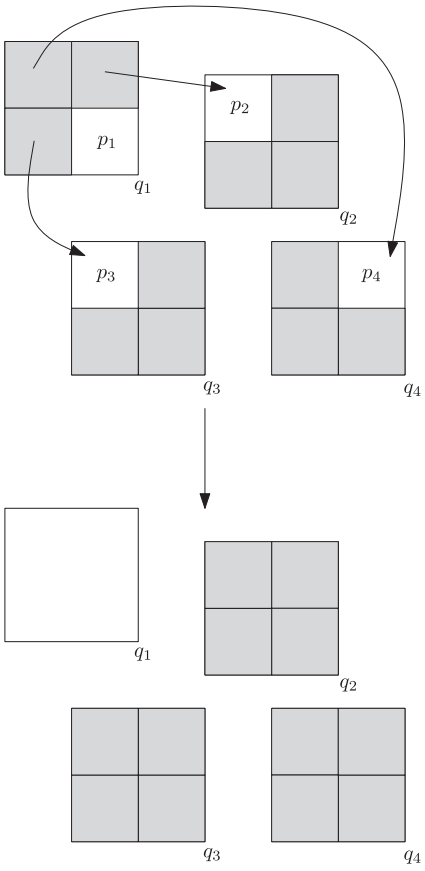

Fig. 2. Illustration to Lemma 9.

Then q_i has at most three children that are not empty. Now, we can move the at most three non-empty subtrees from one of the q_i to the others, $i = 1, 2, 3, 4$. Without loss of generality, we choose q_1. Let a, b and c be the children of q_1 that are not p_1. We move a to p_2, b to p_3 and c to p_4. See Fig. 2 for an illustration. Thus, we get a new configuration T^* with the empty $(j-1)$-pixel q_1 and occupied or fractional pixels q_2, q_3, q_4. Note that p_1 is still empty, but no longer maximally empty, because its parent q_1 is now empty. The construction does not affect any other maximally empty pixels. □

Theorem 10. *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 $\mathrm{cap}(p_1) < 4^{-j}$. 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 $\sum_{i=1}^{k} \mathrm{cap}(p_i) \geq 4^{-j}$. Such a prefix has to exist because of Lemma 3. Note that due to Observation 2 no pixel p_i is contained in another pixel p_j, $i, j \in \{1, 2, \ldots, k\}$, $i \neq j$. Lemma 8 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 9 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. The length of the resulting sequence S'' is three less than the length of S'. This does not change the sum of the capacities, since an empty $(\ell-1)$-pixel has the same capacity as four empty ℓ-pixels. That is, $\sum_{p \in S'} \mathrm{cap}(p) = \sum_{p \in S''} \mathrm{cap}(p)$ holds.

We can repeat these steps until $k < 4$ holds. Then Lemma 8 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 11. 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.

Proof. It is straightforward to see this for volume costs, total or relative: Wherever you can move one ℓ-square you can also move four $(\ell+1)$-squares without causing more cascading moves.

For movement costs a single move of an ℓ-square is less than four moves of $(\ell+1)$-squares, but it can cause cascading moves of three $(\ell+1)$-squares plus the cascading moves caused by the reallocation of an $(\ell+1)$-square and, therefore, does not cause lower costs in total. □

Theorem 12. *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 11 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 13. *Inserting a square into a quadtree configuration has a total volume cost of no more than $3/16 = 0.1875$.*

Proof. Looking at Theorem 12 it is easy to see that the worst case is attained for $i = 1$: $c_{\text{total}} = 3/4 \cdot 4^{-1} \cdot 1 = 3/16 = 0.1875$. □

Theorem 14. *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. We build a quadtree configuration to match the upper bound of Theorem 12. Let $s = 2i$ and consider a subtree rooted at an i-pixel that contains three k-pixels for every $i < k \leq s$. They do not have to be arranged in such a way that the single free s-pixel is in the lower right corner, but the nesting structure is important. Assume all 4^i i-pixels of T are constructed in such a way. Then you have to reallocate three k-squares for every $i < k \leq s$. However, every fractional k-pixel in the configuration in turn contains three k'-pixel for every $k < k' < s$, i.e., moving every k-square causes cascading moves. See Fig. 3 for the whole construction for $s = 6$ and $i = 3$. The reallocated volume without cascading moves adds up to $v_i = \sum_{k=i+1}^{s} 3 \cdot 4^{-k}$.

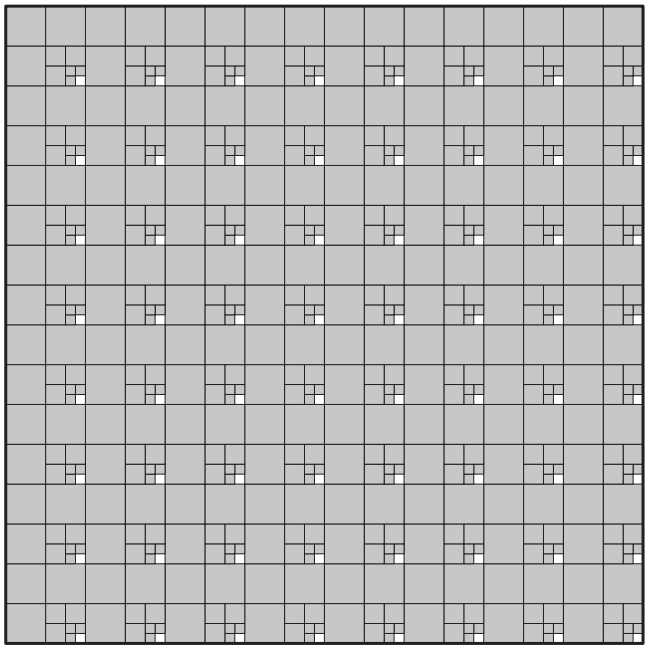

Fig. 3. The worst-case construction for volume cost for $s = 6$ and $i = 3$. Every 3-pixel contains three 4-, 5-, and 6-squares with only one remaining empty 6-pixel.

Including cascading moves we get $x_i = v_i + \sum_{k=i+1}^{s} 3 \cdot x_k$, which resolves to $x_i = 3/4 \cdot 4^{-i} \cdot (s-i)$.

With $s = h(T) - 1$, $i = s/2$ and $|Q| = 4^{-i}$ we get $c_{\text{total, max}} \in \Omega(|Q| \cdot h(T))$. □

As a corollary we get an upper bound for the (relative) volume cost and a construction matching the bound.

Corollary 15. *Inserting an i-square into a quadtree configuration T with sufficient capacity $\text{cap}(T) \geq 4^{-i}$ causes a (relative) volume cost of at most*

$$c_{\text{vol,max}} \leq \frac{3}{4} \cdot \min\{(s-i), i\} \in \Theta(h(T)),$$

when the smallest previously inserted square is an s-square, and this bound is tight, i.e., there are configurations for which the bound is matched.

It is important to note that relative volume cost can be arbitrarily bad by increasing the height of the configuration, as opposed to total volume cost with the upper bound derived in Corollary 13. What is more, large total volume cost is achieved by inserting i-squares for small i, whereas large relative volume cost is only possible for large i (and large $s - i$). This has an interesting interpretation with regard to the structure of the quadtree: Large total volume cost can happen when you assign a square to a node close to the root. To get large relative volume cost you need a high quadtree and assign a square to a node roughly in the middle (with respect to height).

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 $c_{\text{move,max}} \leq 4^{\min\{s-i,i\}} - 1 \in O(2^{h(T)})$. A matching construction is the same as the one in the proof of Theorem 14.

Theorem 16. *The maximum movement 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*

$$c_{\text{move,max}} \leq 4^{\min\{s-i,i\}} - 1 \in O(2^{h(T)})$$

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

Proof. The proof is analogous to the proof of Theorem 12. We can use Observation 11 and formulate a new recurrence. The number of reallocations without cascading moves caused by the insertion of Q can be bounded by $v_i \leq 3(s-i)$ and including cascading moves we get $x_i = v_i + \sum_{j=i+1}^{s} 3x_i$, which resolves to $x_i = 4^{s-i} - 1$.

As we need at least 4^{-i} remaining capacity to insert Q we can again deduce $s \leq 2i$. With $s = h(T) - 1$ we get $\min\{s-i, i\} \leq h(T)/2$, which results in the claimed bound. □

Theorem 17. *For every $i \in \mathbb{N}_0$ there are quadtree configurations T for which the insertion of an i-square Q causes a movement cost of*

$$c_{\text{move,max}} \geq 4^{\min\{s-i,i\}} \in \Omega(2^{h(T)})$$

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

Proof. The example from Theorem 14 works here as well. As every fractional j-pixel, $j < s$, contains three $(j+1)$-pixels, you have to move three squares for every $j = i, \ldots, s-1$ and account for cascading moves. This results in a number of moves $c_{move,max} \geq x_i = 3(s-i) + \sum_{j=i+1}^{s} x_j = 4^{s-i} - 1$, where $s = 2i = h(T) - 1$. □

5. Online packing and reallocation

Applying Theorem 10 repeatedly to successive configurations yields a strategy for the dynamic allocation of aligned squares.

Corollary 18. *Starting with an empty square and given a valid, aligned sequence of requests, there is a strategy that fulfills every request in the sequence.*

Proof. We only have to deal with aligned squares and can use quadtree configurations to pack the squares, since the sequence of requests $\sigma_1, \sigma_2, \ldots, \sigma_k$ is aligned. We start with the empty configuration that contains only one empty 0-pixel. Thus, we have a configuration with capacity 1. We only have to consider insertions, because deletions can always be fulfilled by definition.

As the sequence of requests is valid, whenever a request σ_ℓ demands to insert a j-square s, the remaining capacity of the current quadtree configuration T is at least $1 - \sum_{i=1}^{\ell-1} \text{vol}(\sigma_i) + 4^{-j} \geq 4^{-j}$.

Therefore, we can use Theorem 10 to transform T into a configuration T^* with an empty j-pixel p. We assign s to p. □

This strategy may incur the heavy insertion cost derived in the previous section. However, when we do not have to work with a given configuration and have the freedom to handle all requests starting from the empty unit square, we can use the added flexibility to derive a more sophisticated strategy. In particular, we can use reallocations to clean up a configuration when squares are deleted. This can make deletions costly operations, but allows us to eliminate insertion cost entirely.

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.

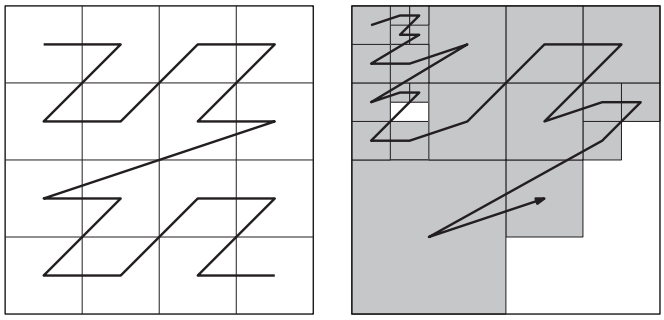

Fig. 4. The z-order for layer 2 pixels (left); a First Fit allocation and the z-order of the occupied pixels – which is not necessarily the insertion order (right).

Given a one-dimensional packing and a request to allocate space for an additional item, First-fit chooses the first suitable location. In one dimension it is trivial to define an order in which to check possible locations. For example, assume your resources are arranged horizontally and proceed from left to right.

Finding an order in two or more dimensions is not as straightforward as it is in 1D. We use space-filling curves to overcome this impediment. Space-filling curves are of theoretical interest, because they fill the entire unit square (i.e., their Hausdorff dimension is 2). More useful to us are the schemes used to create a space-filling curve, which employ a recursive construction on the nodes of a quadtree and become space-filling as the height of the tree approaches infinity. In particular, they provide an order for the nodes of a quadtree. In the following, we make use of the z-order curve [49].

First Fit assigns items to be packed to the next available position in z-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.

In general, the z-order is only a partial order, as it does not make sense to compare nodes with their parents or children. However, there are three important occasions for which the z-order is a total order: If you only consider pixels in one layer, if you only consider occupied pixels, and if you only consider maximally empty pixels. In all three cases pixels are pairwise disjoint, which leads to a total order.

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. 4. 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. 5):

```
S ← {p'}, where p' is the maximally empty pixelcontaining p
while S ≠ ∅ do
    Let a be the element of S that is first in z-order.
    S ← S \ {a}
    Let b be the last occupied pixel in z-order.
    while z(b) > z(a) do
        if the square assigned to b, B, can be packed into a then
            Assign B to the first suitable descendant of a inz-order.
            Unassign B from b.
            Let b' be the maximally empty pixel containing b.
            S ← S ∪ {b'}
            S ← S \ {b'' : b'' is child of b'}
        end if
        Move the pointer z back in z-order to the next occupiedpixel.
    end while
end while
```

The general idea is to reallocate squares from the current end of the z-order to empty spots. As reallocating creates new empty

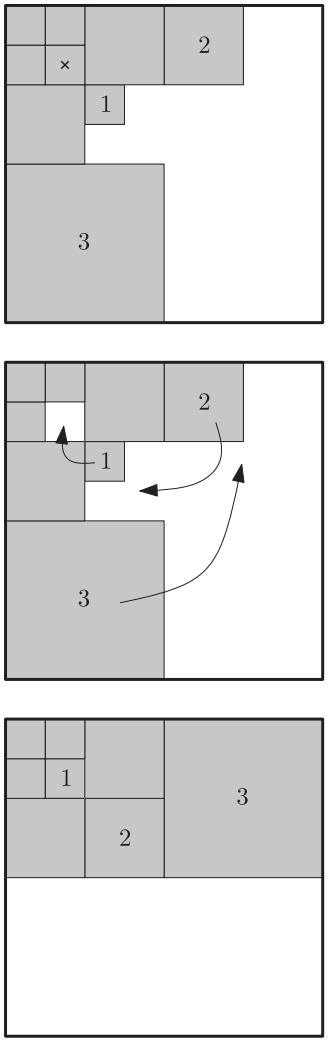

Fig. 5. Deleting a square causes several moves. The deleted square is marked with a cross. Once it is unassigned, the squares are checked in reverse z-order until square 1, which fits. Afterwards, there is a now maximally empty pixel into which square 2 can be moved. Finally, the same happens for square 3.

squares, we need to apply the method repeatedly in what can be considered an inverse case of cascading moves. We ensure termination by always moving the currently considered empty pixel in positive z-order and reallocating squares in negative z-order. We analyze the strategy in more detail now.

Invariant 19. For every empty i-pixel p in a quadtree configuration T there is no occupied i-pixel q with $z(q) > z(p)$.

Lemma 20. *Every quadtree configuration T satisfying Invariant 19 is compact.*

Proof. Assume a quadtree configuration T is not compact. Then it contains two fractional i-pixels, $i \in \mathbb{N}$, p and q with maximally empty children p' and q', respectively. Without loss of generality, assume $z(p) < z(q)$. As q is fractional, there is a j-square, $j > i$, assigned to some descendant of q, say q''. However, p' is an empty $(i+1)$-pixel and therefore contains an empty j-pixel, p''. As $z(p) < z(q)$, we also have $z(p'') < z(q'')$ and Invariant 19 does not hold. □

Lemma 21. *In a compact quadtree configuration T there are at most three maximally empty j-pixels for every $j \in \mathbb{N}_0$.*

Proof. *The statement holds for $j = 0$, since there is only one 0-pixel. For $j > 0$ there is at most one open $(j-1)$-pixel p in T, because T is compact. Therefore, all other $(j-1)$-pixels except for p either do not have an empty child or are maximally empty themselves. Thus, all maximally empty j-pixels have to be children of p. Since p is not empty, there can be at most three.* □

Lemma 22. *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 $\mathrm{cap}(T) \geq 4^{-l}$.*

Proof. The direction from left to right is obvious, as there can be no empty ℓ-pixel if the capacity is less than 4^{-l}. For the other direction assume there is no empty ℓ-pixel in T. Since there is no empty ℓ-pixel, there is also no empty j-pixel for any $j < \ell$. Let the smallest square assigned to a node be an s-square. As T is compact, we can use Lemma 21 and Lemma 3 to bound the remaining capacity of T from above: $\mathrm{cap}(T) \leq \sum_{k=l+1}^{s} 3 \cdot 4^{-k} = 4^{-\ell} - 4^{-s} < 4^{-\ell}$. □

In other words, packing an ℓ-square in a compact configuration requires no reallocations.

Theorem 23. *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 19 holds.*

Proof. The first part follows from Lemmas 22 and 20 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'. □

Comparing our results in Section 4 to those in this section, a major advantage of an empty initial configuration becomes apparent. For all examined cost functions there are configurations into which no square can be inserted at zero cost (cf. Theorem 14, Corollary 15, Theorem 17). This is in contrast to First-fit, which achieves insertion at zero cost (Theorem 23). The downside is the potentially large cost of deletions. The thorough analysis of a strategy with provably low cost for both insertions and deletions is the subject of future work.

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.

To accommodate a non-aligned square, we pack it like an aligned square of the next larger volume. That is, a square of size

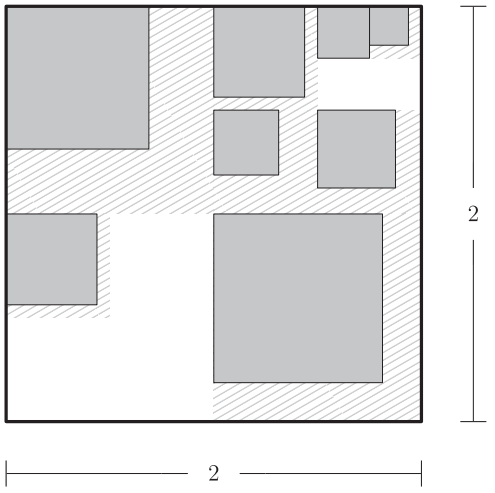

Fig. 6. Example of a dynamically generated quadtree layout. The solid gray areas are packed squares. Shaded areas represent space lost due to rounding.

s with $2^{i-1} < s < 2^i$ for some $i \in \{0, -1, -2, \ldots\}$ is assigned to an i-pixel. This approach results in space that cannot be used to assign squares, even though the remaining capacity would suffice, and we can no longer guarantee to fit every valid sequence of squares into the unit square. However, we can guarantee to pack every such sequence into a 4-underallocated unit square (i.e., a 2×2 square), as every square is assigned to a pixel that can hold no more than four times its volume. Most importantly, our reallocation schemes continue to work in this setting unmodified. An example allocation is shown in Fig. 6, where solid gray areas are assigned squares and shaded areas indicate wasted space.

Note that a satisfactory reallocation scheme for arbitrary squares with no or next to no underallocation is unlikely. Even the problem of handling a sequence of insertions of total volume at most one, without considering dynamic deletions and reallocation, requires underallocation. This problem is known as online square packing and the best known approach results in 5/2-underallocation [50].

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, this theoretical bound is rather pessimistic: the performance in basic simulation runs is considerably better.

7. Simulation results

We carried out a number of simulation runs to get an idea of the potential performance of our approach. For each test, we generated a random sequence of 1000 requests that were chosen as INSERT(·) (probability 0.7) or DELETE(·) (probability 0.3). We apply a larger probability for INSERT(·) 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.

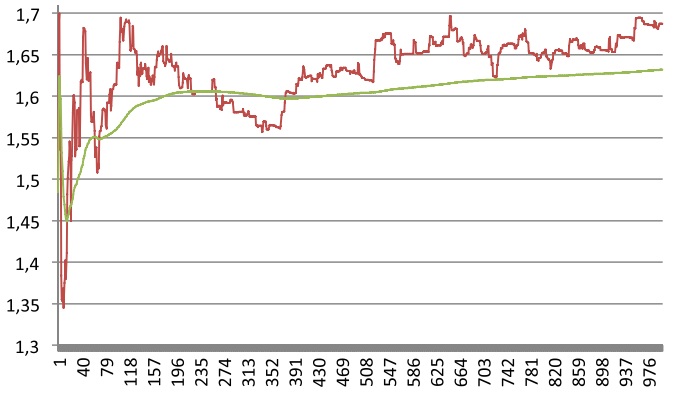

Fig. 7. Number of operations (x-axis) v.s. the inverse value of underallocation (y-axis) for the setting $k = 1$, $b = 0.125$, $c = 219$.

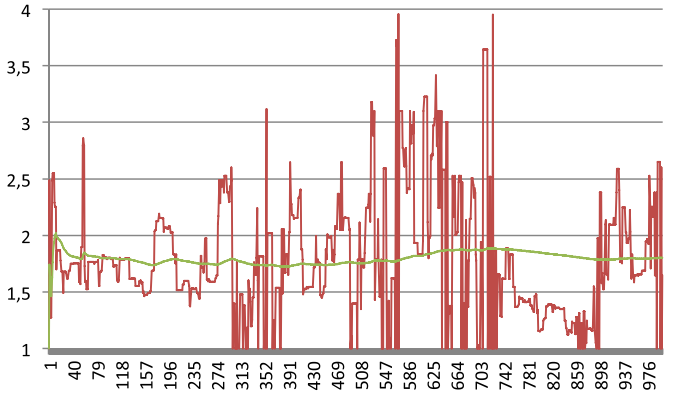

Fig. 8. Number of operations (x-axis) v.s. the inverse value of underallocation (y-axis) for the setting $k = 1$, $b = 1$, $c = 419$.

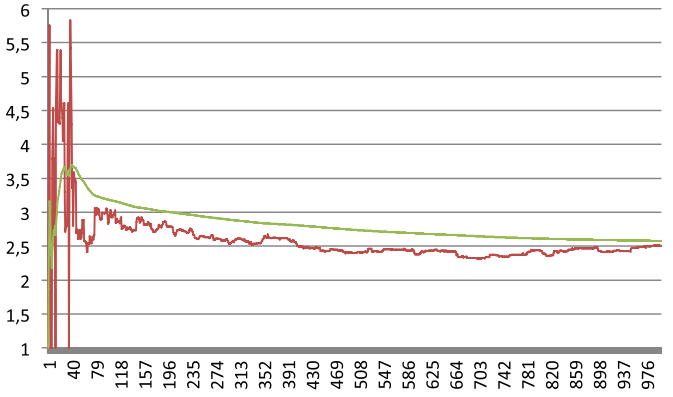

Fig. 9. Number of operations (x-axis) v.s. the inverse value of underallocation (y-axis) for the setting $k = 2$, $b = 0.125$, $c = 232$.

Our simulations indicate the theoretical worst-case bound of $1/4k$ may be overly pessimistic, see Fig. 7–12. In particular, the x-axis represents the number of operations and the y-axis represents the inverse value of under allocations. Furthermore, the red curves illustrate the inverse values of the under allocation and lie below the worst case values of $4k$. Taking into account that a purely one-dimensional approach cannot provide an upper bound on the achievable underallocation, this provides reason to be optimistic about the potential practical performance.

A simulation 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 is shown in Figs. 7–12. Each diagram illustrates the results of an experiment of 1000 requests that are randomly

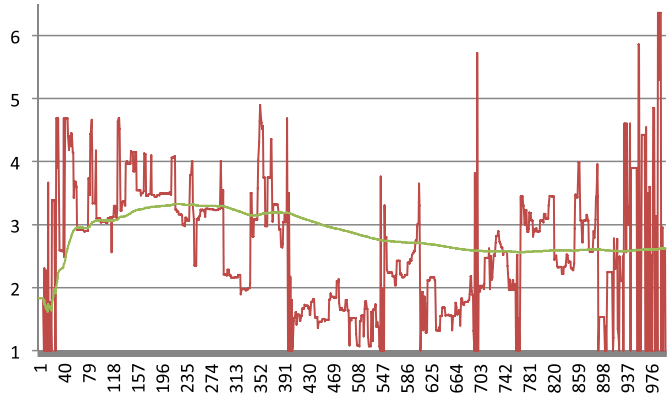

Fig. 10. Number of operations (x-axis) v.s. the inverse value of underallocation (y-axis) for the setting $k = 2$, $b = 1$, $c = 438$.

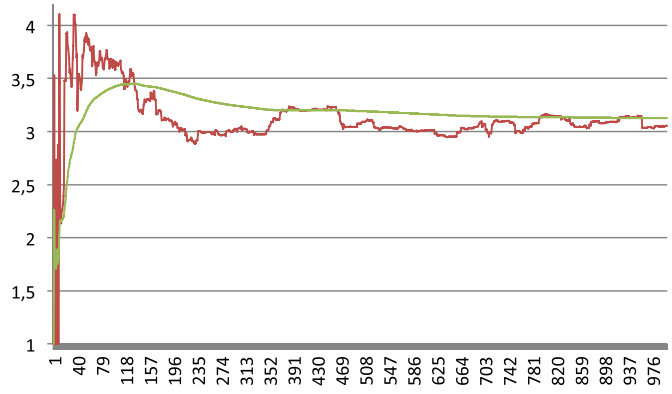

Fig. 11. Number of operations (x-axis) v.s. the inverse value of underallocation (y-axis) for the setting $k = 5$, $b = 0.125$, $c = 264$.

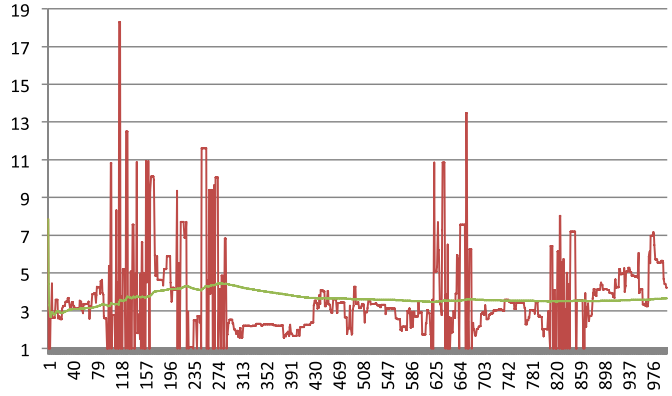

Fig. 12. Number of operations (x-axis) v.s. the inverse value of underallocation (y-axis) for the setting $k = 5$, $b = 1$, $c = 421$.

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 by c the number of collisions, i.e., the situations in that an INSERT$(\cdot)$ cannot be processed.

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, handling heterogeneous chip areas, and advanced algorithmic methods. These will be addressed in future work.

Another aspect of forthcoming work is an explicitly self-refining intermodule wiring. As indicated in Section 3 (and illustrated in Fig. 1), dynamically maintaining this communication infrastructure can be envisioned along the subdivision of the recursive quadtree structure: making the routing a certain proportion of each cell area provides a dynamically adjustable bandwidth, along with intersection-free routing, as shown in Fig. 1 First steps in this direction have been taken with an MA thesis [4], with more work to follow; this also addresses the aspect of robustness of communication in a hostile environment that may cause individual connections to fail.

References

[1] S.P. Fekete, J.-M. Reinhardt, C. Scheffer, An Efficient Data Structure for Dynamic Two-Dimensional Reconfiguration, in: F. Hannig, J.M.P. Cardoso, T. Pionteck, D. Fey, W. SchrÃder-Preikschat, J. Teich (Eds.), Architecture of Computing Systems – ARCS 2016, LNCS, Springer, 2016, pp. 306–318, doi:10.1007/978-3-319-30695-7_23.
[2] I. Kuon, J. Rose, Measuring the gap between FPGAs and ASICs, IEEE Trans. CAD Integr. Circuits Syst. 26 (2007) 203–215.
[3] S.P. Fekete, T. Kamphans, N. Schweer, C. Tessars, J. van der Veen, J. Angermeier, D. Koch, J. Teich, Dynamic defragmentation of reconfigurable devices, ACM Trans. Reconfigurable Technol. Syst. (TRETS) 5 (8) (2012).
[4] C. Meyer, Ausfallsichere Kommunikation in zweidimensionaler Rekonfiguration ("Fail-safe communication in two-dimensional reconfiguration"), Master's thesis. TU Braunschweig, 2016.
[5] G. Bromley, Memory fragmentation in buddy methods for dynamic storage allocation, Acta Inform. 14 (1980) 107–117.
[6] J.A. Hinds, An algorithm for locating adjacent storage blocks in the buddy system, Commun. ACM 18 (1975) 221–222.
[7] D.S. Hirschberg, A class of dynamic memory allocation algorithms, Commun. ACM 16 (1973) 615–618.
[8] K.C. Knowlton, A fast storage allocator, Commun. ACM 8 (1965) 623–625.
[9] K.K. Shen, J.L. Peterson, A weighted buddy method for dynamic storage allocation, Commun. ACM 17 (1974) 558–562.
[10] M.A. Bender, E.D. Demaine, M. Farach-Colton, Cache-oblivious B-trees, SIAM J. Comput. 35 (2005) 341–358.
[11] M.A. Bender, J.T. Fineman, S. Gilbert, B.C. Kuszmaul, Concurrent cache-oblivious B-trees, in: Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures, 2005, pp. 228–237.
[12] M.A. Bender, H. Hu, An adaptive packed-memory array, ACM Trans. Database Syst. 32 (4) (2007), doi:10.1145/1292609.1292616.
[13] M.A. Bender, S.P. Fekete, T. Kamphans, N. Schweer, Maintaining arrays of contiguous objects, in: Fundamentals of Computation Theory, in: LNCS, Springer, 2009, pp. 14–25.
[14] A. Bendersky, E. Petrank, Space overhead bounds for dynamic memory management with partial compaction, ACM Trans. Program. Lang. Syst. 34 (3) (2012) 13:1–13:43, doi:10.1145/2362389.2362392.
[15] N. Cohen, E. Petrank, Limitations of partial compaction: towards practical bounds, in: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, in: PLDI '13, ACM, New York, NY, USA, 2013, pp. 309–320, doi:10.1145/2491956.2491973.
[16] M.A. Bender, M. Farach-Colton, S.P. Fekete, J.T. Fineman, S. Gilbert, Cost-oblivious storage reallocation, in: Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, in: PODS '14, ACM, 2014, pp. 278–288, doi:10.1145/2594538.2594548.
[17] M.A. Bender, M. Farach-Colton, S.P. Fekete, J.T. Fineman, S. Gilbert, Reallocation problems in scheduling, Algorithmica (2014) 1–21, doi:10.1007/s00453-014-9930-4.
[18] S. Fekete, B. Fiethe, S. Friedrichs, H. Michalik, C. Orlis, Efficient reconfiguration of processing modules on FPGAs for space instruments, in: 2014 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2014, pp. 15–22, doi:10.1109/AHS.2014.6880153.
[19] D.E. Knuth, The Art of Computer Programming: Fundamental Algorithms, 1, 3rd, Addison Wesley, Reading, Massachusetts, 1997.
[20] P.R. Wilson, M.S. Johnstone, M. Neely, D. Boles, Dynamic storage allocation: a survey and critical review, in: H. Baker (Ed.), Proc. Internat. Workshop Memory Management, LNCS, 986, 1995.
[21] L. Moser, Poorly formulated unsolved problems of combinatorial geometry, 1966, (Mimeographed).
[22] J. Moon, L. Moser, Some packing and covering theorems, Colloquium Mathematicum 17 (1967) 103–110.

[23] D. Kleitman, M. Krieger, Packing squares in rectangles I, Ann. N. Y. Acad. Sci. 175 (1970) 253–262.
[24] P. Novotný, A note on a packing of squares, Stud. Univ. Transp. Commun. Žilina Math.-Phys. Ser. 10 (1995) 35–39.
[25] P. Novotný, On packing of squares into a rectangle, Arch. Math. (Brno) 32 (2) (1996) 75–83.
[26] S. Hougardy, On packing squares into a rectangle, Comput. Geom. Theory Appl. 44 (8) (2011) 456–463.
[27] J.R. Correa, C. Kenyon, Approximation schemes for multidimensional packing, in: Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA) 2004, 2004, pp. 186–195.
[28] K. Jansen, R. Solis-Oba, A polynomial time approximation scheme for the square packing problem, in: Integer Programming and Combinatorial Optimization, 13th International Conference (IPCO), 2008, pp. 184–198.
[29] R. Harren, Approximation algorithms for orthogonal packing problems for hypercubes, Theor. Comput. Sci. 410 (44) (2009) 4504–4532.
[30] J. Januszewski, M. Lassak, On-line packing sequences of cubes in the unit cube, Geometriae Dedicata 67 (3) (1997) 285–293.
[31] X. Han, K. Iwama, G. Zhang, Online removable square packing, Theory Comput. Syst. 43 (1) (2008) 38–55.
[32] S.P. Fekete, H.-F. Hoffmann, Online square-into-square packing, in: APPROX-RANDOM, 2013, pp. 126–141.
[33] S.P. Fekete, H.-F. Hoffmann, Online square-into-square packing, Algorithmica 77 (3) (2017) 867–901, doi:10.1007/s00453-016-0114-2.
[34] S.P. Fekete, T. Kamphans, N. Schweer, Online square packing, in: 11th International Symposium on Algorithms and Data Structures (WADS), in: LNCS, 5664, Springer Berlin Heidelberg, 2009, pp. 302–314.
[35] S.P. Fekete, T. Kamphans, N. Schweer, Online square packing with gravity, Algorithmica 68 (2014) 1019–1044.
[36] L. Epstein, R. van Stee, Optimal online bounded space multidimensional packing, in: Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004, New Orleans, Louisiana, USA, January 11–14, 2004, 2004, pp. 214–223.
[37] L. Epstein, R. van Stee, Online square and cube packing, Acta Inform. 41 (9) (2005) 595–606.
[38] L. Epstein, R. van Stee, Bounds for online bounded space hypercube packing, Discrete Optim. 4 (2) (2007) 185–197.
[39] Y. Zhang, J.-C. Chen, F.Y.L. Chin, X. Han, H.-F. Ting, Y.H. Tsin, Improved online algorithms for 1-space bounded 2-dimensional bin packing, in: 21st International Symposium on Algorithms and Computation (ISAAC), in: LNCS, 6507, 2010, pp. 242–253.
[40] Y. Azar, L. Epstein, On two dimensional packing, J. Algorithms 25 (2) (1997) 290–310.
[41] A.V. Fishkin, O. Gerber, K. Jansen, R. Solis-Oba, Packing weighted rectangles into a square, in: Mathematical Foundations of Computer Science, International Symposium (MFCS), in: LNCS, 3618, 2005, pp. 352–363.
[42] K. Jansen, G. Zhang, Maximizing the total profit of rectangles packed into a rectangle, Algorithmica 47 (3) (2007) 323–342.
[43] N. Bansal, A. Caprara, K. Jansen, L. Prädel, M. Sviridenko, A structural lemma in 2-dimensional packing, and its implications on approximability, in: Algorithms and Computation, 20th International Symposium, ISAAC, 2009, pp. 77–86.
[44] T. Becker, W. Luk, P.Y. Cheung, Enhancing relocatability of partial bitstreams for run-time reconfiguration, in: Proceedings of the 15th Annual Symposium on Field-Programmable Custom Computing Machines, 2007, pp. 35–44.
[45] M.G. Gericota, G.R. Alves, M.L. Silva, J.M. Ferreira, Run-time defragmentation for dynamically reconfigurable hardware, New Algorithms, Archit. Appl. Reconfigurable Comput. (2005) 117–129.
[46] K. Compton, Z. Li, J. Cooley, S. Knol, S. Hauck, Configuration relocation and defragmentation for run-time reconfigurable systems, IEEE Trans. VLSI 10 (2002) 209–220.
[47] D. Koch, A. Ahmadinia, C. Bobda, H. Kalte, FPGA architecture extensions for preemptive multitasking and hardware defragmentation, in: Proceedings of the IEEE International Conference Field-Programmable Technology, Brisbane, Australia, 2004, pp. 433–436.
[48] M. Koester, H. Kalte, M. Porrmann, U. Ruckert, Defragmentation algorithms for partially reconfigurable hardware, in: International Federation For Information Processing, 240, Publications-IFIP, 2007, p. 41.
[49] G. Morton, A computer oriented geodetic data base and a new technique in file sequencing, Technical Report, IBM Ltd., Ottawa, Ontario, 1966.
[50] B. Brubach, Improved Bound for Online Square-into-Square Packing, in: E. Bampis, O. Svensson (Eds.), Approximation and Online Algorithms, LNCS, Springer International Publishing, 2014, pp. 47–58.

Sándor P. Fekete received the Master's degree (Diploma) in mathematics from the University of Cologne, Cologne, Germany, in 1989, and the Ph.D. degree in combinatorics and optimization from the University of Waterloo, Waterloo, ON, Canada, in 1992, both as a Fellow of the German National Merit Foundation (GNMF). He holds the chair for Algorithmics in the Department of Computer Science, Braunschweig University of Technology, Braunschweig, Germany. His main interests lie in combinatorial optimization, computational geometry, graph algorithms, distributed algorithms, and various applications in computer science, engineering, and economics, such as sensor networks, robot navigation, or traffic control. Following work as a Postdoctoral Research Fellow in computational geometry (SUNY Stony Brook, 1992–1993), he joined the Center for Parallel Computing, the University of Cologne (1993–1999), receiving his habilitation in 1998. After working as an Associate Professor with the Mathematical Optimization Group, TU Berlin, Berlin, Germany (1999–2001), he was a Professor for Mathematical Optimization, Braunschweig (2001–2007), until being appointed to the newly created chair for Algorithmics. He has published over 200 peer-reviewed papers with over 200 coauthors, in publications such as the Journal of the ACM, the SIAM Journal on Computing, Algorithmica, Mathematical Programming, Discrete and Computational Geometry, the Journal of Algorithms, and Operations Research; he has also served on program committees for numerous conferences, including the top-ranking algorithmic meetings SODA, ESA, and SoCG (as co-chair in 2016), and as a reviewer for over 40 journals.

Jan-Marc Reinhardt received his Master's Degree in Computer Science from TU Braunschweig in 2015. He joined the Algorithms Group as a research assistant.

Christian Scheffer received his diploma in Computer Science from the Technische Universität Dortmund, Germany 2011. He received the Ph.D. degree in computer science at the University of Münster, Germany 2014. He became scientific assistant at TU Braunschweig in 2014, and was appointed Academic Council in 2016.