# NO-BREAK DYNAMIC DEFRAGMENTATION OF RECONFIGURABLE DEVICES

*Sándor P. Fekete, Tom Kamphans*[*]
*Nils Schweer, Christopher Tessars*[†],
*Jan C. van der Veen*[*]

Department of Computer Science
Braunschweig University of Technology
Braunschweig, Germany
email: {s.fekete, t.kamphans, n.schweer,
j.van-der-veen}@tu-bs.de

*Josef Angermeier*[‡], *Dirk Koch*[§],
*Jürgen Teich*

Department of Computer Science 12
University of Erlangen-Nuremberg
Erlangen, Germany
email: {angermeier, dirk.koch,
teich}@cs.fau.de

## ABSTRACT

We propose a new method for defragmenting the module layout of a reconfigurable device, enabled by a novel approach for dealing with communication needs between relocated modules and with inhomogeneities found in commonly used FPGAs. Our method is based on dynamic relocation of module positions during runtime, with only very little reconfiguration overhead; the objective is to maximize the length of contiguous free space that is available for new modules. We describe a number of algorithmic aspects of good defragmentation, and present an optimization method based on tabu search. Experimental results indicate that we can improve the quality of module layout by roughly 50% over static layout. Among other benefits, this improvement avoids unnecessary rejection of modules.

## 1. INTRODUCTION

### 1.1. Reconfiguration and Communication

FPGAs suffer from a significant area overhead (monetary cost), a higher power consumption, or a speed penalty as compared to ASIC solutions [1]. Partial runtime reconfiguration is an applicable technique to overcome these issues. By loading just the required modules to an FPGA at runtime, it is possible to build smaller systems, and thus, less power-hungry devices. For instance, an embedded system may start up with some boot-loader and test modules. These modules may be exchanged by a crypto-accelerator to speed up the authentication process of the user. Later, different modules will be loaded to the FPGA by partial runtime reconfiguration with respect to the user demand or the state of the system. Note that a lot of systems provide mutual exclusive functionality (e.g., the record or the play mode of a multimedia device) that is suitable to share some FPGA resources at runtime.

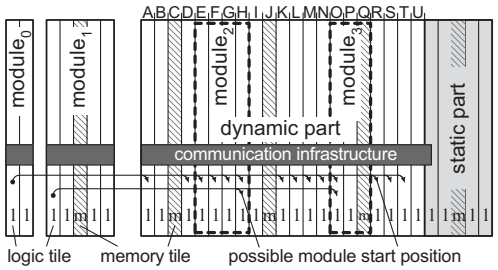


Figure 1: Dynamically reconfigurable system. The system shares some logic tiles $l$ and memory tiles $m$ among a set of modules within the dynamic part of the system. Some modules require a memory tile at a fixed offset with respect to the start position within the modules (e.g., the third tile of $\mathsf{module}_1$ is a memory tile).

However, in order to take more benefit from runtime reconfiguration, such systems should be able to provide the reconfigurable resources in a very flexible way to the modules. Most related work is based on the assumption that a reconfigurable area is used exclusively by one partially reconfigurable module (e.g., [2]) at a point of time. Thus, such approaches do not allow exchanging a large module with multiple smaller ones. This originates from a lack of adequate communication

[*]Supported by DFG grant FE 407/8-2, 8-3, project "ReCoNodes", as part of the Priority Programme 1148, "Reconfigurable Computing".
[†]Supported by BMBF grant 03FEPAI2, project "Advest".
[‡]Supported by DFG grant TE 163/14-2, 14-3, project "ReCoNodes", as part of the Priority Programme 1148, "Reconfigurable Computing".
[§]Supported by DFG grant TE 163/13-2, 13-3, project "ReCoNets", as part of the Priority Programme 1148, "Reconfigurable Computing".

978-1-4244-1961-6/08/$25.00 ©2008 IEEE.

techniques suitable to connect multiple partially reconfigurable modules within the same resource area to the rest of the system.

However, [3] introduces a system that provides a bus based communication for integrating up to 16 partially reconfigurable modules into one large resource area that is divided into 16 individual tiles. Another approach for efficient reconfigurable buses [4] demonstrates that high placement flexibility, low resource overhead, and high throughput can be achieved at the same time. In [5] these authors present a system with a reconfigurable area partitioned into 60 tiles, each being capable to connect a tiny 8 bit module over a so-called ReCoBus to the system. In this system, larger interfaces or modules could be implemented by combining multiple adjacent tiles together (e.g., 4 tiles are required for building a 32 bit interface). In addition, this system can link I/O pins to the partially reconfigurable modules.

When using such systems, an efficient resource management becomes necessary. One problem that has to be solved at runtime is the fragmentation of the tiles due to the time variant execution of some modules on the same resource area. For FPGAs available from the Xilinx Inc., which are best suitable to build dynamically partial reconfigurable systems, the tiles will be vertically aligned column by column (Fig. 1). A module requiring multiple tiles to implement its logic will demand an according consecutive adjacent set of tiles without gaps. This problem is discussed in this paper.

### 1.2. Dynamic Storage Allocation: Old and New

The ever-increasing capabilities of modern reconfigurable devices give rise to a large number of new challenges; solving one of them in turn gives rise to new possibilities and challenges. As described above, there are new solutions for dealing with the communication of relocated devices; this opens up new possibilities for dynamic relocation of modules. The resulting challenge is the dynamic allocation of module requests to a reconfigurable device: given an array-shaped device (e.g., a column-by-column reconfigurable Xilinx Virtex-II FPGA) and a sequence of module requests of varying resource requirements (e.g., logic tiles or memory blocks), assign each module to a contiguous set of slots on the device. (See Fig. 2(a).)

At first glance, this problem has a striking resemblance to one of the classical problems of computing: *dynamic storage allocation* considers a memory array and a sequence of storage requests of varying size, looking for an assignment of each request to a contiguous[1] block of memory cells, such that the length of each block corresponds to the size of the request. Once this allocation has been performed, it is static in space: after a block has been occupied, it will remain to be 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.

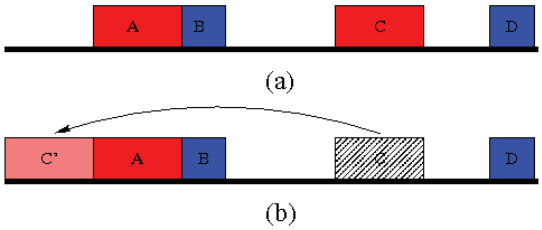


Figure 2: Dynamic storage allocation: (a) Each module occupies a contiguous block of array positions. (b) Moving a module to a new position in order to increase maximum free space.

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 [6] and [7].

Buddy systems partition the storage into a number of standard block sizes and allocate a block in a free subinterval of the smallest standard size sufficient to contain the block. Differing only in the choice of the standard size different, buddy systems are proposed in [8, 9, 10, 11, 12, 6]. Newer approaches that use cache-oblivious structures for allocating space in memory hierarchies include [13, 14].

There are three notable differences between the dynamic allocation of modules to a reconfigurable device and dynamic storage allocation:

1. Using pointers for creating virtual contiguous free blocks is not an option for the placement of modules.
2. In contrast to uniform memory, the reconfigurable device may contain inhomogeneities (e.g., memory tiles).
3. It is possible to relocate modules on a reconfigurable device during runtime, even before a module's lifetime has expired.

There is a certain amount of related work from within the FPGA community. Becker et al. [15] present a method for enhancing the relocability of partial reconfigurability of partial bitstreams for FPGA runtime configuration, with a special focus on heterogeneities. That

[1] Note that this part of the comparison refers to classical research; of course modern storage devices place virtual memory blocks on discontiguous physical space, at the expense of extra overhead for the pointer structures. This approach for allocating discontinuous space is not possible for placing the modules on a reconfigurable device, which is the challenge faced by this paper.

work studies the underlying prerequisites and technical conditions for dynamic relocation. Another relevant approach was presented by Compton et al. [16]; see also Koch et al. [17]. These papers do not consider the algorithmic implications and how the relocation capabilities can be exploited to optimize module layout in a fast, interruption-free, no-break fashion, which is what we consider in this paper. Our approach is significantly different from traditional garbage collection, which would require a freeze of the layout, an offline computation of the new layout, and a complete reconfiguration of all modules. Instead, we just copy one module at a time, and simply switch the running computations to the new module when the move is complete. This leads to *no-break, dynamic defragmentation of module layout*, resulting in much better utilization of the space available for modules.

The rest of this paper is organized as follows. In the following Section 2 we will give a description of the underlying model, giving rise to the problem description in Section 3. As it turns out, solving the corresponding optimization problem is NP-hard, as shown in Section 4; however, for moderate module density, it is still possible to compute optimal results, as shown in Section 5. For higher densities, we develop a heuristic optimization method, based on tabu search and described in Section 6. Experimental results for a Xilinx Virtex-II device are presented and discussed in Section 7. Concluding thoughts are presented in Section 8.

## 2. SCENARIO AND MODEL

When modules are relocated for defragmentation, we have to distinguish between only moving the module configuration, and the configuration and the internal state. In the first case, we just make a copy of the reconfiguration data to the new position and start the next computation on the module at the new position ( e.g., a discrete cosine transformation on the next frame in a video system). In the second case, both modules have to be interrupted and the state (represented by all internal flip-flop and memory values) will be copied to the target module. As compared to the reconfiguration process, the state copying can be performed with short interruption when using hardware-checkpointing [18].

If we would allow overlapping regions for the defragmentation, e.g., the source and the target module may overlap, the interruption time will dominate the reconfiguration process, because we have to copy the routing information and logic settings in addition to the state. As a consequence, we will prevent our defragmentation algorithms to use overlapping regions to place modules.

A further aspect we consider is that common FPGAs typically provide *logic tiles* $l$ and *memory tiles* $m$, as demonstrated in Fig. 1. The placement of a module within the reconfigurable resource area on the FPGA must fit exactly to the particular module. The requirement of a module can be modeled as a string and the search of a valid placement position is then a string matching problem in the reconfigurable resources provided by the dynamic part of the system. This restricts the possible module start positions, and the number of free tiles is not sufficient to determine whether a module can be placed. For instance, $\mathsf{module}_1$ in Fig. 1 has the resource requirement $l\ l\ m\ l\ l$ and can be placed only at the positions A, H, and O, which are currently occupied by the modules $\mathsf{module}_2$ and $\mathsf{module}_3$. In the example, the system has 12 free logic tiles and 2 free memory tiles, but we are currently not able to place $\mathsf{module}_1$, which requires just 4 logic tiles and 1 memory tile, on the FPGA.

## 3. PROBLEM DESCRIPTION

In this paper, we consider a reconfigurable device that allows allocating modules in a contiguous manner on an array $L$ of length $\ell$; modules will be denoted by $M_1, \ldots, M_n$. A module $M_i$ placed in the array occupies a contiguous subinterval, denoted by $L_{M_i}$. Modules are always placed such that $L_{M_i} \cap L_{M_j} = \emptyset$ for $i \neq j$, i. e., two different modules do not overlap.

Modules placed in the array divide $L$ into sections that are occupied by a module and sections that are not occupied; the latter are called *free intervals*. Reconfiguration allows us to relocate a module $M_i$ of size $m_i$ from subinterval $L_{M_i}$ to a new position within a free interval $L_M$ of length $m$ within the array, provided that the following two conditions are fulfilled:

- $L_M \cap \bigcup_{i=1}^{n} L_{M_i} = \emptyset$
- $m \geq m_i$.

Note that the first condition implies that $L_M$ and $L_{M_i}$ are not allowed to intersect, i. e., the subinterval occupied by $M_i$ before the move and the new subinterval have to be disjoint. Furthermore, it ensures that the new position is not occupied by any other module. The second condition ensures that there is sufficient free space to provide a new position for the module.

Now the Maximum Defragmentation Problem (MDP) asks for a sequence of relocation moves, such that the resulting connected free space is as large as possible.

## 4. PROBLEM COMPLEXITY

In this section, we state two complexity results: one for deciding whether one contiguous free block can be formed, and one for the maximization version of the defragmentation problem. We show that the decision version is strongly NP-complete and that no approximation algorithm with a useful approximation factor exists for the maximization version.

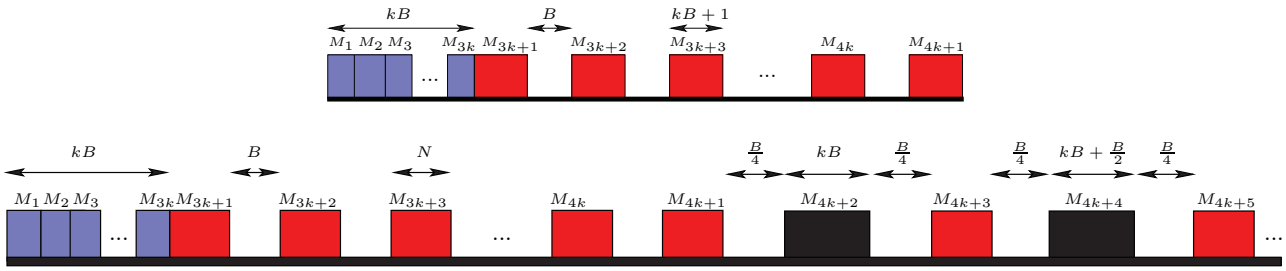


Figure 3: (Top) Reducing 3-Partition to the MDP. (Bottom) Sketch of the inapproximability proof.

The 3-Partition Problem is the main ingredient of the reduction. It belongs to the class of strongly NP-complete and can be stated as follows [19]:

**Given:** A finite set of $3k$ elements $C_1, \ldots, C_{3k}$ with sizes $c_1 \ldots, c_{3k}$, a bound $B \in \mathbb{N}$ such that $c_i$ satisfies $\frac{B}{4} < c_i < \frac{B}{2}$ for $i = 1, \ldots, 3k$ and $\sum_{i=1}^{3k} c_i = kB$.

**Question:** Can the elements be partitioned into $k$ disjoint sets $S_1, S_2 \ldots, S_k$ such that for $1 \leq i \leq k$, $\sum_{C_j \in S_i} c_j = B$?

Because the $c_i$ are lower bounded by $\frac{B}{4}$ and upper bounded by $\frac{B}{2}$, each set $S_j$ contains exactly three elements. We state our complexity result:

**Theorem 1.** *The Maximum Defragmentation Problem with free spaces $F_1, \ldots, F_k$ is strongly NP-complete; moreover, the problem does not allow any deterministic polynomial-time approximation algorithm within any polynomial approximation factor, unless P=NP.*

*Proof.* Given an instance of the 3-Partition problem with input $c_1 \ldots, c_{3k}$, we construct an instance of the MDP in the following way: we place $3k$ modules $M_1, \ldots, M_{3k}$ with $m_i = c_i$, $1 \leq i \leq 3k$, side by side, starting at the left end of $L$. Then starting at the right boundary of $M_{3k}$ we place $k + 1$ modules of size $kB + 1$, alternating with $k$ free spaces of size $B$. We denote these modules by $M_{3k+1}$ to $M_{4k+1}$ and the free spaces by $F_1$ to $F_k$. Fig. 3(Top) shows the overall structure of the constructed instance. Now we ask for the construction of a free space of size $K = kB$. Because the size of the total free space is equal to $kB$, none of the modules $M_{3k+1}, \ldots, M_{4k}$ can ever be moved. Hence, the only way to connect the total free space is to move the modules $M_1$ to $M_{3k}$ to the free spaces. But any solution of this kind implies a solution to the given 3-Partition instance.

This construction can also be used to establish the claimed inapproximability result: as can be seen from Fig. 3 (b), the gap between the possible solution values in case of existence and nonexistence of a 3-Partition can be made arbitrarily big. □

## 5. MODERATE DENSITY

In this section, we consider a special case in which the MDP can be solved with linear computing time and at most $2n$ moves. We define for an array $L$ of length $\ell$ the density to be $\delta = \frac{1}{\ell} \sum_{i=1}^{n} m_i$. We show that if

$$\delta \leq \frac{1}{2} - \frac{1}{2\ell} \cdot \max_{i=1,\ldots,n} \{m_i\} \tag{1}$$

the total free space can always be connected with $2n$ steps by Algorithm 1:

**Algorithm 1**: LeftRightShift

**Input**: A array $L$ with $n$ modules $M_1, \ldots, M_n$ such that (1) is fulfilled.

**Output**: A placement of $M_1, \ldots, M_n$ such that there is only one free space at the left end of $L$.

```
for i = 1 to n do
    Shift M_i to the left as far as possible.
end
for i = n to 1 do
    Shift M_i to the right as far as possible.
end
```

We need the following two observations for proving the correctness of Algorithm 1. Both follow immediately from the definition of the density and from (1); in the following, $f_i$ denotes the size of $F_i$.

$$\sum_{i=1}^{k} f_i \geq \frac{l}{2} + \frac{1}{2} \cdot \max_{i=1,\ldots,n} \{m_i\} \tag{2}$$

$$\delta < \frac{1}{2} \text{ and therefore } \sum_{i=1}^{n} m_i < \sum_{j=1}^{k} f_j \tag{3}$$

**Theorem 2.** *Algorithm 1 connects the total free space with at most $2n$ moves and uses $O(n)$ computing time.*

*Proof.* The number of shifts and the computing time are obvious. We will show that at the end of the first

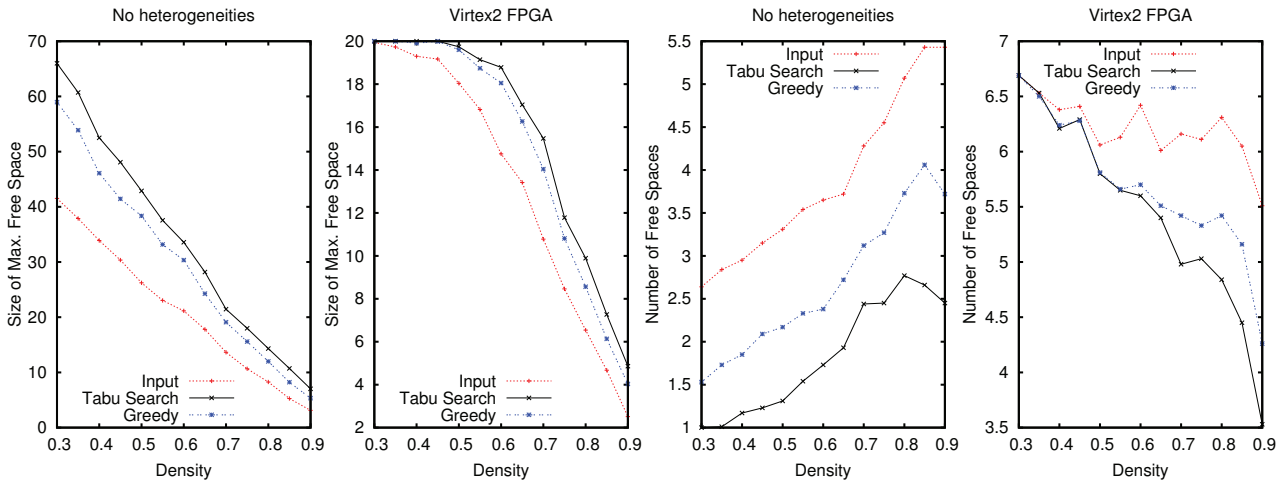


Figure 4: (Left to right) Size of the maximal free space before and after defragmentation using our heuristic and a simple greedy approach in an array with no heterogeneities. Size of the maximal free space before and after defragmentation of the Virtex-II FPGA. Number of free space before and after defragmentation in an array with no heterogeneities. Number of free spaces before and after defragmentation of the Virtex-II FPGA.

loop, the rightmost free space is greater than any module and therefore *all* modules can be shifted to the right in the second loop.

Let $F_1, \ldots, F_k$ denote the free spaces in $L$ at the end of the first loop. Then every $F_i$, $i \in \{1, \ldots, k-1\}$ is bounded to the right by a module $M_j$ with $m_j > f_i$ (otherwise $m_j$ could be shifted). If this would hold for $F_k$ as well, we could conclude that $\sum_{i=1}^{k} f_i < \sum_{i=1}^{n} m_i$, which contradicts (3). Hence, there is no module to the right of $F_k$ and we get with $m^\star = \max_{1,\ldots,n}\{m_i\}$

$$\frac{l}{2} + \frac{1}{2}m^\star \overset{(2)}{\leq} \sum_{i=1}^{k} f_i < \sum_{i=1}^{n} m_i + f_k \overset{(1)}{\leq} \frac{l}{2} - \frac{1}{2}m^\star + f_k$$

implying $m^\star < f_k$. □

## 6. A HEURISTIC METHOD

We implemented a standard tabu search with a tabu list of length $\frac{n}{2}$. In every iteration all homogeneous modules $M_i$ are moved to the left end and the right end of the free subintervals that are greater than or equal to $m_i$. All inhomogeneous modules are moved to any feasible position. Each move is evaluated by a fitness function that divides the size of the maximal free space by the size of the total free space. The move yielding the configuration with the highest fitness is chosen. Ties are broken by choosing the first one. The resulting configuration is added to the tabu list.

If the current solution is the best one found so far it is stored. The heuristic ends if either a fitness of 1.0 (i.e., optimality) is achieved or $2n^2$ iterations have been performed. There are instances where $O(n^2)$ moves are necessary; see [20] for details. Moreover, we conjecture that the number of necessary moves is in $\Theta(n^2)$.

## 7. EXPERIMENTAL RESULTS

We performed experiments on two different arrays, both having 94 slots. The first array does not contain any heterogeneities, while the second one is the Virtex-II FPGA with heterogeneities at positions 3, 24, 45, 50, 71, and 82. Moreover, we compared our heuristic with a simple greedy approach which moves every module to the most promising position (i.e., to the position where the ratio of the size of the maximal free space and the size of the total free space is maximal).

Generating the input was done in two steps, depending on the size of the maximal free subinterval $F^\star$. In the first step the module size is chosen with equal probability from the set $\{1, \ldots, f^\star\}$. This ensures that the modules can be inserted. The exact position is chosen again with equal probability among all feasible positions inside $F^\star$. If the subinterval occupied by the module contains an heterogeneity, this heterogeneity is assigned to the corresponding position of the module. The size of the first module is shrunken by a factor of 0.6 in order to ensure that it can be moved.

For the density ranging from 0.3 to 0.9 with steps of size 0.05 we performed 100 runs of the tabu search for each value and took the average value of the *number of free subintervals* and the *size of the maximal free subinterval*. The results are show in Fig. 4. The left diagrams show the size of the maximal free space of

the array before the defragmentation and the size afterwards. In the array with no heterogeneities there is an improvement of up to 40%. On the Virtex-II FPGA the size of any maximal free space is limited to 20 slots due to the heterogeneities. For a density of less the $\frac{1}{2}$ the tabu search achieves this upper bound for almost all instances. For larger densities it achieves an improvement of approximately 35%.

The change in the number of free spaces before and after defragmentation is displayed in the right charts of Fig. 4. In the array with no heterogeneities there is an increase of 50%. For the Virtex-II FPGA there is almost no improvement for low densities (less then $\frac{1}{2}$) and an improvement of approximately for larger ones.

## 8. CONCLUSION

In this paper we have presented a new approach for defragmenting the module layout on a dynamically reconfigurable device in a no-break fashion. Obviously, improved algorithmic results can lead to further improvements. One of the possible extensions considers a more controlled overall placement of modules, instead of simply fixing fragmentation. As the necessary algorithmic methods are more involved, we leave this to future work.

## 9. REFERENCES


[1] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *IEEE Trans. CAD Integr. Circuits Systems*, vol. 26, pp. 203–215, Feb. 2007.

[2] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford, "Invited paper: Enhanced architecture, design methodologies and CAD tools for dynamic reconfiguration of Xilinx FPGAs," in *Proc. 16th Internat. Conf. Field Programm. Logic Appl.*, Aug 2006, pp. 1–6.

[3] J. Hagemeyer, B. Kettelhoit, M. Koester, and M. Porrmann, "Design of homogeneous communication infrastructures for partially reconfigurable fpgas," in *Proc. Internat. Conf. Engineering of Reconf. Systems and Algor.*, Las Vegas, USA, June 2007.

[4] D. Koch, C. Haubelt, and J. Teich, "Efficient reconfigurable on-chip buses for FPGAs," in *Proc. 16th Annu. IEEE Sympos. Field-Programm. Custom Comput. Mach.*, Palo Alto, CA, USA, Apr. 2008.

[5] D. Koch, C. Beckhoff, and J. Teich, "ReCoBusBuilder — a novel tool and technique to build static and dynamically reconfigurable systems for FPGAs," in *Proc. 18th Internat. Conf. Field Programm. Logic Appl.*, 2008, submitted to.

[6] D. E. Knuth, *The Art of Computer Programming: Fundamental Algorithms*, 3rd ed. Reading, Massachusetts: Addison Wesley, June 1997, vol. 1.

[7] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic storage allocation: A survey and critical review," in *Proceedings of International Workshop on Memory Management*, ser. Lecture Notes in Computer Science, H. Baker, Ed., vol. 986. Kinross, Scotland: Springer-Verlag, Sept. 1995. [Online]. Available: ftp://ftp.cs.utexas.edu/pub/garbage/allocsrv.ps

[8] G. Bromley, "Memory fragmentation in buddy methods for dynamic storage allocation," *Acta Inform.*, vol. 14, pp. 107–117, 1980.

[9] J. A. Hinds, "An algorithm for locating adjacent storage blocks in the buddy system," *Commun. ACM*, vol. 18, pp. 221–222, 1975.

[10] D. S. Hirschberg, "A class of dynamic memory allocation algorithms," *Commun. ACM*, vol. 16, pp. 615–618, 1973.

[11] K. C. Knowlton, "A fast storage allocator," *Commun. ACM*, vol. 8, pp. 623–625, 1965.

[12] K. K. Shen and J. L. Peterson, "A weighted buddy method for dynamic storage allocation," *Commun. ACM*, vol. 17, pp. 558–562, 1974.

[13] M. A. Bender, E. D. Demaine, and M. Farach-Colton, "Cache-oblivious B-trees," *SIAM J. Comput.*, vol. 35, pp. 341–358, 2005.

[14] M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuszmaul, "Concurrent cache-oblivious B-trees," in *Proc. 17th Annu. ACM Sympos. Parallel. Algor. Architect.*, 2005, pp. 228–237.

[15] T. Becker, W. Luk, and P. Y. Cheung, "Enhancing relocatability of partial bitstreams for run-time reconfiguration," in *Proc. 15th Annu. Sympos. Field-Programm. Custom Comput. Mach.*, 2007, pp. 35–44.

[16] K. Compton, Z. Li, J. Cooley, S. Knol, and S. Hauck, "Configuration relocation and defragmentation for run-time reconfigurable systems," *IEEE Transact. VLSI*, vol. 10, pp. 209–220, 2002.

[17] D. Koch, A. Ahmadinia, C. Bobda, and H. Kalte, "FPGA architecture extensions for preemptive multitasking and hardware defragmentation," in *Proc. IEEE Internat. Conf. Field-Programmable Technology*, Brisbane, Australia, 2004, pp. 433–436.

[18] D. Koch, C. Haubelt, and J. Teich, "Efficient hardware checkpointing—concepts, overhead analysis, and implementation," in *Proc. 15th ACM/SIGDA Internat. Sympos. Field-Programm. Gate Arrays*. Monterey, California, USA: ACM, 2007, pp. 188–196.

[19] M. R. Garey and D. S. Johnson, *Computers and intractability; a guide to the theory of NP-completeness*. W.H. Freeman, 1979.

[20] S. P. Fekete, T. Kamphans, N. Schweer, C. Tessars, J. van der Veen, J. Angermeier, D. Koch, and J. Teich, "No-break dynamic defragmentation of reconfigurable devices," submitted, 2008.