Defragmentation Algorithms for Partially Reconfigurable Hardware

Markus Koester[1], Heiko Kalte[2], Mario Porrmann[1], and Ulrich Rückert[1]

[1] Heinz Nixdorf Institute, System and Circuit Technology,
University of Paderborn, Germany
{koester, porrmann, rueckert}@hni.upb.de
[2] School of Computer Science and Software Engineering,
University of Western Australia, Australia
heiko@csse.uwa.edu.au

Abstract. Dynamic reconfiguration is a promising approach for resource efficient utilization of microelectronic systems. Standard platforms for partial dynamic reconfiguration are field-programmable gate arrays (FPGAs). Multiple hardware tasks can share the same FPGA resources over time, which increases the device utilization in comparison to non-reconfigurable systems. Although, similar resource management is already known in the area of operating systems, there is a requirement to adapt these concepts to the special needs of dynamically reconfigurable systems. Additionally, there is a lack of underlying mechanisms, e.g., to suspend hardware tasks and restart them at a different position within the FPGA. In this article we introduce a mechanism for task relocation that includes saving and restoring of state information of the task. Based on this approach we address the problem of defragmentation. We present defragmentation algorithms that minimize different types of costs. With the help of a detailed simulation model and a benchmark, we finally provide realistic simulation results and compare the different algorithms.

1 Introduction

Field Programmable Gate Arrays (FPGAs) are reconfigurable architectures that enable the integration of complete systems on a single chip. Currently available FPGAs have the feature of partial reconfiguration, which offers a high flexibility. Arbitrary functions in form of a hardware task can be configured on demand and can be removed after execution at run-time thus allowing the sharing of FPGA hardware resources over time. With the increasing amount of hardware resources, dynamically exchanging hardware tasks require a resource management and methods for placement and relocation of the tasks. While several approaches address the problem of placing tasks on partial reconfigurable FPGAs [1, 10], there is a lack of underlying mechanisms, e.g., to suspend hardware tasks and restart them at another time or relocate them to another area of the FPGA. In this paper we describe an approach to an efficient task relocation

Koester, M., Kalte, H., Porrmann, M., Rückert, U., 2007, in IFIP International Federation for Information Processing, Volume 240, VLSI-SoC: From Systems to Silicon, eds. Reis, R., Osseiran, A., Pfleiderer, H-J., (Boston: Springer), pp. 41–53.

at run-time. The necessary relocation mechanisms are mainly implemented in hardware allowing to save and restore state information while relocating the hardware task.

Recurrent allocation and de-allocation of various sized tasks cause the free FPGA resources to split into small fragments over time. But for placing a hardware task the FPGA resources need to be available contiguously in a single block. In order to increase the utilization of the FPGA, tasks can be rearranged at run-time by using the relocation mechanisms with the aim to cluster free resources to larger blocks, thus enabling placement of larger hardware tasks. This process is called defragmentation. In this paper we present defragmentation algorithms with the objective to minimize different reconfiguration costs. The defragmentation algorithms have been implemented in our simulation framework SARA. Simulation results show that the defragmentation algorithm we present here can be useful to increase the utilization of the FPGA.

2 Task Relocation

The basic requirement for all kinds of task reorganizations (including defragmentation) on the FPGA fabric is a proper mechanism to stop and relocate a running task. In almost all cases this means that not only the hardware structures of the task have to be relocated, but also the current state information that are stored in registers and memory. In order to relocate a task, the current state information have to be read, the new instance of the task has to be placed, the state information have to be restored, and finally the old instance of the task has to be erased. There are basically two approaches to read and restore state information that are stored in registers and memory all over the FPGA area of the task.

The *Task Specific Access Structures* approach realizes reading and restoring by adding an extra read/write interface to all state registers which leads to extra resource consumption and especially to extra design effort. Consequently, each hardware task has to be redesigned to be used in a reconfigurable environment. However, one advantage of this approach is the high data efficiency, as only the raw state information are read. In [9] Ullmann et al. have presented an implementation of this approach.

In contrast to that, the *Configuration Port Access* approach is based on the bitstream readback facilities of the configuration port (in our case the Xilinx SelectMAP/ICAP interface, see [11]) of the FPGA. This port offers the possibility to read arbitrary columns of the configuration memory including the current register values and the RAM contents. After or during reading the bitstream, the state information have to be filtered out of the readback stream. Before configuring the new instance the preset bits of the flip flops and the RAM content are modified according to the previously extracted state information (see [8]). As the Configuration Port Access approach uses the inherent access structures of the configuration circuitry and the configuration port, no hardware struc-

tures have to be added to the tasks itself. However, one disadvantage of this approach is the low efficiency, as the portion of state information in the read data can be less than 1%.

2.1 Our Relocation Approach

We have developed a relocation approach that is based on the Configuration Port approach. As simply all register values are stored there is no need to know anything about the internal structure or behavior of the task and no extra design effort has to be spent. In contrast to existing implementations that are based on the Configuration Port Access approach (e.g., by Simmler et al. in [8]), our approach does not read all configuration data, but only those that include state information and belong to the task to be suspended. Furthermore, the actual state information extraction is not done after but during reading the configuration data. These differences to other approaches significantly reduce the amount of data to be read back, the data to be stored, and finally the processing time.

Platform Information
The mechanism of task relocation basically depends on the underlying FPGA architecture and on the degree of freedom during the task placement (2D-, 1D-placement or fixed task slots). We use the Xilinx Virtex FPGAs because these are the only devices which combine system level complexity and partial reconfiguration (in a column-wise manner). The internal configuration memory of a Virtex FPGA stores the bitstream and can be visualized as a rectangular array of bits. The bits are grouped into one bit wide vertical columns that extend from the top of the device to the bottom. These so called *frames* are the atomic unit of configuration and are addressed by the major address (MJA) and the minor address (MNA). A detailed description can be found in [11]. The column-wise reconfigurability of the Virtex FPGAs also inspired our reconfigurable system approach [7]. All hardware tasks can be dynamically placed, relocated and erased along a horizontal communication infrastructure (1D-placement). The communication infrastructure is completely homogeneous, which makes it possible to dynamically relocate hardware tasks along the horizontal bus structure. This relocation process can be realized by bitstream manipulations that change the column addresses (MJA) of individual hardware tasks during the download process of the configuration bitstream (see Fig. 1, [7] and [5] for further information).

Architecture Overview
The architecture of our context relocation approach can be seen in Figure 1. There are four main function blocks and a database to perform a relocation process. The main blocks are the *Configuration Manager*, the *State Extraction Filter*, the *State Inclusion Filter* and the *REPLICA Filter*. The first step of a context relocation process is to stop the clock of the particular hardware

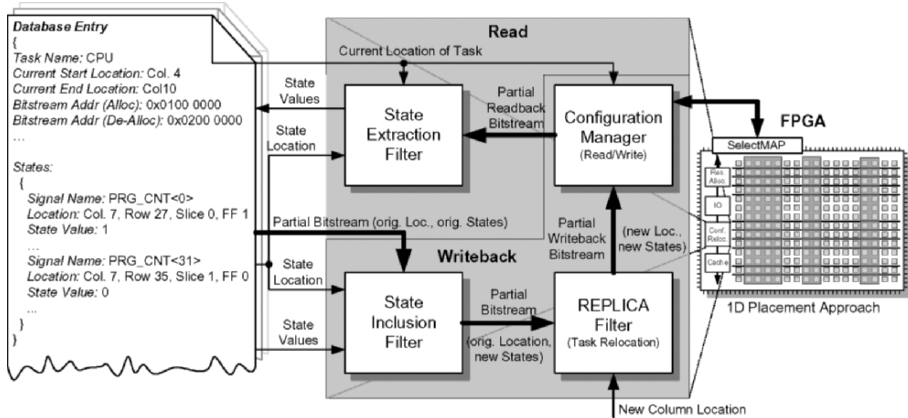

Fig. 1. Relocation Approach Overview.

task or of all hardware tasks to prevent state changes during the read process (e.g., by clock gating). Subsequently, the Configuration Manager initiates the SelectMAP interface to read all frames that contain state information. The addresses of the frames are calculated on basis of the location information given by a database entry of the task. The database stores the current location of each task, the memory addresses of the partial bitstreams and finally the location of all state registers. During the read process, all frames are continuously transferred to the State Extraction Filter, which determines the state value within each frame. The task is now suspended, but not deallocated. That means, a partial "empty" bitstream has to be downloaded to completely erase the circuitry of the task. The restoring process starts with the State Inclusion Filter, which inserts the register values of the database into the original partial bitstream of the hardware task. The resulting bitstream would still allocate the task at its original location, but with the new initial register states. Therefore, the REPLICA Filter relocates the hardware task from its original location to the FPGA column that is determined by the *New Column Location* input. Finally, the new partial bitstream, which is relocated and includes the states, is downloaded by the Configuration Manager. After resetting the hardware task, all registers are set to the proper value and the task can start processing in exactly the state it was interrupted before. In the following, the four blocks are described in more details.

The *Configuration Manager* is connected to the SelectMAP configuration interface to read and write configuration data. When writing a bitstream the Configuration Manager reads 32-bit bitstream words from arbitrary memory locations and converts them to 4×8-bit bitstream words, which are passed to the configuration interface. For performance reasons, this part is implemented in hardware (see [5] for further details). When reading the state information, the Configuration Manager selects only the frames that contain state information.

Therefore, the Configuration Manager takes the column (*Col*), slice (*Slice*) and flip flop (*FF*) values of the database entries for each state bit and generates an address of the frame that contains the current state value. The frame address consists mainly of the major address (MJA) and the minor address (MNA). Equation(1) and (2) show the necessary calculations (*Chip_Cols* determines the maximum CLB column number of the FPGA).

$$MJA = Chip_Cols - Col \cdot 2 + 2 \quad \text{(left chip half and Virtex only)} \tag{1}$$

$$\begin{aligned} MNA = {} & Slice \cdot (12 \cdot FF - 43) - 6 \cdot FF + 45 \\ & \text{with } Slice, FF \in \{0, 1\} \\ & \Rightarrow MNA \in \{2, 8, 39, 45\} \end{aligned} \tag{2}$$

The MNA can have only four different values, which means all flip flop states of one CLB column are stored in only 4 frames. This results in a heavy reduction of the amount of data to be read, as a complete CLB column consist of 48 frames. Consequently, it makes sense to implement tasks in as few CLB columns as possible to ensure a reasonable amount of state information in each frame that is read. The output of the Configuration Manager is finally a stream of single frames that contain the state information of the hardware task.

The *State Extraction Filter* takes the readback stream of the Configuration Manager, extracts the state values and updates the database entries. For extracting the state value, the filter determines the bit index within the readback frames by using the following equation (see also [11]).

$$Bit_idx = (18 \cdot row) + 1 \tag{3}$$

As a result, the bit index only depends on the CLB row of the appropriate flip flop, which means that all flips flop values of the same column and the same type (e.g. Slice=0, FF=1) are located within one frame.

The *State Inclusion Filter* performs the first step of the restoring process. The filter takes the original partial bitstream of the hardware task and inserts all database state values by manipulating the preset bit of the registers. Similar to the state extraction process, the frame address and bit index of all state bits have to be calculated. The computation of the MJA and the bit index are the same as for the state extraction process (cf. (1) and (3)); solely the MNA values are different. See [8] for further information.

The *REPLICA Filter* is capable of relocating tasks by manipulating the partial bitstream of the task. Downloading the output stream of the State Inclusion Filter would allocate the task at its original location (after initial place and route). However, in most cases a new location has to be found according to the current resource allocation. In order to perform the proper manipulations, the REPLICA filter parses the bitstream and replaces the column addresses (MJAs) within the bitstream. The relocation process can only be performed horizontally. The necessary manipulation, including the update of the CRC

(Cyclic Redundancy Check) values within the bitstream, is implemented in hardware and does not cause any extra time overhead. The architecture and the hardware implementation of the REPLICA filter as well as an example application are published in [5].

2.2 Relocation Time Overhead

A key performance issue in a reconfigurable system approach is the time overhead to place or relocate a hardware task. The relocation time in our hardware implemented relocation approach consists of three times: the state capture time, the de-allocation time, and finally the allocation time. The bitstream manipulation processes of state inclusion and task relocation are assumed to be completely hidden in the task allocation time, which has already been shown for the task relocation with the REPLICA filter in [5].

The total time for relocating a task depends on the number of utilized CLB columns N_{cols}, the frame size $N_{Byte/Frame}$, and the SelectMAP frequency $f_{SelectMap}$. For each CLB column, which is to be relocated, 4 frames have to be read for capturing the states of the flip-flops (see Eq. (2)). The first frame of every new read access is always a pad frame, which does not contain any significant data. Hence, in order to capture all states of a CLB column $2 \cdot 4 = 8$ frames have to be read and the total time for capturing the states of the flip flops is:

$$T_{cap} = \frac{8 \cdot N_{cols} \cdot N_{Byte/Frame}}{f_{SelectMap}} \quad (4)$$

For the allocation of a task 48 frames per task column must be written (see [11] for further details) and the allocation time of a task is:

$$T_{alloc} = \frac{48 \cdot N_{cols} \cdot N_{Byte/Frame}}{f_{SelectMap}} \quad (5)$$

If the time for allocating and de-allocating a task is assumed to be the same ($T_{del} = T_{alloc}$), the time for a complete task relocation can be approximated by:

$$T_{reloc} \approx T_{cap} + T_{alloc} + T_{del} = \frac{104 \cdot N_{cols} \cdot N_{Byte/Frame}}{f_{SelectMap}} \quad (6)$$

Equation (6) assumes a de-allocation process for every task relocation, but as described in Section 3, the de-allocation can be avoided if it is ensured that the task area is overwritten anyway (e.g. during a defragmentation process).

In order to give an overview of realistic relocation times we have implemented several designs on an XCV2000E device (see [6] for further details). The frame length of this device is 196 bytes and the SelectMAP frequency is 50 MHz. The task size ranges from 1 (8-bit divider) to 36 (RISC-CPU) CLB columns (30% of the device) and the overall relocation time ranges from 0.4 ms

(1 CLB column) to 14.8 ms (36 CLB columns). For each task the time for capturing the states is only 8.2% of the complete relocation time. This is because the de-allocation and allocation time outweighs the state capturing process.

In the following section various run-time defragmentation algorithms are discussed, that consider the underlying mechanisms and timing models as described in this section. By using the approximation of the relocation times, simulations of a run-time defragmentation can be performed under realistic timing constraints.

3 Defragmentation Algorithms

In dynamic reconfigurable systems recurrent allocations and de-allocations of various sized tasks cause a so called *external fragmentation*, i.e., the contiguous regions of unused reconfigurable cells gradually become scattered in small fragments all over the FPGA. An important criteria for the placement of a requested hardware task is the largest contiguous region of unused reconfigurable cells. Any hardware task larger than that region cannot be placed. A solution to increase the size of the region is to apply run-time defragmentation, i.e., to relocate currently configured hardware tasks aiming to cluster the unused cells in one contiguous region. In [3] Diessel et al. described the one-dimensional order-preserving compaction used for defragmentation in 2D system approaches. The idea of one-way one-dimensional order-preserving compaction is sliding the allocated hardware tasks to be compacted in a single direction along a single dimension while preserving their relative order. The concept of this algorithm can be adapted to the 1D system approach described in [7] since hardware tasks are inherently placed in a single dimension. Algorithm 1 is showing the principle of one-way one-dimensional order-preserving compaction. Consider a set of allocated hardware tasks $M = \{m_1, m_2, ...\}$. In the one-dimensional approach the position $x(m)$ of a hardware task $m \in M$ can be fully described by the leftmost cell column of the task. The width $w(m)$ of a hardware task can be described by the number of cell columns that are used by the task. The defragmentation according to Algorithm 1 is performed within the so called *defragmentation area* from column i_{start} to column i_{end}. M_{defrag} is the set of hardware tasks which are located within the defragmentation area (line 1). i_{cur} is the currently selected column for the placement of the tasks and is initialized by the value i_{end} (line 2). Inside the loop (lines 3-8) the task m is selected, which is located rightmost within the defragmentation area (line 4). The selected hardware task m is relocated by sliding it rightmost within the defragmentation area (lines 5-6). After relocation, the hardware task m is removed from the set M_{defrag} (line 7) and the loop is repeated until all tasks are compacted at the right. As a result of the defragmentation, a single region with unused reconfigurable cells is located starting from position i_{start}.

Although applying the defragmentation to the whole FPGA will result in an optimal situation with no fragmentation, where all unused cells are located

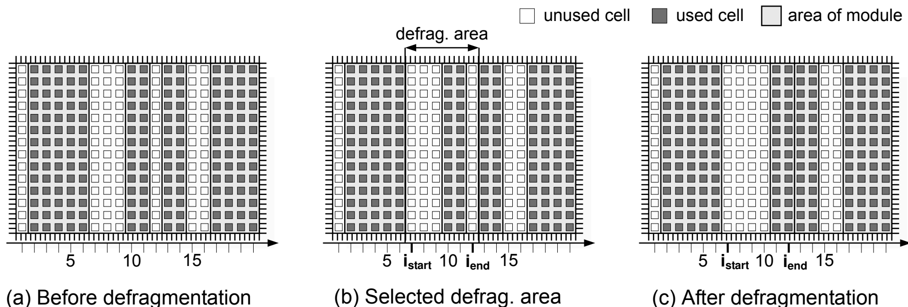

Fig. 2. Example for a locally defragmentation using the 1D system approach.

in a single block, probably all hardware tasks need to be relocated, which will cause a large reconfiguration overhead.

The defragmentation time is derived by the sum of the relocation times of the hardware tasks that are located within the defragmentation area. According to Section 2.2 the relocation time of a hardware task basically depends on the SelectMAP frequency and the task size (number of cell columns). While the SelectMAP frequency is given by the hardware architecture, the only parameter that influences the time for defragmentation is the number of cell columns to be relocated. In order to avoid a long defragmentation time with a large reconfiguration overhead, it is therefore necessary to keep the number of cell columns to be relocated as low as possible.

Whenever a requested hardware task cannot be placed due to fragmentation, sometimes only a few tasks need to be relocated to allow a placement. Hence, to reduce the reconfiguration overhead, the defragmentation can be performed only locally by selecting a suitable defragmentation area. The selection of the defragmentation area can be influenced by the following objectives:

Task Movements: If a requested hardware task m cannot be placed due to fragmentation, one objective for the defragmentation can be to minimize the number of hardware task movements. For this, we need to define the availability vector:

$$b(i) = \begin{cases} 0 \text{ if cell column } i \text{ is used} \\ 1 \text{ if cell column } i \text{ is unused} \end{cases} \tag{7}$$

Consider $w(m)$ is the width of the requested hardware task m, then the bounds of the defragmentation area can be found by solving the following optimization problem:

$$\text{Minimize } |M_{defrag}| \text{ subject to } \sum_{n=i_{start}}^{i_{end}} b(n) = w(m).$$

Column Movements: Minimizing the hardware task movements as described above does not necessarily lead to the least reconfiguration overhead, since the

Input Set of allocated hardware tasks $M = \{m_1, m_2, ...\}$, position of the tasks $x(m)$ (origin:left), width of a task $w(m)$, boundaries of the defragmentation area i_{start} and i_{end} under the condition $b(i_{end}) = 1$ and $b(i_{start}) = 1$.
Output New positions $\tilde{x}(m)$ of the tasks within the defragmentation area.

(1) $M_{defrag} \leftarrow \{m \mid m \in M \ \wedge \ i_{start} \leq x(m) \ \wedge \ x(m) + w(m) \leq i_{end}\}$
(2) $i_{cur} \leftarrow i_{end}$
(3) **while** $M_{defrag} \neq \{\}$
(4) **select an** $m \in M_{defrag}$ **with maximum** $x(m)$
(5) $i_{cur} \leftarrow i_{cur} - w(m)$
(6) $\tilde{x}(m) \leftarrow i_{cur} + 1$
(7) $M_{defrag} \leftarrow M_{defrag} \setminus m$
(8) **end while**

Algorithm 1: 1D defragmentation.

hardware tasks in M_{defrag} can be large and therefore cause a long reconfiguration time. Another approach is to consider the required column movements rather than the required hardware task movements. In this case, the bounds of the defragmentation area can be found by solving a similar optimization problem:

$$\text{Minimize } i_{end} - i_{start} \text{ subject to } \sum_{n=i_{start}}^{i_{end}} b(n) = w(m).$$

Cost: Apart from configuration aspects such as column or hardware task movements mentioned above, the bounds of the defragmentation area can be derived with respect to parameters like, e.g., priorities of the allocated hardware tasks, or the expected remaining time of the allocated hardware tasks.

Let us assume the function $p(m) \in [0, 1]$ describes the priority of the hardware task m. If $p(m) = 0$ the hardware task m has the least priority and if $p(m) = 1$ the hardware task m has the highest priority. In order to find a defragmentation area with a low overall priority, the following optimization problem must be solved:

$$\text{Minimize } \sum_{m \in M_{defrag}} p(m) \text{ subject to } \sum_{n=i_{start}}^{i_{end}} b(n) = w(m).$$

Regardless of the chosen objective – by solving one of the described optimization problems and moving all allocated hardware tasks within column i_{start} and column i_{end} to the right as described by Algorithm 1, the requested hardware task m can be placed at column i_{start}.

An example of the defragmentation is shown in Figure 2. Consider a requested hardware task m with the width $w(m) = 4$ and the reconfigurable architecture is in a configuration as shown in Figure 2(a). In the current configuration the placement of m is not possible although enough free configurable

cells are available. Applying the defragmentation with respect to minimal column movements results in a defragmentation area as shown in Figure 2(b) with $i_{start} = 7$ and $i_{end} = 12$. After defragmentation the allocated hardware task within the defragmentation area is located rightmost, such that an unused region for placing the requested hardware task m is located at position $i_{start} = 7$ as shown in Figure 2(c).

4 Simulation Results

The defragmentation algorithms specified in Section 3 have been implemented in the Simulation Framework for Analyzing Reconfigurable Architectures (SARA). SARA is a discrete event simulator introduced in [4], which enables a realistic simulation of system approaches for partially reconfigurable architectures.

The allocation of a hardware task is performed under real world conditions, i.e., the configuration is done by simulating a SelectMAP interface at a clock frequency of 50 MHz. Only a single hardware task can be configured or removed at a time. The hardware tasks used in the simulations are considered to be implemented on an XCV2000E FPGA and are based on the synthesis results mentioned in [6]. The hardware task size ranges from 1 CLB column (8-bit divider) to 36 CLB columns (RISC-CPU). Each simulation has a length of 4 *sec*, while within this 4 *sec* randomly 200 hardware tasks are requested to be placed on the FPGA. Hardware tasks that cannot be placed due to unavailable FPGA resources will not be placed again later. Defragmentation is initiated, whenever a hardware task cannot be placed due to unavailable contiguous unused CLBs, although the total number of unused CLBs is larger than the size of the requested hardware task. The online placement of a hardware task is done by the Best-Fit algorithm [2]. It is possible to use arbitrary execution times for the hardware tasks. However, for the discussed simulations we decided that the execution times of the hardware tasks linearly depend on the size of the hardware task (e.g. 8-bit divider: 4 ms, RISC-CPU: 115 ms). After execution the hardware tasks are removed from the FPGA as soon as the configuration device is available. In this work we consider defragmentation to be performed as follows:

The relocation is realized as described in Section 2.1. At the beginning the clocks of the hardware tasks that are located within the defragmentation area (M_{defrag} in Alg. 1) are stopped. Subsequently, the state information of the hardware tasks are captured and stored by the configuration device. Then the hardware tasks are relocated to the new positions, which are calculated by the defragmentation algorithm presented in Section 3. During relocation the previously captured states are restored, so that no extra time for the state write-back is necessary. After all hardware tasks are located at their new positions the requested hardware task is placed. Finally, previously used CLB columns, which still contain old configuration data, are erased by a corresponding empty

Table 1. Device utilization and rejected hardware tasks of the simulations.

	Device Utilization (mean)			Rejected Modules (mean)		
CLK_{conf} [MHz]	No Defrag.	Complete Defrag.	Local Defrag.	No Defrag.	Complete Defrag.	Local Defrag.
10	24,88%	**30,78%**	26,89%	**34,16%**	36,01%	34,72%
25	32,63%	**37,14%**	35,95%	18,08%	19,40%	**17,45%**
50	34,25%	36,91%	**37,08%**	14,45%	13,54%	**12,03%**
100	34,91%	**38,27%**	38,25%	13,50%	**8,70%**	**8,70%**
(no)	37,85%	**38,64%**	**38,64%**	9,09%	**7,25%**	**7,25%**

bitstream. Now that the defragmentation is done, the clocks of the relocated hardware tasks are started again.

We consider two different defragmentation algorithms. In the first defragmentation algorithm a *complete defragmentation* is performed by considering the whole FPGA area as the defragmentation area. The second defragmentation algorithm selects the defragmentation area with the objective of minimal column movements to allow a placement of the requested hardware task. Therefore, only a *local defragmentation* is performed.

The simulations have been performed with complete defragmentation, local defragmentation and without defragmentation. For a comparison we considered the metrics *device utilization* and *rejected hardware tasks*. The device utilization $\upsilon = N_{execCLBs}/N_{CLBs}$ is the number of CLBs of the currently executing hardware tasks ($N_{execCLBs}$) compared to the total number of CLBs (N_{CLBs}). In the simulations we used a XCV2000E which has $N_{CLBs} = 80 \cdot 120 = 9600$. The metric rejected hardware tasks $\rho = N_{reject}/N_{hardwaretasks}$ is the number of unplaceable hardware tasks (N_{reject}) divided by the total number of hardware tasks in the simulation ($N_{hardwaretasks}$).

In the simulations we have varied the configuration device clock frequency in order to change the ratio of the configuration times to the execution times of the hardware tasks. The simulation results are shown in Table 1. At a configuration clock speed of 10 MHz defragmentation has a negative effect on the percentage of rejected hardware tasks. In all simulations approximately every third hardware task cannot be placed. However, the simulation with no defragmentation has the least number of rejected hardware tasks.

At a faster configuration clock speed of 25 MHz the local defragmentation has the least number of rejected hardware tasks, while the complete defragmentation results in the largest number of rejected hardware tasks. In this simulation local defragmentation showed an improvement of the number of rejected hardware tasks compared to no defragmentation. At a configuration clock speed of 100 MHz both defragmentation algorithms produced nearly the same simulation results. Although the selected XCV2000E device does not support that configuration clock speed, we intended to analyze the influence of short configuration times compared to relatively long execution times of the hardware tasks. In this simulation there is the largest improvement of the number

of rejected hardware tasks compared to no defragmentation. By assuming that no configuration time is needed and tasks can be configured in 0 *sec* still $9,05\%$ of the tasks cannot be placed and with defragmentation still $7,25\%$ of the tasks are rejected.

In most of the simulations the complete defragmentation leads to the largest device utilization. One reason for this is that hardware tasks are suspended longer due to the higher reconfiguration overhead of complete defragmentation. Therefore, they remain longer on the FPGA and cause a higher device utilization. But this does not result in fewer hardware task rejections.

5 Conclusion

In this paper we have described our approach to run-time relocation. Hardware tasks can be placed along a one-dimensional communication structure by manipulating the partial bitstream during configuration of the hardware task. When relocating a hardware task the internal state information is preserved by a state extraction and state inclusion filter. To save the internal states no extra hardware structure have to be added to a hardware task and there is no need to have detailed knowledge about the internal structure or behavior of the hardware task.

By using our hardware task relocation and context saving methods, run-time defragmentation can be realized. We have described a defragmentation method with the objective to minimize the reconfiguration time overhead. We have implemented the defragmentation method in a simulation framework. Simulation results have shown: If the configuration time of a task equals the execution time of the task defragmentation is not beneficial. If the execution time of a task is greater than the configuration time of the task, local defragmentation becomes useful. In any simulation local defragmentation performed better compared to complete defragmentation.

Acknowledgment

This work was partially supported by the Graduate College 776 "Automatic Configuration in Open Systems", the Collaborative Research Center 614 "Self-Optimizing Concepts and Structures in Mechanical Engineering" of the University of Paderborn, and the Research Fellowship Programm of the German Research Foundation (DFG).

References

1. K. Bazargan, R. Kastner, and M. Sarrafzadeh. Fast template placement for reconfigurable computing systems. *IEEE Design and Test of Computers*, Vol. 17, No. 1:68–83, 2000.

2. E. G. Coffman, M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing: A survey. In D. Hochbaum, editor, *Approximation algorithms*. PWS Publishing Company, 1997.
3. O. Diessel and H. ElGindy. Run-time compaction of FPGA designs. In *Field-Programmable Logic and Applications. 7th Int. Workshop*, volume 1304, London, U.K., 1997. Springer.
4. H. Kalte, M. Koester, B. Kettelhoit, M. Porrmann, and U. Rückert. A comparative study on system approaches for partially reconfigurable architectures. In *Proc. of the Int. Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '04)*. CSREA Press, 2004.
5. H. Kalte, G. Lee, M. Porrmann, and U. Rückert. Replica: A bitstream manipulation filter for module relocation in partial reconfigurable systems. In *Proc. of the 19th International Parallel and Distributed Processing Symposium*, 2005.
6. H. Kalte, M. Porrmann, and U. Rückert. Study on column wise design compaction for reconfigurable systems. In *Proceedings of the IEEE International Conference on Field Programmable Technology (FPT'04)*, 2004.
7. H. Kalte, M. Porrmann, and U. Rückert. System-on-programmable-chip approach enabling online fine-grained 1D-placement. In *11th Reconfigurable Architectures Workshop (RAW 2004)*, Santa F, New Mexico, 2004.
8. H. Simmler, L. Levinson, and R. Manner. Multitasking on FPGA coprocessors. In *Proceedings of the 10th International Workshop on Field-Programmable Logic and Applications*, pages 121–130, London, UK, 2000. Springer.
9. M. Ullmann, M. Hübner, B. Grimm, and J. Becker. An FPGA run-time system for dynamical on-demand reconfiguration. In *Proc. of the 18th International Parallel and Distributed Processing Symposium*. IEEE Computer Society, 2004.
10. H. Walder and M. Platzner. Non-preemptive multitasking on FPGAs: Task placement and footprint transform. In *Proc. of the Int. Conference on Engineering of Reconfigurable Systems and Architectures*, pages 24–30. CSREA Press, 2002.
11. Xilinx Inc. Application notes 151. Virtex series configuration architecture user guide, 2000.