

Relocation and Defragmentation for Heterogeneous Reconfigurable Systems

Markus Koester, Mario Porrmann
*Heinz Nixdorf Institute,
System and Circuit Technology,
University of Paderborn, Germany*
{koester, porrmann}@hni.uni-paderborn.de

Heiko Kalte
*School of Computer Science
and Software Engineering,
University of Western Australia, Australia*
heiko@csse.uwa.edu.au

Abstract

Current FPGAs are heterogeneous partially reconfigurable architectures, consisting of several resource types, e. g., logic cells and embedded memory. By using partial reconfiguration, arbitrary hardware tasks can be placed and removed at run-time, causing the free FPGA resources to become fragmented over time. This fragmentation can prevent a requested task from being placed, if the required FPGA resources are not available in a sufficiently large contiguous region. A solution to this problem is to relocate the currently placed tasks for being able to place the requested task. This paper introduces a run-time defragmentation algorithm, which relocates currently placed tasks on a heterogeneous FPGA area. Additionally, the necessary hardware mechanism for relocating a task at run-time are described. Simulation results for dynamically reconfiguring Xilinx Virtex-II FPGAs are presented, which show the improvement of the placement when using the proposed defragmentation algorithm.

1. Introduction

Partially reconfigurable Field Programmable Gate Arrays (FPGAs) offer a high flexibility in implementing a complete system on a single chip. Arbitrary hardware tasks can be configured on demand and removed after execution at run-time, allowing the sharing of FPGA hardware resources over time. Due to the various sizes of the hardware tasks, the free resources are split into multiple fragments over time. Consider a task is requested to be placed on the FPGA. Although the total number of free resources may be larger than the number of resources required by the requested task, the free resources can be scattered in small fragments, which are not large enough to accommodate the requested task. Since the remaining execution time of the currently configured tasks is assumed to be unknown, defragmentation can be applied by relocating the currently configured tasks with

the objective to maximize the area of contiguously free resources. Defragmentation in this sense is equivalent to a strip-packing problem (SPP). Diessel et al. [1] proposed an online algorithm to this problem preserving the relative order of the tasks as they appear on the FPGA area. In [11] van der Veen et al. described an offline algorithm to maximize the available contiguous free space to optimality. But the algorithm can only be used at recurring idle times where the task configuration of the system is constant for a reasonable long time.

The above mentioned approaches are based on homogeneous FPGA architectures. But current FPGAs are multi-grained reconfigurable architectures, which include, e. g., embedded processors, arithmetic units, and embedded memory. In contrast to homogeneous FPGAs, hardware tasks that use embedded memory cannot be placed anywhere on the reconfigurable array because their feasible positions are limited by the locations of the embedded memory.

The defragmentation algorithm, proposed in this paper, is suitable for heterogeneous FPGA architectures. If a requested task cannot be placed, the algorithm basically relocates hardware tasks, which are placed at a feasible position of the requested task. Relocating a hardware task at run-time requires a suitable mechanism to preserve the internal states of the hardware task during the relocation process. In Section 2 a context saving and restoring mechanism is depicted, which is suitable for Xilinx Virtex-II/-Pro FPGAs. Besides preserving the states, the relocation of a hardware task additionally requires a heterogeneous placement algorithm, which determines a new position of the task. Section 3 specifies a corresponding heterogeneous placement approach, which is suitable for heterogeneous FPGAs. The proposed methods are specified with respect to a column-oriented one-dimensional placement model, where tasks vary in their width only (cf. [2, 3, 6]). In our approach task communication is established by a bus-based homogeneous horizontal communication infrastructure (cf. [6]). In Section 4 the defragmentation algorithm is

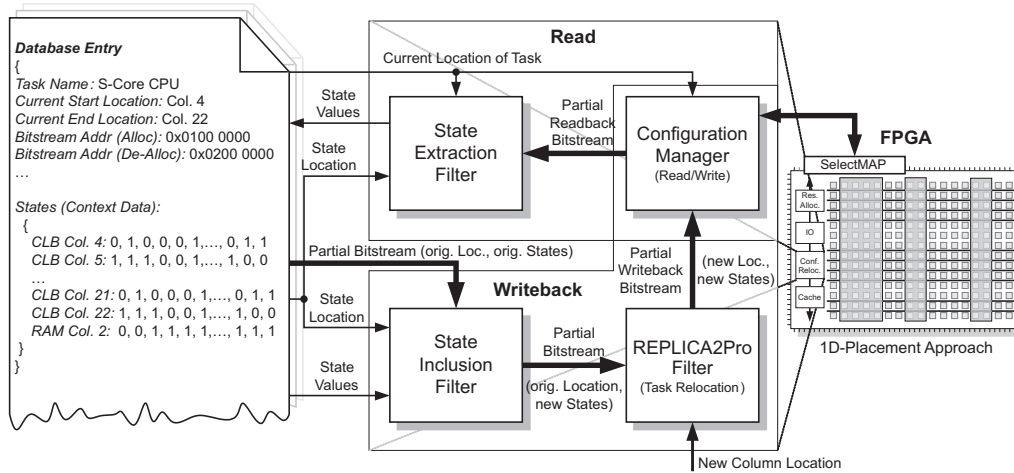


Figure 1. Relocation Approach Overview.

described and simulation results are presented that show the improvement of the placement when using the proposed algorithm.

2. Context Saving and Restoring

In order to relocate a placed hardware task on a Xilinx Virtex-III/Pro FPGA at run-time, there are basically two approaches to read and restore state information that are stored in registers and memory. The *Task Specific Access Structures* approach realizes reading and restoring by a read/write interface to all registers, requiring additional resources and design effort. Consequently, each hardware task has to be redesigned to be used in a reconfigurable system. However, one advantage of this approach is the high efficiency, as only the raw state information are read. Ullmann et al. [10] 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 (cf. [12]). This port offers the possibility to read arbitrary columns of the configuration memory including the current register values and the RAM contents, and to extract the state information from the read configuration data. Subsequently, the extracted state information can be restored in a relocated instance of the task by manipulating the preset bits of the corresponding flip-flops as well as the RAM content in the bitstream to be downloaded. As the Configuration Port Access approach uses the inherent access structures of the configuration circuitry and the configuration port, no hardware structures have to be added to the tasks itself. However, a disadvantage of this approach is the low efficiency, as the portion of state information in the read data can be less than 1%. Configuration Port Access approaches for the Virtex/-E architecture have been presented, e. g., by Simmler et al. [9], and Kalte et al. [7].

2.1. Configuration Port Run-time Relocation

In the following, the Configuration Port Access approach of our reconfigurable system, which is based on Virtex-II/Pro FPGAs, is described. The Virtex-II/Pro FPGAs can only be configured column-wise and the atomic unit of configuration is a *frame* [12]. A frame is the smallest portion of the configuration memory that can be written to or read from. In contrast to the existing Configuration Port Access approach by Simmler et al. [9] our approach does not read the configuration data of the whole FPGA, but only those frames that include the state information of the task to be relocated. Furthermore, the current state information is not extracted after, but while reading back the configuration data.

Figure 1 shows the architecture of our context relocation approach consisting of four main function blocks and a database. The main blocks are the *Configuration Manager*, the *State Extraction Filter*, the *State Inclusion Filter* and the *REPLICA2Pro Filter*. The first step of the context relocation process is to stop the clock of the particular hardware 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/ICAP interface to read only those frames that contain state information of the hardware task. The addresses of the frames are calculated based on 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 all flip-flop states of the occupied CLB columns. For each CLB column of the task the Configuration Manager generates all necessary frame addresses that contain state values. All flip-flop states of a CLB column are located in 2 frames. Consequently, the Configuration Manager reads only the corresponding frames instead of a complete CLB column (22 frames). The output of the Configu-

ration Manager is a stream of single frames that contain the state information of the hardware task. The frames are continuously transferred to the State Extraction Filter, which determines the state value within each frame. For extracting the state value, the filter determines the bit index within the read frames. The task is now suspended, but not de-allocated. That means, a partial "empty" bitstream has to be downloaded to completely erase the circuitry of the task at its former location.

For each task there is only one *pre-implemented partial bitstream*, which is modified at run-time to allow the placement of the task at its feasible positions. The restoring process starts with the State Inclusion Filter, which takes the pre-implemented partial bitstream of the hardware task and includes all database state values by manipulating the preset bit of the registers. Similar to the state extraction process, the frame address and the bit index of all state bits have to be calculated.

Subsequently, the modified partial bitstream is passed to the REPLICA2Pro Filter [4], which is implemented as a hardware filter. The REPLICA2Pro Filter [4] is able to relocate a hardware task to the *New Column Location* by manipulating the partial bitstream of the task. A heterogeneous placement algorithm to determine a suitable New Column Location for the task is depicted in Section 3. The bitstream manipulation is done during the configuration process of the task and therefore does not require any additional time.

Finally, the Configuration Manager writes the resulting partial bitstream of the relocated task with the corresponding state information to the FPGA. After resetting the hardware task, all registers are set to the proper values and the task can continue processing in exactly the state it was interrupted before.

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 in [4] for the task relocation with the REPLICA2Pro filter.

The total time for relocating a task depends on the number of utilized CLB columns N_{Clb} , the number of RAM columns N_{Ram} , the frame size $N_{B/F}$ in byte per frame, and the SelectMAP/ICAP frequency f_{SelMAP} . As mentioned earlier, for each occupied CLB column N_{Cols} , two frames have to be read for capturing the states of the flip-flops. The first frame of every new read access is always a pad frame which does not contain significant

Table 1. Worst-case relocation times.

	LDPC- Decoder	16-bit Divider	FIR- Filter	Rijndael	S-Core CPU
Design size [slices]	191	281	845	2129	6061
Min. Columns [CLB/RAM]	1/0	1/0	3/1	7/0	19/0
Available Flip Flips	640	640	1920	4480	12160
Used Flip Flops	44	211	944	788	2287
Alloc. Bitstream Size	19,3 kB	19,3 kB	75,4 kB	125,0 kB	354,0 kB
Allocation Time	386 μ s	386 μ s	1508 μ s	2500 μ s	7080 μ s
De-Alloc. Bitstream Size	2,9 kB	2,9 kB	5,44	6,3 kB	65,0 kB
De-Allocation Time	58 μ s	58 μ s	109 μ s	126 μ s	1300 μ s
Frames to Read	2	2	70	14	38
Read Data [Byte] [*]	3296	3296	62624	23072	62624
Read Time	66 μ s	66 μ s	1252 μ s	461 μ s	1252 μ s
Complete Relocation	0,5 ms	0,5 ms	2,9 ms	3,1 ms	9,6 ms
Worst Case Calculation	0,8 ms	0,8 ms	6,3 ms	5,5 ms	15,0 ms

^{*} includes one pad frame per read CLB frame

data. Hence, in order to capture all states of a CLB column 4 frames must be read. To capture the state of a BlockRAM column, the memory content (64 frames + 1 pad frame) must be read. The total time for capturing the states of the flip-flops and BlockRAM is

$$T_{cap} = \frac{(4N_{Clb} + 65N_{Ram}) \cdot N_{B/F}}{f_{SelMAP}} \quad (1)$$

For the allocation of the task at most 22 frames per CLB column and 86 frames per BlockRAM column must be written (see [13] for further details). However, this does not consider the bitstream compression (Multiple Frame Write command), which was introduced with Virtex-II-Pro architecture. Consequently, the time for allocating a task can be approximated by

$$T_{alloc} = \frac{(22N_{Clb} + 86N_{Ram}) \cdot N_{B/F}}{f_{SelMAP}} \quad (2)$$

The time for allocating and de-allocating a task is assumed to be the same ($T_{del} = T_{alloc}$). At most the complete task relocation time T_{reloc} is

$$\begin{aligned} T_{reloc} &\approx T_{cap} + T_{alloc} + T_{del} \\ &= (48N_{Clb} + 237N_{Ram}) \frac{N_{B/F}}{f_{SelMAP}} \end{aligned} \quad (3)$$

Table 1 shows worst case times, and realistic relocation times for some example hardware tasks, based on an XC2V4000 FPGA. The frame length of this device is 824 bytes and the SelectMAP frequency is set to 50 MHz. In addition to the size of the task and the flip-flop utilization, the table lists the realistic allocation and de-allocation bitstream sizes (as generated by Bitgen) and configuration times. Additionally, the amount of data to be read and the resulting read time is listed. The *Complete Relocation* describes the realistic relocation times that consider the read, allocation, and de-allocation times based on the generated bitstreams for the tasks. The last row finally depicts the worst case relocation times as calculated by (3). The differences between the two relocation times is exclusively caused by

the compression of the allocation and de-allocation bitstreams. Particularly, the de-allocation bitstreams benefit from the compression; in some cases these bitstreams can be reduced to 5% of the uncompressed bitstream. As shown in Figure 1, the proposed relocation approach needs a new column location for the task as an input parameter. In the following section a heterogeneous task placement algorithm is described, which is used to determine a suitable new column for the task.

3. Utilization Probability Placement

As mentioned before, the present FPGAs are typically not completely homogeneous. While most of the placement algorithms that have been published so far rely on homogeneous architectures, we present a heterogeneous placement algorithms, which is used in the defragmentation algorithm explained in Section 4. The placement algorithm is suitable for reconfigurable architectures with n different resource types. Without loss of generality, we assume two different resource types in the following. With respect to the Virtex-II FPGA series, the different resource types are, e.g., configurable cells (slices) and memory cells (BlockRAM). In [8] the placement algorithm is described for a general two-dimensional placement. In the following the algorithm is adapted to the one-dimensional placement.

In the 1D-system approach, the atomic unit for a partial reconfiguration is a cell column and tasks can only be placed along the horizontal axes of the FPGA. Based on the initial position of the pre-implemented partial bitstream (cf. Section 2.1), a task can just be placed at positions, where the same pattern of the different cell types occurs on the FPGA. This restricts the placement of a hardware task to the set of its *feasible positions* as illustrated in Figure 2. Consider M is the set of hardware tasks, where each hardware task $m \in M$ has a given set of feasible positions $X_{pos}(m) = \{x_1, x_2, \dots\}$, which define the leftmost column of the task in relation to the leftmost column of the FPGA. Then the feasible positions of hardware task m_1 are $X_{pos}(m_1) = \{1, 5\}$ and the feasible positions of hardware task m_2 are $X_{pos}(m_2) = \{1, 4, 5, 8\}$.

The probability that a cell column is utilized by a correspondingly configured task depends on the number of feasible positions that cause a utilization of this cell column. E.g., the rightmost cell column cannot be utilized by the task m_1 and can only be utilized by the task m_2 , when it is placed at the feasible position $x = 8$. In the following, we refer to the number of feasible positions of a hardware task m that use the cell column at position x as the *coverage of feasible positions* $o_{pos}(m, x)$.

Table 2 shows the coverage of feasible positions for the hardware tasks m_1 and m_2 . Consider W is the number of cell columns of the FPGA, $x \in [1, W]$ is the horizontal position of a cell column, and $a_x(m)$ is the horizontal

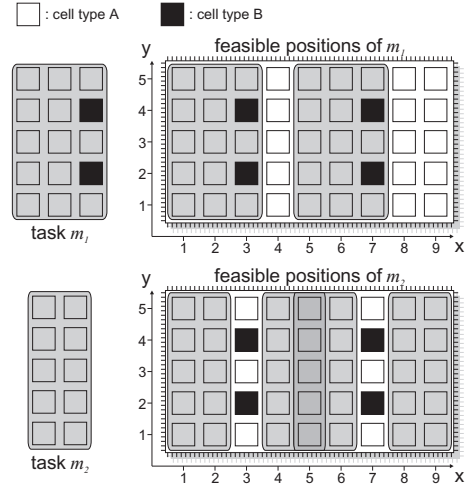


Figure 2. Feasible positions for the hardware tasks m_1 and m_2 .

Table 2. Coverage of feasible positions for the hardware tasks m_1 and m_2 .

	x	1	2	3	4	5	6	7	8	9
$o_{pos}(m_1, x)$		1	1	1	0	1	1	1	0	0
$o_{pos}(m_2, x)$		1	1	0	1	2	1	0	1	1

size of a hardware task $m \in M$. The set $X_{pos}(m, x)$ is the set of the feasible positions of the task m , which utilizes the cell column at position x . The coverage of feasible positions $o_{pos}(m, x)$ is the number of elements, of the set $X_{pos}(m, x)$, i. e.,

$$o_{pos}(m, x) = |X_{pos}(m, x)|, \text{ where} \quad (4)$$

$$X_{pos}(m, x) = \{i : i \in X_{pos}(m) \wedge i \leq x < i + a_x(m)\}$$

Besides the coverage of feasible positions the utilization probability of a cell column also depends on the probability of a task being requested by the system. When the system requests a task to be placed on the FPGA, the probability that the hardware task $m \in M$ is requested is characterized by the *allocation probability* $p_{alloc}(m)$, i. e., $\sum_{m \in M} p_{alloc}(m) = 1$.

If the allocation probabilities of the tasks are unknown, we can assume, e. g., uniformly distributed task requests and thus a constant allocation probability of $p_{alloc}(m) = 1/|M|$. Based on the allocation probability and the coverage of feasible positions of the hardware tasks, the *static utilization probability* $p_{pos}(x)$ of the cell column $x \in [1, W]$ is defined as follows:

$$p_{pos}(x) = \sum_{m \in M} \frac{p_{alloc}(m) \cdot o_{pos}(m, x)}{|X_{pos}(m)|} \quad (5)$$

Suppose all cells of the FPGA are available and no hardware task is placed. Now, if a hardware task is requested to be placed, $p_{pos}(x)$ describes the probability that the

Table 3. Static utilization probability p_{pos} and corresponding position weights w_{pos} for m_2 .

x	1	2	3	4	5	6	7	8	9
$p_{pos}(x)$	0.35	0.35	0.2	0.15	0.5	0.35	0.2	0.15	0.15
$w_{pos}(m_2, 1) = 0.35$									
$w_{pos}(m_2, 4) = 0.369$									
$w_{pos}(m_2, 5) = 0.432$									
$w_{pos}(m_2, 8) = 0.15$									

cell column x will be utilized by the hardware task when using a random position selection. Table 3 shows an example of the static utilization probability $p_{pos}(x)$ for the previously mentioned hardware tasks $M = \{m_1, m_2\}$ using the allocation probabilities $p_{alloc}(m_1) = 0.4$ and $p_{alloc}(m_2) = 0.6$.

According to the fact that the cell column at position $x = 5$ is the only one, which can be utilized by the feasible positions $x_{pos} = 5$ of m_1 and by the feasible positions $x_{pos} = 4$ and $x_{pos} = 5$ of m_2 , it therefore has the largest static utilization probability of $p_{pos}(5) = 0.5$. Consequently, if this cell column is occupied by a hardware task, all of the previously mentioned feasible positions are unavailable for a placement of a requested task. When placing a task by selecting a free feasible position, the static utilization probability of the cell columns of the selected feasible position should be low.

In this respect, each feasible position $x_{pos} \in X_{pos}(m)$ of a hardware task m is given a so called *position weight* $w_{pos}(m, x_{pos})$, which is the quadratic mean of the static utilization probability p_{pos} of the corresponding cell columns:

$$w_{pos}(m, x_{pos}) = \sqrt{\frac{1}{a_x(m)} \sum_{i=x_{pos}}^{x_{pos}+a_x(m)-1} p_{pos}(i)^2} \quad (6)$$

The weights w_{pos} of the feasible positions of task m_2 are shown in Table 3. The lowest weight is $w_{pos}(m_2, 8) = 0.15$, which indicates that the cell columns of the feasible position $x_{pos} = 8$ are least utilized compared to the other feasible positions of m_2 .

A heterogeneous placement algorithm that uses the utilization probability as described above is the *Static Utilization Probability Fit* algorithm (SUP Fit) [8]. The algorithm determines the position to place a requested task by the position weights of its feasible positions. Therefore, the feasible positions of all tasks are sorted before run-time according to their position weights, so that the lowest-indexed position has the least weight. In our example the order of feasible positions for hardware task m_2 is 8, 1, 4, 5. Once, the order of the positions is determined, it is not necessary to keep the static utilization probability or the position weights in memory at run-time. The placement of a task at run-time is done by selecting the first free feasible position using the determined order of position weights.

4. Partial Displacement Defragmentation

In many cases a complete defragmentation by maximizing the area of contiguously free resources is not necessary for the placement of a requested task, rather the aim is to relocate the currently placed tasks to obtain a reasonable amount of contiguous free resources for placing the requested task. In current FPGA architectures hardware tasks can only be configured sequentially by the configuration interface and the allocation time of a task is in the order of milliseconds (cf. Table 1). Therefore, a crucial criteria of run-time defragmentation is to minimize the total reconfiguration time overhead. Thus, the requested task is configured as soon as possible and the delay to the beginning of its execution is minimized. The reconfiguration time of a task basically depends on the number of its CLB columns. Therefore, minimizing the total reconfiguration overhead of the defragmentation is achieved by minimizing the total number of CLB columns to be relocated. In [7] we introduced a run-time defragmentation algorithm aiming to minimize the reconfiguration time overhead in a homogeneous 1D-system approach. However, when considering a heterogeneous FPGA architecture, tasks cannot be placed at any position but basically are constrained to their feasible positions.

In the following we present a heuristic approach to run-time defragmentation on heterogeneous FPGA architectures aiming to minimize the reconfiguration overhead. The algorithm uses an virtual mapping of the current task allocation of the FPGA. Basically, the following steps are performed on the virtual mapping:

1. A feasible position of the requested task is selected and the tasks that intersect with the feasible position are virtually removed from the mapping of the FPGA.
2. The requested task is virtually placed at the selected feasible position.
3. The algorithm tries to place the previously removed tasks at new positions. If the placement of the previously removed tasks is successful, then a feasible solution is found and stored.
4. The original task configuration is restored again and the algorithm is repeated with selecting the next feasible position of the requested task unless all feasible positions have been considered.

If several solutions are found, the one with the least reconfiguration overhead is selected. Finally, the solution is realized on the real FPGA. Due to the principle of displacing previously placed tasks to allow the placement of the requested task, we refer to the algorithm as *partial displacement defragmentation*.

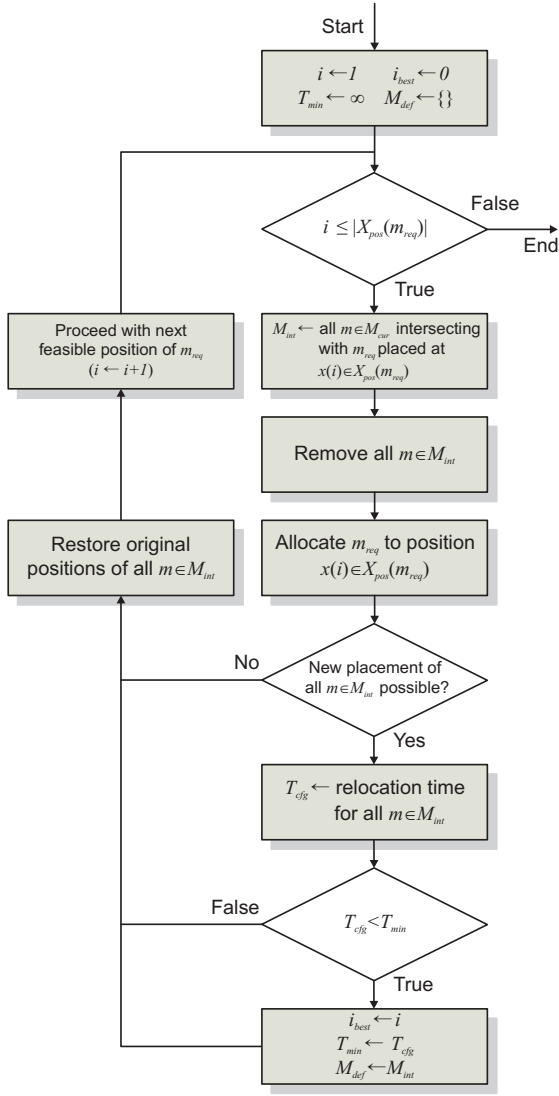


Figure 3. Partial displacement defragmentation algorithm.

The algorithm does not find a solution in any case, but it succeeds in many cases as demonstrated by simulation results in Section 5. A flow chart summarizing the algorithm is shown in Figure 3. M_{int} is the set of tasks that intersect with the requested task m_{req} placed at the currently selected feasible position $x(i) \in X_{pos}(m_{req})$. i is the index variable for the currently selected feasible position. The best found solution is represented by i_{best} which is the index of the feasible position, and M_{def} which is the set of tasks to be relocated, where T_{min} is the time overhead for relocating the tasks of M_{def} .

The execution time of the algorithm basically depends on the number of feasible positions of the requested hardware task. In a one-dimensional placement the number of feasible positions for a task is at the most equal to the number of columns of the FPGA. Hence, the time for the computation of the solution is low compared to the relocation times of the tasks (cf. Table 1).

5. Experimental Results

The proposed defragmentation algorithm has been tested with the SUP Fit placement algorithm and evaluated by simulation. The results are compared to the SUP Fit placement algorithm, and a heterogeneous Best Fit algorithm – both without defragmentation. The simulations are based on the designs described in [5]. The smallest design is a 32-bit Divider with 844 slices and the largest design is a 32-bit RISC Processor S-Core with 5730 slices. The designs are synthesized for the 1D-system approach using a Xilinx XC2V4000 FPGA. The resulting hardware tasks therefore have a constant height equal to the height of the FPGA.

The simulations are based on a schedule of 100 randomly requested tasks. In the simulation the configuration time of a task is derived based on the assumption of a SelectMap configuration at 50MHz. If a defragmentation is performed, the simulations consider the necessary relocations times of the tasks as described in (3).

For the two placement algorithms without defragmentation, the scheduling is done as follows: If a task cannot be placed, it will be set on a placement queue and its placement will be carried out when sufficient resources are available. The scheduling for the run-time defragmentation approach is done as follows: If a task cannot be placed, but enough free resources are available, a run-time defragmentation is performed. If the defragmentation fails, or not enough free resources are available, the task is set on a placement queue and its placement will be carried out after reasonable enough resources become available. As long as a task is in the placement queue the following requested tasks are set on the queue as well to maintain the order of the schedule. When a task finishes its execution it is removed as soon as the configuration interface is available.

The simulations are compared in terms of the total time for finishing the execution of all tasks (T_{all}). The more efficient the tasks are placed, the less number of tasks placements are delayed due to unavailable feasible position, which finally results in a shorter total execution time. Additionally, we take into account the resource utilization, which is the number of actively used resources compared to the total number of available resources. A large resource utilization indicates a more efficient usage of the given resources. Table 4 shows the simulation results for three different task schedules. Although the run-time defragmentation requires additional reconfiguration time overhead caused by relocating already placed tasks, the overall execution time is reduced in all three simulations. Most of all, in task schedule A the total execution time is reduced to 87.1% in comparison to the SUP Fit placement approach without defragmentation. Compared to the Best Fit approach, the maximum number of tasks in the placement queue is significantly reduced. When the reconfiguration overhead is small compared to

Table 4. Comparison of the defragmentation approach (SUPD Fit) to heterogeneous placement approaches without defragmentation.

Task Sched.	$\frac{T_{cfg}}{T_{exec}}$	Placement Algorithm	T_{all}	Defrag-mentations	Resource Utilization	max. Tasks in Queue
A	~3%	SUPD Fit	2.120s	13	54.44%	12
		SUP Fit	2.433s	0	46.94%	21
		Best Fit	2.569s	0	44.46%	27
B	~10%	SUPD Fit	2.340s	16	53.19%	15
		SUP Fit	2.507s	0	48.37%	20
		Best Fit	2.780s	0	42.61%	30
C	~75%	SUPD Fit	1.522s	13	9.03%	16
		SUP Fit	1.594s	0	6.93%	17
		Best Fit	1.595s	0	6.93%	17

the execution time of the tasks, defragmentation can improve the placement of the tasks.

With an increasing ratio of the reconfiguration time T_{cfg} to the execution time T_{exec} of the tasks, the improvement of defragmentation reduces. In task schedule *B* the total execution time compared to the SUP Fit approach without defragmentation is only 93.3%. In schedule *C* the improvement of the total execution time T_{all} using run-time defragmentation is negligible. Moreover, the average resource utilization of all placement approaches is very low compared to the task schedules *A* and *B*. The delay caused by the sequential partial reconfiguration of the tasks is that long, that only a small number of resources actively perform an operation, while the remaining resources are in an idle state, reserved for a future placement, or occupied by an executed task. As a result run-time defragmentation is only useful, if the ratio of T_{cfg} to T_{exec} is sufficiently low.

6. Conclusion

Similar to memory fragmentation, the fragmentation of the reconfigurable fabric has a serious impact on the placement performance. In order to overcome this problem, hardware tasks must be relocated on the FPGA at run-time to obtain a sufficiently large amount of contiguous free resources. However, defragmentation on an FPGA is quite different to memory defragmentation, mainly because of the heterogeneous arrangement of the resources.

In this paper we have presented a context saving and restoring mechanisms that is necessary to relocate a hardware task including all its state information at run-time. The approach utilizes the standard configuration port which has the advantage that all hardware tasks remain unchanged and no extra hardware resources are wasted for the tasks. Additionally, we introduced a new defragmentation algorithm that was developed to cope with heterogeneous FPGA architecture, as known placement and defragmentation algorithms are not applica-

ble in these environments. Simulations with realistic parameters have shown a great benefit of the proposed defragmentation algorithms for heterogeneous reconfigurable systems over no defragmentation if the configuration times of the tasks are small compared to their execution times .

References

- [1] O. Diessel and H. ElGindy. Run-time compaction of FPGA designs. In *Field-Programmable Logic and Applications. 7th Int. Workshop*, volume 1304, pages 131–140, London, U.K., 1997. Springer-Verlag.
- [2] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The Chimaera reconfigurable functional unit. In *Proceedings of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '97)*, pages 87–97. IEEE Computer Society Press, 1997.
- [3] J. R. Hauser and J. Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In *Proc. of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE Computer Society Press, 1997.
- [4] H. Kalte and M. Pormann. REPLICA2Pro: Task relocation by bitstream manipulation in Virtex-II/Pro FPGAs. In *Proc. of the ACM International Conference on Computing Frontiers*, 2006.
- [5] H. Kalte, M. Pormann, and U. Rückert. Study on column wise design compaction for reconfigurable systems. In *Proc. of the IEEE Int. Conference on Field Programmable Technology (FPT'04)*, 2004.
- [6] H. Kalte, M. Pormann, and U. Rückert. System-on-programmable-chip approach enabling online fine-grained 1D-placement. In *11th Reconfigurable Architectures Workshop*, Santa Fé, New Mexico, 2004.
- [7] M. Koester, H. Kalte, and M. Pormann. Run-time defragmentation for partially reconfigurable systems. In *Proc. of the Int. Conference on Very Large Scale Integration of System-on-Chip (IFIP VLSI-SoC)*, 2005.
- [8] M. Koester, H. Kalte, and M. Pormann. Task placement for heterogeneous reconfigurable architectures. In *Proceedings of the IEEE 2005 Conference on Field-Programmable Technology (FPT'05)*, 2005.
- [9] H. Simmler, L. Levinson, and R. Manner. Multitasking on FPGA coprocessors. In *Proc. of the 10th Int. Workshop on Field-Programmable Logic and Applications*, pages 121–130, London, UK, 2000. Springer.
- [10] 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*, 2004.
- [11] J. van der Veen, S. Fekete, M. Majer, A. Ahmadinia, C. Bobda, F. Hannig, and J. Teich. Defragmenting the module layout of a partially reconfigurable device. In *Proc. 2005 Int. Conference on Engineering of Reconfigurable Systems and Algorithms*. CSREA Press, 2005.
- [12] Xilinx Inc. Application notes 151. Virtex series configuration architecture user guide, 2000.
- [13] Xilinx Inc. Virtex-II platform FPGA user guide, 2005.