Maintaining Virtual Areas on FPGAs using Strip Packing with Delays

Josef Angermeier*, Sándor P. Fekete*, Tom Kamphans*, Nils Schweer, Jürgen Teich*

1 Department of Computer Science 12, University of Erlangen-Nuremberg
Erlangen, Germany
{angermeier, teich}@cs.fau.de
2 Department of Computer Science, Braunschweig University of Technology
Braunschweig, Germany
{s.fekete, n.schweer}@tu-bs.de, tom@kamphans.de

Abstract—The computing resources available on dynamically partially reconfigurable devices increase every year enormously. In the near future, we expect that many applications run on a single reconfigurable device. In this paper, we present a concept for multitasking on dynamically partially reconfigurable systems called virtual area management. We explain its advantages, show its challenges, and discuss possible solutions. Furthermore, we investigate one problem in more detail: Packing modules with time-varying resource requests. This problem from the reconfigurable computing field results in a completely new optimization problem not tackled before. ILP-based and heuristic approaches are compared in an experimental study and the drawbacks and benefits discussed.

I. INTRODUCTION

Reconfigurable devices offer more and more space and functionality over time and will probably continue to do so in the future. Yet, huge hardware applications can already be instantiated on the reconfigurable chips, or even arrays of processors. Furthermore, nowadays reconfigurable chips are mostly used to instantiate a single application with multiple modules, which might not all be necessary at each instant in time. But as reconfigurable devices evolve further, the offered resources will be large enough to also run completely different applications simultaneously. This development also took place in the software world: In the beginning of the nineties, most personal computers used operating systems such as MS-DOS, which allowed just one single software application to be executed, alone with some drivers. Since then, the increased computing resources have allowed us to run multiple applications simultaneously.

Similar to the software world, multitasking will lead reconfigurable devices to higher efficiency, but will also raise several challenges that must be solved. For example, in order to provide reliability and security, we must make sure that no application can violate the execution of another one. In software this is solved by running each application in its own virtual address space. Each application does not know the physical position of its modules, but just the relative positions. Thus, it cannot reconfigure any region that belongs to a different application and is concerned only with its own resources. Intermodule communication takes place only in the region of one application. Thus, the modules that must communicate with each other are automatically grouped by position to each other.

Communication between the partial modules and with the input and output periphery is an important point. We assume that the application modules are provided with some means to communicate with each other and to the FPGA’s I/O-ports independently from their position (e.g., as described in [1]). There are several possible ways to achieve this goal. We solved the communication problems by using our self-developed platform called ESM (see [2],[3]). It offers—amongst others—a so called crossbar device, which dynamically routes the input and output signals to the position of the corresponding module. Moreover, the modules can communicate to each other using the crossbar. Thus, we can assume that the modules can be placed independent from the positions of the I/O periphery and other the positions of other modules.
A. Related Work

1) Reconfigurable Computing: Brebner [4, 5] addressed the problems involved in presenting to a software-oriented user a larger virtual hardware resource that is implemented using smaller physical FPGA hardware. Their approach is based on using swappable units, and a prototype operating system is described that demonstrates operational steps. In contrast to that work, this paper does not address the problem of overcoming the physical constraints given by a small FPGA, but tackles the problem, how to run multiple applications on a single reconfigurable resource.

Bazargan et al. [6] present fast online placement methods for dynamically reconfigurable systems, as well as offline 3D placement algorithms. Hereby, partial modules are to be placed completely independent of each other, and the inter-module communication problem is not addressed. Steiger et al. [7] and Diessel et al. [8] further improve scheduling methods for partially reconfigurable systems. Our approach is based on the differentiation between applications and modules. Modules belonging to the same application are placed nearby, such that inter-module communication can be as efficient as possible. Different scheduling subproblems have been addressed meanwhile; for example, scheduling with respect to the reconfiguration overhead [9]. Banerjee et al. [10] take into account hardware-software partitioning decisions for a fast execution of an application. But in contrast to our paper, all these approaches still focus on executing a single application.

Some operating systems for reconfigurable embedded platforms were developed [11, 12, 13]. Such an operating system provides a minimal programming model and a runtime system. The runtime system performs online task and resource management. Scheduling problems are formulated for the 1D and 2D resource models and developed heuristics are compared to each other. Resources of the operating system and the user applications are clearly differentiated, but that is not the case for the resource access of multiple applications running on the FPGA. Our paper suggests a compromise for the inter-module communication problem: Modules belonging to the same application should placed nearby to each other, as that they can exchange data efficiently. Modules belonging to different applications are not necessarily placed nearby. Additionally, in contrast to our application model, no application can shrink and grow during runtime. The focus in former works was put more on hardware and software abstraction, here it is on securely running multiple, dynamic hardware applications.

Many recent works focused on achieving optimal performance to put multiple tasks onto one FPGA within this context. Cordone et al. [14] and Redaelli et al. [15] specified a new model for partitioning and scheduling on partially dynamically reconfigurable hardware. The different applications are represented by a task graph and the aim is to obtain a total execution time near optimallity by taking reconfiguration and communication times into account. However, this approach does not allow dynamic behavior, such that according to the current state of the resources, the tasks can select on their own which module implementation to reconfigure next and at which nearby position to place it.

The approach by Cardoso [16] also considers the topic of resource virtualization on FPGA devices, achievable due to dynamic reconfiguration capabilities. Hereby, a new temporal partitioning algorithm is proposed. The model by Banerjee et al. [17] furthermore also supports HW/SW partitioning of the tasks. However, both approaches are also based on the assumption that the running times of each task can be estimated roughly, and that the worst-case execution time does not differ too much from the average case. In contrast to that, our approach may also be engaged for the online case, where this must not be the case.

2) Packing: It turns out that placing hardware modules with growing and shrinking area resources amounts to strip packing. The classical strip packing problem was first considered by Baker et al. [18]. In this problem a set of rectangles must be packed into a strip of semi-infinite height and width 1 such that the total height of the packing is minimized. They showed that in the online case the bottom left heuristic does not guarantee a constant competitive ratio for packing a sequence of rectangles. For the offline case they proved an upper bound of 3 for a sequence of rectangles and of 2 on the competitive ratio for a sequence of squares; both analyses require the elements to be sorted. Later Kenyon and Remila designed a fully polynomial time approximation scheme for the offline setting [19]. For the online case Baker and Schwarz [20] introduced the so-called shelf algorithms with a competitive ratio that can be made arbitrarily close to 1.7. Csirik and Woeginger [21] showed a lower bound of 1.69103 for any shelf algorithm and introduced an algorithm whose asymptotic worst-case ratio comes arbitrarily close to this value.

In the classical game of Tetris the aim is to find online placements for a sequence of objects—not all having rectangular shape—such that space is utilized as well as possible. In this process, no item can ever move upward, no collisions between objects must occur, an item will come to a stop if and only if it is supported from below, and each placement must be placed before the next item arrives. Obviously, there is a slight difference in the objective function, as Tetris aims at filling rows. In actual optimization scenarios, this is less interesting, as it is not critical whether a row is used to precisely 100%. Even when disregarding the difficulty of ever-increasing speed, Tetris is notoriously difficult: As shown by Breukelaar et al. [22], Tetris is PSPACE-hard, even for the original, limited set of different objects. Azar and Epstein [23] considered Tetris-like online packing of rectangles into a strip where each item must be moved on a collision-free path to its final position which does not have to supported from below. Just like in Tetris, they considered the situation with or without rotation of objects. For the case without rotation, they showed that no constant competitive ratio is possible, unless there is a fixed-size lower bound of $\varepsilon$ on the side length of the objects, in which case there is an upper bound of $O(\log \frac{1}{\varepsilon})$. For the case in which rotation is possible, they showed a 4-competitive strategy, based on shelf-packing methods, with all rectangles being rotated to be placed on their narrow sides. Coffmann, Downey, and Winkler [24] considered probabilistic aspects of online rectangle packing with Tetris constraint.
without allowing rotations. If rectangle side lengths are chosen uniformly at random from the interval \([0, 1]\), they showed that there is a lower bound of \((0.3138273 \ldots)n\) on the expected height of the strip. Using another kind of level-type strategy, which arises from the bin-packing–inspired Next Fit Level, they established an upper bound of \((0.3697642 \ldots)n\) on the expected height. Fekete et al. \[25\] considered an Tetris-like online packing with gravity; that is, every item must be supported from below in its final position. For squares they gave an algorithm with competitive ratio 2.6154.

Note that none of the previous works allows stretching the objects in any direction.

II. VIRTUAL AREA MANAGEMENT

A. Main Idea

Our concept is aimed at partially dynamically reconfigurable architectures, where modules loaded onto the reconfigurable device may be exchanged at runtime. A typical structure of such a device is given in Fig. 1. Yet devices are available which allow the reconfiguration of columns only, while newer platforms also allow reconfiguration of certain contiguous cells. The former is called 1D reconfiguration, the latter 2D reconfiguration. Our concepts are applicable to both architectures. Furthermore, the platform should consist of one control CPU, which may be placed externally or be included into the reconfigurable device.

To run different applications on the reconfigurable device, we propose to partition the available reconfigurable area into so called virtual areas (VAs). For performance reasons, the different VAs should consist of contiguous reconfigurable area units. As the required amount of resources of an application changes over time, the size of the virtual area may change dynamically over the running time of its application. The mapping of virtual area to physical reconfigurable area is done by the control processor. Each hardware application being executed on the reconfigurable device has its own software control thread running on this CPU, see Fig. 2. These threads request the initially required area and transmit changes in the requirements. An operating system service on the software side takes the requests and is in charge of the management of the virtual areas. This secure operating system unit maps the virtual area units to the corresponding physical dynamically reconfigurable area units. The virtual area management unit can be compared to the memory management unit (MMU) in the software world: both handle the translation of virtual to physical memory positions. Furthermore, the corresponding application software threads do not know the actual physical positions of the reconfigured modules, only the relative positions of each reconfigurable module to each other. Each application simply requests to load a specified module to a virtual position in the assigned virtual area. See the example in Fig. 2. The second application with its virtual area VA2 issues a request to load a specified bit-file called “X” to the virtual address (here called: VSlot) ‘2’. It does not know that this virtual address corresponds physically to the last reconfigurable unit on the reconfigurable device. It knows only that the module loaded there is on the right side of a module loaded to the virtual address (VSlot) ‘1’. As each application is allowed to specify only a virtual address corresponding to its virtual area in which to place the module, it cannot place a module into an area belonging to a different application.

The concept is transparent to the applied intermodule communication in each virtual area: Each application can choose its preferred communication method (e.g., bus system, neighbor to neighbor communication over bus macros). When only contiguous virtual areas are used, the communicating modules are grouped together automatically and communication overhead is minimized. Furthermore, communication between different applications or heterogeneities of the reconfigurable
area is supported easily by extending the operating system.
Yet there are many other approaches to schedule tasks of
different applications, but in contrast to them our approach
of virtual areas combines a localization strategy, putting tasks
which belong to one application together, and highly dynamic
applications, which can individually take different module
selection and placement decision based on the current context.

B. Advantages

The idea of virtual area management offers the following
advantages when running multiple applications on a single
dynamically partially reconfigurable device.

- **Resource accounting**: The concept can be easily used to
prevent applications from using too many resources at
runtime. One might want to restrict the resources granted
to an application for each execution or just in a certain
context in order. The goal is, for example, to reduce
the power consumption or to accelerate more important
applications in a system where certain applications have
lower priorities than others. The concept of virtual area
management provides a limitation mechanism by granting
just a certain amount of, for example, reconfigurable area
to an application.

- **Resource protection**: Each application controls only the
reconfiguration within its virtual area, as it can only
specify those virtual positions that are valid in its own VA. The virtual area management unit checks for each
request if the virtual address is valid and translates it into
a physical position on the reconfigurable device. This
way, no application can load any module to the area of an-
other application and the applications are protected. Thus,
an error in one application cannot harm the execution
of all other applications and lead to a complete system
breakdown.

- **Support for differing scheduling and placement proce-
dures**: Different applications require different scheduling
and placement methods to increase performance. Instead
of designing complex scheduling algorithms that try to
meet all the requirements (e.g., periodic or aperiodic,
with or without deadlines) by introducing various priority
classes, each application gets its own virtual area and
specifies the best scheduling strategy. This may include
a simpler and faster implementation of new hardware
applications within an existing system.

- **Area position transparency and programming dynamic
applications**: The absolute positions’ independence of the
executed modules, in the following called *area position
transparency*, offers a new programming model. It allows
to write dynamic applications, which subject to the cur-
rent resource context decide on their own how to proceed.
Depending on the assigned area, an application can either
use an implementation that offers more performance but
needs more area, or another one that takes longer but uses
less area. Another new possibility is that the application
can decide how to increase and shrink depending on
the current area usage context. An application can ask
the virtual area management unit, if it can grow to the
left or right, or at the bottom, and, depending where
some unused area is available, it can decide to put a
partial module there and instantiate some appropriate
communication module to transfer data there and back.
Thus, at each run, an application can operate differently in
its amount of resource usage. Before, trade-off decisions
where also possible for a single application, but the new
idea is to let each application decide in its own control
program its next steps depending on the behavior of the
other applications.

C. Challenges

First, there are some technical problems to be solved. For
area position transparency, the placement of a module should
not be limited to one single position. Furthermore, there might
be heterogeneities on the reconfigurable device which require
different implementation for different positions. A possible
solution to this problem is to generate implementations for
the module for each possible position where the corresponding
virtual area can be placed. The corresponding module imple-
mentation bit-files can be compressed to save some space.
Another option is to apply a single generated module bit-file
and relocate this file; that is, adapt it to the corresponding
position. We solved this technical issue in the following way:
Our experimental board is equipped with a reconfiguration
manager device that manipulates the corresponding bit-files
before the reconfiguration of the corresponding device.

A further technical challenge lies in the communication
of the partial modules to the external periphery (e.g., video,
audio) over the input/output pins at the border of the FPGA.
Our experimental board has a crossbar device that routes I/O
signals dynamically from the periphery to the current position
of the partial module.

A larger challenge is to fulfill the changing resource re-
quirements of the different applications. Every application
has a different resource usage profile which depends on the
inputs specified. An application called with a larger problem
instance to solve needs also more resources. Additionally,
the resource usage profile also depends on the execution
context. The application may behave differently depending
on how many area resources are available at each position.
Furthermore, the resource requirement depending on the inputs
of an application may or may not be known in advance (or at
least can be estimated, e.g., in a numerical application based
on the required precision of the solutions). The former is called
the offline case, the latter the online case.

In the online case, a deadlock is possible when two applica-
tions currently running on the reconfigurable device can only
proceed in their task when an resource increase is granted,
see Fig. 5. The blocks represent the occupied area of one
application at different points in time. The leftmost application
needs two more area units, as does the application second
from the left. Furthermore, no application can give up its
accumulated resources, as saving and restoring states is widely
considered too expensive on FPGAs. Such a deadlock must be
prevented. A commonly used solution is to allow hardware
task preemption: If not enough resources are available to
meet increased resource demands, the state of one application is stored in external memory. At a later point in time, the application is loaded again on the reconfigurable device and the state be restored again to continue its processing. Another approach is to wait until a deadlock scenario actually happens, but to check beforehand that it cannot get this far. Assuming that the maximal size of a request is known, an area shared between two applications is granted exclusively to just one application, but not one part of it to the one application, and another part to the other application.

In the following, we consider the offline problem: The resource usage profiles for specific inputs of a series of applications is known a priori, or can be estimated roughly. An example of application resource profiles is given in Fig. 4. Note that in the offline case deadlocks cannot occur: We search for feasible solutions (i.e., schedules where no resource requests overlap) only.

Hardware task preemption can be used to increase the area usage. However, saving and restoring the states of the hardware applications can be very costly in time and memory. An approach that balances reconfiguration costs and efficient resource usage is to allow that requests may be delayed by the scheduler. The application keeps its currently occupied area, but remains idle until the request is fulfilled. Compare the two schedules for our example shown in Fig. 5: The schedule shown on the left hand side is a solution for scheduling the application modules without delaying requests. On the right hand side, we allow that requests are postponed. Using this option for the forth request of the forth module, we achieve a better makespan.

III. PACKING APPLICATION MODULES

We consider the problem FPGATris: Scheduling modules whose resource requests (i.e., space on an FPGA) varies over time. This may be, for example, a router module that needs more resources if the traffic increases. We assume that a module occupies a certain number of slots on the FPGA, but requests only complete slots. Thus, we model the FPGA as a one-dimensional array. Furthermore, we assume that time is discrete; that is, requests are multiples of a fixed-size time slot. Now, scheduling a sequence of modules with time-varying resources corresponds to strip packing: The width of the strip is the number of slots on the FPGA; the height corresponds to the time axis. Thus, we use height and time synonymously. Moreover, we assume that every module occupies a base slot and extends to the left or to the right of the base slot.

We are allowed to delay a request; that is, we may stretch the modules along the time axis; see Fig. 6. Our goal is to minimize the makespan (i.e., the time needed to fulfill all requests). For the strip-packing problem, this goal corresponds to minimizing the height of the occupied part of the strip.

For convenience, we consider only the case of growing either to the left or to the right of the base slot. The generalization to both sides is straightforward.
A. An ILP

We are given a strip of width \(N\) (e.g., an FPGA with \(N\) slots and a time axis) and want to place \(M\) modules. Each module, \(m_i\), is given as a sequence of requests. Let \(\ell_i\) denote the length of the request sequence for module \(m_i\) and \((i, j)\) the \(j\)th request of \(m_i\). The size of \((i, j)\) is given by \(r_{ij} \in \mathbb{Z} \setminus \{0\}, 1 \leq j \leq \ell_i\). For \(r_{ij} > 0\) the module expands to the right of the base slot, for \(r_{ij} < 0\) the module expands to the left of the base slot.

For the ILP, we introduce four kinds of variables:

- the slot assignment variables, \(x_{si}\)
- the time assignment variables, \(y_{tij}\)
- the occupancy variables, \(z_{sti}\)
- the usage variables, \(u_t\)

The first two types specify when and where a request is scheduled. More precisely, setting \(x_{si}\) to 1 indicates that module \(m_i\) is scheduled in slot \(s\). Setting \(y_{tij}\) to 1 indicates that request \((i, j)\) of module \(m_i\) is scheduled at time \(t\). That is, module \(m_i\) occupies the following slots at time \(t\):

\[
s, \ldots, s + r_{ij} - 1 \text{ for } r_{ij} > 0,
\]

\[
s + r_{ij} + 1, \ldots, s \text{ for } r_{ij} < 0,
\]

where \(s\) is the base slot of module \(m_i\). For every \(i\) there is exactly one \(x_{si}\) with \(x_{si} = 1\) and for every \((i, j)\) there is exactly one \(y_{tij}\) with \(y_{tij} = 1\).

Usually (i.e., if \(|r_{ij}| > 1\)) a request occupies more than one slot when executed. Moreover, if the request \((i, j+1)\) is delayed, \((i, j)\) remains on the FPGA for more than one time unit (i.e., it occupies more than the one time row). To keep track of the occupied slots, we set \(z_{sti}\) to 1, if slot \(s\) is occupied by request \((i, j)\) at time \(t\). The usage variables simply specify which time steps are used.

Clearly, the FPGA’s size, \(N\), and the number of modules, \(M\), strongly determine the size of an ILP and, in turn, the time needed to solve it. In addition, we assume that an upper bound, \(T\), on the number of time steps is given. The closer this bound is to the optimum, the smaller the resulting ILP. This upper bound can be obtained, for example, using the tabu search in Sect. III-B4.

1) Constraints:

a) Assignment Constraints: Each request must be scheduled exactly once. That is, for every \((i, j)\) we have to set exactly one \(x_{si}\) to 1 (to assign a slot for \((i, j)\)) and exactly one \(y_{tij}\) (to assign a start time for \((i, j)\)). The following constraints express these conditions:

\[
\sum_{s=1}^{N} x_{si} = 1 \quad \forall i = 1, \ldots, M; \tag{1}
\]

\[
\sum_{t=1}^{T} y_{tij} = 1 \quad \forall i = 1, \ldots, M, \ j = 1, \ldots, \ell_i. \tag{2}
\]

b) Boundary Constraints: Next, we ensure that a request does not exceed the FPGA’s boundary by forcing all slot assignment variables that would cause an infeasible placement to be zero.

\[
\forall i, j, s = s_{low}, \ldots, s_{up}: x_{si} = 0 \tag{3}
\]

where

\[
s_{low} := \begin{cases} N - r_{ij} + 2, & r_{ij} > 0 \\ 1, & r_{ij} < 0 \end{cases} \text{ and } s_{up} := \begin{cases} N, & r_{ij} > 0 \\ -r_{ij} - 1, & r_{ij} < 0 \end{cases}.
\]

c) Order Constraints: Now, we ensure that the processing order is maintained; that is, request \((i, j)\) of \(m_i\) is not scheduled before request \((i, j-1)\) is finished. For every \((i, j)\) there is exactly one \(t\) such that \(y_{tij} = 1\). Thus, summing up \(t \cdot y_{tij}\) over \(t\) for fixed \(i\) and \(j\) yields the time step where request \((i, j)\) is scheduled. This yields:

\[
\sum_{t=1}^{T} t y_{tij} - \sum_{t=1}^{T} t y_{tij-1} > 0 \quad \forall i, j > 0. \tag{4}
\]

d) Occupancy Constraints: If \(x_{si} = 1\) and \(y_{tij} = 1\), the request \((i, j)\) occupies \(r_{ij}\) slots adjacent to \(s\) at time \(t\); see Fig. 7. The first step to prevent other modules from overlapping with \(m_i\) is to set the appropriate occupancy variables as follows.

\[
\forall i = 1, \ldots, M, \ j = 1, \ldots, \ell_i, \ s = 1, \ldots, N, \ t = 1, \ldots, T, \ s' = s_{low}, \ldots, s_{up}:
\]

\[
x_{si} + y_{tij} - z_{sti} \leq 1, \tag{5}
\]

with

\[
s_{low} := \begin{cases} s, & r_{ij} > 0 \\ \max\{1, s + r_{ij} + 1\}, & r_{ij} < 0 \end{cases} \quad \text{and} \quad s_{up} := \begin{cases} \min\{N, s + r_{ij} - 1\}, & r_{ij} > 0 \\ s, & r_{ij} < 0 \end{cases}.
\]

![Fig. 7. Occupancy Constraints: If \(x_{si} = 1\) and \(y_{tij} = 1\), the request \((i, j)\) occupies \(r_{ij}\) slots left or right to \(s\)---depending on \(\text{sgn}(r_{ij})\).]
e) **Exclusive Constraints:** By setting the appropriate occupancy variables with the occupancy constraints, we can ensure that requests do not overlap. We allow at most one occupancy variable for a fixed slot and a fixed time to be 1.

\[ \forall t = 1, \ldots, T, s = 1, \ldots, N : \sum_{i=1}^{M} \sum_{j=1}^{\ell_i} z_{stij} \leq 1 \]  

f) **Delay Constraints:** If a request \((i, j, t+1)\) is delayed, the preceding request \((i, j)\) remains on the FPGA until \((i, j+1)\) is scheduled. Thus, if \(z_{stij} = 1\) either \(z_{s(t+1)ij}\) must be set to 1—because the module still occupies space on the FPGA—or \((i, j+1)\) is scheduled at time \(t+1\); that is, \(y_{(t+1)i(j+1)} = 1\) holds. The following constraints keep track of delayed requests.

\[ \forall i = 1, \ldots, M, j = 1, \ldots, \ell_i - 1, s = 1, \ldots, N, t = 1, \ldots, T - 1 : \]

\[ z_{stij} - z_{s(t+1)ij} - y_{(t+1)i(j+1)} \leq 0 \]  

g) **Usage Constraints:** Finally, we introduce some constraints that define our usage variables. Let \(u_t\) be 1, if at least one \(y_{tij}\) is 1 or if \(u_{t+1}\) is 1.

\[ \forall t = 1, \ldots, T, i = 1, \ldots, M, j = 1, \ldots, \ell_i : \]

\[ u_t - y_{tij} \geq 0 \]  

and \(\forall t = 2, \ldots, T : \)

\[ u_{t-1} - u_t \geq 0 . \]  

2) **Objective Function:** To minimize the makespan, we use the following ILP:

\[ \min \sum_{i=1}^{T} u_t \quad \text{subject to} \quad \text{Eq. (10)} \]

\[ x_{si} \in \{0, 1\} \]

\[ y_{tij} \in \{0, 1\} \]

\[ z_{stij} \in \{0, 1\} \]

\[ u_t \in \{0, 1\} . \]  

**B. Heuristic Methods**

We implemented several heuristics for our problem: A simple FirstFit with and without delaying requests, and two more elaborated heuristics, BestFit and TabuSearch. Our methods pack the given modules in a semi-infinite strip. The width of the strip is given by the number of slots on the FPGA; the height of the strip corresponds to the time axis.

1) **FirstFit:** Probably the simplest heuristic is to place the modules, one by one, in a first-fit way into the strip: beginning with \(s = 1\) and \(t = 1\), we test for every position if the module that must be placed overlaps with already-placed modules. We choose the first position where no overlap occurs. Note that we disregard the possibility of delaying requests.

\[ S = (1, \ldots, M) \]

\[ \text{for } i = 0 \text{ to } M/2 \]

\[ \text{Found} = 0 \]

\[ \text{for } j = 1 \text{ to } M \]

\[ \text{if } (j, ((i + j) \mod M + 1) \in S \]

\[ \text{Swap items at pos. } j \text{ and } ((i + j) \mod M + 1) \text{ in } S \]

\[ \text{Calculate makespan of BestFit} \]

\[ \text{if makespan is the best so far then} \]

\[ \text{Found} = j \]

\[ \text{Undo swapping} \]

\[ \text{if found > 0 then} \]

\[ \text{Swap pos. found and } ((\text{found} + i) \mod M + 1) \text{ in } S \]

\[ \text{Store } \text{found}, ((\text{found} + i) \mod M + 1) \text{ in the tabu list} \]

![Fig. 8. Tabu Search](image)

2) **FirstFit with delays:** This method works the same as the method above, but allows the delaying of requests. That is, if for a certain start position the requests \(0, \ldots, j - 1\) fit into the strip without overlap, but request \(j\) does not fit in time step \(t'\), we search for the largest \(j' < j\) such that request \((i, j')\) fits in \(t'\) and delay every request \(j''\) from \(j'' = j' + 1\), ... (i.e., we move them upwards in the strip); see Fig. 6.

3) **BestFit:** Similar to FirstFit with delays, we try to find a nonoverlapping position by testing every possible position. But now, we do not choose the first feasible position, but we evaluate every position as follows: We separately count the unoccupied cells left and right to the placed module and take the minimum of these two values as a score for the given position. For example, for the placement of \(m_t\) in Fig. 5 there are 4 unoccupied cells left to \(m_t\) and 14 unoccupied cells right to \(m_t\), yielding a value of 4 for his placement. We choose the position that yields the minimal score and break ties by preferring the (first) position with least number of delays.

To avoid that every module is placed on the left or right side (yielding a score of 0), we maintain an upper limit, \(t_{\max}\), for the time. Before we place a new module, \(m_t\), we increase \(t_{\max}\) by \(t_{\ell_i}/2\) and try to place \(m_t\) within the given time bound. If this is not possible, we increase \(t_{\max}\) by \(t_{\ell_i}\) and try again.

4) **TabuSearch:** BestFit inserts the modules in the given order (i.e., \(m_1, m_2, \ldots, m_M\)). Obviously, the result of BestFit highly depends on the insertion order, so we may get a better result if we permute the insertion order of the modules. Thus, we use a tabu search to try several BestFit runs, each one using a different order for the insertion of modules. Starting with the sequence \(S = (1, \ldots, M)\), we swap two items of the sequence and compute the makespan that is achieved by BestFit. More precisely, we maintain a swapping distance, \(i\), ranging from \(i = 0\) to \(M/2\). For a fixed \(i\), we swap the items at positions \(j\) and \(((j + i) \mod M + 1)\) for \(j = 1, \ldots, M\), keeping track of the best makespan achieved so far. We accept the swap that achieves the best makespan known so far. A tabu list ensures that we do not swap an already accepted pair again; see Fig. 6.

**C. Experimental Results**

An example instance and solutions are shown in Fig. 9. The corresponding ILP was solved in approximately 6 hours on
an Intel(R) Xeon(TM) 3.20 GHz CPU running ILOG CPLEX 10.00 under Linux. Note that TabuSearch yields the same result in less than one second.

To test our heuristics, we conducted a set of experiments. For upper limits on the size of a request, $r_{\text{max}}$, ranging from 10% to 90% in steps of 10, we randomly generated sequences, each of 20 modules. For each value of $r_{\text{max}}$, we shuffled 20 sequences as follows: For every module, we choose its length, $\ell _i$, randomly, by normal distribution with expected value $\mu = 10$ and variance $\sigma ^2 = 5$. The size for every request, $r_{ij}$, was chosen by normal distribution, too, with an expected value of $\mu = r_{\text{max}}/2$ and variance $\sigma ^2 = r_{\text{max}}/4$. We present the results for $N = 50$, other FPGA widths showed similar results.

### Table I

| Heuristic      | Average running time |
|----------------|----------------------|
| FirstFit       | 0.23 s               |
| FF with delays | 0.33 s               |
| BestFit        | 2.09 s               |
| Tabu Search    | 1125.06 s            |

#### IV. Conclusion

Reconfigurable devices increasingly offer enough computing resources, so that in the near future, multiple applications may be executed on them concurrently, instead of just a single application. However, until now there is a general lack of research on how to successfully achieve secure and flexible execution of multiple applications on dynamically partially reconfigurable devices. We present an approach called virtual area management, which is heavily influenced by multitasking and operating system concepts of the software. Advantages of our concept (e.g., support for accounting of resources, resource protection, multiple scheduling and placement strategies, and a new programming model) are explained. Furthermore, challenges posed by this concept and possible solutions are discussed. Afterwards, a specific approach to virtual area management in the offline case is presented in more detail. It is based on the assumption that most applications can handle also a delayed resource grant. The corresponding optimization problem to minimize the total makespan is solved with an ILP and heuristics. Both approaches are compared in an experimental study.

The presented concept is a practicable solution to the considered important problem; the proposed methods can be applied to nowadays available reconfigurable devices. We do not rely on the assumption that saving and storing hardware task states will no longer be considered as too expensive in the future. Furthermore, programming models for reconfigurable architectures, resource accounting and protection, bitstream relocation and position independence, are all formidable research problems on their own, however the concept is compatible and extendable to different solution approaches to these problems.

#### References

[1] 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., 2008.

[2] J. Angermieier, D. Göhringer, M. Majer, J. Teich, S. P. Fekete, and J. V. der Veen, “The Erlangen Slot Machine — A platform for interdisciplinary research in dynamically reconfigurable computing,” Information Technology, vol. 49, pp. 143–148, 2007.
Fig. 10. A comparison of our heuristics—FirstFit (without delays), FirstFit (with delays), BestFit, and TabuSearch—in settings with different densities (i.e., maximal value for a request) averaged over 100 runs per densities. For comparison, a lower bound (ratio of total area by number of slots) is shown.

Fig. 11. Mean-, maximal-, and minimal value averaged over 100 runs per densities for BestFit and TabuSearch.