2D Defragmentation Heuristics for Hardware Multitasking on Reconfigurable Devices

Julio Septién[1], Hortensia Mecha[1], Daniel Mozos[1] and Jesús Tabero[2]

[1]Universidad Complutense de Madrid
28040 Madrid, Spain
{jseptien,horten,mozos}@dacya.ucm.es

[2]Instituto Nacional de Técnica Aeroespacial
28850 Madrid, Spain
taberogj@inta.es

Abstract

This paper focuses on the fragmentation problem produced in 2D run-time reconfigurable FPGAs when hardware multitasking management is considered. Though allocation heuristics can take fragmentation into account when a new task arrives, the free area becomes inevitably fragmented as the tasks finish and exit the FPGA. The main contributions of our work are a fragmentation metric able to estimate when the FPGA fragmentation status has become critical, and several heuristics to decide when to perform defragmentation and how to perform it. This defragmentation heuristics can be of a preventive kind, driven by alarms that fire when isolated islands appear or a high fragmentation status is reached. It can be also an on-demand process produced when a task allocation fails though there is enough free area in the FPGA to accommodate it.

1. Introduction

The continuous increase in FPGA features and capabilities has led in recent years to the study of how to exploit such advances in better ways, by performing hardware multitasking through space multiplexing. A single FPGA can be foreseen, in the near future, as executing several tasks or applications concurrently, possibly from different users. The problem of HW multitasking management involves decisions such as the allocation of FPGA resources for each incoming task, the scheduling of the task at an instant where its time constraints are satisfied, and others that have been studied in detail in [1].

A relevant problem occurs when a task finishes and has to leave the FPGA, leaving a rectangular hole that has to be incorporated to the FPGA free area. It becomes inevitable that such process, repeated once and again, generates an external fragmentation that can lead to difficult situations where new tasks are unable to find room in the FPGA though there is enough free area for them.

In order to anticipate or solve such situations, two things have to be done. First, a fragmentation metric that gives an accurate estimation of the FPGA fragmentation status anytime must be developed. This metric can be used to design alarms that initiate, when fired, a routine defragmentation process. On the other side, if an incoming task cannot be accommodated inside the FPGA due to fragmentation, an on-demand defragmentation process can be initiated as well. In such situations, the goal of the defragmentation heuristics used can be different. The first one can have a more ambitious defragmentation goal than the second one, that must give a quick solution to the immediate problem.

The problem of defragmentation is different when FPGAs managed in one or two dimensions are considered. Current commercial FPGAs are oriented to a single dimension (i.e., Xilinx Virtex are only column-programmable, though they consist of a 2D block array), and defragmentation heuristics are very similar to memory defragmentation techniques used in SW multitasking. Compton et al. [2], Brebner et al. [3] or Koch et al. [4] have proposed architectural features to perform 1D defragmentation through relocation of complete columns or rows.

The different aspects of 2D HW multitasking, with special emphasis on online placement, have been studied by several researchers in recent years such as [5], [6], [7], [8], [9] or [10]. Some have even applied 2D techniques to commercial 1D FPGAs, by considering the special ways in which the frame reconfiguration process affects the 2D tasks execution [11]. Finally, most have omitted the I/O or the interconnection problems, though other authors have explicitly proposed the use of some kind of global interconnection network [12]. But only very recently the specific problems posed by fragmentation have begun to be dealt with.

Several researchers such as Walder et al. [13], Tabero et al. [14] or Handa et al. [15] have proposed fragmentation metrics that are used to help their allocation algorithms to choose the more suitable location for the arriving task, though they do not perform an explicit defragmentation.

Gericota et al. propose in [16] architectural changes to a classical 2D FPGA to permit task relocation by

1-4244-0054-6/06/$20.00

replication of CLBs, in order to solve fragmentation problems. But they do not solve the problems of how to choose a new location or how to decide when this relocation must be performed.

Ejnioui et al. [17] have proposed a fragmentation metric adapted from the one shown in [9]. They propose to use this estimation to schedule a defragmentation process when a given threshold is reached. They comment several possible ways of defining such threshold, though they do not seem to choose any of them. Though they suggest several methodologies, they do not give any experimental results.

Van der Veen et al. [11] use a branch-and bound approach with restrictions, to accomplish a global defragmentation process that searches for an optimal module layout. It is aimed to 2D FPGAs, though column-reconfigurable as current Virtex FPGAs. This process seems to be quite time-consuming, of an order of magnitude of seconds. The authors do not give any information about how to insert such defragmentation process, in a HW management system.

2. Our approach to HW management

Our approach to reconfigurable HW management is summarized in Figure 1. Our environment is an extension of the operating system that consists of several modules. The Task Scheduler controls the tasks currently running in the FPGA and accepts new incoming tasks. Tasks can arrive anytime and must be processed on-line. The Vertex-List Updater keeps track of the available FPGA free area with a Vertex-List (VL) structure that has been described in detail in [9], updating it whenever a new event happens. Such structure can be travelled with different heuristics ([9] and [13]) by the Vertex Selector in order to choose the vertex where each arriving task will be placed. Finally, a permanent checking of the FPGA status is made by the Free Area Analyzer. Such module estimates the FPGA fragmentation and checks for isolated islands appearing inside the hole defined by the VL, every time a new event happens.

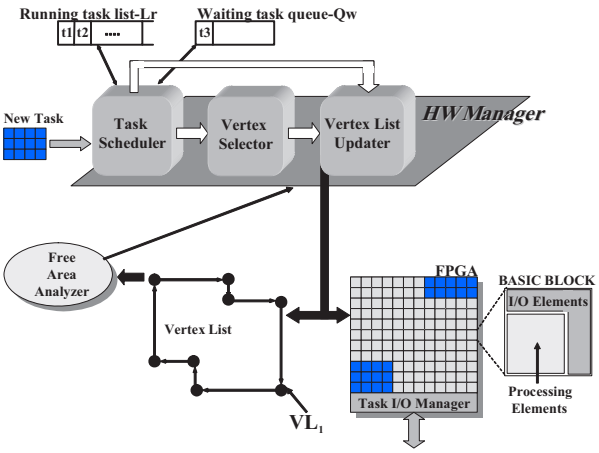


Figure 1. HW management environment.

As Figure 1 shows, we suppose a 2D-managed FPGA, with rectangular relocatable tasks made of a number of basic reconfigurable blocks, each block including processing elements and I/O elements as part of a global interconnection network. Each incoming task $\boldsymbol{T_i}$ is originally defined by the tuple of parameters:

$$\boldsymbol{T_i} = \{w_i, h_i, t\_ex_i, t\_arr_i, t\_max_i\}$$

where w_i times h_i indicates the task size in terms of basic reconfigurable blocks, t_ex_i is the task execution time, t_arr_i the task arrival time and t_max_i the maximum time allowed for the task to finish execution. These parameters are characteristic for each incoming task.

If a suitable location is found, task $\boldsymbol{T_i}$ is finally allocated and scheduled for execution at an instant t_start_i. If not, the task goes to the queue Qw, and it is reconsidered again at each task-end event or after defragmentation. We call the current time *T_curr*. All the times but t_ex_i are absolute (referred to the same time origin). We calculate t_conf_i as the time needed to load the configuration of the task, that it is proportional to its size: $t\_conf_i = \boldsymbol{k} * w_i * h_i$.

We also define t_marg_i, as the margin each task has available before its time-out (defined by t_max_i) is reached. If the task has been scheduled at time t_start_i it must be computed as:

$$t\_margi = t\_maxi - (t\_starti + t\_confi + t\_exi) \qquad (1)$$

But if the task has not been allocated yet, *T_curr* should be used instead of t_start_i. In this case, t_marg_i value decreases at each time cycle as *T_curr* advances. When t_marg_i reaches a value of 0 the task must be definitively rejected and deleted from Qw.

3. Fragmentation analysis

The fragmentation status of the free FPGA area is directly related to the possibility of being able to find a suitable location for an arriving task. We have identified a fragmentation situation by the occurrence of several circumstances. First, proliferation of the number of independent free area holes, each one represented in our system by a different VL. And second, increasing complexity of the hole shape, that we relate with the number of vertices. A particular instance of a complex hole is created when it contains an occupied island inside.

The Free Area Analyzer module, shown in Figure 1, estimates continuously the fragmentation status of the FPGA. This estimation is done with the following metric, very similar to the one we presented in [14]:

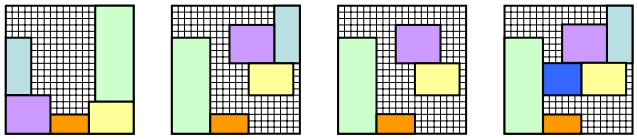
$$\boldsymbol{F} = 1 \; - \Pi_h \; [\; (4/V_h)^n * (A_h/A_{F\_FPGA})] \qquad (2)$$

Where the term between brackets represents a kind of "suitability" for a given hole h, with area A_h and V_h vertices:

- $(4/V_h)^n$ represents the suitability of the shape of hole h to accommodate rectangular tasks. Notice that any hole with four vertices has the best suitability. For most of our experiments we employ n=1, but we can use higher or lower values if we want to penalize more or less the occurrence of holes with complex shapes and thus difficult to use.
- (A_h/A_{F_FPGA}) represents the relative normalized hole area. AF_FPGA stands for the whole free area in the FPGA. That is AF_FPGA = $\sum$ Ah.

This fragmentation metric penalizes the proliferation of holes in the FPGA, as well as the task placements that generate holes with complex shapes and small sizes. Figure 2 shows several fragmentation situations in an example FPGA of 20x20 basic blocks, and the fragmentation values estimated by the formula in (2).

A new estimation is done every time a new event occurs, that is, when a new task is placed in the FPGA, when a finishing task leaves the FPGA, or when relocation decisions are taken during a defragmentation process.



a) F= 0.6 b) F= 0.75 c) F= 0.75 d) F= 0.93

Figure 2. Different FPGA situations and the fragmentation values given by our metric.

The estimation can be used to help in the vertex selection process, as is done in [14], or to check the FPGA status in order to fire a defragmentation process when needed. In the next sections we will focus in how we accomplish defragmentation.

4. Defragmentation

Even if we use intelligent (even fragmentation-aware) heuristics to select the location for each incoming task, it is unavoidable that situations where fragmentation becomes a real problem will eventually arise.

In order to be able to defragment the free area available in an FPGA with several running tasks, we are making some considerations: we will suppose a pre-emptive system, that is, that we have the resources needed to interrupt anytime a currently running task, relocate or reload the task configuration at a different location without modifying its status, and then continue its execution.

We will consider two different defragmentation heuristis, each one for a different situation:

- First, a routine, **preventive defragmentation** will be initiated if an alarm is fired by the Free Area Analyzer module. This alarm has two possible causes: the appearing of an occupied island inside a free hole, as in Figure 2.c, or a high fragmentation FPGA status detected by the metric above, as in Figure 2.b or 2.d . This preventive defragmentation is desired but not urgent, and will be performed only if time constraints for currently running tasks are not too severe.
- Second, an urgent **on-demand defragmentation** will be initiated, if an arriving task cannot find a suitable location in the FPGA, though there is enough free area to accommodate it. This emergency defragmentation will try to get room by moving a single currently running task.

4.1 Defragmentation time-cost estimation

It becomes clear that defragmentation is a time-consuming process, and therefore an estimation of the defragmentation time T_D will be needed in order to decide when, how or even if defragmentation will be performed. We must state also that we will not consider the time spent by the defragmentation algorithms themselves, which run in software in parallel with the tasks in the FPGA.

We have supposed that the defragmentation time cost due to each task will be proportional to the number of basic blocks of the task. And thus the total defragmentation time cost could be estimated as:

$$T_D = 2 * \sum_i t\_conf_i = 2\boldsymbol{k} * \sum_i (w_i * h_i) \quad (3)$$

for all the tasks $\boldsymbol{T_i}$ in the FPGA to be relocated.

The proportionality factor $\boldsymbol{k}$ will depend on the technique we use to relocate the task configuration and on the configuration interface features (for example, the 8-bit SelectMap interface for Virtex FPGAs described in [18]). The factor of 2 appears because we have supposed that configuration reloading is done for each task through a readback of the task configuration and status from the original task location, that are later copied to the new one.

We would get a lower $2\boldsymbol{k}$ value if relocation could be done inside the FPGA, with the help of architectural changes such as the buffer proposed by Compton et al. in [2]. Such buffer, though, poses problems because relocation of each task must take into account the locations of other tasks in the FPGA. But we suppose it is not done by a task shifting technique such as the one explained in [6], because in such case relocation time would depend for each task on the initial and final task locations.

The solution that would get the most significant reduction of $2\boldsymbol{k}$ would be using an FPGA architecture with two different contexts, a simplified version of the classical multicontext architecture proposed by Trimberger [19]. A second context would allow to schedule and accomplish a global defragmentation with a minimal time cost. The configuration load in the second context could be done while tasks go on running, and we would have to add only the time needed to transfer the status of each currently running task from the active context to the other one.

4.2 Preventive defragmentation

This defragmentation is fired by the Free Area Analyzer module, with two possible alarms being the cause: an **island alarm**, or a **fragmentation metrics alarm**. Such situations correspond, respectively, to the examples shown in Figures 2.c and 2.b/d. The first alarm checked is the island alarm. An island is made of one or more tasks that have become isolated when all the tasks surrounding them have already finished. An island can appear only when a task-end event happens. It is obvious that to remove an island by relocating its tasks can lead to a significant reduction of the fragmentation value, and thus we treat it separately.

The second alarm cause is that the fragmentation value rises above a certain threshold. This can happen as a consequence of several different events, and the system will try to perform, if possible, a global or quasi-global relocation of the currently running tasks.

This routine defragmentation is not urgent, or at least it is not fired by the immediate need to allocate an incoming task, and its goal is to get a significantly lower fragmentation FPGA status by taking one of the mentioned actions.

This process will be performed only if the free area is large enough, and it will try first to relocate islands inside the free hole, if they exist, or to relocate most of the currently running tasks if possible.

4.2.1 Island alarm management. Though islands are not going to appear frequently, when they appear inside a hole they must be dealt with before any other consideration is done. An island inside a hole is represented in our system as part of the hole frontier, its vertices belonging to the VL defining the hole as all the other vertices do. We connect the island vertices with the external ones by using two virtual edges, which do not represent, as normal vertices do, a real frontier, and thus they are not considered when intersections are checked. Figure 3.a shows an example with a simple island made of two tasks and its VL is shown in figure 3.b. The island alarm is then only a bit that is set whenever the Free Area Analyzer module detects the presence of a pair of virtual edges in VL, that in the example appear as discontinued arrows.

If the island alarm has been fired, we check first if we can relocate it or not, by demanding that for every task $\boldsymbol{T_i}$ in the island the following condition is satisfied:

$$\textbf{C1:} \quad t\_marg_i \geq T_{D\_island} \quad (4)$$

Where t_marg_i is computed as in (1) and T_{D_island} is the time needed to relocate the complete island, proportional to the island block size and computed as in (3). If condition C1 is satisfied, then new locations for the island tasks are selected by the 3D-adjacency allocation heuristic explained in [14]. The tasks are allocated by decreasing values of t_rem_i, the time the will still remain in the FPGA, that is given by $t\_rem_i = t\_start_i + t\_conf_i + t\_ex_i - T\_curr$. Figure 3.c shows the FPGA status after the island has been removed. Usually, the fragmentation estimation after island removal will lower substantially, below the alarm firing value, and thus we can consider the defragmentation accomplished.

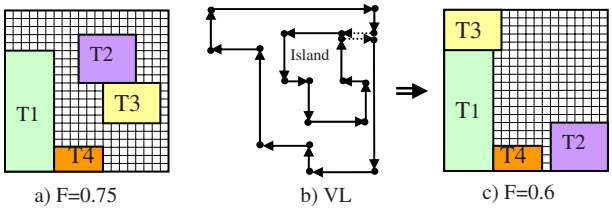


Figure 3. FPGA status with an island (a) and its vertex list (b), and FPGA status after defragmentation (c).

If the island cannot be moved because the C1 condition is not met, then the defragmentation process will not be done.

4.2.2 Fragmentation alarm firing. The Free Area Analyzer module checks continuously the fragmentation status of the FPGA, estimating its value with the metric in (2). The fragmentation alarm fires whenever the estimated value surpasses a given threshold.

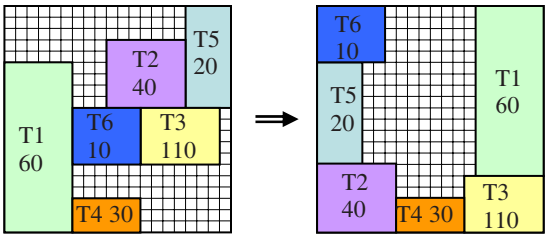
One of our problems has been to set a value for this threshold. We have considered the formula in (2), and analyzed how it behaves in several situations. If n=1, we can see that for two rectangular holes the fragmentation becomes 0.75 or higher. Also, for a single hole, it estimates a fragmentation of 0.6 for a VL with 10 vertices, or 0.75 for a VL with 16 vertices. For the examples shown in this paper, with an average running task number between four and five tasks, we have chosen as threshold a value of 0.75, supposing that a VL with 16 vertices or more, or with several holes, is highly fragmented. With this threshold value, fragmentation alarm would be fired for Figure 2 cases b) and d).

Finally, even when the fragmentation estimation reaches a high value, we have set another condition in order to decide if defragmentation is started: we only perform it if the hole has a significant size. We have set a minimum size value of two times the average task size: $A_{F\_FPGA} \geq 2 * average(Ai)$. Only when this happens the theoretical fragmentation value can be taken as truly significant, and the alarm is really fired.

4.2.3 Running tasks analysis. If a high fragmentation alarm has fired, the system can try a global FPGA defragmentation. In order to decide if such a defragmentation is possible, it must check if all the currently running tasks can be relocated or not, by demanding that for every task T_i in the FPGA the following condition is satisfied:

$$\textbf{C2:}\quad t\_marg_i \geq T_D \qquad (5)$$

Where T_D is the time needed to relocate all the running tasks computed as in (3). If all the tasks satisfy condition C2, then a **global defragmentation** is performed where all the tasks are relocated, starting from an empty FPGA. The task configurations are readback first, and then are relocated at their new locations. In order to reduce the probability of a new fragmentation situation too soon, tasks are relocated in order of decreasing values of t_rem_i, and the allocation heuristic used is based on the 3D-adjacency concept described in [14]. Figure 4.a shows the FPGA status of Figure 2.d with t_rem_i example values for each task $\boldsymbol{T_i}$.



a) F= 0.93 b) F= 0.6

Figure 4. FPGA status before (a) and after (b) a global defragmentation.

A global defragmentation will lead to the situation of Figure 4.b. We have supposed all tasks meet condition C2.

On the contrary, if there are one or more tasks $\boldsymbol{T_j}$ not meeting the condition above, we say these tasks have **severe time constraints**. In such case, a global immediate defragmentation cannot be made and we have to try a different approach. Then we set as a reference the time interval defined by the average time-lapse between consecutive task arrivals, *T_av*. Two situations can happen, depending on the instant the problematic tasks are going to finish, related to *T_av*. If the condition:

$$\textbf{C3:}\quad t\_rem_j < T\_av \qquad (6)$$

is met by all tasks $\boldsymbol{T_j}$ not satisfying C2, a **delayed global fragmentation** will be tried. If this is not the case, an **immediate quasi-global defragmentation** will be performed, affecting only the non-problematic tasks.

4.2.4 Delayed global defragmentation. This heuristic is used when condition C3 is met by all tasks T_j not satisfying C2, that is, the task or tasks T_j with severe time constraint will end "soon". If all the problematic tasks end before this reference threshold is reached, then we can wait the largest t_rem_j value and accomplish a delayed global defragmentation. During this defragmentation we do not perform new incoming task allocations. The tasks arriving during this time-lapse (it would be most probably only one task) will be directly copied to Qw, if they have no severe time constraints. When a task with a severe time constraint arrives the defragmentation process is instantly aborted. Figure 5.a shows a situation derived from Figure 4.a, where condition C2 is not met now by task T6, though it satisfies C3. The situation depicted corresponds to a time instant after 10 cycles when task T6 has already finished. We also suppose no tasks have arrived later than task T5. The figure shows how it is possible to get a much better fragmentation status, though not immediately.

4.2.5 Immediate quasi-global defragmentation. This approach is chosen if the tasks with severe time constraints will finish "late", that is, the condition C3 is

not met. In such case, a quasi-global defragmentation is done immediately, by relocating all the tasks except the problematic ones. Such defragmentation is not optimal, but can reduce the fragmentation value very soon. The configurations of the tasks to be relocated are readback, and then they are relocated as in a global defragmentation, but with a Vertex-List including the problematic tasks, instead of with an empty FPGA.

Figure 5.b shows a situation also derived from Figure 4.a, where condition C2 is not met now by tasks T5 and T6, and thus they cannot be moved. The resulting FPGA fragmentation status is not so good as the delayed one of Figure 5a, but is immediate.

4.3 On-demand defragmentation

The on-demand defragmentation is only accomplished on an urgent basis, when a new task $\boldsymbol{T_N}$ cannot fit inside the FPGA due to fragmentation in spite of all the preventive measures already explained. Reasons for such failing can be the presence of many tasks with severe time constraints in the FPGA, or a fragmentation level below the alarm threshold. Then, as a final action, we try to move a single task in order to get room for the new one.

First, it must be guaranteed that the real problem is fragmentation and not the lack of space. Thus, we will take defragmenting actions only if the free FPGA area is two times the area of the incoming task:

$$A_{F\_FPGA} \geq 2 * (w_i * h_i) \qquad (7)$$

If this condition is met, we choose as **best candidate task for relocation,** $\boldsymbol{T_R}$, the task $\boldsymbol{T_i}$ with the highest percentage of its perimeter P_i belonging to the hole borders, what we have called its **relative adjacency** $radj_i$, that can be actually moved.

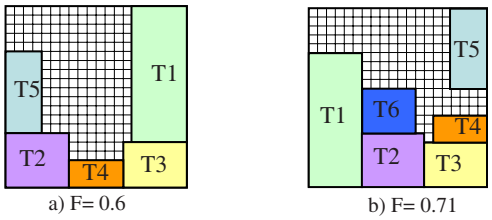


Figure 5. FPGA status derived from Figure 4.a, after a delayed global defragmentation (a), and an immediate quasi-global defragmentation (b)

The $radj_i$ value is computed by the allocation algorithm for every task in the hole border as:

$$radj_i = [(P_i \cap VL) / 2(w_i + h_i)] \qquad (8)$$

$\boldsymbol{T_R}$ will be thus the task $\boldsymbol{T_i}$ with the maximal value of *radj*. The allocation algorithm keeps continuous track of such relocation candidate, anytime the VL is modified, considering only values of $radj_i$ greater than 0.5. Any task forming an island would give the highest possible value of $radj_i$, that is 1. Good candidates would be tasks "joined" with a single side to the rest of the hole perimeter. Figure 6.a shows a candidate $\boldsymbol{T_R}$ intermediate between such two situations, with a *radj* value of 0.9286. On the contrary in Figure 6.c, with all tasks having a *radj* value of 0.5 or lower, no candidate $\boldsymbol{T_R}$ is available any longer because an advantageous quick task move is not obvious.

Moreover, $\boldsymbol{T_R}$ must satisfy: $t\_marg_R \geq t_{DR}$, t_{DR} being the relocation time of the candidate task $\boldsymbol{T_R}$. A similar condition must be satisfied by the incoming task $\boldsymbol{T_N}$ as well: $t\_marg_N \geq t_{DR}$. If these two conditions are met, $\boldsymbol{T_R}$ is relocated with a 3D-adjacency heuristic, and then the new task $\boldsymbol{T_N}$ is considered again, and a suitable location perhaps can be found as in Figure 6.c.

If there is not a valid $\boldsymbol{T_R}$ candidate, though, then the on-demand defragmentation will not take place and the task $\boldsymbol{T_N}$ will go directly to Qw, in hope of a future chance before its t_marg_N is spent. It happens the same if the defragmentation does not give the desired results.

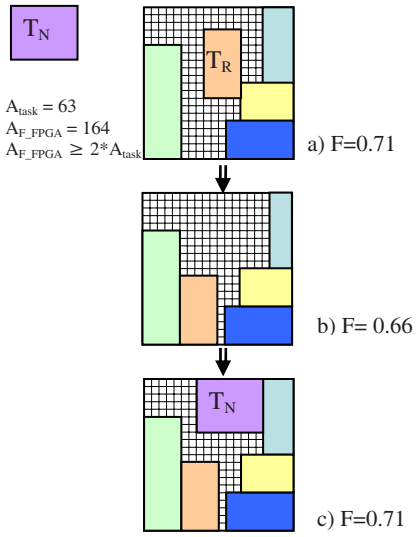


Figure 6. FPGA status before (a) and after (b, then c) an on-demand defragmentation.

6. Conclusions and Future Work

We have presented an approach to 2D hardware multitasking that estimates the fragmentation FPGA status and takes defragmentation decisions when needed. Two basic approaches have been shown: preventive and on-demand defragmentation. Preventive heuristics try to anticipate to possible allocation problems due to fragmentation. They can be fired by two alarm sources: the presence of an island in the vertex list, or a high fragmentation value given by the metric. When fired, it performs an immediate global or quasi-global defragmentation, or a delayed global one depending on the time constraints of the involved tasks. On-demand heuristics try an urgent move of a single candidate task, the one with the highest relative adjacency with the hole border. Such battery of defragmentation measures can help avoiding most problems produced by fragmentation in HW multitasking on 2D reconfigurable hardware.

Future work-plans include the substitution of the fragmentation metric of section 3, vertex-based, by another one based on the relationship between the hole area and its perimeter length, and the study of new architectural features in order to reduce the value of ***k*** described in section 4.1, based on a two-context approach.

7. Acknowledgements

This work has been supported by Spanish Government research grants TIC2002-00160 and TEC2005-04752.

8. References

[1] G. Wigley, D. Kearney, “Research Issues in Operating Systems for Reconfigurable Computing”, Proc. of the *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA’02),* Las Vegas, USA. June 2002.

[2] K. Compton, J. Cooley, S. Knol, S. Hauck, "Configuration Relocation and Defragmentation for Reconfigurable Computing", *Transactions on VLSI Systems*, Vol. 10, No. 3, pp. 209-220, June 2002.

[3] G. Brebner, O. Diessel, “Chip-Based Reconfigurable Task Management”, Proc. of the *Int’l Workshop on Field Programmable Gate Arrays (FPL’01)*, pages 182–191, 2001.

[4] D. Koch, A. Ahmadinia, C. Bobda, H. Kalte, “FPGA Architecture Extensions for Preemptive Multitasking and Hardware Defragmentation”. Proc. of the *IEEE International Conference on Field-Programmable Technology*. Brisbane, Australia, pp. 433-436, December 2004.

[5] K. Bazargan, R. Kastner, M. Sarrafzadeh. “Fast Template Placement for Reconfigurable Computing Systems”, *IEEE Design and Test of Computers*, volume 17, pages 68–83, 2000.

[6] O. Diessel, H. ElGindy, M. Middendorf, H. Schmeck, B. Schmidt, “Dynamic scheduling of tasks on partially reconfigurable FPGAs2. *IEE Proc.-Computer Digital Technology*, Vol. 147, No. 3, May 2000, pp. 181-188.

[7] C. Steiger, H. Walder, M. Platzner. “Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-Time Tasks”. *IEEE Transactions on Computers* vol. 53 (11), pp.1393-1407, 2004.

[8] A. Ahmadinia and J. Teich, “Speeding up online placement for XILINX FPGAs by reducing configuration overhead”, Proc. of the IFIP International Conference on VLSISOC. Darmstadt, Germany: IFIP, Dec. 2003, pp. 118–122.

[9] J. Tabero, J. Septien, H. Mecha, D. Mozos, and S. Roman, "Efficient Hardware Multitasking through Space Multiplexing in 2D RTR FPGAs," Euromicro Digital System Design Conference, September 2003.

[10] M. Handa and R.Vemuri "An Efficient Algorithm for Finding Empty Space for Online FPGA Placement". In *41st Design Automation Conference (DAC’04)*, San Diego, CA, USA. June 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,” Proc. of the *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA´05)*, Las Vegas, USA, June 2005.

[12] C. Bobda, A. Ahmadinia, "Dynamic Interconnection of Reconfigurable Modules on Reconfigurable Device", *IEEE Design and Test of Computers*, vol. 22 (5), pp. 443-451, 2005.

[13] H. Walder, M. Platzner, “Non-preemptive Multitasking on FPGAs: Task Placement and Footprint Transform”, *Engineering of Reconfigurable Systems and Algorithms (ERSA’02)*, June 2002

[14] J. Tabero, J. Septien, H. Mecha, D.Mozos. “A Low Fragmentation Heuristic for Task Placement in 2D RTR HW Management”, Proc. of the *International Conference on Field-Programmable Logic and Applications (FPL’04)*, ser. Lecture Notes in Computer Science, vol. 3203. Springer-Verlag, 2004, pp. 241-250.

[15] M. Handa and R. Vemuri, "Area Fragmentation in Reconfigurable Operating Systems", *Engineering of Reconfigurable Systems and Algorithms (ERSA’04)*, Las Vegas, USA, June 2004.

[16] M. Gericota, G. Alves, M. Silva, and J. Ferreira, “Run-Time Management of Logic Resources on Reconfigurable Systems”, *Design, Automation and Test In Europe Conference (DATE’03)*, Munich, Germany, 3-7 March 2003, pp. 974-979.

[17] A. Ejnioui and R. F. DeMara, “ Area Reclamation Metrics for SRAM-based Reconfigurable Device,” in the Proceedings of The *International Conference on* Engineering of Reconfigurable Systems and Algorithms (ERSA’05), Las Vegas, USA, June 2005.

[18] www.xilinx.com

[19] S. Trimberger, D. Carberry, A. Johnson, and J. Wong, "A timemultiplexed FPGA", Proc. of the *5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM 97),* April 1997, pp. 22--28.