Run–Time Compaction of FPGA Designs

Oliver Diessel[1] and Hossam ElGindy[2]

[1] Department of Computer Science and Software Engineering
[2] Department of Electrical and Computer Engineering
The University of Newcastle, Callaghan NSW 2308, AUSTRALIA

Abstract. Controllers for dynamically reconfigurable FPGAs that are capable of supporting multiple independent tasks simultaneously need to be able to place designs at run–time when the sequence or geometry of designs is not known in advance. As tasks arrive and depart the available cells become fragmented, thereby reducing the controller's ability to place new tasks. The response times of tasks and the utilization of the FPGA consequently suffer. In this paper, we describe and assess a task compaction heuristic that alleviates the problems of external fragmentation by exploiting partial reconfiguration. We identify a region of the chip that can satisfy the next request after the designs occupying the region have been moved. The approach is simple and platform independent. We show by simulation that for a wide range of task sizes and configuration delays, the response of overloaded systems can be improved significantly.

1 Introduction

Recently, FPGA architectures have become partitionable and dynamically reconfigurable — chips have become capable of supporting several independent or interdependent tasks/designs at a time, and parts of the chip can be reconfigured relatively quickly while the remaining tasks continue to execute [6, 1]. These new capabilities promise exciting new application areas and pose several challenging engineering problems, including the design of suitable controllers [4]. When the sequence of tasks to be performed by the chip is known in advance the designer can optimize the use of resources off–line and design an appropriate static controller. However, when the sequence is not predictable, or the task designs are not fixed, the controller needs to make allocation decisions on–line. Unfortunately, on–line allocation schemes that allocate contiguous resources suffer from external fragmentation as variously sized tasks are allocated and deallocated. Tasks end up waiting in a queue despite there being sufficient, albeit non–contiguous resources available to service them. The time to complete tasks is consequently longer, and the utilization of resources is lower than it could be, thereby contributing to the selection of larger, less utilized chips.

We are interested in alleviating this problem. Our aim is to satisfy the next allocation request to a dynamically reconfigurable FPGA that is executing a set of tasks when it is not possible to satisfy the request without compaction. Two subproblems naturally arise:

1. how to identify a good *allocation site*, a sub–array of the requested size efficiently, and
2. how to schedule the compaction so as:
   (a) to free the allocation site of other executing tasks as quickly as possible,
   (b) to delay the tasks that are to be moved as little as possible, and
   (c) to complete compacting the tasks as quickly as possible.

On a grid, it is NP–complete to decide whether or not a set of *rectangular* tasks can be placed orthogonally without overlap [3]. Efficient heuristics are therefore sought. Partial task compaction to reduce fragmentation on meshes has been investigated by Youn et al [7]. However, effective heuristics needed to carry out the arbitrary rearrangements generated by Youn's approach with minimum delay to FPGA tasks are still being sought. In this paper we present a more structured heuristic that is simple, effective, and platform independent. We describe a method that has the effect of sliding the set of tasks to be compacted in a single direction along a single dimension while preserving their relative order. Without loss of generality, we describe compacting the tasks to the right along the rows of the FPGA cells. Fig. 1 contains an example of such a one–way one–dimensional order–preserving compaction [2].

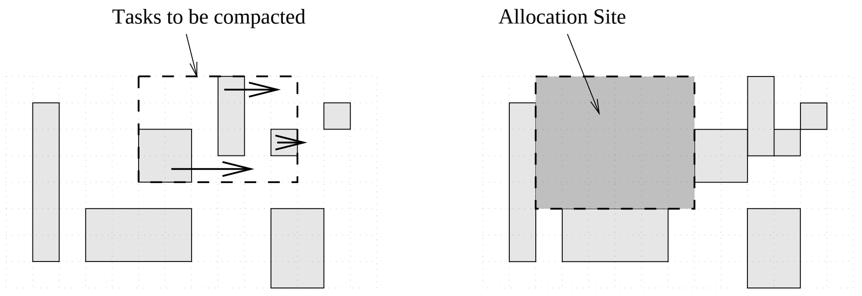


Fig. 1. An example of a one–way one–dimensional order–preserving compaction. The initial arrangement on the left shows the tasks to be compacted so as to allocate a task of size 5×6. The final arrangement on the right indicates the allocation site.

In the following section we describe the assumptions leading to the FPGA task allocation model. In Section 3 we describe and analyze the algorithms used by the controller to compact FPGA tasks at run–time. We show that it is possible to reduce the number of potential allocation sites from $O(H \times W)$ to $O(n^2)$, which is a considerable saving when the number of tasks is relatively small. Thereafter, we describe the construction of a visibility graph over the executing tasks that allows us to determine the feasibility and cost of freeing the executing tasks from each candidate site in $O(n)$ time. In the worst case, we therefore spend $O(n^3)$ time determining whether the incoming task can be allocated with compaction. The site that can be freed of executing tasks in minimum time, and thus satisfies the request quickest of all, can be identified at the same time. In Section 4 we demonstrate by simulation the improvement in performance achievable by use of the compaction heuristic. Our conclusions appear in Section 5, which also mentions areas for future research.

2 Model

For the remainder of this paper we use the terms "task" and "design" synonymously. At the cost of possibly introducing internal fragmentation, we assume that a task and the used routing resources surrounding its perimeter, which may or may not be associated with the task, can be modeled as a rectangular sub–array of arbitrary yet specified dimensions. We assume that the time needed to configure a sub–array (place a design) is proportional to the size (area) of the sub–array since at worst, cells are configured sequentially. Since the delay properties of commercially available chips are isotropic and homogeneous, we assume that the time needed to configure a task and route I/O to it is independent of the task's location.

Moving a task involves: suspending input to the task and waiting for the results of the last input to appear, or waiting for the task to reach a checkpoint; storing register states if necessary; reconfiguring the portion of the FPGA at the task's destination; loading stored register states if necessary; and resuming the supply of input to the task for execution. We do not consider tasks with deadlines, and therefore assume that any task may be suspended, with its inputs being buffered and necessary internal states being latched until the task is resumed. The time needed to wait for the results of an input to appear, or for the task to reach a checkpoint, is considered to be proportional to the size of the task, which, in the absence of feedback circuits, is the worst case. However, since the time to configure a cell and associated routing resources is typically at least one order of magnitude greater than the signal delay of a cell or the latency of a wire, the latency of the design is considered negligible compared with the time needed to configure the task. We investigate the effectiveness of reconfiguring the destination region of a task by reloading the configuration stream with a new offset. This approach naturally re–incurs the cost of placing the task, but is applicable to any device. In this paper we do not address the problem of rerouting I/O to a task that is moved. If I/O to tasks is performed using direct addressing, then tasks not being moved may be delayed by reloading the configuration stream of tasks being moved. We ignore this phenomenon here.

Overall management of tasks is accomplished in the following way. Tasks are queued by a sequential controller as they arrive. A task allocator, executing on the controller, attempts to find a location for the next pending task. If some executing tasks need to be compacted to accommodate the task, then a schedule for suspending and moving them is computed by the allocator. The allocator coordinates the partial reconfiguration of the FPGA according to the compaction schedule, and associates a control process with the new task and its placement. If a location for the next pending task cannot be found, the task waits until one becomes available following one or more deallocations as tasks complete processing.

We use the following notation in this paper: The FPGA of size $F(H, W)$ consists of H rows and W columns of configurable cells arranged in a grid. A task $T_i(s_i, b_i)$ of size $s_i = (r_i, c_i)$ and base $b_i = A_{y_i, x_i}$ is allocated to a submesh of r_i rows and c_i columns of cells with bottom–leftmost cell A_{y_i, x_i}, $1 \leq y_i, 1 \leq x_i$ and top–rightmost cell $A_{y_i+r_i-1, x_i+c_i-1}$ with $y_i + r_i - 1 \leq H$ and $x_i + c_i - 1 \leq W$. The task T_i is said to be based at b_i. We denote the i^{th} row of cells R_i and the j^{th} column of cells C_j. The intersection of R_i and C_j is the cell $A_{i,j}$. The interval of cells $A_{i,k}, A_{i,k+1}, \ldots, A_{i,k+m}$ in the i^{th} row is denoted $R_i[k, k+m]$. A similar definition applies to $C_j[l, l+n]$. The intervals $R_i[k, k+m]$ and $C_j[l, l+n]$ intersect at cell $A_{i,j}$ iff $l \leq i \leq l+n$ and $k \leq j \leq k+m$.

3 Algorithms

3.1 Identifying Potential Allocation Sites

Definition 1. For the request of size $s_{n+1} = (r_{n+1}, c_{n+1})$ we define a *top cell interval* for each executing task $T_i((r_i, c_i), A_{y_i, x_i}), 1 \leq i \leq n$, that consists of the set of possible base locations for T_{n+1} were the bottom edge of T_{n+1} to abut the top edge of T_i. The existence and extent of the top cell interval for T_i is constrained by the boundaries of the chip but disregards the intersection of T_{n+1} with other executing tasks. A top cell interval is also defined with respect to the bottom edge of the chip. We thus define the set of *top cell intervals for* s_{n+1} to be the set $\mathcal{T} = \{R_{r_i+y_i}[\max(1, c_i - c_{n+1} + 1), \min(c_i + x_i - 1, W - c_{n+1} + 1)] : 1 \leq i \leq n, r_i + y_i \leq H - r_{n+1} + 1\} \cup R_1[1, W - c_{n+1} + 1]$.

We similarly define for each executing task T_i a *right cell interval*, consisting of the the set of possible base locations for T_{n+1} were the left edge of T_{n+1} to abut the right edge of T_i. A right cell interval is also defined with respect to the left edge of the chip. The set of *right cell intervals for* s_{n+1} is thus defined to be the set $\mathcal{R} = \{C_{c_i+x_i}[\max(1, r_i - r_{n+1} + 1), \min(r_i + y_i - 1, H - r_{n+1} + 1)] : 1 \leq i \leq n, c_i + x_i \leq W - c_{n+1} + 1\} \cup C_1[1, H - r_{n+1} + 1]$.

The set of cells at the intersection of the set of top and right cell intervals is denoted $\mathcal{B} = \mathcal{T} \cap \mathcal{R}$.

These intervals, which define the minimum cost locations for placing the base of the incoming task T_{n+1} if it is to be allocated in the neighbourhood of T_i, are illustrated in Fig. 2.

Theorem 2. *If T_{n+1} can be allocated by means of compaction, then the cost of freeing the executing tasks is minimized for an allocation site based at some cell in $\mathcal{B}$.*

Proof. The proof considers the time needed to free the space for the incoming task for all of its possible base positions as it is shifted along a row from the left edge of the FPGA to the right. Our assumption is that the cost to free an

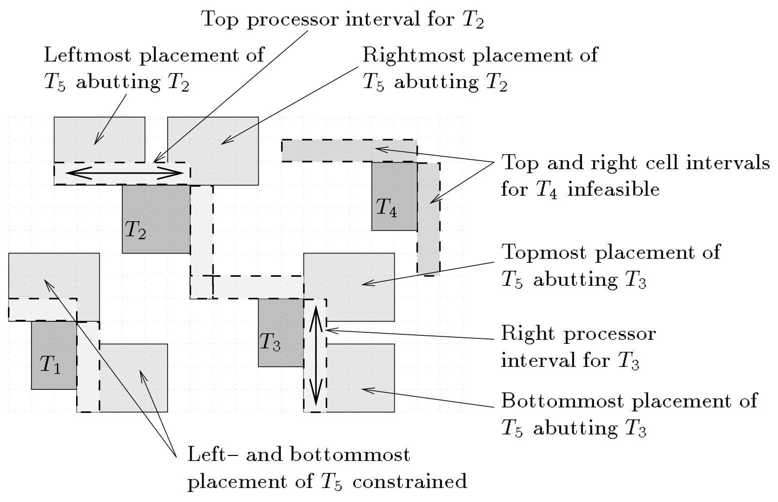

Top processor interval for T_2
Leftmost placement of T_5 abutting T_2
Rightmost placement of T_5 abutting T_2
Top and right cell intervals for T_4 infeasible
T_4
T_2
Topmost placement of T_5 abutting T_3
Right processor interval for T_3
T_3
T_1
Bottommost placement of T_5 abutting T_3
Left– and bottommost placement of T_5 constrained

Fig. 2. Definition of top and right cell intervals for four executing tasks and an incoming task T_5 of size 3×4.

allocation site is at least proportional to the area of tasks that need to be moved out of the allocation site.

Let us consider the allocation site based at $A_{r,1}, 1 \leq r \leq H - r_{n+1} + 1$. Executing tasks lying within the allocation site are to be compacted to the right. Assume the leftmost allocated cell(s) within the allocation site are in column C_c. As the base of the allocation site is shifted to the right from $A_{r,1}$ to $A_{r,c}$, additional allocated tasks potentially become covered by the right edge of the allocation site, thereby increasing the time to free the site of occupying tasks. However, it is not until the first task occupying the allocation site, T_l say, is completely uncovered by the left edge of the allocation site that the time needed to free the site of occupying tasks potentially decreases, since while any cells of T_l remain within the allocation site T_l must be moved to the right of it. Thus it is only necessary to check allocation sites based in the columns of cells $C_{x_i+c_i}$ immediately to the right of executing tasks T_i. The base is constrained from moving to either side with the potential of reducing the cost to free the allocation site by the presence of T_i to the left, and the possibility of covering additional tasks to the right. However, the columns $C_{x_i+c_i}$ need only be checked over the interval in which the task T_i potentially intersects the allocation site, namely $C_{x_i+c_i}[\max(1, r_i - r_{n+1} + 1), \min(r_i + y_i - 1, H - r_{n+1} + 1)]$.

By a similar argument it follows that it is only necessary to check sites in the rows of cells $R_{y_i+r_i}$ immediately above executing tasks T_i. These rows $R_{y_i+r_i}$ need only be checked in the interval in which the task T_i intersects the allocation site, $R_{y_i+r_i}[\max(1, c_i - c_{n+1} + 1), \min(c_i + x_i - 1, W - c_{n+1} + 1)]$.

Consider the top cell interval associated with a task T_i. The allocation site is constrained from moving below or above it without potentially increasing the cost to free the site. Allocation sites based to the right or left of the interval are potentially more costly than sites based within a column that intersects other top cell intervals. A similar argument applies to a right cell interval. Therefore if we consider potential bases within the top interval, the cost to free the allocation site is least where it intersects right intervals. These intersections are guaranteed to exist due to the fact that T_{n+1} cannot be allocated without compaction. $\square$

Constructing the set $\mathcal{B}$ of potential bases for the incoming task requires $O(n^2)$ time if each member of the set of right cell intervals is used to check for intersections against each member of the set of top cell intervals. Since $O(n^2)$ potential base locations have to be identified, this is optimal in the worst case.

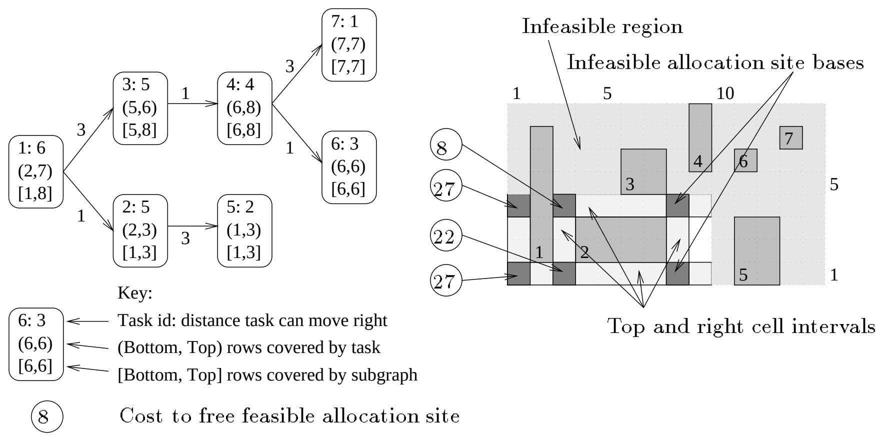
3.2 Assessing Allocation Site Feasibility

Allocation sites based at cells in $\mathcal{B}$ are not guaranteed to be feasible since it may not be possible to compact the executing tasks within the allocation site to the right due to a lack of free cells. An efficient way of answering this question for the $O(n^2)$ possible sites is to build a visibility graph of the executing tasks thereby allowing the feasibility of a site to be determined in $O(n)$ time.

Definition 3. (After [5]) Task V *dominates* a task T if, for some cell A_{r_V,c_V} of V and some cell A_{r_T,c_T} of T, $r_V = r_T$ and $c_V > c_T$. Where V dominates T, we say that V *directly dominates* T if there is no task U such that V dominates U and U dominates T. A *visibility graph* is the directed graph having the collection of executing tasks as vertex set; for each pair of tasks T and V it contains an edge from T to V iff V directly dominates T.

We build the visibility graph in $O(n^2)$ time from its roots to its leaves in the following way. The list of executing tasks is sorted into increasing base column order, where if two or more tasks share a column, they are sorted into increasing row order. For each task we create a graph vertex and insert it in sorted order. A vertex already in the graph has associated with it the bottom– and topmost rows covered by tasks in its subgraph. Vertex insertion can therefore be done in linear time by a depth first search of vertices not visited before to determine whether the task is to the right of the subgraph or not. For each edge inserted, we associate the distance from the parent to the newly added child. After the graph has been built, we compute and store at each vertex the maximum distance the task can be moved to the right by summing the edge distances in a bottom–up fashion. Note that the distance the terminal nodes can be moved is given by their base columns and their widths. This final step, which takes $O(n)$ time, saves time during searching by eliminating the need to determine the cost of compaction for allocation sites that cannot be freed of executing tasks. Fig 3 depicts the visibility graph for our example.

For each potential base $b \in \mathcal{B}$, those subgraphs whose covered rows intersect the allocation site based at b are searched depth first down to the leftmost task(s) that intersect the allocation site. The possibility of moving each of these out of the way of the incoming task is then checked in $O(n)$ time per base b.



1: 6
(2,7)
[1,8]
3: 5
(5,6)
[5,8]
4: 4
(6,8)
[6,8]
7: 1
(7,7)
[7,7]
6: 3
(6,6)
[6,6]
2: 5
(2,3)
[1,3]
5: 2
(1,3)
[1,3]
Key:
Task id: distance task can move right
(Bottom, Top) rows covered by task
[Bottom, Top] rows covered by subgraph
Cost to free feasible allocation site
Infeasible region
Infeasible allocation site bases
Top and right cell intervals
8
27
22
27

Fig. 3. Visibility graph for arrangement of tasks on the right, with cost to free feasible allocation sites for an incoming task of size 5×6.

3.3 Selecting the Best Allocation Site

We can identify the set of tasks to be compacted and the distance each is to be moved by traversing each subgraph rooted in the potential allocation site. This traversal is needed to accumulate the cost of freeing the site as well since it is assumed to be proportional to the sum of the areas of the tasks that have to be moved. These traversals, which take $O(n)$ time, are required for each feasible allocation site. The cost of freeing each of the potential allocation sites for our example is illustrated in Fig. 3.

3.4 Scheduling the Compaction

Given a set of tasks that are to be compacted, the scheduling policy we investigate moves tasks according to the visibility graph: a task is not moved until all tasks in its subgraph that must move have moved. The policy attempts to minimize delays to executing tasks by suspending each task that is to be moved for the period needed to reload the configuration, and by moving a task onto a region of the FPGA that does not overlap any other executing tasks. Note that the scheduling policy moves the tasks occupying the allocation site last of all, and therefore does not minimize the time needed to free the allocation site. Scheduling the compaction is straightforward and requires time linear in the number of tasks to be compacted.

4 Experimental Evaluation

We evaluated the performance of a simulated FPGA chip operating with and without compaction. The simulator queued and then allocated a set of random requests for service using two allocation methods:

1. With Compaction — attempted to satisfy the next pending request by compaction whenever the task could not be allocated by the Bottom–left method, and
2. Bottom–left Allocation — allocated the next pending task to the bottom–leftmost free subarray whenever possible.

Experiments were conducted to measure the performance under varying load, varying configuration delays, and varying task sizes. All experiments involved generating 10,000 requests for service to an $F(64, 64)$ chip. The service period generated for each task ranged uniformly between 1 and 1,000 time units. The task size per side and the intertask arrival period were independently generated random variables. Each experiment involved fixing two of the maximum task size, maximum intertask arrival period, and configuration delay per cell, and varying the third. The amount of time a task spent waiting at the head of the queue to begin entering the chip, the elapsed time between a task arriving and completing processing, and the chip utilization were measured.

Effect of System Load on Allocation Performance Task sizes were allowed to range up to 32 cells per side and the configuration delay per cell was set to 1/1,000 of a time unit, giving a mean configuration delay per task of about 0.3 time units. See Fig. 4(a). At maximum inter–task arrival periods below 50 time units the FPGA was saturated with work as tasks arrived more frequently than they could be allocated. Compaction resulted in a reduction in mean allocation delay of approximately 19% at saturation, which caused a reduction in mean response time. The reduction in mean response time due to compaction increased from approximately 26% in the saturated region to over 75% at loads where the chip was coming out of saturation before falling to zero in the unsaturated region. Fig. 4(b) illustrates that the utilization was roughly 25% higher in the saturated region due to compaction. This benefit rapidly decreased to zero as the chip came out of saturation because compaction cannot influence the inter–task arrival period. Significantly, when compaction is used, the system has a higher load–bearing capacity, as evidenced by decreases in response times and utilization at greater task arrival frequencies.

Effect of Configuration Delay on Allocation Performance The performance benefits due to compaction, though significant in saturated systems, are greatest when the system is coming out of saturation. We investigated the response times in this operating range as the configuration delay per cell was increased from 1/1,000 time unit to 1,000 time units. Task sizes were again allowed to range up to 32 cells per side and the maximum intertask arrival period was set to 120 time units. Fig 4(c) indicates that at configuration delays of less than 50% of the mean service period, the performance benefit due to compaction was significant.

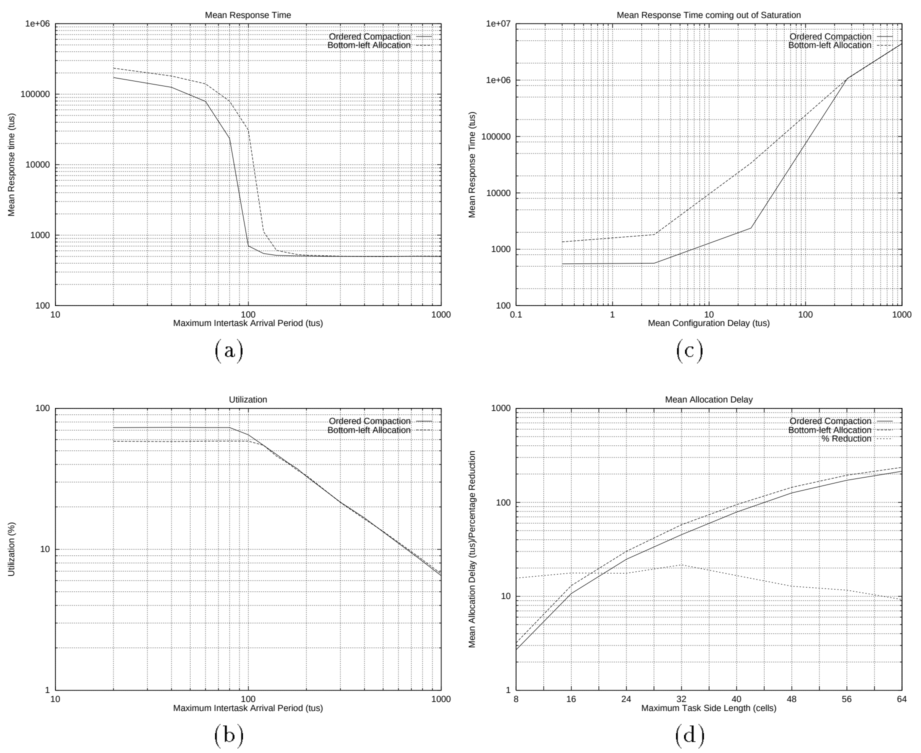


Fig. 4. (a) Mean response time and (b) chip utilization for 10,000 tasks of size U(1,32) $\times$ U(1,32), service period U(1,1000) time units (tus), and uniform intertask arrival periods on F(64,64). (c) Dependency of response time on configuration delay for above tasks with intertask arrival period of U(1,120) tus. (d) Dependency of mean allocation delay on task size for above tasks arriving with intertask arrival period of 1 tu.

Effect of Task Size on Allocation Performance To give some indication of the relative performance benefits as the maximum task size changes, Fig. 4(d) plots the mean allocation delay at saturation as the maximum task size was increased from 8 to 64 cells per side. The configuration delay per cell was set to 1/1,000 time unit, and tasks arrived at intervals of 1 time unit. The figure indicates that the performance benefit due to compaction increased as the maximum task size was increased from 8 since small tasks are more easily accommodated without compaction. The benefit decreased again as the maximum task size approached the chip size, suggesting that less opportunities for free fragment combination by one–dimensional ordered compaction presented themselves.

5 Concluding Remarks

In this paper we described an effective heuristic for alleviating fragmentation by task migration in partially reconfigurable FPGAs. The one–dimensional ordered partial task compaction method is generally applicable because it operates independently of the task scheduling and allocation method. It is platform independent since task are moved by reloading their configurations. Simulations indicate that significant performance benefits can be gained as the FPGA becomes saturated with work even when configuration delays are large. Further evaluation under real run–time conditions is desired.

Many research problems remain to be solved. These include: examining the possibility of improving the time complexity of compaction algorithms; developing methods to relocate the tasks occupying an allocation site to arbitrary locations on the FPGA, which has the potential of improving the performance of compaction; taking into account tasks with deadlines; and, determining practical means of rerouting I/O to migrated tasks.

Acknowledgments

This research was partially supported by grants from the Australian Research Council, and the RMC at The University of Newcastle.

References

1. Atmel: AT6000 FPGA configuration guide
2. Diessel, O., ElGindy, H.: Run–time compaction of FPGA designs. Technical report 97–02, Department of Computer Science and Software Engineering, The University of Newcastle (1997). Available by anonymous ftp: `ftp.cs.newcastle.edu.au/pub/techreports/tr97-02.ps.Z`
3. Li, K., Cheng, K. H.: Complexity of resource allocation and job scheduling problems on partitionable mesh connected systems. Technical report UH-CS-88-11, Department of Computer Science, University of Houston, Houston, TX (1988). Proceedings 1st IEEE Symposium on Parallel and Distributed Processing (1989) 358–365
4. Lysaght, P., McGregor, G., Stockwood, J.: Configuration controller synthesis for dynamically reconfigurable systems. IEE Colloquium on Hardware–Software Cosynthesis for reconfigurable systems, London, UK, (1996)
5. Sprague, A. P.: A parallel algorithm to construct a dominance graph on nonoverlapping rectangles. International Journal of Parallel Programming **21** (1992) 303–312
6. Xilinx: XC6200 field programmable gate arrays. Technical report, Xilinx, Inc., (1996)
7. Youn, H.-y., Yoo, S.-M., Shirazi, B.: Task relocation for two–dimensional meshes. Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems (1994) 230–235