Partial FPGA Rearrangement by Local Repacking

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

Partial rearrangement of executing tasks has been proposed as a means of alleviating the fragmentation of free logic elements that occurs on space–shared run–time reconfigurable FPGA systems. In this paper, we present and assess a new solution to this strategy. Local repacking of executing tasks aims to free sufficient contiguous resources for the next waiting task so as to minimize allocation and execution delays. Heuristics for the NP–hard problems of identifying and scheduling optimal task rearrangements are described and assessed by comparison with known methods.

1 Introduction

Partial configuration and context switching are two features of current FPGAs that permit efficient implementation of run–time reconfiguration.

Partial reconfiguration has been variously used to recycle resources that are not currently used for circuits that are currently needed. A good example of this technique is the DISC system, which makes use of a well–defined global context to implement relocatable tasks of arbitrary size [9]. When a new task is to be executed and there is insufficient contiguous space for it, the least recently used task is removed from the system. The effective area of the FPGA is increased by simulating many FPGAs, or a much larger FPGA, on a small one. More recently, interest has grown in exploiting partial reconfiguration to share the FPGA amongst multiple simultaneous tasks and/or users [6].

Switching between configurations to time–share the device amongst several tasks or users is under investigation for the Garp system [5]. The Garp system time slices between four contexts, each of which is dedicated to a single task at a time. While DISC and Garp allow the low–level parallelism inherent in applications to be exploited, much of the FPGA resource may remain idle because tasks are processed sequentially. To be able to utilize the unused portions of devices, and, more importantly, to reduce response times by processing tasks in parallel, future FPGA systems must also consider employing space–sharing. Such systems process multiple tasks simultaneously, allocating resources according to need. Tasks must wait for sufficient resources to become available before they can be loaded.

Space–shared FPGA systems will have to overcome several design hurdles if they are to become viable, and fulfill their potential. Of critical interest is the management of

shared resources such as I/O pins, wires, and logic blocks. To maximize utilization of space–shared FPGAs, for example, they will have to overcome the resource fragmentation problem, which occurs as variously sized tasks are loaded onto and unloaded from the array. As a consequence of fragmentation, a large fraction of the array can remain idle, and tasks may be forced to wait for sufficiently many contiguous resources to become available. In [4] we proposed using partial reconfiguration to alleviate fragmentation by compacting a subset of the tasks executing on the FPGA. In this paper, we develop and assess local repacking, a new approach to rearranging a subset of the tasks executing on the FPGA to free a sufficiently large contiguous block for the next waiting task.

Local repacking proceeds in two steps. The first step identifies a rearrangement of the tasks executing on the FPGA that frees sufficient space for the waiting task, and the second schedules the movements of tasks so as to allocate the waiting task as quickly as possible and minimize the delays to executing tasks. We use a quadtree decomposition [7] of the free cells within the FPGA to identify those sub–arrays that could potentially accommodate the waiting task if the tasks executing within the sub–array were repacked. Well–known two–dimensional bin packing algorithms [2] can then be used to attempt repacking the tasks occupying the sub–array together with the waiting task. If a feasible rearrangement is found, we schedule the movement of tasks so as to minimize the maximum delay to executing tasks when the waiting task is allocated immediately. Subject to these constraints, scheduling the rearrangement of FPGA tasks is NP–complete [3], thus we describe a polynomial time approximation algorithm.

In the following section we describe our FPGA model and terminology. Section 3 describes the local repacking approach to finding and scheduling FPGA task rearrangements. An experimental study of the performance of local repacking is reported on in Section 4. Concluding remarks appear in Section 5.

2 Model

Definition 1 *An FPGA of width W and height H is a two–dimensional grid of configurable cells denoted $G^2[(1,1),(W,H)]$ with bottom–left cell labeled $(1,1)$, and top–right cell labeled (W,H).*

We assume that an FPGA 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. Tasks are assumed to be independent.

Definition 2 *The FPGA task $t[l_1,l_2], l_1, l_2 \in Z^+, l_1 \leq l_2$ requires an array of size $l_1 \times l_2$ to execute.*

An FPGA is said to be partitionable when non–overlapping orthogonally aligned rectangular sub–arrays can be allocated to independent tasks. Each task is allocated a sub–array of the required size within a larger partitionable array. Usually a sub–array will simply be referred to as an array as well.

Definition 3 *The orientation, $\mathrm{or}(t) = (x,y)$, of a task, t, specifies the number of cell columns, x, and rows, y, allocated to the task from the array.*

Tasks may be rotated and relocated. Task $t[l_1, l_2]$ may be oriented such that $\text{or}(t) = (l_1, l_2)$, or such that $\text{or}(t) = (l_2, l_1)$. If $\text{or}(t) = (l_1, l_2)$, then it may be allocated to any array, $G^2[(x, y), (x + l_1 - 1, y + l_2 - 1)]$ where $1 \leq x \leq W - l_1 + 1$ and $1 \leq y \leq H - l_2 + 1$.

We assume that the time needed to configure a sub–array,

$$t_{\text{conf}}(G^2[(x_1, y_1), (x_2, y_2)]) = \text{CD} \times (x_2 - x_1 + 1)(y_2 - y_1 + 1), \tag{1}$$

is proportional to the configuration delay per cell, *CD*, and the size 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 *CD* is a constant i.e., the time needed to configure a task and route I/O to it is independent of the task's location and orientation.

Definition 4 *Let* $T = \{t_i[l_{1i}, l_{2i}] : 1 \leq i \leq n\}$ *be a set of tasks allocated to an FPGA* $G^2[(1, 1), (W, H)]$.

The arrangement of tasks, $A(G^2[(1, 1), (W, H)]) = \{a(t_i) : t_i \in T\}$*, is the set of non–overlapping orthogonally aligned rectangular allocations,* $a(t_i) = G^2[\text{bl}(t_i), \text{tr}(t_i)]$*, in the array* $G^2[(1, 1), (W, H)]$*. The allocation for task* t_i *is said to be based at the cell allocated to the bottom–left corner of the task,* $\text{bl}(t_i)$*, and to extend to the cell allocated to the top–right corner of the task,* $\text{tr}(t_i)$*.*

Rearranging the tasks executing on an FPGA requires moving them. 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 an 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 configuring the task, given above in Equation 1, 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 rearranged 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 rearrangement 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.

3 Local Repacking

Local repacking begins by finding a feasible rearrangement of the tasks. Deciding whether or not a set of orthogonal rectangles can be packed into a larger rectangle without overlap is NP–complete [**?**]. It is therefore also NP–complete to decide whether or not a waiting task can be allocated following task rearrangement. Efficient heuristics for finding rearrangements are consequently sought. We use a hierarchical decomposition of the array called a free area tree to keep track of the number of free cells within each sub–array. In so doing, regions that contain sufficient free area to accommodate the waiting task are identified by depth–first traversal, and a rearrangement of the tasks they contain is attempted. A two–dimensional bin packing algorithm with good absolute performance bounds is used for this last step. The tasks, viewed as rectangles, are packed from scratch into an infinitely long strip whose width is one side of the sub–array. If the tasks are packed using total height less than the other side of the sub–array, the rearrangement is feasible, and its cost can then be assessed.

When a feasible rearrangement has been identified, the rearrangement of the tasks is scheduled. The cost of a rearrangement is measured in terms of the maximum of the individual execution delays to those tasks that are to be moved. This cost is governed by the set of tasks to be moved, the overlap between the initial and final arrangements, and the sequence in which they are moved. We develop an ordered depth–first heuristic search algorithm to minimize the cost of rearranging a set of tasks. Having determined the cost of one rearrangement, the search for a minimum cost rearrangement could proceed by attempting to rearrange other sets of tasks in the course of traversing the free area tree.

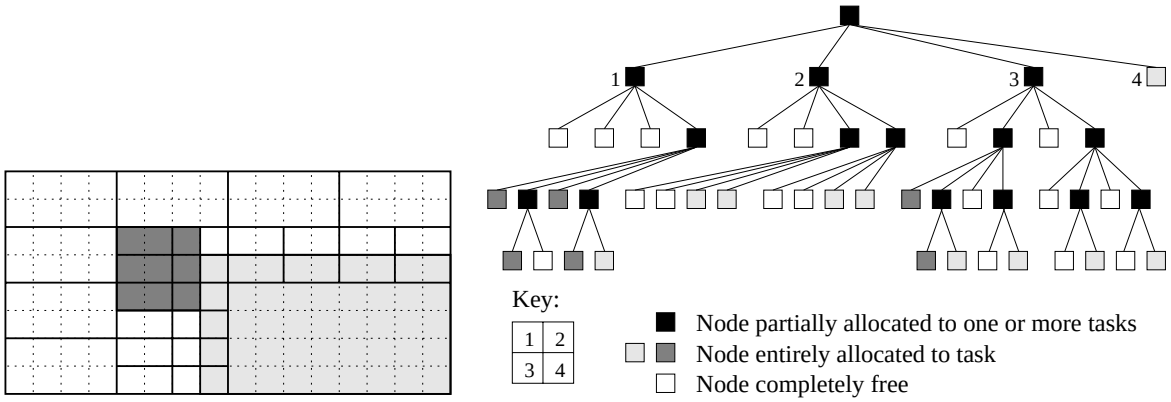
This section concludes with a brief comparison of the algorithmic complexity of local repacking and ordered compaction.

3.1 Identifying feasible rearrangements

3.1.1 Free Area Trees

A *free area tree* is a type of quadtree [7, 10], that need not necessarily be defined over a square grid, and whose leaves may have just one rather than three siblings. Each node of the tree represents a portion of the array, stores the number of free cells contained within the region, and pointers to its children. If the array covered by a node is completely free, or if it is entirely allocated to a single task, then it is not further decomposed. Otherwise, the array represented by the node is partitioned evenly into two or four disjoint sub–arrays, depending upon its size, and represented by child nodes. Figure 1 depicts the arrangement of a pair of tasks and the corresponding free area tree.

The local area repacking method commences by building the free area tree for an arrangement of tasks on the array. Next the tree is searched for nodes that contain more free cells than are needed by the waiting task. For each such node a repacking of the tasks allocated to the array covered by the node is attempted. These tasks are found in linear time by checking for intersections with the node's array.



(a) (b)

Figure 1: Arrangement of two tasks, (a), and corresponding free area tree, (b).

Tasks which only partially intersect the array covered by a node need to be handled in some way. Should they be included in the packing, moved elsewhere, or left where they are to be packed around? The approach adopted here is to attempt to repack these tasks completely into the rectangular array covered by the node as well. At each node therefore, the area available for the waiting task needs to account for the total area of tasks that are only partially covered by the region. If the free area less this attached area at a node exceeds the area of the waiting task, then a packing into the array is attempted.

3.1.2 Building the Free Area Tree

The free area tree is expanded iteratively by inserting each of the executing tasks into an initially empty root. The task insertion procedure updates the free and attached area for the current node and task, and recurses with those children that are partially intersected by the task. It expands the tree by creating the children that don't already exist.

3.1.3 Searching the Free Area Tree

It is desirable that the free area tree be searched in some way that allows promising regions to be discovered early in the search. Ideally we want to discover that region which costs least to repack first of all, knowing that it is the best available. Searching the tree breadth–first allows schedules affecting successively fewer tasks to be discovered, and allows the search to be abandoned at a time when the marginal benefit of finding arrangements with lower allocation and execution delays is offset by the growing allocation delay due to the search.

In Section 4 we report on the performance of a local repacking method that implements a depth–first search of the free area tree, and which abandons the search once the first feasible arrangement is found.

3.1.4 Repacking the Tasks

Given a set of oriented rectangles and a two–dimensional bin of a given width and unbounded height, the strip–packing problem is to find a minimum height non–overlapping orthogonal packing of the rectangles into the bin. This variant of the two–dimensional

bin–packing problem is NP–complete; much attention has therefore been given to finding polynomial approximation algorithms with good performance bounds. For L an arbitrary list of rectangles, let $\mathrm{OPT}(L)$ denote the minimum possible bin height into which the rectangles in L can be packed, and let $\mathrm{A}(L)$ denote the height actually used by a particular algorithm when applied to L. An *absolute performance bound*, β, for A is a bound of the form $\mathrm{A}(L) \leq \beta\mathrm{OPT}(L)$.

So as to minimize the frequency with which an algorithm fails to find a feasible arrangement when such an arrangement is possible, their absolute performance bounds should be as small as possible. The algorithm should also be efficient so as to keep the scheduling component of the allocation delay to a minimum. Sleator proposed an $O(n \log n)$ time strip–packing algorithm with $\mathrm{A}(L) \leq 2\mathrm{OPT}(L) + 0.5h_{\mathrm{tall}}$, where h_{tall} is the height of the tallest rectangle [8]. Since $h_{\mathrm{tall}} \leq \mathrm{OPT}(L)$, $\mathrm{A}(L) \leq 2.5\mathrm{OPT}(L)$ in the worst case. Asymptotically, however, $\mathrm{A}(L) \Rightarrow 2\mathrm{OPT}(L)$ as $h_{\mathrm{tall}} \Rightarrow 0$.

We report on the effectiveness of using Sleator's algorithm to attempt the repacking. Given the node $G^2[(x_1, y_1), (x_2, y_2)]$ has been identified as a likely candidate, we first try a "strip" of width $(x_2 - x_1 + 1)$. While the orientation of the allocated tasks relative to the width of the strip needs to be preserved to obtain the performance of known strip–packing algorithms, a packing with each orientation of the waiting task is attempted. A feasible rearrangement results if the height of the packing is less than $(y_2 - y_1 + 1)$. Otherwise, the orientation of the strip is flipped so that its width is considered to be $(y_2 - y_1 + 1)$, and a packing within a height of $(x_2 - x_1 + 1)$ is attempted.

3.2 Scheduling the rearrangement

Our goal is to minimize the delays to executing tasks with the constraint that the waiting task is to be placed first of all. The problem of scheduling FPGA task rearrangements to realize this goal is NP–complete. Formulating the scheduling problem as a search for a path in a state–space tree suggests adopting the use of a depth–first search with a simple estimator of path cost to give an approximate solution in polynomial time.

3.2.1 FPGA rearrangement scheduling is NP–complete

Definition 5 *Given two arrangements of a set of FPGA tasks, an initial arrangement* $A(G^2[(1,1),(W,H)]) = \{a(t_i) : t_i \in T\}$ *and a final arrangement* $A'(G^2[(1,1),(W,H)]) = \{a'(t_i) : t_i \in T\}$*, the intersection set of task* t_i*,* $I(t_i) \subseteq T - \{t_i\}$*, is the set of tasks in the initial arrangement that are intersected by* t_i *when it is placed into its final position, i.e.,* $I(t_i) = \{t_j : a(t_j) \cap a'(t_i) \neq \emptyset\}$*.*

Given an initial and a final arrangement of a set of FPGA tasks, we are motivated to find a method for rearranging the tasks, i.e., moving all tasks from their initial to their final allocations, that minimizes the delay (defined below) to tasks under the following constraints:

C1: A task must be removed from its initial position on the array before it can be placed into its final position. The removal of a task from the array is instantaneous.

C2: Only one task at a time can be placed. A task can only be placed into its final position and the placement of a task cannot be interrupted. The time needed to place a task is equal to its size $s(t_i) = w(\text{or}(t_i)) \times h(\text{or}(t_i))$ since the configuration delay per cell is the same for all tasks.

C3: Any tasks in $I(t_i)$ that have not yet been removed from the array at the instant the placement of t_i commences are simultaneously removed from the array.

C4: The waiting task, t_{n+1}, which is assumed to be initially removed from the array and therefore without an initial position, is the first task placed into its final position.

Definition 6 *The elapsed time between the removal of a task from the array and the* commencement *of its placement represents a delay to the task.*

Let $r(t_i)$ be the time t_i is removed from the array, $p(t_i)$ be the time the placement of t_i commences, and $d(t_i) = p(t_i) - r(t_i)$ be the delay to t_i.

FPGA REARRANGEMENT SCHEDULING
INSTANCE: A set $T = \{t_1, \ldots, t_{n+1}\}$ of tasks and a delay bound $D \in Z^+$. For each task $t_i \in T$, a size $s(t_i) \in Z^+$, and an intersection set $I(t_i) \subseteq T - \{t_i\}$.
QUESTION: Is there a schedule $p : T \rightarrow Z_0^+$ subject to C1 through C4 with $\max\{p(t_j) - p(t_i) : t_j \in I(t_i)\} \leq D$ for all i?

Theorem 1 *FPGA REARRANGEMENT SCHEDULING is NP–complete.*

Corollary 1 *With the constraint of placing the waiting task first of all, scheduling the ordered compaction of FPGA tasks to minimize delays to executing tasks is NP–complete in one or two dimensions.*

Corollary 2 *Without the constraint of placing the waiting task first of all, FPGA rearrangement scheduling is NP–complete.*

The reader is referred to [3] for details of the proofs.

3.2.2 FPGA rearrangement scheduling as heuristic search

The FPGA rearrangement scheduling problem may be thought of as a search for a task reconfiguration sequence that minimizes the maximum delay to tasks. With n tasks to rearrange after configuring the waiting task, there are $n!$ different ways of sequencing the rearrangement. Each schedule can be viewed as a path from the root of a tree to a leaf, where a node, $c_i, 0 \leq i \leq n$, represents the ith sequencing choice.

In FPGA rearrangement scheduling, each path has a cost associated with it, which is the maximum of the execution delays to the tasks when they are relocated in the sequence given by the path. The FPGA rearrangement scheduling problem is to find a cost–minimal path, which is known as a solution path.

At a node, the search for a cost–minimal path proceeds by calculating the cost associated with each arc leaving the node. This process is called expanding the node. After a node has been expanded, a decision is made about which node to expand next.

Optimal search algorithms such as A* [1] potentially require exponential time and space because they attempt to make a globally optimal choice of the most promising node at each step. An *ordered depth–first search* [1] achieves an acceptable solution most of the time for polynomial cost. The idea is to make a locally optimal choice of the next node to expand by always expanding the most promising successor of the last node expanded. The choice of next node to expand is based on an underestimate, f^*, of the cost of a solution path passing through each possible successor of the current node.

Procedure **Approximate FPGA Rearrangement Scheduling**(TaskList, WaitingTask)
begin

1. Create a current state with the waiting task on a list of reconfigured tasks. Place the tasks intersected by the waiting task on a list of suspended tasks, and place the remaining tasks on a list of executing tasks.
2. Repeat n times:
   (a) Initialize $f^*_{\min}$ to a large value.
   (b) For each task, t_i, not relocated yet:
      i. Create an open state copy of the current state with task t_i removed from its source list and appended to the list of reconfigured tasks. Remove the remaining executing tasks intersected by t_i from the list of executing tasks and insert them into the list of suspended tasks.
      ii. Calculate f^* for the open state and save it and a new value for $f^*_{\min}$ if $f^* < f^*_{\min}$.
   (c) Copy the saved state to the current state.
3. Report the sequence in which tasks are reconfigured.

end

It remains for us to describe the nature of the evaluation function, f^*. The cost of reaching a node is given by the maximum of the delays to the relocated tasks, which is known. A simple estimator of the minimal–cost path to reach a leaf from the node is also available: we ignore those tasks that have not yet been moved or suspended, and determine the maximum amount by which the suspended tasks could be delayed. Ignoring the list of tasks that are yet to be suspended or relocated allows us to optimally schedule the suspended tasks in polynomial time as a result of the following lemma.

Lemma 1 *If $r(t_i)$ is the time at which an FPGA task, t_i, is removed from the array and $s(t_i)$ is its size, then the suspended tasks are optimally scheduled in non–decreasing $r(t_i) + s(t_i)$ order if none of them causes additional tasks to be suspended.*

To apply Lemma 1, it is useful to keep the list of suspended tasks in sorted order, and therefore to implement it using a priority queue with $\Theta(\log n)$ insertion and deletion time. Note that Step 2 examines all possible next states from the previously expanded state and closes all but the best. We felt our estimator may not look far enough ahead to be useful when the number of tasks to be rearranged is large, and consequently implemented and tested an algorithm which looks two states ahead in Section 4, the drawback being that another factor of n is added to the time complexity of the algorithm.

3.3 Complexity comparison with ordered compaction

For the FPGA of width W and height H, with $m = \max\{W, H\}$, and n executing tasks, the local repacking method requires $O(mn)$ time to build the free area tree. With $O(m)$ nodes, the tree can be searched in $O(mn \log n)$ time for the existence of a feasible rearrangement. Since m is a constant, this time complexity compares favorably with the $O(n^3)$ needed by ordered compaction to determine whether a feasible compaction exists or not.

Without the constraint of placing the waiting task first of all, ordered compaction needs $O(n)$ time to schedule the rearrangement so as to minimize the delays to the executing tasks, whereas local repacking requires $O(n^2 \log n)$ time with one–state lookahead, or $O(n^3 \log n)$ time with two states of lookahead. When the waiting task is to be placed first of all, both methods need to use an approximate scheduling method. In each case, the schedule cannot be executed until after it has been computed.

4 Experimental Evaluation

Simulating the arrangement of tasks over time on an FPGA chip allows us to compare the ability of a general FPGA model to complete given work sets with various approaches to allocating and rearranging tasks. In this work, we compared the performance of a first–fit task allocator with task allocators using ordered compaction [4] and local repacking. In outline, the simulator's operation is as follows. The simulator generates a random set of tasks within specified parameters and queues them in arrival order. When a site for the task at the head of the queue is found using the allocation method under test, it is loaded onto the FPGA. The task remains allocated until its service period has finished, whereupon it is removed from the FPGA.

Three experiments were conducted to compare the performance of the different allocation methods. An experiment consisted of a specified number of runs for a fixed set of task and FPGA parameters: maximum task side length, L, maximum inter–task arrival period, P, and configuration delay, CD. Averaging the results of a number of runs was used to reduce the uncertainty in the result.

Figures 2(a) through 2(d) show the effect on performance of varying the service load with nominal configuration delay. In broad terms, we observed marginally better performance for local repacking than for ordered compaction. The mean allocation delay for local repacking was almost 24% less than for first fit, and over 3% better than for ordered compaction. The queue delay and response times are consequently less, and the utilization greater than for ordered compaction and first fit.

Figures 3(a) through 3(d) show the effect on performance of varying the configuration delay at different system loads. Local repacking performed better than ordered compaction when the mean configuration delay was very low (less than 1% of the mean service period). Local repacking began performing worse than first fit at mean configuration delays less than 5% of the mean service period. By comparison, ordered compaction performed worse than first fit at mean configuration delays greater than 10% of the mean service period.

The dependency of performance upon task size at saturation with nominal configuration delay was also investigated. An improvement in performance was observed to increase from $L = 8$ up to $L = 32$, beyond which the improvement in performance decreased. For $L < 32$

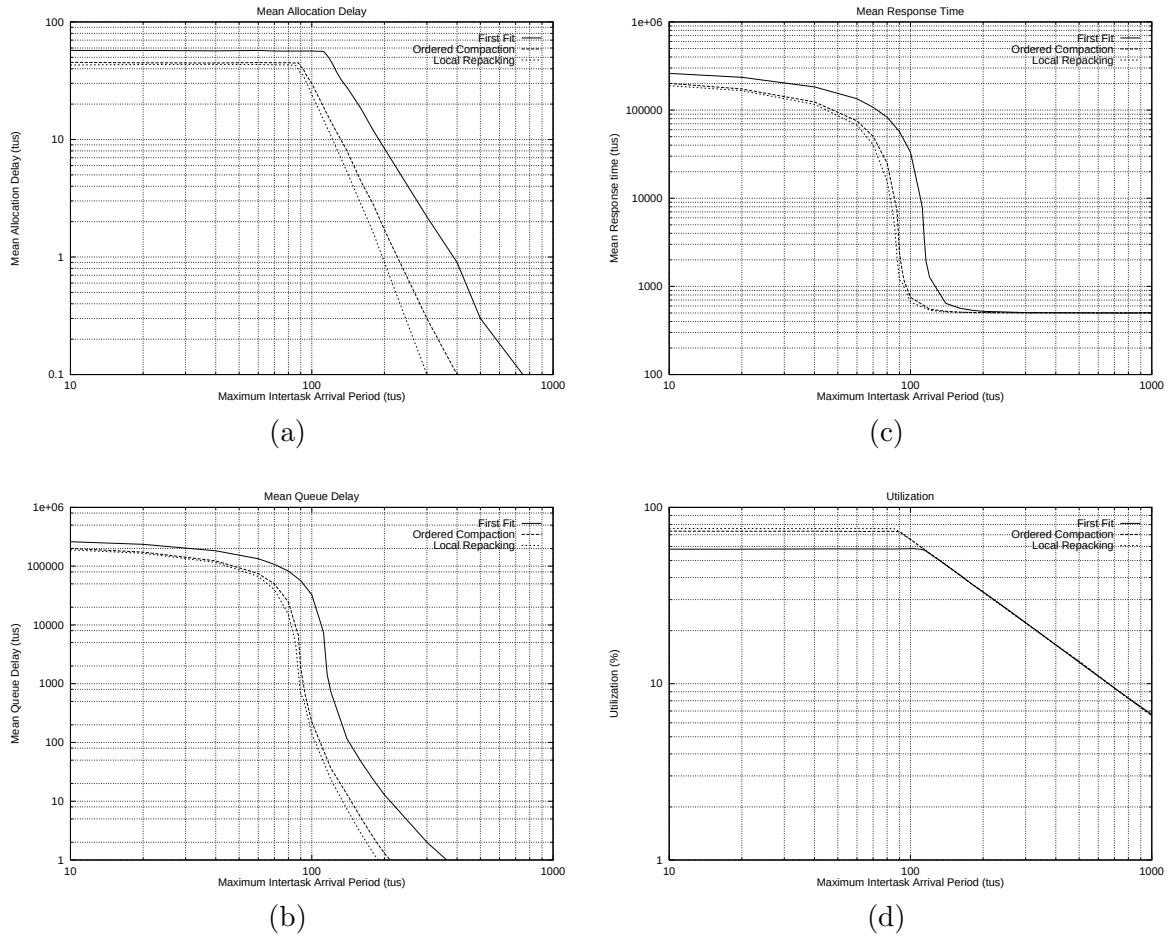


Figure 2: (a) Mean allocation delay, (b) mean queue delay, (c) mean response time, and (d) utilization for first fit, ordered compaction, and local repacking as the maximum inter–task arrival period was varied. 10,000 tasks of size U(1,32) $\times$ U(1,32), service period U(1,1000) time units (tus), arriving at uniformly distributed time intervals were allocated to a simulated 64×64 cell FPGA with a configuration delay of 1/1000 tu/cell.

local repacking performed better than ordered compaction, but for $L > 32$, it performed worse.

5 Concluding Remarks

Future run–time reconfigurable FPGAs will need to consider space–sharing to increase utilization and obtain speedup by processing tasks in parallel. To overcome the fragmentation of FPGA resources that occurs with on–line task allocation, partially rearranging the array was previously proposed [4]. Local repacking is a new method for this strategy. The method was motivated by the desire to find a more effective alternative to the ordered compaction approach, which only collects available resources in a single dimension. Our assessment does not clearly establish which of these methods is better.

Both local repacking and ordered compaction work in two steps by identifying a feasible rearrangement, and scheduling the task movements. While local repacking requires less

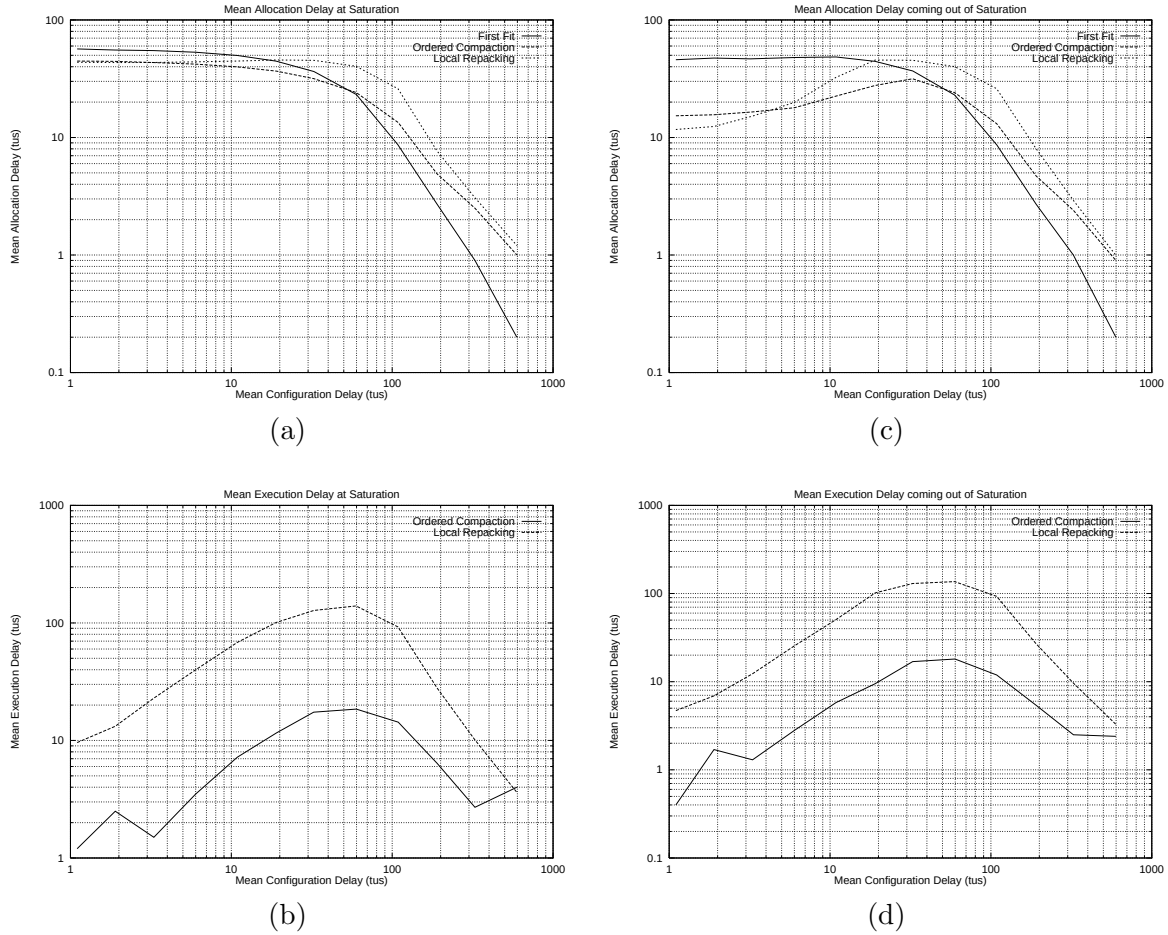


Figure 3: Mean allocation delay and mean execution delay at saturation and coming out of saturation for first fit, ordered compaction, and local repacking as the mean configuration delay per task was varied. 10,000 tasks of size U(1,32) $\times$ U(1,32), service period U(1,1000) time units (tus), arriving at intervals of U(1,40) tus ((a) and (b)), and arriving at intervals of U(1,120) tus ((c) and (d)), were allocated to a simulated FPGA of size 64×64.

computation time to perform the first step, it is possible for the scheduling to be performed more quickly by ordered compaction. Local repacking appears to be more effective at finding feasible rearrangements when task side lengths are smaller than half the array side lengths, the FPGA is saturated with work, and configuration delays are small relative to service periods. On the other hand, ordered compaction performs better as task sizes and configuration delays increase.

There appear to be many avenues for improvement. We are interested in discovering whether algorithms with better run time or performance can be found. Finding a faster, better scheduling algorithm is of particular interest. The performance of local repacking when rearrangement schedules are not constrained to place the waiting task first of all should also be evaluated.

There is also ample scope for further work. The effectiveness of the methods would be enhanced if the number of movements individual tasks are subjected to could be reduced, and if arbitrary rearrangements that affect less tasks could be developed. The applicability of partial FPGA rearrangement needs to be extended by developing methods that take real–

time and dependent tasks into account. Developing overall specifications for the capabilities of space–shared FPGA operating systems, and developing the hardware support for multiple simultaneous tasks are long–term goals.

References

[1] A. Barr and E. A. Feigenbaum, editors. *The Handbook of Artificial Intelligence*, volume I. William Kaufmann, Inc., Los Altos, CA, 1981.

[2] E. G. Coffman Jr., M. R. Garey, and D. S. Johnson. Approximation algorithms for bin-packing – an updated survey. In G. Ausiello, M. Lucertini, and P. Serafini, editors, *Algorithm Design for Computer System Design*, pages 49 – 106. Springer-Verlag, Vienna, Austria, 1984.

[3] O. Diessel and H. ElGindy. Partial FPGA rearrangement by local repacking. Technical report 97–08, Department of Computer Science and Software Engineering, The University of Newcastle, Sept. 1997. Available by anonymous ftp: `ftp.cs.newcastle.edu.au/pub/techreports/tr97-08.ps.Z`.

[4] O. Diessel and H. ElGindy. Run–time compaction of FPGA designs. In W. Luk, P. Y. K. Cheung, and M. Glesner, editors, *Field–Programmable Logic and Applications, 7th International Workshop, FPL'97 Proceedings*, pages 131 – 140, Berlin, Germany, 1997. Springer–Verlag.

[5] J. R. Hauser and J. Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In K. L. Pocek and J. M. Arnold, editors, *The 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'97)*, pages 24 – 33, Los Alamitos, CA, Apr. 1997. IEEE Computer Society.

[6] P. Lysaght, G. McGregor, and J. Stockwood. Configuration controller synthesis for dynamically reconfigurable systems. In *IEE Colloquium on Hardware–Software Cosynthesis for Reconfigurable Systems*, pages 1 – 9, London, UK, Feb. 1996. IEE.

[7] H. Samet. The quadtree and related hierarchical data structures. *Computing Surveys*, 16(2):187 – 260, June 1984.

[8] D. D. K. D. B. Sleator. A 2.5 times optimal algorithm for packing in two dimensions. *Information Processing Letters*, 10(1):37 – 40, Feb. 1980.

[9] M. J. Wirthlin and B. L. Hutchings. Sequencing run–time reconfigured hardware with software. In *FPGA'96 1996 ACM Fourth International Symposium on Field Programmable Gate Arrays*, pages 122 – 128, New York, NY, Feb. 1996. ACM Press.

[10] Y. Zhu. Fast processor allocation and dynamic scheduling for mesh multiprocessors. *Computer Systems Science and Engineering*, 11(2):99 – 107, Mar. 1996.