Partial FPGA Rearrangement by Local Repacking

Oliver Diessel[1] and Hossam ElGindy[2]

Technical Report 97–08
September, 1997

[1]Department of Computer Science and Software Engineering
[2]Department of Electrical and Computer Engineering
The University of Newcastle, Callaghan NSW 2308, Australia
{odiessel@cs,hossam@ee}.newcastle.edu.au

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 [16]. 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 [13].

Switching between configurations to time–share the device amongst several tasks or users is under investigation for the Garp system [11]. 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 [8] 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 [14] 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 [4] 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 [7], 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 $\mathrm{or}(t) = (l_1,l_2)$, or such that $\mathrm{or}(t) = (l_2,l_1)$. If $\mathrm{or}(t) = (l_1,l_2)$, 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,

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

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, $a(t_i) = G^2[\mathrm{bl}(t_i),\mathrm{tr}(t_i)]$, 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, $\mathrm{bl}(t_i)$, and to extend to the cell allocated to the top–right corner of the task, $\mathrm{tr}(t_i)$.

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

The idea behind local repacking is to repack the tasks initially allocated to some rectangular region of the array so as to accommodate the waiting task within the region as well. In this section we describe the local repacking procedure.

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 [12]. 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

Following a definition of a free area tree, and a description of its use, the algorithms for constructing and then searching the free area tree are presented. We then describe the use of known strip–packing algorithms to attempt a repacking of sets of tasks found in the course of the tree search.

3.1.1 Free Area Trees

A *free area tree* is a type of quadtree [14, 18], 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. A formal definition of a free area tree follows.

Definition 5 *Array* $G_1^2[(x_1, y_1), (x_2, y_2)]$ *is said to intersect array* $G_2^2[(x_3, y_3), (x_4, y_4)]$ *iff* $(x_1 \leq x_4)$ *and* $(x_3 \leq x_2)$ *and* $(y_1 \leq y_4)$ *and* $(y_3 \leq y_2)$.

The intersection of arrays, $G_1^2[\,(x_1, y_1),\, (x_2, y_2)] \cap G_2^2[\,(x_3, y_3),\, (x_4, y_4)]$, *is the array* $G^2[\,(\max(x_1, x_3), \max(y_1, y_3)), (\min(x_2, x_4), \min(y_2, y_4))]$, *if* $(x_1 \leq x_4)$ *and* $(x_3 \leq x_2)$ *and* $(y_1 \leq y_4)$ *and* $(y_3 \leq y_2)$, *otherwise it does not exist and is defined to be* $\emptyset$.

Definition 6 *The area of the array* $G^2[(x_1, y_1), (x_2, y_2)]$ *is* $\mathrm{ar}(G_1^2[(x_1, y_1), (x_2, y_2)]) = (x_2 - x_1 + 1)(y_2 - y_1 + 1)$. *By definition,* $\mathrm{ar}(\emptyset) = 0$.

Definition 7 *The free area of the array* $G^2[(x_1, y_1), (x_2, y_2)]$ *is the number of unallocated cells,* $\mathrm{fa}(G^2[(x_1, y_1), (x_2, y_2)])$, *in the array.*

For the arrangement of tasks, $A(G^2[(1, 1), (W, H)]) = \{a(t_i) : 1 \leq i \leq n\}$, *the free area* $\mathrm{fa}(G^2[(x_1, y_1), (x_2, y_2)]) = \mathrm{ar}(G^2[(x_1, y_1), (x_2, y_2)]) - \sum_{i=1}^{i=n} \mathrm{ar}(G^2[(x_1, y_1), (x_2, y_2)] \cap G^2[\mathrm{bl}(t_i), \mathrm{tr}(t_i)])$.

Definition 8 *The predicate $h(G^2[(x_1, y_1), (x_2, y_2)])$ is defined to be true if some task, t_i, exists such that some but not all of the cells in $G^2[(x_1, y_1), (x_2, y_2)]$ are allocated to it, i.e., if for some allocated task t_i, $0 < \mathrm{ar}(G^2[(x_1, y_1), (x_2, y_2)] \cap G^2[\mathrm{bl}(t_i), \mathrm{tr}(t_i)]) < \mathrm{ar}(G^2[(x_1, y_1), (x_2, y_2)])$.*

Definition 9 *(After [18]) The free area tree, $F[(x_1, y_1), (x_2, y_2)]$, covering $G^2[(x_1, y_1), (x_2, y_2)]$ is defined recursively as follows:*

1. *$F[(x, y), (x, y)]$ is a leaf node.*
2. *$F[(x, y_1), (x, y_2)]$ with $y_1 < y_2$ is a node.*
   If $h(G^2[(x, y_1), (x, y_2)])$, then $F[(x, y_1), (x, y_2)]$ has two children:
   (a) *$F[(x, y_1), (x, \lfloor (y_1 + y_2)/2 \rfloor)]$, and*
   (b) *$F[(x, \lfloor (y_1 + y_2)/2 \rfloor + 1), (x, y_2)]$.*
3. *$F[(x_1, y), (x_2, y)]$ with $x_1 < x_2$ is a node.*
   If $h(G^2[(x_1, y), (x_2, y)])$, then $F[(x_1, y), (x_2, y)]$ has two children:
   (a) *$F[(x_1, y), (\lfloor (x_1 + x_2)/2 \rfloor, y)]$, and*
   (b) *$F[(\lfloor (x_1 + x_2)/2 \rfloor + 1, y), (x_2, y)]$.*
4. *$F[(x_1, y_1), (x_2, y_2)]$ with $x_1 < x_2$ and $y_1 < y_2$ is a node.*
   If $h(G^2[(x_1, y_1), (x_2, y_2)])$, then $F[(x_1, y_1), (x_2, y_2)]$ has four children:
   (a) *$F[(x_1, y_1), (\lfloor (x_1 + x_2)/2 \rfloor, \lfloor (y_1 + y_2)/2 \rfloor)]$,*
   (b) *$F[(\lfloor (x_1 + x_2)/2 \rfloor + 1, y_1), (x_2, \lfloor (y_1 + y_2)/2 \rfloor)]$,*
   (c) *$F[(x_1, \lfloor (y_1 + y_2)/2 \rfloor + 1), (\lfloor (x_1 + x_2)/2 \rfloor, y_2)]$, and*
   (d) *$F[(\lfloor (x_1 + x_2)/2 \rfloor + 1, \lfloor (y_1 + y_2)/2 \rfloor + 1), (x_2, y_2)]$.*

Figure 1 depicts the arrangement of a pair of tasks on a rectangular array. The array is partitioned into the regions delimiting the extent of the leaf nodes in the free area tree representation of the arrangement.

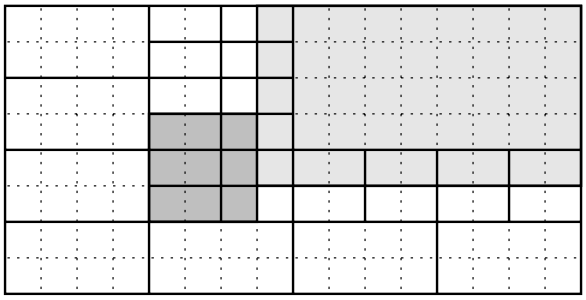

Figure 1: Arrangement of tasks on array with free area tree leaves marked.

The free area tree corresponding to the arrangement of Figure 1 is illustrated in Figure 2. The left to right order of nodes on each level corresponds to the order in which the children of a node are listed in Definition 9.

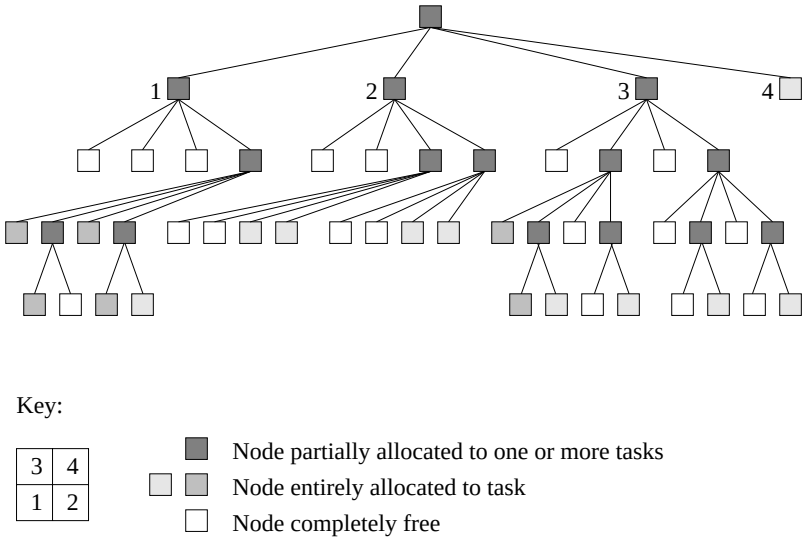

Figure 2: Free area tree for arrangement of Figure 1.

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. If the new arrangement accommodates the waiting task within the array covered by the node as well, then the rearrangement of the tasks to achieve the packing can be scheduled in order to evaluate its optimality.

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.

Definition 10 *If the allocation for task t_i intersects the array $G^2[(x_1, y_1), (x_2, y_2)]$, then the uncovered area,* $\mathrm{ua}(t_i, G^2[(x_1, y_1), (x_2, y_2)]) = \mathrm{ar}(G^2[bl(t_i), tr(t_i)]) - \mathrm{ar}(G^2[\mathrm{bl}(t_i), \mathrm{tr}(t_i)] \cap G^2[(x_1, y_1), (x_2, y_2)])$. *Otherwise,* $\mathrm{ua}(t_i, G^2[(x_1, y_1), (x_2, y_2)]) = 0$

Definition 11 *The attached area,* $\mathrm{aa}(F[(x_1, y_1), (x_2, y_2)]) = \sum_{i=1}^{i=n} \mathrm{ua}(t_i, G^2[(x_1, y_1), (x_2, y_2)])$, *is the total number of cells allocated to the uncovered portion of tasks that are partially intersected by the free area tree node $F[(x_1, y_1), (x_2, y_2)]$.*

If the free area less the attached area at a node exceeds the area of the waiting task, then a packing into the array covered by the node of all the tasks intersected by it and the waiting task 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[1]. The procedure *InsertTaskIntoFAT* 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.

Procedure **InsertTaskIntoFAT**

Input A pointer to a node in a free area tree and a pointer to the descriptor of a task that is to be inserted into the tree.

Output A free area tree that has been expanded or modified to account for the task inserted into it.

begin

1. compute the area of intersection (AI) between the node array and the task
2. update the free area for the node
3. if the area of the task is greater than AI (the node does not wholly contain the task), then
   (a) update the attached area for the node
4. if the area of the node is greater than AI (the node is not completely allocated to the task), then
   (a) if the children of the node have not yet been created, then
      i. create child nodes with free area initialized to the child's area and attached area set to zero
   (b) for each child that intersects the task
      i. recurse with the task descriptor and a pointer to the child node
5. return

end

Except for step 4(b), each step requires constant time, and since the cost of step 4(b) can be attributed to the ancestors of the node input to the procedure, the time spent updating each node in the tree is a constant.

Dyer has analyzed the size of the quadtree representation of square objects in square images [9]. He showed that $O(2^{p+2} - p + q)$ nodes at worst are needed to represent a square of size $2^p \times 2^p$ in an image of size $2^q \times 2^q$. This expression accounts for the perimeter of the square,

[1] An empty node has free area set to the area of the array covered by the node and attached area set to zero.

the logarithm of its diameter, and the height of the root of the tree above the expanded subtree covering the object. If we let $m = \max(W, H)$, it therefore follows that $O(m)$ nodes of the free area tree are updated per task insertion since no task can be larger than the array. Thus $O(mn)$ time is needed to build the free area tree for n tasks. The worst case is attained by any arrangement of tasks that occupy entire rows of a square array, for example, $A(G^2[(1,1)(W,W)]) = \{a(t_i) = G^2[(1,i),(W,i)] : 1 \leq i \leq n \leq W\}$.

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. A "deepest layer first" search examines those nodes which affect the least number of tasks first, but which have the least chance of accommodating the waiting task. An ideal search therefore starts somewhat higher in the tree and works its way upwards.

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

The search of the free area tree identifies those sub–arrays that might accommodate the waiting task if the tasks allocated to it are rearranged. We propose using well–known strip–packing algorithms for the purpose of checking whether such an arrangement exists.

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 [2]. This variant of the two–dimensional bin–packing problem is NP–complete; much attention has therefore been given to finding polynomial approximation algorithms, i.e., fast algorithms that come within a constant times the height used by an optimal packing [4]. 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 [5]. An *absolute performance bound* β for A is a bound of the form $\mathrm{A}(L) \leq \beta\mathrm{OPT}(L)$. On the other hand, in *asymptotic performance bounds* of the form $\mathrm{A}(L) \leq \beta\mathrm{OPT}(L) + \gamma$, the constant β is intended to characterize the behaviour of the algorithm as the ratio between $\mathrm{OPT}(L)$ and the maximum height rectangle in L goes to infinity. The height of the tallest rectangle is usually normalized to 1, whereby any other choice would only affect the constant γ.

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 [15]. 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 $h_{\mathrm{tall}} \Rightarrow 0$. Sleator suggested using his algorithm together with the Split–Fit algorithm of Coffman et al [5] that has better asymptotic performance. Their algorithm, which has time complexity $O(n \log n)$, is characterized by the equation $\mathrm{A}(L) \leq 1.5\mathrm{OPT}(L) + 2$ when the height of the tallest rectangle is normalized to 1. Should even better asymptotic performance be desired the $O(n \log n)$ time 5/4 algorithm of Baker et al [1] has the characteristic equation $\mathrm{A}(L) \leq 5/4\mathrm{OPT}(L) + 53/8h_{\mathrm{tall}}$. We extend the suggestion of Sleator to include an $O(n \log n)$ stacking algorithm due to Coffman and Shor [6] that has good asymptotic average case performance. For their algorithm the expected height of a strip–packing of rectangles from the uniform model of rectangles (side lengths in [0,1]) is $\mathrm{E}[\mathrm{A}(L)] = n/4 + \Theta(\sqrt{n})$, i.e., $\Theta(\sqrt{n})$ space is wasted.

In this report, we report on the effectiveness of using Sleator's algorithm to attempt the repacking. Given the node $F[(x_1, y_1), (x_2, y_2)]$ 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.

As mentioned in Section 3.1.1, we attempt to pack tasks that are partially intersected by the sub–array into the array as well. If, however, a partially intersected task couldn't possibly fit because one of its sides is too long, the repacking is aborted.

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. We begin by showing that 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 A* algorithm to obtain an exact (optimal) solution. A depth–first search of the state–space using a simple estimator of path cost yields an approximate solution in polynomial time.

3.2.1 FPGA rearrangement scheduling is NP–complete

Definition 12 *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 partitions, 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 $s(t_i) = w(\text{rm or}(t_i)) \times h(\text{rm or}(t_i))$ 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 13 *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. The sequencing constraints can then be formulated in the following way:

$$
\begin{array}{ll}
r(t_i) \leq p(t_i) & \text{(C1)}, \\
p(t_i) > p(t_j) \Rightarrow p(t_i) \geq p(t_j) + s(t_j) & \text{(C2)}, \\
\forall t_j \in I(t_i), r(t_j) \leq p(t_i) & \text{(C3)}, \\
r(t_i) \leq \min\{p(t_i), \{p(t_j) | t_i \in I(t_j)\}\} & \text{(C1 \& C3), and} \\
r(t_{n+1}) = p(t_{n+1}) = 0 & \text{(C4)}.
\end{array}
$$

Our problem now is to determine the complexity of finding a schedule $p : T \rightarrow Z_0^+$ that minimizes $\max\{d(t_1), d(t_2), \ldots, d(t_n)\}$.

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.*

Proof: It is easy to see that FPGA REARRANGEMENT SCHEDULING is in NP, since a non–deterministic algorithm need only guess a schedule and then check in polynomial time that the placement constraints and the delay bound are met. To show that the FPGA REARRANGEMENT SCHEDULING is NP–complete, we transform the well–known PARTITION problem to it [10].

Let the non–empty set $A = \{a_1, \ldots, a_n\}$ with size $s(a_i) \in Z^+$ for each $a_i \in A$ constitute an instance of PARTITION, and let $S = \sum_{i=1}^{n} s(a_i)$. We construct an instance of the FPGA REARRANGEMENT SCHEDULING problem consisting of $n + 3$ tasks with $D =$

$4S + \lfloor S/2 \rfloor$ such that the delay bound can be met if and only if the set A can be partitioned into two subsets $A' \subset A$ and $A - A'$ such that $|A'| = |A - A'| = \lfloor S/2 \rfloor$.

Let us denote by $t_i = [w(\text{rm or}(t_i)), h(\text{rm or}(t_i))]$ a task oriented with width $w(\text{rm or}(t_i))$, height $h(\text{rm or}(t_i))$, and size $s(t_i) = w(\text{rm or}(t_i)) \times h(\text{rm or}(t_i))$. We set

$t_1 = t_{n+3} = [2S, 2]$,
$t_2 = [S, 1]$,

and construct a further n tasks corresponding to the items in A,

$t_{i+2} = [s(a_i), 1], 1 \leq i \leq n$.

The initial arrangement of the tasks for a particular instance is illustrated in Figure 3. Task t_1 initially occupies the second and third rows from the bottom of an array of width $2S$ and height 5. Tasks t_2 through t_{n+2} are arranged in sequence from left to right along the fourth row from the bottom of the array.

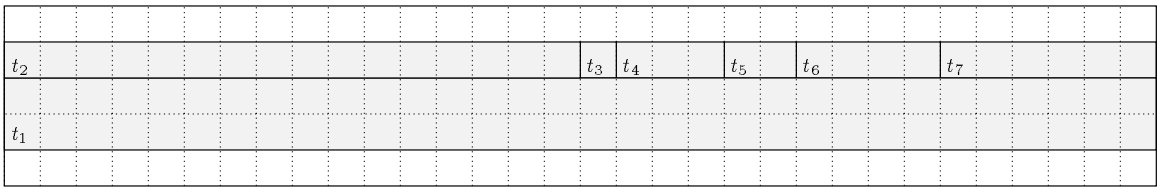

Figure 3: The initial arrangement of the tasks for an instance of PARTITION with $A = \{1, 3, 2, 4, 6\}$.

The final arrangement of the tasks for our example is shown in Figure 4. The bottom–left corner of task t_{n+3} is aligned with the bottom–left corner of the array, and tasks t_1 through t_{n+2} have been shifted up a row.

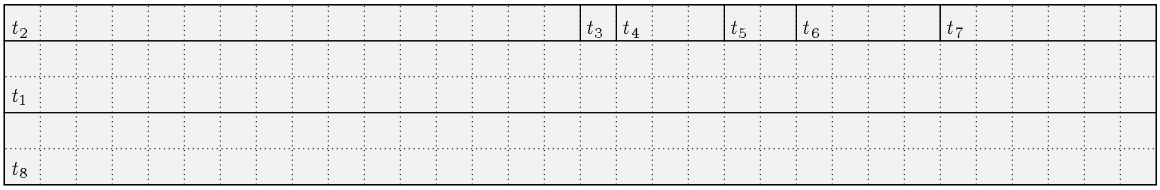

Figure 4: The final arrangement of the tasks for an instance of PARTITION with $A = \{1, 3, 2, 4, 6\}$.

From the initial and final arrangements, it can be seen that the intersection set of t_1 is $I(t_1) = T - \{t_1, t_{n+3}\}$, of t_{n+3} is $I(t_{n+3}) = \{t_1\}$, and of all other tasks is empty. The tasks $t_{i+2}, 1 \leq i \leq n$, their intersection sets, and the intersection set of task t_1 can be constructed in a linear scan of the set A. The magnitude of S can be established at the same time, whereupon, the tasks t_1, t_2, and t_{n+3} can be constructed in constant time. The initial and final arrangement of tasks need not be computed since the intersection information is conveyed by the intersection sets. The transformation from an instance of PARTITION to an instance of FPGA REARRANGEMENT SCHEDULING can thus be done in linear time.

As t_{n+3} is placed first of all, t_1 is removed from the array at time $p(t_{n+3}) = 0$. If t_1 is placed next, then the remaining tasks are removed from the array at time $p(t_1) = p(t_{n+3}) + s(t_{n+3})$, and can only be placed after t_1 has been placed. In order to minimize the delay to the remaining tasks, they are best placed in non–decreasing order of size. But this ordering places t_2 last of all, and delays it for a total of $5S$ (the size of tasks t_1 and t_3 through t_{n+2}). If this delay is to be reduced, some tasks from $T - \{t_1, t_{n+3}\}$ must therefore be placed before t_1 is placed. However, tasks totaling no more than $\lfloor S/2 \rfloor$ in size can be placed after t_{n+3} and before t_1 if the delay on t_1 is not to exceed D.

Since $s(t_2) = S$, it must be placed after t_1, and placing t_2 before any other tasks that are also chosen to be placed after t_1 forces those tasks to be delayed by more than $5S$ since they would be delayed by the time taken to place tasks t_1 and t_2 at least. Any correct schedule must therefore place t_2 last of all. Placing tasks with a total size of less than $\lfloor S/2 \rfloor$ after t_{n+3} and before t_1 leaves tasks with a total size of more than $\lfloor S/2 \rfloor$ to be removed from the array with t_2 when t_1 is placed, thereby delaying task t_2 by more than $4S + \lfloor S/2 \rfloor$.

Therefore, if the tasks t_3 through t_{n+2}, can be partitioned into a pair of disjoint sets of size $\lfloor S/2 \rfloor$ in total, then a schedule satisfying the bound D can be found. Since there is a one–to–one correspondence between these tasks and the elements of A in the given PARTITION instance, it can be seen that if and only if a partition of the set exists, a schedule meeting the bound can be found.

Similarly, if the bound of the FPGA REARRANGEMENT SCHEDULING problem corresponding to an instance of the PARTITION problem can be met, then one of the possible partitions is given by the elements corresponding to the tasks placed respectively between tasks t_{n+3} and t_1, and between t_1 and t_2 in the schedule. If a valid partitioning of the set A does not exist, then no schedule can meet the bound. ∎

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.*

Proof: The construction of the proof of Theorem 1 orderly rearranges the tasks, so it is clear the proof holds in two dimensions.

The proof is easily adapted to one dimension by converting the two–dimensional arrangements of Figures 3 and 4 to arrangements in one dimension. The conversion maps the cell of each array in row major order from the bottom up to the linear array from left to right. ∎

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

Proof: Consider an arrangement in which the tasks representing the elements of the PARTITION set completely fill a single row, and an additional task, which also fills a single row, is to be rearranged such that the two rows containing the tasks are to be interchanged. In this case there is a cyclic dependency between the single task, and the tasks of the PARTITION set, and delays to tasks are minimized by moving tasks totaling half the area of the PARTITION set before placing the singleton. ∎

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 of these 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. From the specification of the problem, the waiting task, t_{n+1}, is chosen to be placed at the root, c_0. The initially executing tasks are then chosen to be reconfigured in the sequence $c_1, c_2, \ldots, c_n$. The state of the search at any node, c_i, can be deduced from the unique path, $c_0, c_1, c_2, \ldots, c_i$, taken from the root to c_i. The sizes of the tasks determine the times at which a choice can be carried out, and thus the time at which tasks are suspended as they become intersected. It is therefore also possible to determine which tasks have not yet been suspended or relocated, and by how much the placed tasks have been delayed. 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. For the search for a solution path to be efficient, as little as possible of the tree is expanded. Searching for a cost–minimal path blindly in a breadth–first or depth–first manner is impractical because there are $n - i$ possibilities for the next sequencing choice at node c_i — one for each task remaining to be placed into its final position. However, the search can be made more efficient through the use of heuristic information to guide the choice of node to expand next. The idea is to expand the node that seems most promising. Such a search is called an *ordered search* or *best–first search* [3]. One way of judging the promise of a node is to estimate the cost of a solution path which includes the node being evaluated. This estimate, made by an evaluation function, is based on the current state, and knowledge about the problem domain. How well the evaluation function discriminates between promising and unpromising nodes determines the effectiveness of the search.

3.2.3 Optimal heuristic search — the A* algorithm

A well–known optimal ordered search algorithm applicable to finding minimal–cost paths in directed acyclic graphs is the A* algorithm [3]. Its distinctive feature is its definition of the evaluation function, f^*. In a tree, the evaluation function, $f^*(c_i)$, estimates the minimal cost of a path from the root to a leaf passing through node c_i by summing the exact cost of reaching the node from the root, $g(c_i)$, and an estimate, $h^*(c_i)$, of the minimal cost of reaching a leaf from c_i. It can be shown that if h^* is a nonnegative under–estimator of the minimal cost of reaching a leaf from the node being evaluated, and all arc costs are positive, then A* is guaranteed to find a solution path. Although $h^*(c_i)$ is required to be a lower bound on $h(c_i)$, the actual cost of reaching a leaf from c_i, the more nearly h^* approximates h the better the algorithm performs. Algorithm A_1 is said to be more informed than algorithm A_2 if, whenever a node, $c_i, 0 \leq i < n$, is evaluated, $h_1^*(c_i) > h_2^*(c_i)$. If algorithm A* is more informed than algorithm A, then A* never expands a node that is

also expanded by A. It is in this sense that A* is considered optimal. A description of an algorithm which finds a solution path and is based on the A* algorithm follows.

Procedure **Exact FPGA Rearrangement Scheduling (EFRS)**

Input A list of n tasks to be rearranged, and a description of the waiting task. For each task, its size and intersection set is given.

Output A sequence in which the tasks ought to be rearranged so as to minimize the maximum execution delay to tasks.

Note The algorithm is based on the A* heuristic search algorithm.

begin

1. Create an open state with the waiting task on the list of reconfigured tasks. Place the tasks intersected by the waiting task on the list of suspended tasks, and place the remaining tasks on the list of executing tasks.
2. Calculate f^* for this state and place the state on the list of open states.
3. Clear a flag for indicating that a solution path has been found.
4. While a solution path has not been found:
   (a) Remove that state from the list of open states for which f^* is minimal, and save it as the current state.
   (b) If the list of suspended tasks and the list of executing tasks for the current state are both empty, set the solution path found flag and iterate.
   (c) For each task, t_i, not yet relocated:
      i. Create an open state copy of the current state with task t_i removed from its source list[2] 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. Update f^* for the open state and place it on the list of open states.
   (d) Discard the current state.
5. 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, $g(c_i)$, 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: in calculating $h^*(c_i)$, 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. This

[2] If t_i is suspended, its source list is the list of suspended tasks, otherwise it is the list of executing tasks.

approach is examined in more detail below. With this estimate in hand, the minimum cost of a path to a leaf through c_i is then given by $f^*(c_i) = \max\{g(c_i), h^*(c_i)\}$.

Ignoring the list of tasks that are yet to be suspended or relocated allows us to optimally schedule the suspended tasks in polynomial time. We prove this fact next by proving a slightly stronger result. Since the suspended tasks are scheduled such that the maximum delay to them is minimized, $h^*(c_i)$ is guaranteed to be an underestimate of the actual delays to them when tasks that remain to be moved are considered as well.

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.*

Proof: Consider the expressions for the delay to suspended tasks t_i and t_j assuming $r(t_i) + s(t_i) \leq r(t_j) + s(t_j)$, and neither causes additional tasks to be suspended. If at time τ, t_i commences reconfiguration before t_j, then t_i is delayed for $\tau - r(t_i)$ time units, and t_j is delayed for at least $(\tau + s(t_i)) - r(t_j)$ time units. On the other hand, were t_j scheduled first, it would be delayed for $\tau - r(t_j)$ time units, and t_i would be delayed for at least $(\tau + s(t_j)) - r(t_i)$ time units. Our assumption is that $s(t_i) - r(t_j) \leq s(t_j) - r(t_i)$, hence $\tau + s(t_i) - r(t_j) \leq \tau + s(t_j) - r(t_i)$. Clearly, $\tau - r(t_i) \leq \tau + s(t_j) - r(t_i)$, as well, so the delays to tasks when t_i is scheduled first are no greater than the delay to t_i were t_j scheduled first.

To see that the ordering is sub–optimal if additional tasks are suspended, consider the example in which $r(t_i) = r(t_j) = 1$, and $1 < s(t_i) < s(t_j)$. Let us assume that t_i is reconfigured at $\tau = 2$, and that it causes a task t_k with $s(t_k) \geq s(t_j)$ to be suspended when it is placed, thus $r(t_k) = 2$. The specified ordering schedules t_j to be reconfigured after t_i at time $2 + s(t_i)$, and t_k to be reconfigured at $2 + s(t_i) + s(t_j)$, giving delays of $1, 1 + s(t_i)$, and $s(t_i) + s(t_j)$ respectively for t_i, t_j, and t_k. The maximum delay can, however, be reduced by scheduling t_j to be placed first of all at $\tau = 2$. Task t_i is then reconfigured at $2 + s(t_j)$, and t_k is only suspended at $2 + s(t_j)$ and reconfigured at time $2 + s(t_j) + s(t_i)$. The delays to t_i, t_j, and t_k are then $1 + s(t_j), 1$, and $s(t_i)$ respectively, which are all less than $s(t_i) + s(t_j)$. ∎

3.2.4 Local versus global choice of the most promising node

The running time of the A* algorithm potentially requires exponential time and space because it attempts to make a globally optimal choice of the most promising node at each step. In this section we present a simplification of the algorithm that achieves an acceptable solution most of the time for lower 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. Such a search is known as an *ordered depth–first search* [3]. The algorithm is as follows:

Procedure **Approximate FPGA Rearrangement Scheduling (AFRS)**

Input The list of n tasks to be rearranged, and the waiting task. For each task, its size and intersection set is given.

Output An approximately minimal–cost sequence for rearranging the tasks.

Note The algorithm is an ordered depth–first heuristic search.

begin

1. Create a current state with the waiting task on the list of reconfigured tasks. Place the tasks intersected by the waiting task on the list of suspended tasks, and place the remaining tasks on the 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

If the evaluation function, f^*, of the exact algorithm is used by algorithm AFRS as well, 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. Step 1 of AFRS then requires $O(n \log n)$ time in the worst case, since $O(n)$ tasks may be intersected by the waiting task. Step 2(b)i will also require $O(n \log n)$ time, and Step 2(b)ii requires a scan of the suspended task list in $O(n)$ time. The running time of algorithm AFRS is therefore $O(n^2 \log n)$.

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 therefore examined the performance of AFRS with one– and two–state lookahead in Section 4. The drawback with looking two states ahead in Step 2 is that it adds another factor of n 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

In this section we first investigate the performance of the approximate FPGA rearrangement scheduling algorithm and then compare the performance of the local repacking allocation heuristic with that of ordered compaction.

4.1 Approximate FPGA rearrangement scheduling

Tests were conducted to compare the performance of the approximate FPGA rearrangement scheduling (AFRS) algorithm with that of the exact FPGA rearrangement scheduling (EFRS) algorithm. The comparison focused upon the speed of the algorithms (number of states expanded to reach a solution) and the quality of the solution (the difference between solutions on identical problem instances). These tests were carried out using solution path cost estimators, f^*, with a lookahead of one and two states respectively.

The algorithms were implemented with two optimizations to reduce the number of expanded states: all executing tasks with empty intersection sets were relocated when the suspended task list was empty, and the suspended tasks were relocated according to Lemma 1 when no executing tasks remained to be moved.

A test consisted of submitting a randomly generated problem instance to each of the algorithms and examining the magnitude of the maximum delay to a task for the schedule produced by the algorithm, as well as the number of states expanded to obtain the solution. While the exact algorithm produces an optimal solution, it may require exponential space and time to find it. The operation of the exact algorithm was therefore aborted when the open state list exceeded 4MB. Note that the number of states examined by the cost estimator were not counted in the assessment.

The parameters governing a randomly generated problem instance were: the number of tasks in the test; the maximum size for the independently chosen, uniformly distributed side lengths of a task; and the likelihood of extending the intersection set for a task, t_i, beyond length l, given by $P[|I(t_i)| \geq l] = x^l$ with radix, $0 < x < 1$.

4.1.1 One–state lookahead

The performance of EFRS and AFRS was measured on 23,040 randomly generated problem instances. The number of tasks in a problem instance and the maximum task side lengths ranged from 5 to 20, and the radix for extending the intersection set ranged from 0.1 to 0.9. For each parameter combination, 10 tests were performed.

The significant findings were:

Relative Performance Achieved	Number of Instances	Percentage of Total
1.0	17,200	81.0
1.1	18,225	85.6
1.2	19,021	89.6
1.3	19,566	92.2
1.4	19,976	94.1
1.5	20,286	95.6
2.0	21,000	98.9
3.0	21,185	99.8
4.0	21,207	99.9
5.0	21,219	99.9
10.0	21,230	100.0
10.4	21,231	100.0

Table 1: Relative performance of algorithm AFRS with a lookahead of one state.

- The exact algorithm failed to find a solution using less than 4MB of working storage in 1,809 (7.9%) of 23,040 tests.
- The performance of the approximate relative to the exact algorithm is summarized in Table 1. The relative performance is the maximum execution delay of the approximate solution divided by the maximum execution delay of an optimal solution. The table lists the number of instances achieving the relative performance specified, or better.
- Of 23,040 tests, the exact method expanded less states than the approximate method in 17,053 cases (74.0%).

4.1.2 Two–state lookahead

The performance of the exact and approximate algorithms using an estimate for the cost of compaction based on a lookahead of one state (EFRS–1 and AFRS–1 respectively) was compared with algorithms using a lookahead of two states (EFRS–2 and AFRS–2 respectively).

Each algorithm was tested on a set of 2,560 randomly generated FPGA compaction instances. The parameters used to generate the problem set were selected to be able to compare the performance when there was a good chance of having non–trivial intersection patterns. Between 11 and 14 tasks were generated, task side lengths ranged up to a maximum of between 5 and 20, and the radix for extending the intersection set ranged from 0.5 to 0.8.

Our findings were:

- Of 2,560 tests, in 1,489 cases (58.1%) EFRS–2 required less states than EFRS–1 to minimize the maximum compaction delay. The mean reduction in the number of states required was 32.8%, with a standard deviation of 0.3%.

Relative Performance Achieved	Number of AFRS–1 Instances	Percentage of AFRS–1 Instances	Number of AFRS–2 Instances	Percentage of AFRS–2 Instances
1.0	1,307	53.2	1,551	62.2
1.1	1,573	64.0	1,881	75.4
1.2	1,797	73.1	2,099	84.1
1.3	2,017	82.1	2,262	90.7
1.4	2,150	87.5	2,359	94.6
1.5	2,245	91.3	2,420	97.0
2.0	2,408	97.8	2,486	99.6
2.5	2,445	99.5	2,495	100.0
3.0	2,455	99.9		
4.0	2,457	100.0		
5.0	2,458	100.0		

Table 2: Relative performance of algorithm AFRS with lookahead of one and two states.

- Of 2,560 tests, in 73 cases (2.9%) EFRS–2 required more states than EFRS–1 to minimize the maximum compaction delay. The mean increase in the number of states required was 7.2% with a standard deviation of 0.5%.
- Of 2,560 tests, 65 (2.5%) were unsolved by EFRS–2 with less than 4MB of open state information. EFRS–1 was unable to solve 37 additional cases (4.0% in total) within 4MB of working storage.
- Of 2,458 tests solved by EFRS–1 and EFRS–2, in 692 cases (28.2%) the relative performance of AFRS–2 was less (better) than that of AFRS–1. The relative performance is the ratio of the maximum delay to a task obtained by the approximate method compared with the maximum delay found by the optimal algorithm. The mean reduction in relative performance from AFRS–1 to AFRS–2 was 17.2% with a standard deviation of 0.5%.
- Of 2,458 tests solved by EFRS–1 and EFRS–2, in 126 cases (5.1%) the relative performance of AFRS–2 was greater (worse) than that of AFRS–1. The mean increase in relative performance from AFRS–1 to AFRS–2 was 15.5% with a standard deviation of 1.8%.
- Table 2 summarizes the relative performance of AFRS–1 and AFRS–2. For each relative performance figure, the table lists the number of instances which achieved the figure or better.
- Of 2,560 tests, EFRS–2 expanded less states than AFRS–2 in 1,404 cases (54.8%) whereas EFRS–1 expanded less states than AFRS–1 in 1,306 cases (51.0%).

4.1.3 Discussion

We draw the following conclusions from the results. Since the approximate algorithm achieved better than twice the maximum delay of an optimal solution in more than 95% of

trials, it appeared to perform well on small instances. A lookahead of two states is desirable because it affects the quality of the solution significantly by reducing the gap between the exact and approximate solutions. The speed of the exact method suggests a practical approach to frequently obtaining a good solution quickly: expand up to $n(n-1)$ states using the exact algorithm, and if a solution has not yet been found, use the approximate method.

4.2 Evaluation of local repacking

In this section we report on simulation experiments conducted to gauge the performance of FPGA task rearrangement strategies. First the operation of the simulator and its components are outlined. Then we describe the experiments, and briefly summarize our findings. After reporting on the results in detail, we conclude with some final remarks about the experiments and suggestions for further improvement.

4.2.1 Overview of simulator

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 [17] task allocator with task allocators using ordered compaction [8] 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. Time–stamping the significant events in a task's life cycle allows various performance metrics to be calculated for the task set as a whole. We go on to describe the simulator in more detail.

Each simulation run commences with the generation of 10,000 independent tasks that may be characterized by 4 independent uniformly distributed random variables. A task is a rotatable rectangular request for a sub–array of cells with independently chosen side lengths $x \times y = \mathrm{U}_1(1, L) \times \mathrm{U}_2(1, L)$, where L is a specified maximum side length. The task executes for a service period of $\mathrm{U}_3(1, 1000)$ time units. The period of time between task arrivals, the inter–task arrival period, is $\mathrm{U}_4(1, P)$ time units, where P is a specified maximum. Task sets are generated using uniformly distributed random variables because expected workload characteristics are not known, and it is usually easy to draw conclusions from the results. Since task sets are generated, all measurements of time are scaled to a common time unit. The results are independent of the magnitude of a time unit.

The simulated FPGA has its size fixed at 64×64 cells, and the configuration delay per cell, cd, may be varied. Tasks are loaded or reconfigured via a single I/O port. The time to load or reload task is therefore $x \times y \times$ cd time units, and the mean configuration delay is given by the product of the mean task size and the configuration delay per cell.

Three allocation methods were compared. They were:

First fit allocates the waiting task to the bottom–leftmost block of free cells large enough to support the task. Both orientations of the waiting task are tried as soon as the

previous task has loaded, and after each deallocation. First fit is a good standard for comparison because it has complete recognition capability (it always finds an allocation site when one exists).

Ordered compaction attempts first fit and then allocating the waiting task by orderly compacting a subset of the executing tasks. Ordered compaction is attempted in each of the compass directions with both orientations of the waiting task as soon as the previous task has loaded and after each deallocation. Tasks are delayed for the minimum amount of time needed to move them, and the waiting task is loaded last of all.

Local repacking attempts first fit and then allocating the waiting task by locally repacking a subset of the executing tasks. Local repacking is tried with both orientations of the waiting task and both orientations of the sub–array being repacked as soon as the previous task has loaded and previous task movements have completed, and then after each deallocation. Sleator's algorithm was used to attempt a repacking of all the tasks partially intersected by the sub–array. A two–state lookahead cost estimator to perform the scheduling was used.

Note that instead of searching for the minimum cost allocation site for ordered compaction and local repacking, the simulator allocated at the first feasible site found. It also did not abort compaction or repacking when the waiting task could have been allocated more quickly using the first fit method following subsequent deallocations. The simulations do not account for the time required to find a bottom–left allocation site or a rearrangement of the tasks.

The simulator recorded the time a task arrived, the time the allocation for a task commenced (when the allocation algorithm first began looking for an allocation site, which for first fit and ordered compaction was the time the previously loaded task completed loading, and for local repacking was the time the previously loaded task completed loading and the rearrangements following the previous task load completed), the time the task commenced loading, and the time the task finished processing (accounting for delays due to task movements). From these records the following performance measures were computed:

Mean allocation delay: the mean over all tasks of the time between the allocation for a task commencing and the loading of the task commencing.

Mean queue delay: the mean over all tasks of the time between a task arriving and the allocation of the task commencing.

Mean response time: the mean over all tasks of the time between a task arriving and the task finishing processing.

Utilization: the mean amount of time an FPGA cell spent executing tasks as a percentage of the time needed to finish processing all tasks.

4.2.2 Experiments conducted and summary of results

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 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. The experiments were designed to investigate the following effects.

1. The effect on performance of varying load with nominal configuration delay.

   Performance was measured for a configuration delay of 1/1000 time unit per cell as the maximum inter–task arrival period, P, was varied from 20 to 1000 time units. The maximum side length, L, was fixed at 32 cells.

   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.

2. The effect on performance of varying the configuration delay at different system loads.

   Performance was measured at maximum inter–task arrival periods of 40 and 120 time units, corresponding to loads for which the FPGA was saturated with work and coming out of saturation respectively. While the maximum side length, L, was fixed at 32 cells, the configuration delay was varied from 1/250 time unit per cell to 2.2 time units per cell, at which the mean configuration delay per task exceeded the mean service period.

   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 sustained mean configuration delays approximately 10% of the service period in magnitude before performing worse than first fit.

3. The dependency of performance upon task size at saturation with nominal configuration delay.

   Performance was measured at a maximum inter–task arrival period of $P = 1$ time unit, and a configuration delay of 1/1000 time unit per cell, as the maximum task side length, L, was varied from 8 to 64 cells.

   An improvement in performance was observed to increase from $L = 8$ up to $L = 32$, beyond which the improvement in performance decreased. For $L < 32$ local repacking performed better than ordered compaction, but for $L > 32$, it performed worse.

Detailed results of the experiments follow.

4.2.3 Effect of system load on allocation performance

Figures 5(a) through 5(d) plot the performance of the three allocation methods as the system load was varied by altering the inter–task arrival period.

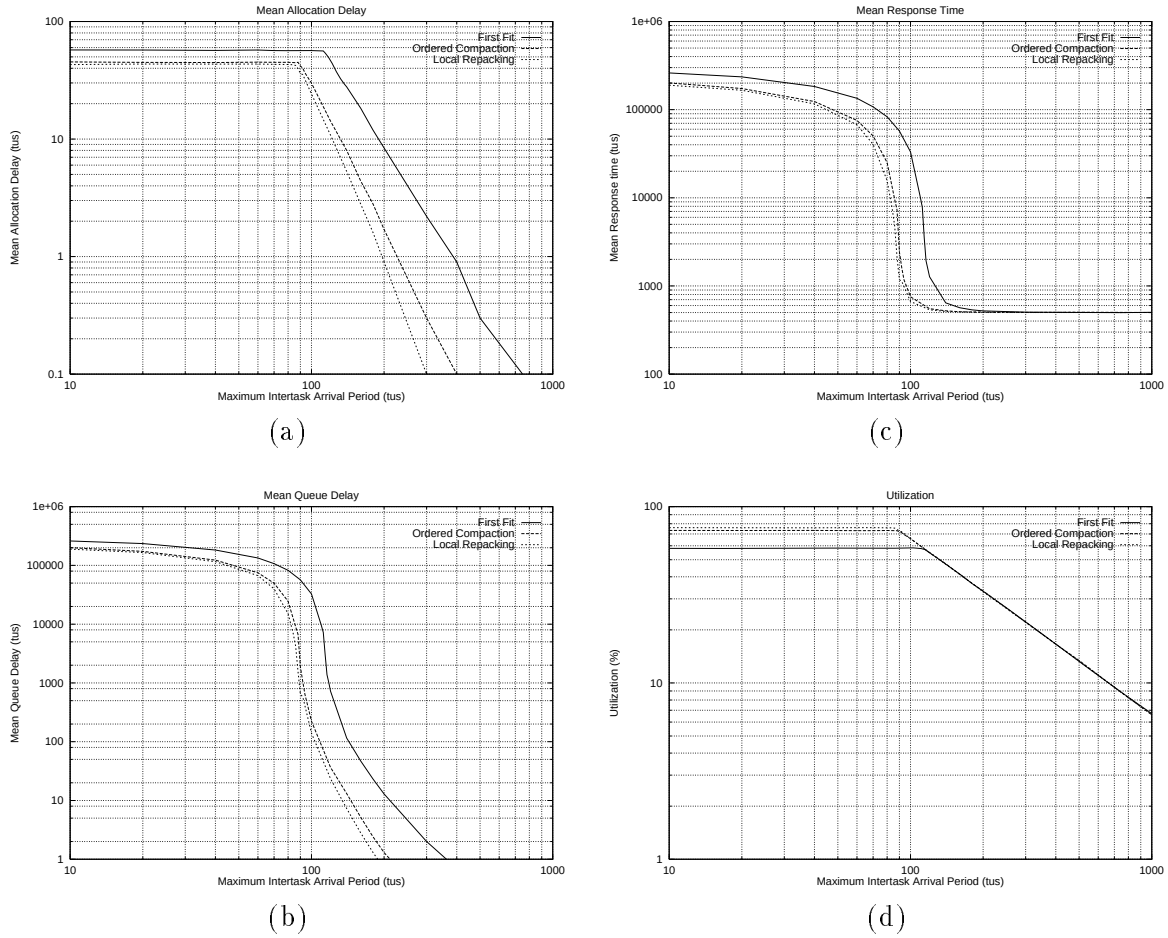

Figure 5: (a) Mean allocation delay, (b) mean queue delay, (c) mean response time, and (d) utilization for first fit, ordered compaction, and local repacking as the maximum inter–task arrival period was varied. 10,000 tasks of size U(1,32) $\times$ U(1,32), service period U(1,1000) time units (tus), arriving at uniformly distributed time intervals were allocated to a simulated 64×64 cell FPGA with a configuration delay of 1/1000 tu/cell.

At small inter–task arrival periods, tasks arrive far more quickly than they can be processed by the FPGA. The FPGA consequently saturates with work, meaning tasks need to wait before they can be allocated. Although the mean allocation delay at saturation depends upon the size of the chip, the mean task size and service period, the ability of the allocation method to find allocation sites, and the time needed to load tasks, it is independent of inter–task arrival period. In our simulations, the mean allocation delays at saturation for the first fit, ordered compaction, and local repacking allocation methods were 57.2, 44.9 and 43.5 time units respectively. Local repacking was almost 24% quicker at allocating a task than first fit, and over 3% quicker than ordered compaction

As the mean inter–task arrival period (half the maximum inter–task arrival period) increases beyond the mean allocation delay at saturation, tasks begin to arrive less frequently than

they can be accommodated on the FPGA, and the chip quickly comes out of saturation. The mean allocation delay dropped below 1 time unit at maximum inter–task arrival periods less than 400, 300 and 200 time units respectively for first fit, ordered compaction, and local repacking. Since Figure 5(a) is a log–log plot, it hides the fact that mean allocation delay dropped to zero at maximum inter–task arrival periods of 500 tus for ordered compaction and local repacking, and 1000 tus for first fit.

In the saturated region, the queue delay to the kth task is approximately k times the difference between the mean allocation delay at saturation and the mean inter–task arrival period. Thus the mean queue delay is given by $(\bar{a} - \bar{p}) \times (n+1)/2$ where $\bar{a}$ is the mean allocation delay, $\bar{p}$ is the mean inter–task arrival period, and n is the number of tasks. Since the allocation delay drops to zero as the FPGA comes out of saturation, so too does the mean queue delay.

The mean response time is given by the sum of the mean queue delay, the mean allocation delay, the mean time needed to load the task, the mean service period, and the mean time needed by ordered compaction and local repacking to move tasks. At low configuration delays per cell (1/1000 time unit), the last component is negligible.

Utilization is the ratio of cell usage to cell capacity. It is given by the formula

$$\text{utilization} = 100 \times \frac{\sum_{\mathrm{i=1}}^{\mathrm{n}} \mathrm{e_i} \times \mathrm{s_i}}{\text{FPGA size} \times \max\{\mathrm{f_i} : 1 \leq \mathrm{i} \leq \mathrm{n}\}}$$

where n is the number of tasks processed, e_i is the execution time (service period), s_i is the size, and f_i is the completion time of the ith task. In the saturated region, an estimate for the completion time is given by the mean allocation delay multiplied by the number of tasks, and thus the utilization is constant, and can be approximated by multiplying the mean task size by the mean service period, and dividing the result by the the FPGA size and the mean allocation delay. This model predicts values approximately 5% lower than the utilization of 58.0%, 73.2%, and 75.9% observed respectively for first fit, ordered compaction, and local repacking. We cannot explain the gap without further experimentation to check the completion time for a task set.

When tasks arrive less quickly on average than they can be accommodated, the completion time of the last task to finish depends upon its arrival time. As the FPGA comes out of saturation, we therefore observe a drop in utilization proportional to the rise in the inter–task arrival period.

4.2.4 Effect of configuration delay on allocation performance

Figures 6(a) through 6(d) illustrate the performance of the allocation methods as the configuration delay per cell of the FPGA was increased from 1/250 time unit to 2.2 time units. This range corresponds approximately to a mean configuration delay per task of between 1 and 600 time units.

At mean configuration delays below 1% of the mean service period, the performance benefits of local repacking and ordered compaction are similar to those observed at saturation with negligible configuration delay. However, local repacking begins to perform worse than

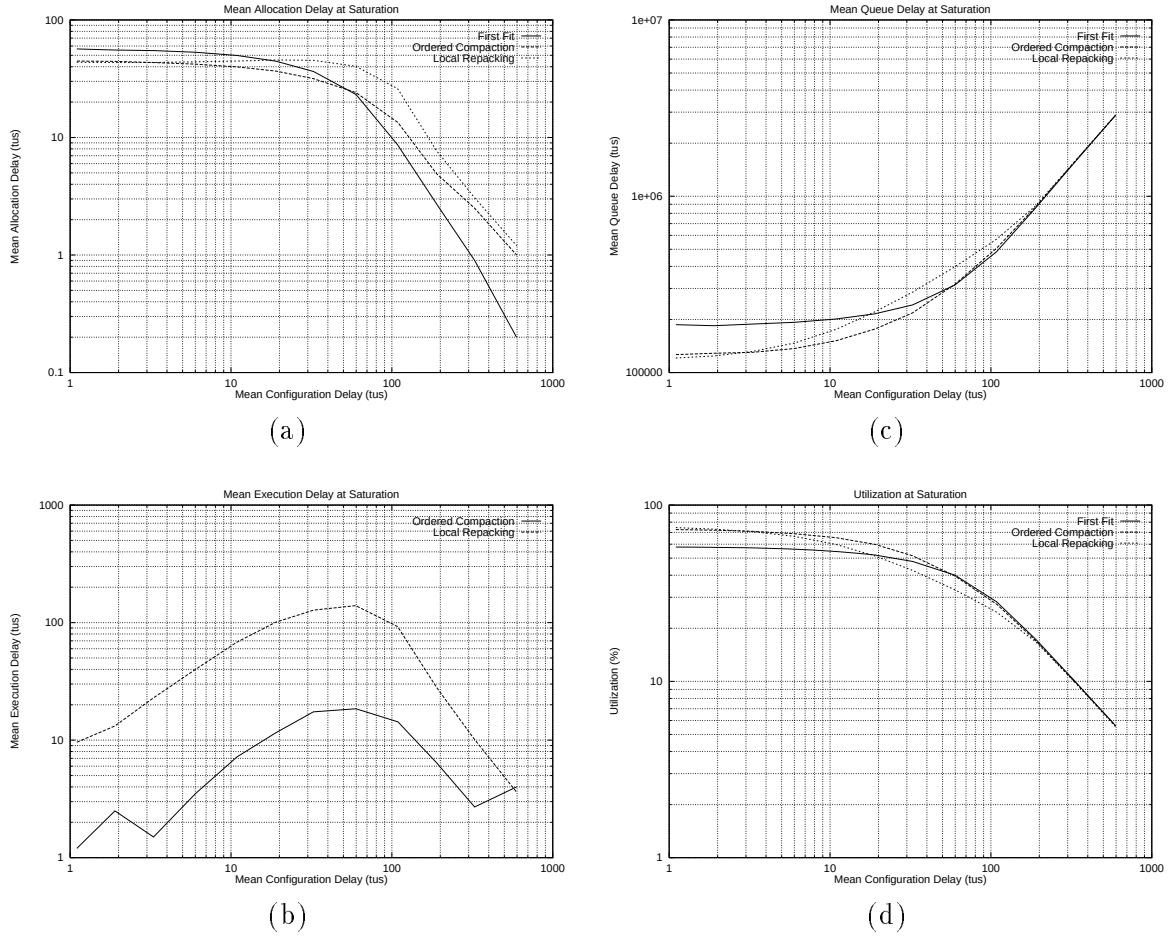

Figure 6: (a) Mean allocation delay, (b) mean execution delay, (c) mean queue delay, and (d) utilization at saturation for first fit, ordered compaction, and local repacking as the mean configuration delay per task was varied. 10,000 tasks of size U(1,32) $\times$ U(1,32), service period U(1,1000) time units (tus), arriving at intervals of U(1,40) tus were allocated to a simulated FPGA of size 64×64.

ordered compaction at very low configuration delays. Two factors contribute to this reversal. First, the delays to executing tasks are larger for local repacking than for ordered compaction since the latter can be scheduled so as to suspend each task no more than is needed to move it. Second, whereas few waiting tasks benefit from orderly compacting all of the tasks executing on the FPGA, local repacking often rearranges all of them to allocate the waiting task. With local repacking the additional delay to executing tasks worsens as the configuration delay increases, and because more cells need to be reconfigured, the next waiting task is delayed longer while it waits for the rearrangement following the previous allocation to complete.

As the configuration delay rises, the mean time to load a task increases, and the allocation delay for first fit falls since it becomes more likely that suitable free blocks will have been deallocated before allocation of the next task is attempted. The increased likelihood of finding an allocation site without the need to rearrange executing tasks also reduces the mean allocation delay and mean execution delay for ordered compaction and local repacking.

The mean execution delay curve was obtained by subtracting the mean queue, allocation, and configuration delay, as well as the mean service period from the mean response time. While the shape of the curve is explicable, we cannot explain the magnitude of the maximum, nor why it should occur at a mean configuration delay of 60 time units. At mean configuration delays below 100 tus, the plot suggests that each task is relocated multiple times.

The allocation performance was also examined at operating loads where the FPGA is neither saturated nor unsaturated with work. The results appear plotted in Figures 7(a) through (d).

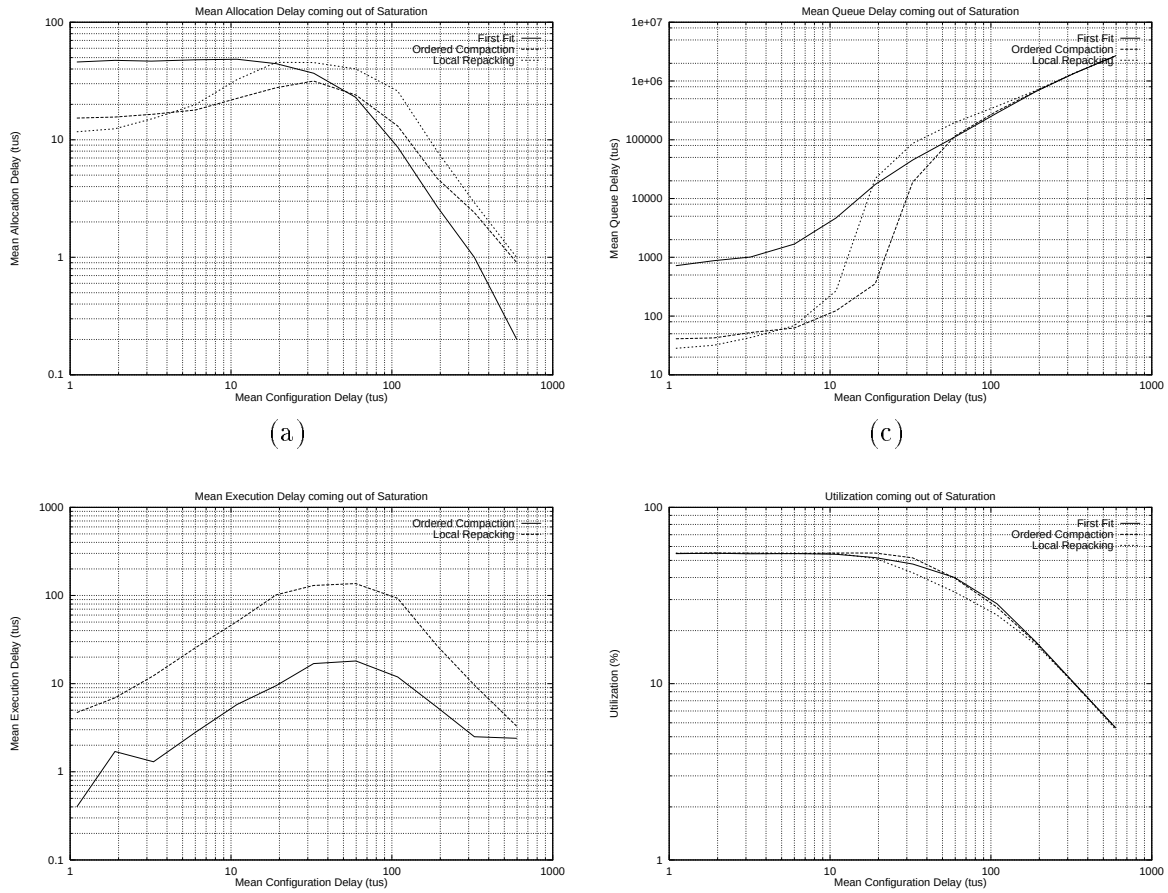

Figure 7: (a) Mean allocation delay, (b) mean execution delay, (c) mean queue delay, and (d) utilization coming out of saturation for first fit, ordered compaction, and local repacking as the mean configuration delay per task was varied. 10,000 tasks of size U(1,32) $\times$ U(1,32), service period U(1,1000) time units(tus), arriving at intervals of U(1,120) tus were allocated to a simulated FPGA of size 64×64.

Despite the much larger performance gap between local repacking and first fit, its usefulness seems limited to mean configuration delays of less than 5% of the mean service period. Ordered compaction performed best of all when the mean configuration delay was between 1% and 10% of the mean service period.

It is interesting to note that the mean execution delay curves for the saturated system and the system coming out of saturation are almost identical. We have been unable to explain the similarity.

4.2.5 Effect of task size on allocation performance

Figure 8 plots the mean allocation delay as the maximum task size is increased to fill the FPGA. The FPGA is saturated by allowing the tasks to arrive at 1 time unit intervals.

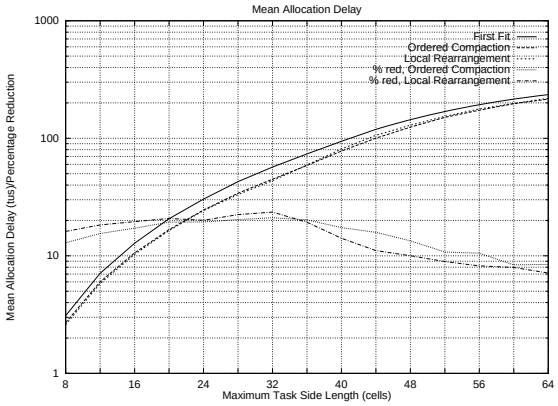

Figure 8: Mean allocation delay for first fit, ordered compaction, and local repacking as the maximum task size was varied. 10,000 tasks with uniformly distributed side lengths, service period U(1,1000) time units(tus), arriving at intervals of 1 tu were allocated to a simulated FPGA of size 64×64.

The chart indicates that tasks are better allocated using local repacking when the longest task side is less than half the array side length. However, they are more easily allocated by ordered compaction when larger tasks occur. We guess that free fragments trapped between small tasks are easily gathered by repacking, but cannot necessarily be coalesced by ordered compaction. We also suspect that Sleator's method frequently fails to repack the tasks into the array when large tasks occur because when the tasks wider than half the array are stacked on top of one another large regions are left unused and those tasks that occupied these regions before the repacking cannot be squeezed in between the top of the stack and the top of the array.

4.2.6 Discussion of the experimental results

We have attempted to evaluate and compare the known approaches to FPGA task rearrangement as completely as possible primarily because the target applications and their associated task mix is not known at this time. It could be argued that this uncertainty is sufficient to question the validity of our method — is it to be expected that huge workloads queue up for an FPGA to process, or does work arrive in bursts? How should the service period of tasks be modeled?

Nevertheless, now that we have some feeling for the performance of local repacking relative to ordered compaction, we wonder whether improvements are possible to the local repacking method. Points to be considered include handling partially intersected tasks differently,

using different packing algorithms, modifying the scheduling goals and algorithm, searching for the best rearrangements, aborting the rearrangement when the waiting tasks would be allocated sooner by waiting for further deallocations, and reducing the number of times a task is moved. Some of these measures would improve the performance of local repacking relative to that of ordered compaction, while the last three would improve the performance of both methods. Attempting to slide partially intersected tasks out of the sub–array by ordered compaction would not perform any worse assuming the scheduling costs, which we have ignored in our study, can be borne. Similarly, trying other packing algorithms would only cost scheduling time. Improvements would flow from switching the order in which scheduling goals are applied. If we were to attempt to minimize the delays to executing tasks first of all, and then insert the waiting task into the schedule as early as possible without increasing the maximum delay to the executing tasks, the mean execution delay to tasks could only decrease. With tasks spending less time on the array, waiting tasks would be allocated sooner.

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, we have proposed partially rearranging the array. In this paper we described and assessed the local repacking solution to rearranging a subset of the executing tasks. This method was motivated by the desire to find a more effective alternative to the ordered compaction approach to partial FPGA rearrangement, which 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 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 finding out whether the running time of our algorithms could be improved. Specifically, can the free area tree be maintained dynamically, as tasks are deallocated/allocated/repacked? How can the free area tree be searched efficiently? Can a faster scheduling heuristic be found? We are interested in improving the performance of our algorithms. Two–dimensional bin packing algorithms are irrotational — can they be improved by allowing task rotations? What is the performance bound on our scheduling heuristic? Can a better scheduling heuristic be found? Better allocation performance may be possible if a different approach to handling partially intersected tasks and different packing algorithms were tried, if the scheduling algorithm tried to minimize the delays to moving tasks before attempting to allocate the waiting task as early as possible, if least cost rearrangements were performed, and if rearrangement were aborted when allocation could occur earlier following task deallocations.

There is also ample scope for further work. To compare methods more effectively, future simulations should take the time needed by the allocator to compute rearrangements into account. The effectiveness of the methods would be enhanced if means of reducing the number of movements individual tasks are subjected to could be found, and if more targeted arbitrary rearrangements that move a small number of 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.

Acknowledgments

We thank Mathew Wojko, Carsten Steckel, and Jens Utech for interesting discussions about this work.

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

References

[1] B. S. Baker, D. J. Brown, and H. P. Katseff. A 5/4 algorithm for two–dimensional packing. *Journal of Algorithms*, 2:348–368, 1981.

[2] B. S. Baker, E. G. Coffman Jr, and R. L. Rivest. Orthogonal packings in two dimensions. *SIAM Journal on Computing*, 4(9):846–855, Nov. 1980.

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

[4] 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, Wien, 1984.

[5] E. G. Coffman Jr, M. R. Garey, D. S. Johnson, and R. E. Tarjan. Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM Journal on Computing*, 9(4):808–826, Nov. 1980.

[6] E. G. Coffman Jr and P. W. Shor. Packings in two dimensions: Asymptotic average–case analysis of algorithms. *Algorithmica*, 9:253–277, 1993.

[7] 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`.

[8] 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, 1997. Springer–Verlag.

[9] C. R. Dyer. The space efficiency of quadtrees. *Computer Graphics and Image Processing*, 19(4):335–348, Aug. 1982.

[10] M. R. Garey and D. S. Johnson. *Computers and Intractability — A Guide to the Theory of NP–Completeness.* W.H.Freeman, San Fransisco, California, 1979.

[11] J. R. Hauser and J. Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In *Fifth Annual IEEE Symposium on Field–Programmable Custom Computing Machines*, pages 24–33, Los Alamitos, CA, Apr. 1997. IEEE Computer Society.

[12] K. Li and K. H. Cheng. 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. Appeared in Proceedings 1st IEEE Symposium on Parallel and Distributed Processing, 1989, pp. 358–365.

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

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

[15] 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.

[16] 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.

[17] Y. Zhu. Efficient processor allocation strategies for mesh–connected parallel computers. *Journal of Parallel and Distributed Computing*, 16(4):328–337, Dec. 1992.

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