International Journal of Foundations of Computer Science

ON DYNAMIC TASK SCHEDULING FOR FPGA–BASED SYSTEMS

OLIVER DIESSEL
School of Computer Science and Engineering, University of New South Wales, Sydney, NSW 2052, Australia.

and

HOSSAM ELGINDY
School of Computer Science and Engineering, University of New South Wales, Sydney, NSW 2052, Australia.

Received
Revised
Communicated by

ABSTRACT

The development of FPGAs that can be programmed to implement custom circuits by modifying memory has inspired researchers to investigate how FPGAs can be used as a computational resource in systems designed for high performance applications. When such FPGA–based systems are composed of arrays of chips or chips that can be partially reconfigured, the programmable array space can be partitioned among several concurrently executing tasks. If partition sizes are adapted to the needs of tasks, then array resources become fragmented as tasks with varying requirements are processed. Tasks may end up waiting despite their being sufficient, albeit fragmented resources available. We examine the problem of repartitioning the system (rearranging a subset of the executing tasks) at run–time in order to allow waiting tasks to enter the system sooner.
In this paper, we introduce the problems of identifying and scheduling feasible task rearrangements when tasks are moved by reloading. It is shown that both problems are NP–complete. We develop two very different heuristic approaches to finding and scheduling suitable rearrangements. The first method, known as Local Repacking, attempts to minimize the size of the subarray needing rearrangement. Candidate subarrays are repacked using known bin packing algorithms. Task movements are scheduled so as to minimize delays to their execution. The second approach, called Ordered Compaction, constrains the movements of tasks in order to efficiently identify and schedule feasible rearrangements. The heuristics are compared by time complexity and resulting system performance on simulated task sets. The results indicate that considerable scheduling advantages are to be gained for acceptable computational effort. However, the benefits may be jeopardized by delays to moving tasks when the average cost of reloading tasks becomes significant relative to task service periods. We indicate directions for future research to mitigate the cost of moving executing tasks.

Keywords: FPGA, partitionable array, dynamic reconfiguration, task scheduling, task rearrangement, dynamic allocation, scheduling complexity, scheduling heuristic

1. Introduction

FPGA–based computer systems are typically composed of workstation class processors that host circuit boards comprising arrays of FPGAs, distributed memory to support high bandwidth I/O, and low latency networks to allow flexible and rapid communications between system components. Such systems have proved effective at performing compute intensive tasks by loading the FPGAs with special purpose logic circuits that exploit the parallelism present in many grand challenge problems and overcome the "von Neumann bottleneck" of sequentially fetching and decoding instructions. The design of FPGA–based systems poses many challenges and attracts considerable attention from researchers interested in system architectures that provide high bandwidth and low latency to computational resources, effective mappings of algorithms to the compute resource, and the run–time management of systems for high performance.

An FPGA consists of a two–dimensional grid of configurable logic cells and a programmable routing network that can be programmed to implement any digital circuit. The logic cells are typically capable of implementing simple boolean functions of their inputs and one or more registers for storing constants or intermediate results. Cells are usually directly connected to their grid neighbours but often also provide some means of obtaining inputs or supplying results over longer wires that may connect several cells or convey signals for longer distances. An FPGA is programmed by configuring the logic functions of the cells and setting the routing switches to implement a digital circuit.

The conventional approach to configuring FPGAs is to load the configuration for the complete chip in an uninterrupted fashion taking time proportional to the area of the chip. This approach does not scale well, and with the development of configuration stores based on static RAM that can be randomly accessed, partially reconfigurable devices have been produced. Partially reconfigurable FPGAs allow part of the FPGA to be reconfigured while circuits occupying the rest of the device continue to operate. The benefits of this development have been to reduce the reconfiguration time to within a factor of the area requiring reconfiguration and to allow reconfiguration to overlap with operation. Such FPGAs are therefore also called dynamically reconfigurable.

Dynamically reconfigurable FPGAs can be used in several ways. If a circuit does not fit onto a chip, it may be possible to partition it into components that can be "paged" onto the device as control passes to them and they become active. Another use is to recycle inactive regions of the FPGA for use by other tasks. It has thus become feasible to configure and operate regions of an FPGA independently of other regions. Interest has therefore grown in sharing FPGA devices and FPGA–based systems among multiple simultaneous tasks. Similar goals for mesh multiprocessor systems led to the notion of a space–shared system which partitions the mesh to share the resource among multiple simultaneously operating tasks.

The space–shared FPGA model allows the FPGA area to be partitioned among multiple tasks that are each allocated a region of the FPGA in which they may execute as if they were the sole task running on an FPGA of the required size.

Efficient use of the FPGA is gained by allocating as much of the resource to running tasks as possible. Rather than waiting until the system is available for their exclusive use, tasks benefit from being able to run as soon as there are sufficient contiguous free resources available to support their processing needs.

When the logic resource of an FPGA is to be shared among multiple tasks, each having its own spatial and temporal requirements, the resource becomes fragmented. If the requirements of tasks and their arrival sequence is known in advance, suitable arrangements of the tasks can be designed and sufficient resource can be provided to process tasks in parallel. However, when placement decisions need to be made on–line, it is possible that a lack of contiguous free resource will prevent tasks from entering although sufficient resource in total is available. Tasks are consequently delayed from completing, and the utilization of the FPGA is reduced because resources that are available are not being used. The system designer may be tempted to provide additional resource, thereby increasing the physical and economic needs of the system.

To maintain system speed, and to contain size and cost, we propose rearranging a subset of the executing tasks when doing so allows a waiting task to be processed sooner. Our goal is to increase the rate at which waiting tasks are allocated while minimizing disruptions to executing tasks that are to be moved.

In the next section, a model of a space–shared FPGA system and its task management is developed. We use this model to formulate the problem and our solution goals in Section 3. In this section it is also shown that the problems of identifying feasible FPGA rearrangements to accommodate a waiting task and the scheduling of task movements so as to minimize execution delays are NP–complete. We then present two heuristics to overcome the complexity of finding suitable rearrangements. Each uses a different scheduling technique. Section 4 describes the local repacking heuristic that attempts to minimize the FPGA area that is to be rearranged to accommodate the waiting task. To schedule the movements, an ordered search heuristic is used. In Section 5 we present a second technique, ordered compaction, which restricts the type of movements permitted so as to efficiently identify potential rearrangements and schedule their movements. Section 6 assesses the time complexity and reports on the performance of the methods. The results are discussed in Section 7 and our conclusions are presented in Section 8.

2. Model & Assumptions

2.1. Space–Shared FPGA Model

The configurable logic cells of common dynamically reconfigurable FPGAs are laid out in a two–dimensional grid and are usually directly connected with their neighbours to the north, south, east, and west via nearest neighbour links [9, 1].

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

It is assumed that an FPGA task and the used routing resources surrounding its perimeter can be modelled as a rectangular subarray of arbitrary yet specified dimensions. Some internal fragmentation therefore results when task designs cannot be optimized to a rectangular shape. The size of a task is assumed fixed for the duration of its execution.

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

Tasks are assumed to be independent. However, when a task is decomposed into several reconfigurable subtasks, they are allocated to the largest bounding box required throughout the task's instantiation. In this way, routing conflicts and interference with other tasks are avoided.

Each task is allocated a subarray of the required size within a larger partitionable array. Usually a subarray 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. Given the orientation of a task, its width* $w(\mathrm{or}(t)) = x$ *and height* $h(\mathrm{or}(t)) = y$ *are known.*

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 $1 \leq y \leq H - l_2 + 1$. If tasks make use of hierarchical routing, then they might not in practice be relocated anywhere. Our FPGA abstraction assumes the routing interface to all cells is identical.

No limit is placed upon the number of tasks that can execute simultaneously. To support multi–tasking, the FPGA should be able to support multiple simultaneous I/O streams. The idealized model assumes any number of I/O streams can be supported without slowdown.

Tasks are assumed to be deadline–free and to have unknown service periods. However, it is possible to check whether or not tasks with known service periods can be rearranged without exceeding deadlines.

Configuration involves loading the lookup table and/or selectors for the multiplexors associated with each cell to select the cell's function. Several bytes of configuration data per cell are serially loaded as a configuration bit–stream via pins on the periphery of the chip. The setting of the routing switches is also determined by the configuration data.

The dynamically reconfigurable FPGA model assumes the I/O architecture permits random access to the configuration memory of a single logic cell or routing switch in a single step. Moreover, it is assumed that a cell or switch can be configured in a constant amount of time.

It is therefore assumed that the time needed to configure a subarray,

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

is proportional to the configuration delay per cell, CD, and the size of the subarray. Since the delay properties of commercially available chips are isotropic and homo-

geneous, *CD* is assumed to be 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.

2.2. Task Management with Partial Rearrangement

The management of tasks is depicted in Fig. 1.

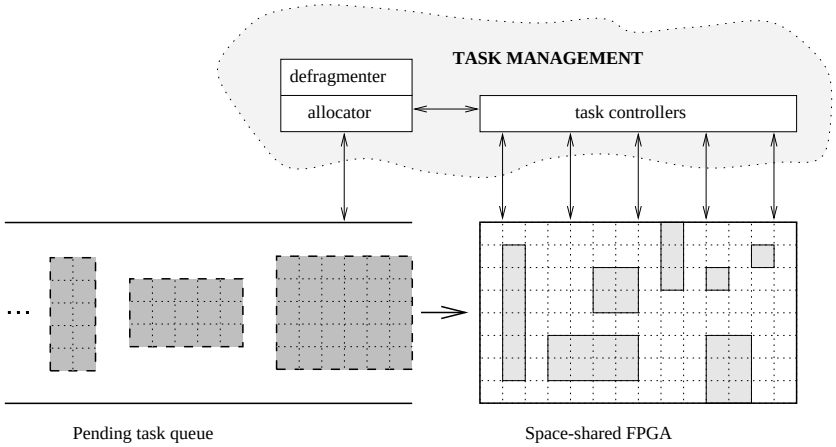


Fig. 1. An overview of task management with partial rearrangement.

Requests for service are queued in arrival order by a sequential host. A task allocator, executing on the host, attempts to find an allocation site to satisfy the next pending request when it arrives. If the allocator succeeds in finding a suitable site, it associates a controller with the new task and its partition and allows the controller to assume responsibility for loading the task and establishing I/O to it.

When the allocator fails to find a suitable allocation site for the next pending request, it invokes a defragmenter, that determines whether or not the request can be satisfied with partial rearrangement. If allocation with rearrangement is possible, the defragmenter performs the rearrangement and returns the allocation site thereby created to satisfy the request. If on the other hand allocation with rearrangement is not possible, then all requests are blocked until allocation is attempted once more.

Subsequent allocation attempts are made whenever a task completes and there is a request for service pending. If some executing tasks can be rearranged to accommodate the task, then a schedule for suspending and moving them is computed. The defragmenter then coordinates the partial reconfiguration of the FPGA by signalling individual task controllers to suspend a task's operation, save the task's context, move the task and its context to a new location, and to resume the task's operation.

For the sake of fairness and simplicity, requests for service are processed in first–come first–served (FCFS) order. However, the methods we describe do not depend upon the scheduling method. Non–FCFS methods with better performance could therefore be used.

The task allocator uses bottom–left allocation, which is a first fit method [10].

The bottom–leftmost free block large enough to satisfy the request is allocated to the task. The advantages of the first fit method are that it is simple and that it has complete recognition capability, i.e., it recognizes all possible allocation sites. Many other contiguous allocation schemes have been proposed. It should be noted that partial rearrangement can be successfully used with any allocation method.

Definition 4 *Let* $T = \{t_i[l_{1,i}, l_{2,i}] : 1 \leq i \leq n\}$ *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 partitions* $a(t_i) = G^2[\mathrm{bl}(t_i), \mathrm{tr}(t_i)]$ *allocated to each task in* T. *The allocation for task* t_i *is said to be based at the cell allocated to the bottom–left corner of the task* $(x(\mathrm{bl}(t_i)), y(\mathrm{bl}(t_i)))$ *and to extend to the cell allocated to the top–right corner of the task* $(x(\mathrm{tr}(t_i)), y(\mathrm{tr}(t_i)))$.

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 assume I/O to tasks is performed by register access. The interesting problem of rerouting I/O wires from the chip periphery to a task that is moved is therefore not considered.

Since tasks are assumed to be without deadlines, any task is considered preemptable and may therefore 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 is the worst case in the absence of feedback circuits. However, since with current technology 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.

3. Problem Statement and Complexity

The purpose of partial FPGA rearrangement is to allocate waiting tasks as quickly as possible so as to reduce task response time. This benefit to the waiting task is at the cost of delays to the executing tasks since they must be interrupted in order to be moved. It is therefore desirable to delay as few executing tasks as possible and to minimize the maximum amount any single task is delayed. The FPGA model moves tasks by reloading their configuration bit streams with new offsets. This approach determines the time needed to complete the rearrangement since the time to move a task is assumed to be proportional to its area and the tasks that need to be moved must be reloaded one after another. The time to complete a rearrangement is therefore proportional to the total area of the tasks moved. Since the rate at which tasks can be allocated is limited by the rate at which allocations can be found or rearrangements can be performed, it is desirable to complete rearrangements as quickly as possible. These factors contribute to the

formulation of the FPGA rearrangement problem as follows.

FPGA REARRANGEMENT PROBLEM
INPUT: A set of executing tasks $T = \{t_1, \ldots, t_n\}$, an arrangement $A(G^2[(1,1), (W,H)]) = \{a(t_i) : t_i \in T\}$ of the executing tasks, and a waiting task $t_{n+1}[l_{1,n+1}, l_{2,n+1}]$ that cannot be allocated to the array without overlapping the allocation of some other task in T.
OUTPUT: A new arrangement $A'(G^2[(1,1),(W,H)]) = \{a'(t_i) : 1 \leq i \leq n+1\}$ of the tasks, if possible, and a schedule $p : T \rightarrow Z_0^+$ for moving the tasks in $\{t_i : a(t_i) \neq a'(t_i)\}$ such that:

1. the delay to the waiting task is minimized,
2. the maximum delay to moving tasks is minimized, and
3. the time needed to complete the rearrangement is minimized.

The statement of the problem suggests two subproblems are to be solved. First, a new arrangement of the executing tasks that accommodates the unallocated or waiting task is needed. Second, a schedule for moving the tasks from their initial to their final allocations needs to be found. The work involved in solving these problems represents an overhead to the system. An additional requirement, therefore, is to find efficient solutions.

3.1. Identifying Feasible Rearrangements

A feasible rearrangement of the executing tasks is a new arrangement of the tasks that allows the waiting task to be allocated as well. Although it is assumed that tasks may be rotated before allocation, the rearrangements considered in this work do not rotate the executing tasks.

In [5], Li and Cheng show that it is NP–complete to decide the RECTANGLE PACKABILITY problem, which is to determine whether or not a set of oriented rectangles can be orthogonally packed without overlap into a larger containing rectangle. Their proof was by reduction from the PARTITION problem [4]: the sizes of the elements of a given PARTITION instance determine the widths of corresponding rectangles having height 1/4. These can be packed into an array of width one half the total size of the PARTITION elements and height 1/2 in polynomial time if and only if P = NP. The following theorem thus follows.

Theorem 1 *([5]) RECTANGLE PACKABILITY is NP–complete.*

Corollary 1 *REARRANGEMENT FEASIBILITY, the problem of deciding whether a set of executing FPGA tasks can be rearranged to accommodate the next waiting task, is NP–complete.*

Proof. By equivalence with RECTANGLE PACKABILITY. A procedure for deciding RECTANGLE PACKABILITY can decide whether or not the set of executing tasks taken together with the next waiting task can be packed into the array. Similarly, an algorithm for deciding REARRANGEMENT FEASIBILITY

can be used iteratively to determine RECTANGLE PACKABILITY. The problems are therefore computationally equivalent. □

Since the problem of deciding REARRANGEMENT FEASIBILITY is NP–complete, it is unlikely to have a polynomial time solution. The corresponding optimization problem, that of finding a feasible rearrangement, is therefore also unlikely to be easy. Consequently, heuristic solutions are sought.

3.2. Scheduling FPGA Rearrangements

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, a method for rearranging the tasks, i.e., moving the tasks from their initial to their final partitions, is sought that minimizes the delay (defined below) to tasks subject to the following constraints. These constraints arise as a consequence of the FPGA model and the scheduling goals.

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 its placement cannot be interrupted. The time needed to place a task is equal to its size $s(t_i) = w(\text{or}(t_i)) \times h(\text{or}(t_i))$.

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. The sequencing constraints can then be formulated in the following way:

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

The 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 2 *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, the well–known PARTITION problem is reduced to it [4].

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 $S = \sum_{i=1}^{n} s(a_i)$. Then 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 $t_i = [w(\mathrm{or}(t_i)), h(\mathrm{or}(t_i))]$ denote a task oriented with width $w(\mathrm{or}(t_i))$, height $h(\mathrm{or}(t_i))$, and size $s(t_i) = w(\mathrm{or}(t_i)) \times h(\mathrm{or}(t_i))$. Set

$$\begin{aligned} t_1 &= t_{n+3} = [2S, 2], \\ t_2 &= [S, 1], \end{aligned}$$

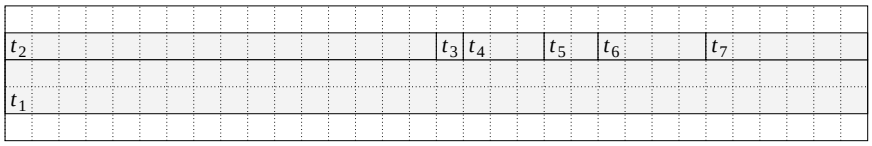
and construct a further n tasks corresponding to the items in A,

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

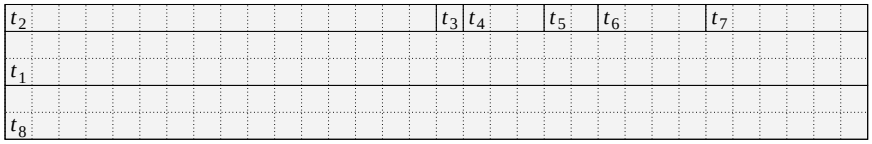
The initial arrangement of the tasks for a particular instance is illustrated in Fig. 2(a). Task t_1 initially occupies the second and third rows from the bottom of an array of width $2S$ and height 5. Tasks t_2 through t_{n+2} are arranged in sequence from left to right along the fourth row from the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

The final arrangement of the tasks of the example is shown in Fig. 2(b). 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.

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



(a)



(b)

Fig. 2. The initial (a) and final (b) arrangement of the tasks for an instance of PARTITION with $A = \{1, 3, 2, 4, 6\}$.

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 moved 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 $5S$ at least since they would be delayed by the time taken to place tasks t_1 and t_2. 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 2 *Without the constraint of placing the waiting task first of all, FPGA*

rearrangement scheduling is NP–complete.

4. Local Repacking

Local repacking is a heuristic approach to finding feasible rearrangements that limits the area of the FPGA that needs to be rearranged in order to accommodate the waiting task. 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 subarray as well. A hierarchical decomposition of the array known as a free area tree is used to keep track of the number of free cells within each subarray. In so doing, regions that contain sufficient free area to accommodate the waiting task can be identified quickly, and rearrangements of the tasks they contain can be attempted. Two–dimensional bin packing algorithms with good performance bounds are used for this last step: the tasks, viewed as rectangles, are packed from scratch into an infinitely long strip whose width is determined by the length of one side of the subarray. If the tasks are packed using total height less than the length of the other side of the subarray, then the rearrangement is feasible and its cost is assessed. An ordered depth–first heuristic search algorithm is used to schedule and compare the cost of feasible FPGA rearrangements.

4.1. Free Area Trees

A free area tree is a type of quadtree [6, 11] 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, which 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 subarrays, depending upon its size, and represented by child nodes.

Fig. 3(a) depicts the arrangement of a pair of tasks on a rectangular array. The array is partitioned to show the regions delimiting the extent of the leaf nodes in the free area tree representation of the arrangement. The free area tree corresponding to the arrangement of Fig. 3(a) is illustrated in Fig. 3(b).

When invoked, the local 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 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. The approach we adopt is to attempt to repack these tasks completely into the rectangular array covered by the node as well. This approach

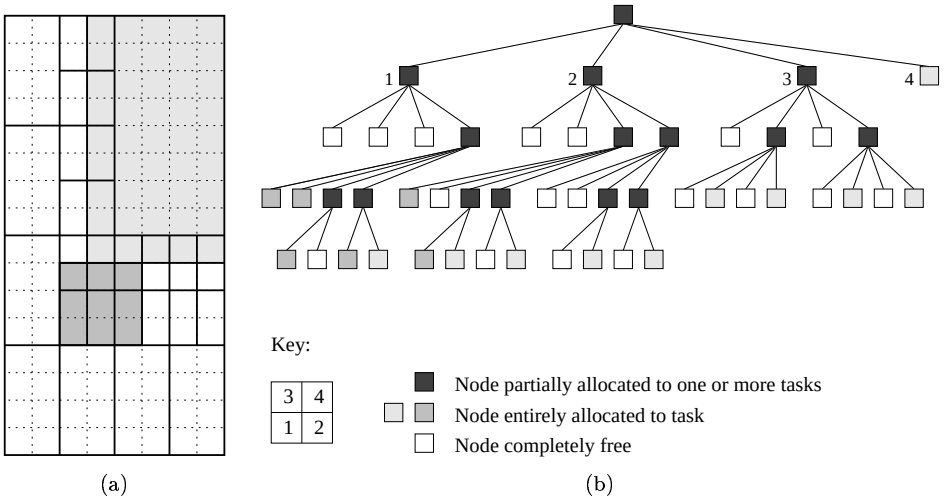


Fig. 3. (a) The arrangement of a pair of tasks on an array with free area tree leaves marked. (b) The free area tree for the arrangement of (a).

avoids further searching and avoids the complexity of packing into arbitrary rectilinear polygons. At each node, therefore, the area available for the waiting task needs to account for the total area of tasks that are partially covered by the region as well.

4.2. 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 the region that is known to cost least to repack should be discovered first. 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 that affect the least number of tasks but have the least chance of accommodating the waiting task first of all. An ideal search therefore starts somewhat higher in the tree and works its way up.

4.3. Repacking the Tasks

The search of the free area tree identifies those subarrays that might accommodate the waiting task if the tasks allocated to it are rearranged. Well–known strip–packing algorithms can be used to check 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. This variant of the two–dimensional bin–packing problem is NP–complete. Much attention has therefore been given to finding polynomial time approximation algorithms, i.e., fast

algorithms that come within a constant times the height used by an optimal packing [3]. 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.

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 [7]. 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, a "strip" of width $(x_2 - x_1 + 1)$ is tried first. 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 4.1, the algorithm attempts to pack tasks that are partially intersected by the subarray 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 not attempted.

4.4. 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, in which 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. The idea is to expand the node that seems most promising. Such a search is called an *ordered search* or *best–first search* [2]. One way of judging the promise of a node is to estimate the cost of a solution path that 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.

A well–known optimal ordered search algorithm applicable to finding minimal–cost paths in directed acyclic graphs is the A* algorithm [2]. 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 A* is guaranteed to find a solution path 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.

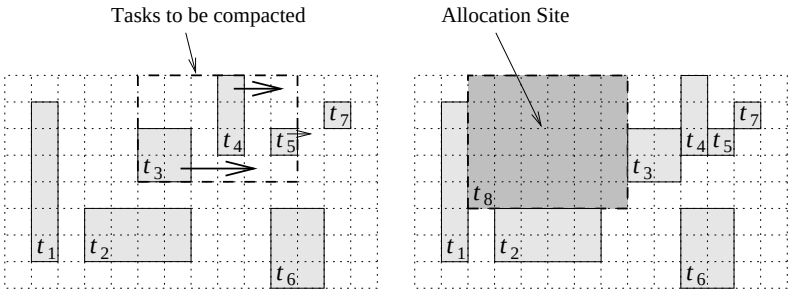
The cost of reaching node $g(c_i)$ is given by the maximum of the delays to the relocated tasks, which is known. A simple estimator of the minimal–cost path to reach a leaf from the node is also available: in calculating $h^*(c_i)$, ignore the executing tasks, and determine the maximum amount by which the suspended tasks could be delayed if they were scheduled in non–decreasing $r(t_i) + s(t_i)$ order. The minimum cost of a path to a leaf through c_i is then estimated by $f^*(c_i) = \max\{g(c_i), h^*(c_i)\}$.

Unfortunately 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. However, an acceptable solution can be found most of the time for lower cost by making a locally optimal choice of the next node to expand. Such a search is known as an *ordered depth–first search* [2]. The quality of the solution is determined by the depth to which the algorithm looks ahead before making a choice. Better quality can thus be gained at higher search cost.

5. Ordered Compaction

Ordered compaction constrains the movement of tasks in order to identify feasible rearrangements efficiently. The ordered compaction heuristic places the waiting task at a favourable location and moves those tasks that initially occupy the allocation site off to one side only. Ordered compaction therefore has the effect of moving a subset of the executing tasks closer together while preserving their relative order. Without loss of generality, ordered compaction to the right is considered. Fig. 4 depicts an example of a right ordered compaction.

In order to minimize the time to complete a compaction we show it is best to



(a) Initial arrangement (b) Final arrangement

Fig. 4. An example of a right ordered compaction. The initial arrangement on the left shows the tasks to be compacted so as to allocate a waiting task of size 6×5. The final arrangement on the right indicates the allocation site for the waiting task.

attempt to place the waiting task adjacent to a pair of tasks such that one abuts the allocation site on its left and the other abuts the allocation site below. The number of potential allocation sites worth checking is thus significantly reduced. It is then shown how the feasibility of a site can be decided by searching a visibility graph defined over the executing tasks. An optimal schedule for freeing the allocation site of executing tasks is also obtained during the search.

In the following, the right ordered compaction of tasks for a given orientation of the waiting task is discussed. It is assumed that $\mathrm{or}(t_{n+1}) = (w, h)$. However, it is necessary to consider both orientations of the waiting task and to consider compacting the executing tasks to the left, top, and bottom of the array as well in order to find the best allocation site. In each case the method is identical, albeit with orientations and directions switched in the natural way. For the remainder of this section the term *compaction* is used to refer to right ordered compaction.

5.1. Identifying Potential Allocation Sites

The following definitions are used to pinpoint the minimum cost locations for placing the base of the waiting task t_{n+1} if it is to be allocated in the neighbourhood of t_i.

The first definition arises from considering where the waiting task can be based were it to abut t_i on the right.

Definition 7 *For the waiting task* t_{n+1}*, assumed to be oriented such that* $\mathrm{or}(t_{n+1}) = (w, h)$*, and for each executing task* $t_i, 1 \leq i \leq n$*, the right cell interval for* t_i*,* $\mathrm{rci}(t_i, t_{n+1})$ *consists of the set of possible base locations for* t_{n+1} *were some cell in its leftmost column placed adjacent to and in the same row as a cell in the rightmost column of* t_i*.*

The existence and extent of the right cell interval for t_i given $\mathrm{or}(t_{n+1})$ is constrained by the boundaries of the array but disregards the intersection of t_{n+1} with other executing tasks.

Definition 8 *The right cell interval*

$$\begin{aligned}\mathrm{rci}(t_i, t_{n+1}) &= G^2[(x(\mathrm{tr}(t_i)) + 1, \max\{1, y(\mathrm{bl}(t_i)) - h + 1\}),\\ &\quad (x(\mathrm{tr}(t_i)) + 1, \min\{y(\mathrm{tr}(t_i)), H - h + 1\})]\end{aligned}$$

exists iff $x(\mathrm{tr}(t_i)) + 1 \leq W - w + 1$.

Definition 9 *The right cell intervals for* t_{n+1} *is the set*

$$\mathcal{R}(t_{n+1}) = \{\mathrm{rci}(t_i, t_{n+1}) : 1 \leq i \leq n, x(\mathrm{tr}(t_i)) \leq W - w\} \cup G^2[(1,1), (1, H - h + 1)],$$

which includes an interval that is defined with respect to the left edge of the array.

Similar definitions can be made regarding the placement of the waiting task in the vicinity of the other edges of executing tasks. Those that can be made with respect to the topmost row of an executing task follow.

Definition 10 *For the waiting task* t_{n+1} *the top cell interval for* t_i *is the set of possible base locations* $\mathrm{tci}(t_i, t_{n+1})$ *were some cell in the bottommost row of the waiting task placed adjacent to and in the same column as a cell in the topmost row of* t_i.

The top cell interval

$$\begin{aligned}\mathrm{tci}(t_i, t_{n+1}) &= G^2[(\max\{1, x(\mathrm{bl}(t_i)) - w + 1\}, y(\mathrm{tr}(t_i)) + 1),\\ &\quad (\min\{x(\mathrm{tr}(t_i)), W - w + 1\}, y(\mathrm{tr}(t_i)) + 1)]\end{aligned}$$

exists iff $y(\mathrm{tr}(t_i)) + 1 \leq H - h + 1$.

The top cell intervals for t_{n+1} *is the set*

$$\mathcal{T}(t_{n+1}) = \{\mathrm{tci}(t_i, t_{n+1}) : 1 \leq i \leq n, y(\mathrm{tr}(t_i)) \leq H - h\} \cup G^2[(1,1), (W - w + 1, 1)],$$

which includes an interval defined with respect to the bottom edge of the array.

The cells at the intersection of the set of right and top cell intervals for t_{n+1} are of particular interest.

Definition 11 *The set of cells at the intersection of the set of right and top cell intervals for* t_{n+1} *is denoted* $\mathcal{I}(t_{n+1}) = \mathcal{R}(t_{n+1}) \cap \mathcal{T}(t_{n+1})$.

Definition 12 *Let the set* $\mathcal{B}(t_{n+1})$ *denote the union of the set of cells in* $\mathcal{I}(t_{n+1})$ *with the bottommost cells of each* $\mathrm{rci}(t_i, t_{n+1}) \in \mathcal{R}(t_{n+1})$ *and the leftmost cells of each* $\mathrm{tci}(t_i, t_{n+1}) \in \mathcal{T}(t_{n+1})$.

Theorem 3 *If the waiting task* t_{n+1} *can be allocated by right ordered compaction, then the time needed to complete the compaction is minimized for an allocation site based at some cell in* $\mathcal{B}(t_{n+1})$.

Proof. The proof considers the time needed to free the allocation site for all possible base locations of the waiting task. The assumption is that the time needed to complete the compaction is proportional to the area of tasks that need to be moved out of the allocation site.

The right cell interval for t_i is the leftmost column in t_i's neighbourhood where t_{n+1} can be placed without intersecting t_i. Refer to Fig. 5. Were the placement of

t_{n+1} to intersect t_i, t_i would have to be moved to the right of the allocation site for t_{n+1} by the right ordered compaction rule, thereby increasing the time needed to complete the compaction. Placing t_{n+1} to the right of the right cell interval for t_i does not reduce the cost of freeing the area needed by t_{n+1}. Indeed, it could increase the cost by intersecting additional tasks on the right boundary of the allocation site. For example, see task t_k in Fig. 5. These additionally intersected tasks would need to be moved as well, thereby increasing the time needed to complete the compaction.

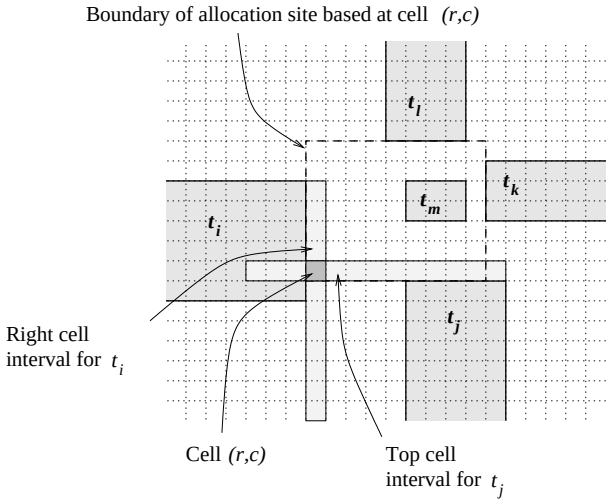


Fig. 5. Allocation sites based at the intersection of right and top cell intervals are locally optimal. Minor displacements from these local optima can force additional tasks to have to be removed from the allocation site.

If the right cell interval for t_i intersects the top cell interval for t_j at cell (r, c) say, then if t_{n+1} were based at (r, c), it would, as described above, be constrained from being placed further to the left or right without potentially increasing the time needed to free the allocation site. The waiting task would also be constrained from being located above or below the top cell interval for t_j by a similar argument since a slightly lower placement of the site would intersect t_j, forcing it to be moved, and a slightly higher placement would force the movement of any tasks that become intersected at the top edge of the site. In Fig. 5, for example, task t_l is intersected if the allocation site is based above the top cell interval for t_j.

If the right cell interval for t_i is not intersected by a top cell interval, then it is possible for an allocation site based at a cell in the interval to intersect another task in one way only. The site could intersect a task t_j, to the right of and in the vicinity of t_i, whose top edge is flush with, or above the top edge of t_i. See Fig. 6 for an example. Basing the waiting task at the bottommost cell of the right cell interval avoids the need for compaction if the site does not intersect such a task. On the other hand, no more time is needed to complete the compaction for a site based at this cell than at any other cell in the right cell interval for t_i since each location forces t_j to have to move.

A similar argument can be used to show that it is only necessary to check the

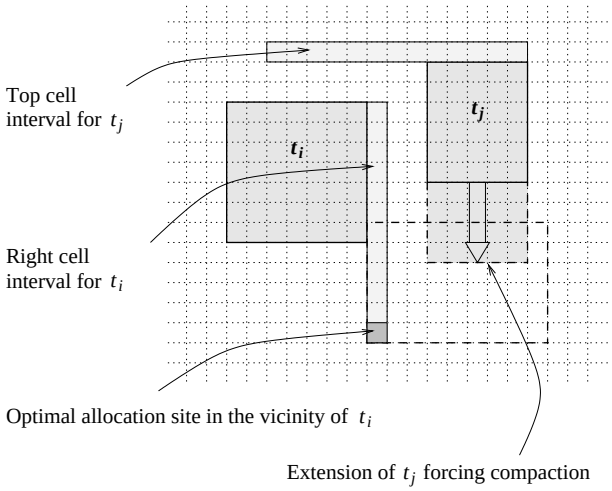


Fig. 6. Right cell intervals that are not intersected may offer opportunities for allocating the waiting task without compaction. In any case, checking the bottommost cell is optimal.

leftmost cell of each top cell interval when the interval is not intersected by a right cell interval. □

5.2. Assessing Allocation Site Feasibility

Allocation sites based at cells in $\mathcal{B}(t_{n+1})$ are not guaranteed to be feasible because it may not be possible to compact the executing tasks within such a site to the right due to lack of space. An efficient way of assessing feasibility is to build a visibility graph of the executing tasks.

Definition 13 *(After [8]) A task v is said to dominate a task t if, for some cell (r_v, c_v) of v and some cell (r_t, c_t) of t, $r_v = r_t$ and $c_v > c_t$. Where v dominates t, v is said to directly dominate t if there is no task u such that v dominates u and u dominates t. A visibility graph is a directed graph having the collection of executing tasks as vertex set and for each pair of tasks t and v it contains an edge from t to v iff v directly dominates t.*

Fig. 7 depicts the visibility graph for the example of Fig. 4. For each potential base $b \in \mathcal{B}(t_{n+1})$, the subgraphs that span rows intersected by the allocation site were it based at b are searched. The leftmost tasks that intersect the potential allocation site can be identified by a depth first search. Once they have been found, the feasibility of moving them right the required distance can be checked. If the potential base admits a feasible rearrangement, the set of tasks that need to be moved can be listed by searching the subgraphs of the leftmost tasks intersecting the allocation site.

The order in which potential allocation site bases are searched influences the efficiency of the ordered compaction method. It is desirable to search the bases as they are identified in a left to right sweep across the array because tasks that intersect potential allocation sites based closer to the left edge have a better chance

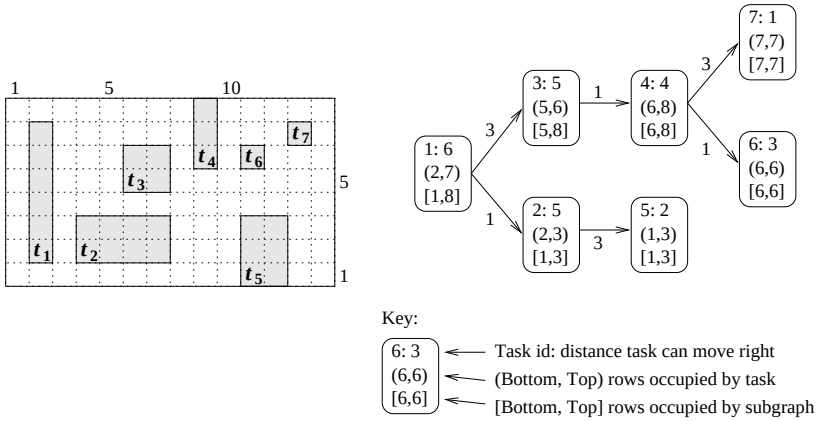


Fig. 7. An arrangement of tasks on the left, with its visibility graph depicted on the right.

of being accommodated on the right. To this end, it is useful to check potential allocation site bases in the order generated when right cell intervals are chosen in increasing column order. However, sites closer to the left may involve moving a greater total area of tasks than sites further to the right. These potentially less costly sites become more sparse as the sweep progresses because it becomes more difficult to compact the intersected tasks. The search for the best allocation site could therefore be abandoned when the cost of further searching exceeds the marginal benefit of finding less costly compaction schedules.

5.3. Scheduling Ordered Task Movements with Minimum Delay

Given a set of tasks that are to be orderly compacted it is possible to move the tasks without delay according to the visibility graph if a task is not moved until all tasks in its subgraph that must move have moved. This scheduling policy minimizes delays to executing tasks by suspending each task that is to be moved just for the period needed to reload it, and by moving it onto a region of the FPGA that does not overlap any other executing tasks. Scheduling the compaction is straightforward and requires time linear in the number of tasks to be compacted. The only drawback with the policy is that it moves the tasks occupying the allocation site last of all and therefore does not minimize the time needed to free the allocation site. However, this may not be a serious disadvantage since it is the rate at which compactions are completed that determines the rate at which waiting tasks can be allocated.

6. Evaluation of Partial Rearrangement Heuristics

6.1. Time Complexity

For an FPGA of width W and height H with $m = \max\{W, H\}$ and n executing tasks, the local repacking heuristic 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. Ordered compaction, on the other hand, needs $O(n^2)$ time to identify potential allocation sites and build the visibility graph. Each of the potential sites can be checked in $O(n)$ time, which leads to a time complexity of $O(n^3)$ for ordered compaction to determine whether a feasible compaction exists or not. These costs can be reduced by dynamically maintaining the free area tree and visibility graph.

In the worst case it is difficult to know which method requires more time. However, in practice only a few nodes at the root of the free area tree are searched, which means $O(mn)$ time is spent building the tree, and a few searches at a cost of $O(n \log n)$ time each are performed. For ordered compaction, the visibility graph needs to be built, and if the potential allocation sites are checked in a left to right sweep, the search can be abandoned after checking the right cell interval on the left array border, which usually suffices to determine compaction feasibility. The cost for ordered compaction is therefore more likely to be $O(n^2)$, which is unlikely to be greater than the $O(mn)$ time needed by local repacking.

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 $O(n^3 \log n)$ time with one–state lookahead, or $O(n^4 \log n)$ time with two states of lookahead. When the waiting task is to be placed first of all, both methods need to use the approximate scheduling method.

6.2. Empirical Performance

Simulation experiments were carried out to compare the performance of the static first fit allocation method with dynamic allocation methods employing local repacking and ordered compaction whenever first fit failed. Local repacking used an ordered depth–first search with two–state lookahead to schedule rearrangements, while ordered compaction adopted a strategy that minimized the delay to executing tasks. Both local repacking and ordered compaction were programmed to abandon the search for feasible rearrangements when the first feasible rearrangement was found. This rearrangement was then scheduled.

Several experiments were conducted to compare the performance of the different allocation/rearrangement methods. For each experiment, sets of 10,000 tasks characterized by 4 independently chosen uniformly distributed random variables were generated. The random variables represented the two task side lengths, the inter-task arrival period, and the task service period. The tasks were queued and placed in arrival order to a simulated FPGA of size 64×64. The time needed to load a task was determined by the availability of space and the time used to configure the cells needed by the task. The configuration delay per cell was thus also a parameter. Each experiment averaged the results of 10 runs.

The first experiment investigated the effect on performance of varying task loads with nominal configuration delays (see Fig. 8(a)). Both local repacking and ordered compaction perform significantly better than first fit when the FPGA is saturated with work as tasks arrived more quickly than they could be processed. Varying

the maximum task size indicates local repacking performs marginally better than ordered compaction when tasks are small, whereas ordered compaction performs better when task sizes as large as the array size are permitted (see Fig. 8(b)). When the FPGA is saturated with work, partial rearrangement reduces the mean allocation delay by up to 24%. Response times are correspondingly lower and the utilization correspondingly higher. When tasks arrive less frequently than they can be processed, the benefits of rearrangement are insignificant.

A second experiment investigated the effect on performance of varying the configuration delay in the saturated operating region (see Fig. 9(a)). Both methods performed well when the mean configuration delay is very low (less than 1% of the mean service period). However, local repacking begins to perform worse than first fit at mean configuration delays of less than 5% of the mean service period. By comparison, ordered compaction sustains mean configuration delays corresponding to approximately 10% of the service period before performing worse than first fit. The very high execution delays experienced by tasks using the local repacking method is the main factor contributing to its poor performance (see Fig. 9(b)). Ordered compaction, which delays no task longer than is needed to move it, and which may move tasks with less total area, delays tasks much less. It therefore sustains better performance than first fit over a larger range of configuration delays.

7. Discussion

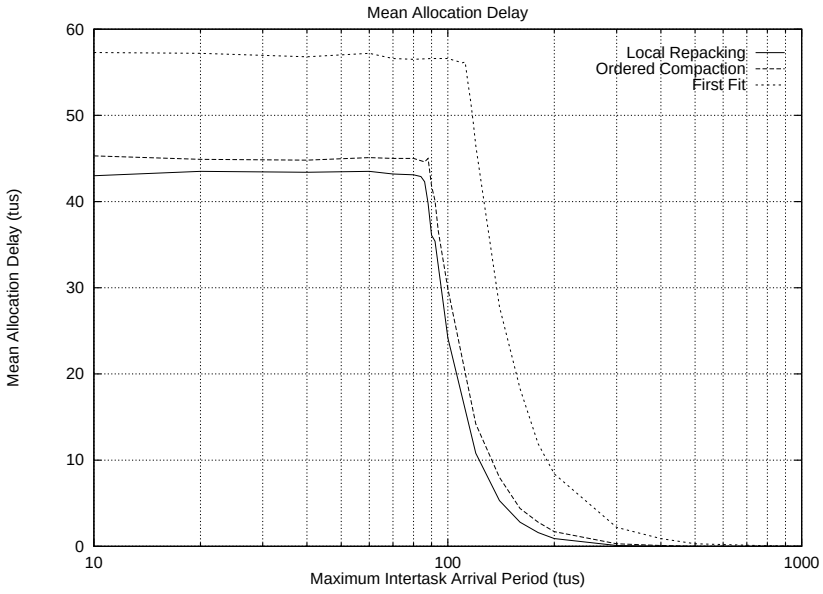
Partial rearrangement has the potential to offer considerable performance advantages with acceptable computational effort. Unfortunately, these benefits may be jeopardized by execution delays which render partial rearrangement ineffective at configuration delays that are relatively small compared with the mean task service period.

These execution delays need to be substantially reduced if partial rearrangement is to be broadly applied. Since the ratio of the configuration delay to the computational latency of cells is unlikely to change significantly, unless heuristics that reduce the total area of tasks involved in rearrangements are found, new approaches to moving tasks are needed.

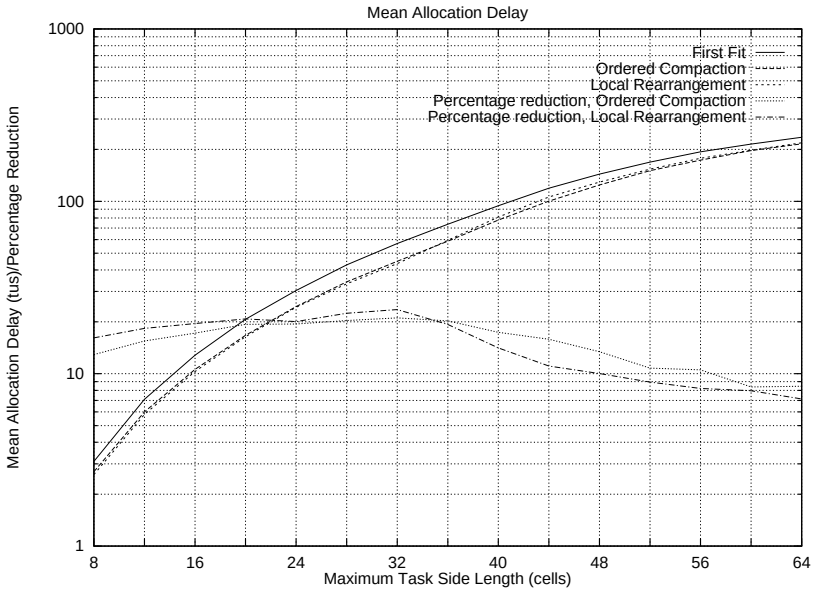
The main bottleneck with the approach is that tasks are reloaded from off–chip in a sequential process that takes time proportional to the area of each task. If it were possible to use the configurable interconnect for moving tasks, then it would be possible to eliminate the sequential reload step and to reconfigure a number of target cells at a time. Furthermore, several tasks might make use of the available bandwidth to move at the same time. Additional performance gains could then be expected from the reduced delays to individual tasks and rearrangement schedule lengths.

8. Conclusions

Partial FPGA rearrangement aims to reduce allocation delays for waiting tasks by reducing the fragmentation of free cells through the movement of executing

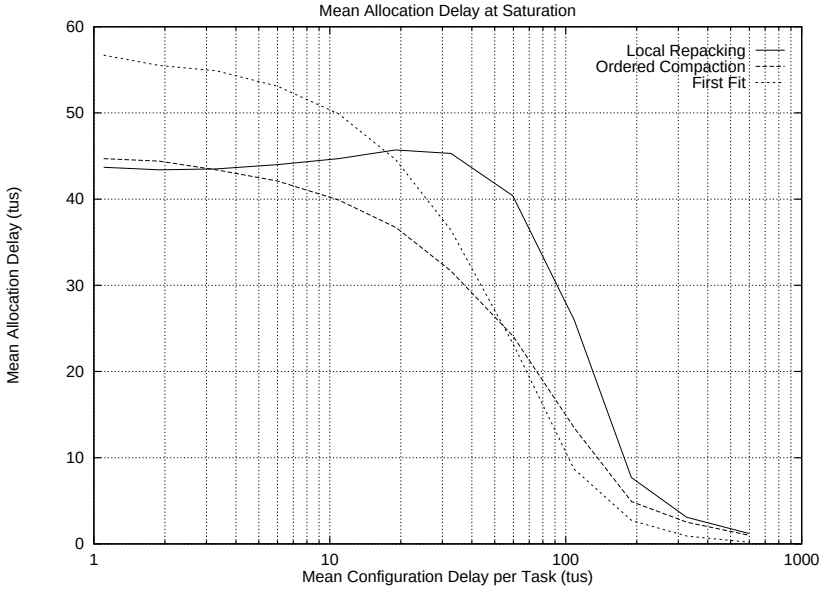


(a)

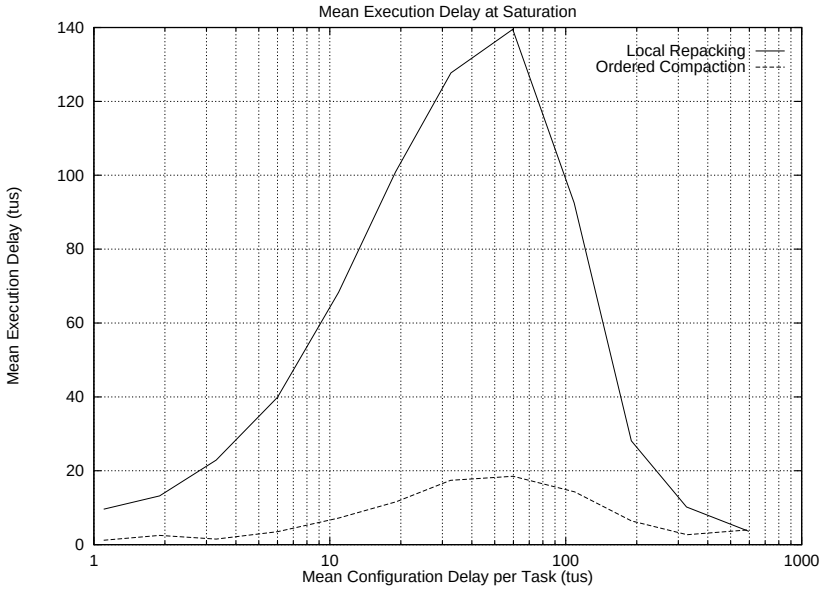


(b)

Fig. 8. (a) Mean allocation delay for local repacking, ordered compaction, and first fit as the maximum inter–task arrival period was varied. Task sizes were chosen from U(1, 32) $\times$ U(1, 32) and CD was 1/1,000 time unit (tu). (b) Mean allocation delay for local repacking, ordered compaction, and first fit as the maximum task size was varied. The intertask arrival period was set to 1 tu. For both plots, the task service period was chosen from U(1, 1000) tus.



(a)



(b)

Fig. 9. (a) Mean allocation delay at saturation for local repacking, ordered compaction, and first fit as the mean configuration delay per task was varied. (b) Mean execution delay at saturation for local repacking and ordered compaction as the mean configuration delay per task was varied. For both plots, task sizes were chosen from U(1, 32) $\times$ U(1, 32) and intertask arrival period was set to 40 time units (tus). The task service period was chosen from U(1, 1000) tus.

tasks. In order to minimize the delay to waiting tasks, the rearrangements should be performed as quickly as possible. When tasks are reloaded, this aim equates to minimizing the total area of the set of tasks to be moved. So as not to offset the benefits of rearrangement, the delay to executing tasks that are to be moved needs to be minimized as well.

Considering identifying feasible rearrangements of FPGA tasks is NP–complete, two heuristics were developed. Local repacking uses a quadtree decomposition of the FPGA to identify subarrays that may be capable of accommodating the waiting task if they are repacked. Known strip–packing algorithms are used to attempt the repacking. Ordered compaction, on the other hand, searches a visibility graph of the executing tasks to determine whether tasks can be moved together in one direction so as to fit the waiting task in the resulting gap — the method can also be used to identify allocation sites when compaction is not required. The fixed costs associated with constructing the data structures needed to search for feasible rearrangements can be reduced by dynamically maintaining them. However, the efficiency of both methods depends upon the search strategy used. It is not clear how the search effort can be minimized.

Scheduling rearrangements so as to minimize the delays to the executing tasks that must be moved was shown to be NP–complete. Scheduling heuristics based on state-space search strategies were therefore proposed, and a simple linear time cost function for guiding the order in which nodes are expanded was described. The more constrained task movements of ordered compaction allow a scheduling method that minimizes the delay to the moving tasks to be used.

It is difficult to distinguish between the heuristics for identifying rearrangements on the basis of worst case performance. It is likely that the cost of identifying feasible ordered compactions is no more than the cost of identifying feasible local repackings. However, the existence of a linear time scheduling algorithm for ordered compaction gives it a scheduling advantage. Analytical performance bounds on the ordered depth–first scheduling heuristic are needed.

An experimental assessment of the performance of the methods indicates dynamic allocation by partial rearrangement can be of significant benefit when the configuration delay is a small fraction of the mean service period and when tasks arrive more quickly than they can be processed. Local repacking appears to be slightly more effective than ordered compaction when task sizes are small, however, both methods become ineffective with modest increases in the configuration delay.

In order to increase the range of application, it is proposed to examine moving tasks on–chip as a means of overcoming the I/O bottleneck of reloading configuration bit streams from off the chip. The architectural support and scheduling techniques needed to move tasks on–chip are being further investigated.

References

1. Atmel, "AT6000 FPGA configuration guide," Document 0436B, Atmel, 1997.
2. A. Barr and E. A. Feigenbaum, eds. *The Handbook of Artificial Intelligence*, Volume I (William Kaufmann, Los Altos, 1981).

3. E. G. Coffman Jr., M. R. Garey, and D. S. Johnson, "Approximation algorithms for bin-packing – an updated survey," in *Algorithm Design for Computer System Design*, eds. G. Ausiello, M. Lucertini, and P. Serafini (Springer, Vienna, 1984) pp. 49 – 106.

4. M. R. Garey and D. S. Johnson. *Computers and Intractability — A Guide to the Theory of NP–Completeness* (W. H. Freeman, San Francisco, 1979).

5. 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, University of Houston, 1988.

6. H. Samet, "The quadtree and related hierarchical data structures," *ACM Comput. Surv.* **16** (1984) 187 – 260.

7. D. D. K. D. B. Sleator, "A 2.5 times optimal algorithm for packing in two dimensions," *Inf. Process. Lett.* **10** (1980) 37 – 40.

8. A. P. Sprague. "A parallel algorithm to construct a dominance graph on nonoverlapping rectangles," *Int. J. Parallel Program.* **21** (1992) 303 – 312.

9. Xilinx, "XC6200 Field Programmable Gate Arrays," Technical report, Xilinx, 1996.

10. Y. Zhu, "Efficient processor allocation strategies for mesh–connected parallel computers," *J. Parallel Distrib. Comput.* **16** (1992) 328 – 337.

11. Y. Zhu, "Fast processor allocation and dynamic scheduling for mesh multiprocessors," *Comput. Syst. Sci. Eng.* **11** (1996) 99 – 107.