Dynamic Scheduling of Tasks on Partially Reconfigurable FPGAs

Oliver Diessel
School of Computer and Information Science, University of South Australia,
Mawson Lakes SA 5095, Australia

Hossam ElGindy
School of Computer Science and Engineering, University of New South Wales,
Sydney 2052, Australia

Martin Middendorf, Hartmut Schmeck, Bernd Schmidt
Institute of Applied Computer Science and Formal Description Methods,
University of Karlsruhe, D-76128 Karlsruhe, Germany

August 27, 1999

Abstract

Field–Programmable Gate Arrays (FPGAs) that allow partial reconfiguration at run–time can be shared among multiple independent tasks. When the sequence of tasks to be performed is unpredictable the FPGA controller needs to make allocation decisions on–line. Since on–line allocation suffers from fragmentation, tasks can end up waiting despite there being sufficient, albeit non–contiguous resources available to service them. The time to complete tasks is consequently longer and the utilization of the FPGA is lower than it could be.

We propose rearranging a subset of the tasks executing on the FPGA when doing so allows the next pending task to be processed sooner. We describe and evaluate methods for overcoming the NP–hard problems of identifying feasible rearrangements and scheduling the rearrangements when moving tasks are reloaded from off–chip.

1 Introduction

Dynamically reconfigurable field–programmable gate arrays (FPGAs) are composed of uncommitted logic cells and routing resources whose functions and interconnections are determined by user–defined configuration data stored in static RAM. This memory can be modified at run–time, thereby allowing the configuration for some part of the chip to be altered while other circuits operate without interruption.

The ability to reconfigure parts of a chip while it is operating allows functional components/tasks to be swapped in and out of the chip as needed, thereby reducing required chip area at the cost of some reconfiguration overhead, control circuitry, and memory. Embedded applications that have successfully exploited this feature to conserve hardware include an image processing system [16] and a reconfigurable crossbar switch [7]. Successful designs for cryptographic applications [15], video communications [14], and neural computing [8], attest to the suitability of the architecture for high performance array–based computations.

As more ambitious systems are developed, it is conceivable that it becomes possible and desirable for related or even disparate functions to share a single hardware platform [2, 6]. A purely time–shared approach to multi–tasking may not be appropriate because of the overhead in loading a configuration, and the limited availability of on-chip memory for caching. Space–sharing is a way of partitioning the FPGA logic resource so that each function or task obtains as much resource as it needs and executes independently of all others as if it were the sole application executing on a chip just large enough to support it. When the logic resource of an FPGA is to be shared among multiple tasks, each having their 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 [13]. 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 the next 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. We describe three methods by which feasible rearrangements may be identified that allow the waiting task to be accommodated as well. We present techniques for scheduling the task movements so as to minimize delays to the moving tasks when their configuration bit streams are reloaded at new locations. We conclude with a summary and directions for further investigation.

2 The techniques

Rearranging a subset of executing tasks, or partial rearrangement, 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 chosen tasks so as to minimize the delays to their execution. The schedule for each feasible rearrangement is evaluated for the maximum delay to the executing tasks and the time needed to complete the schedule.

The following assumptions are made. A space–shared dynamically reconfigurable FPGA is modelled as a rectangular array of configurable logic and routing resources that may be partitioned among multiple independent tasks [1, 17]. Each task is controlled by a process executing on a host. Tasks are queued and processed in arrival order; they are assumed to be independent and to be contained within orthogonally aligned, non–overlapping, rectangular sub–arrays of the FPGA. Interdependent sub–tasks are assumed to be confined to the task's bounding box. We assume I/O with individual tasks is handled via user defined registers rather than through wires routed from the chip's periphery. The interesting problem of rerouting I/O to a task after it is moved is outside the scope of this paper and will not be considered.

3 Identifying feasible rearrangements

The problem of deciding whether or not a waiting task can be accommodated on an FPGA executing a set of tasks is equivalent to the problem of deciding whether a set of non–overlapping orthogonal rectangles can be packed into a larger rectangle, which is NP–complete [9]. Heuristic solutions are therefore sought. In the following, we present three methods including two heuristics — which we refer to as local repacking and ordered compaction — and an evolutionary approach by a genetic algorithm.

3.1 Local Repacking

The local repacking method [4] attempts to repack the tasks within a sub–array so as to accommodate the waiting task as well. A quadtree decomposition of the free space in the array is used to identify those sub–arrays capable of accommodating the waiting task by virtue of the total number of free cells they contain. Every node in the quadtree corresponds to a sub–array and stores the number of free cells in its sub–array.

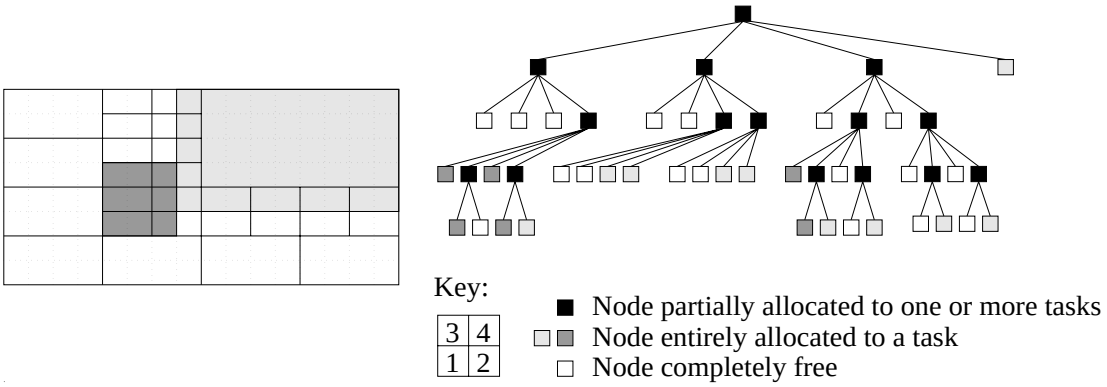


Figure 1: Task arrangement (left) and corresponding quadtree (right)

Tasks which only partially intersect a node's sub–array need to be handled in some way. Should they be included into the packing, moved elsewhere, or left where they are to be packed around? The approach adopted in this technique is to attempt to repack these tasks completely into the sub–array. Sub–arrays that contain sufficient free cells to potentially accommodate the waiting task as well as the total area of all intersected tasks thus become candidates for repacking. This approach avoids further searching and allows

the use of a fast strip–packing heuristics for the repacking.

For the repacking we use a two–dimensional strip–packing method of Sleator [12]. Given a set of oriented rectangles and a two-dimensional bin of fixed width, strip–packing involves finding a non-overlapping orthogonal packing of minimal height. The method of Sleator has a good worst case bound of 2 times the optimal height plus the height of the largest rectangle. While the orientation of the allocated tasks relative to the width of the strip needs to be preserved to obtain the performance bounds, a packing with each orientation of the waiting task is attempted. The algorithm results in a feasible rearrangement if the height of the packing is less than the height of the sub-array. Otherwise, the orientation of the strip is flipped so that its width is considered to be the height of the sub–array and packing within the width of the sub–array is attempted. If the resulting packing represents a feasible rearrangement of the tasks, movement of the tasks can be scheduled in order to evaluate the costs of the rearrangement. To find nodes in the quadtree that correspond to promising sub–arrays, a depth–first search strategy is followed. For an FPGA of width W and height H, with $m = \max\{W, H\}$, and n executing tasks, the local repacking heuristic requires $O(mn \log n)$ time to check for the existence of a feasible rearrangement.

3.2 Ordered compaction

The ordered compaction heuristic [3] places the waiting task at a favourable location, and moves those tasks initially occupying the site off in one direction. Ordered compaction therefore has the effect of sliding the executing tasks that are to be compacted closer together. Without loss of generality, we describe ordered compaction to the right hand side.

It can be shown that it suffices to attempt to place the waiting task adjacent to a pair of tasks (or the border of the array) such that one task abuts the allocation site on its left, and the other abuts the allocation site below. Let S be the set of such sites (see Figure 2). It can be shown that S contains a feasible allocation site which minimizes the total area of all tasks that intersect it (if one exists). Such a site is a good candidate because

the time to complete the compaction is proportional to the size of the tasks. The number of allocation sites which have to be checked is also significantly reduced to $O(n^2)$.

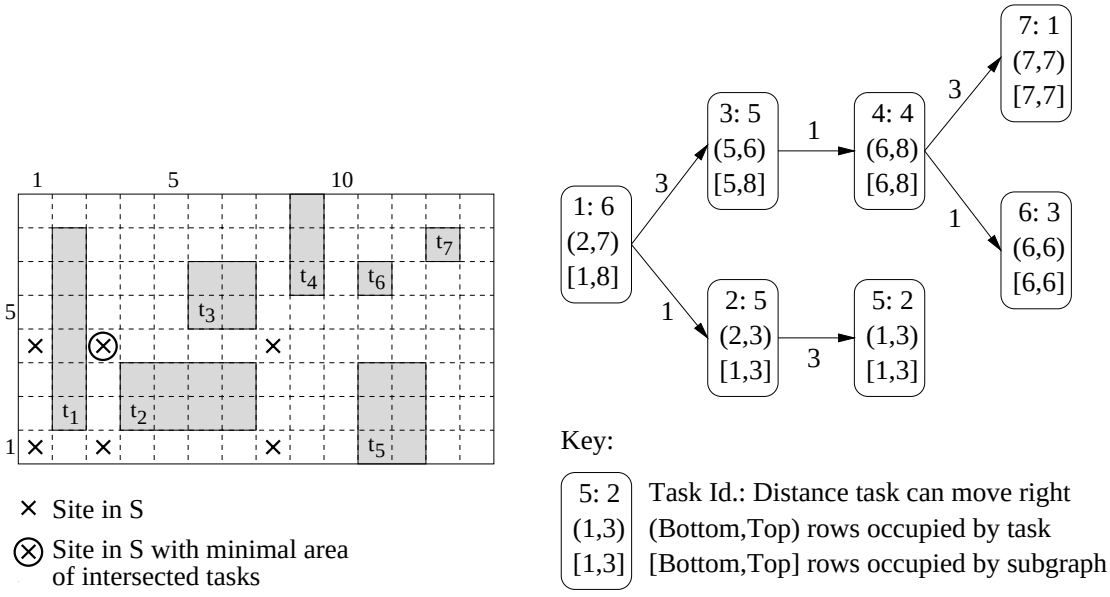


Figure 2: Task arrangement with set S for waiting task of size 6x5 (left); visibility graph for the arrangement (right)

The feasibility of a site is decided by searching a visibility graph that is defined over the executing tasks. Two nodes (tasks) t_i and t_j of the visibility graph are connected by an arc (t_i, t_j) if t_i directly dominates t_j. Task t_i is said to directly dominate t_j if cells c_j of t_j and c_i of t_i exist such that they are in the same row, cell c_i is to the left of c_j, and no other task separates them. For each node the following parameters are computed: i) the maximal number of columns the task can move to the right, ii) the range of rows occupied by the task, and iii) the range of rows occupied by some task in the subtree rooted at the node.

For each site $s \in S$ the subgraphs that span rows intersected by s are searched. The leftmost tasks intersecting a site can be identified by a depth first search. Once found, the feasibility of moving them right the required distance can be checked in time $O(n)$. The order in which the sites in S are searched influences the efficiency of the ordered compaction method. In general it is advantageous to search sites in a left to right sweep because intersecting tasks based closer to the left have a better chance of being accommodated on

the right.

Set S can be determined in time $O(n^2)$, the visibility graph can be built in time $O(n^2)$, and a feasible site with minimal total area of intersected tasks can be found in time $O(n^3)$.

3.3 Genetic Algorithm

A genetic algorithms (GA) is a probabilistic search method based on an evolutionary approach. A simplified structure of a GA is presented in Figure 3.

```
τ := 0
initialize P(0)

while stopping condition not met do
        τ := τ + 1

        /* create new population P(τ) */
        evaluation
        selection
        crossover
        mutation
end while
```

Figure 3: Structure of a genetic algorithm

In a GA possible solutions (individuals) are represented by a data structure called a chromosome. All individuals living at a specific time form a population. With regard to the chronological development, populations are also known as generations. An initial population of possible solutions is created by means of a specific initialization method. As long as the stopping condition (e.g. exceeding a given number of generations) has not been met a new generation is created. This involves determining the fitness of each individual by the application of an evaluation function. By means of the obtained fitness values

the individuals in the population can be compared with each other. Individuals which represent desirable solutions (high fitness values) are selected with high probability to produce offsprings. In a so–called crossover process, some parts of the parent chromosomes are combined to create a child chromosome. Additionally, in a mutation process the child's chromosome is changed at random in order to introduce new genetic information. The children created by crossover and mutation are inserted into the new population thereby replacing other low-fitness individuals. The GA used for finding rearrangements — called R-GA — has the following characteristics [11].

i) Representation: R-GA works only on the subset of possible arrangements that allow a specific genetic representation, called a slicing tree. Such a task arrangement — called a slicing task arrangement — is recursively defined as either a single task, or there exists a vertical or horizontal line segment dividing it into two slicing task arrangements. Each slicing task arrangement can be represented by a slicing tree (see Figure 4). A leaf represents a task with fixed orientation. A parent node corresponds to the minimal bounding box containing two horizontally or vertically aligned patterns represented by the children. Such a pattern can be either a task or another bounding box. The leaves are labelled with the index of the task and its orientation. Parent nodes contain the cut direction (vertical or horizontal).

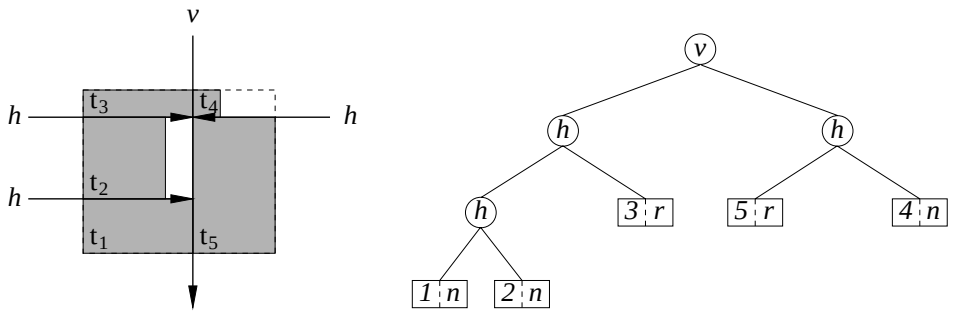


Figure 4: Arrangement (left) with corresponding slicing tree (right)

ii) Initialization: An initial population is formed by individuals (fixed slicing trees) which are built bottom–up by random pairing, i.e. starting with the leaves, two randomly chosen nodes are linked together by a newly created parent node until a complete slicing tree is obtained.

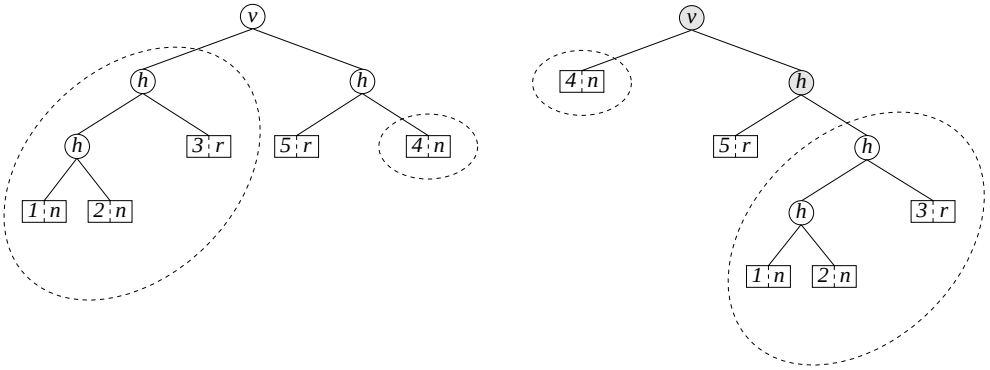
iii) Evaluation: The evaluation function considers the ratio of the tasks completely allocated inside the FPGA border, the compactness of the arrangement and the total number of cells of the individuals that have to be rearranged.

iv) Crossover: The crossover operation is based on gene–pool crossover [10]. All subtrees (including leaves, excluding the complete trees) of the two parents are inserted into a gene–pool. Duplicate subtrees are removed. A new individual is created from the subtrees in the gene–pool. Each subtree is evaluated by a rating function which considers the compactness, the number of tasks allocated inside the FPGA, and the total number of tasks in the subtree. The resulting subtree with the highest rating is chosen first. All subtrees in the gene–pool containing at least one task of the selected subtree are removed. The next disjoint subtree can then be drawn as the highest ranked subtree of the remaining gene–pool. The two selected subtrees are combined by checking each node of one subtree as a potential insertion point of the other subtree. The insertion point with the highest rating value of the resulting combined subtree is chosen. By repeatedly adding disjoint subtrees to the resulting subtree the crossover operator terminates with a complete slicing tree containing all tasks.

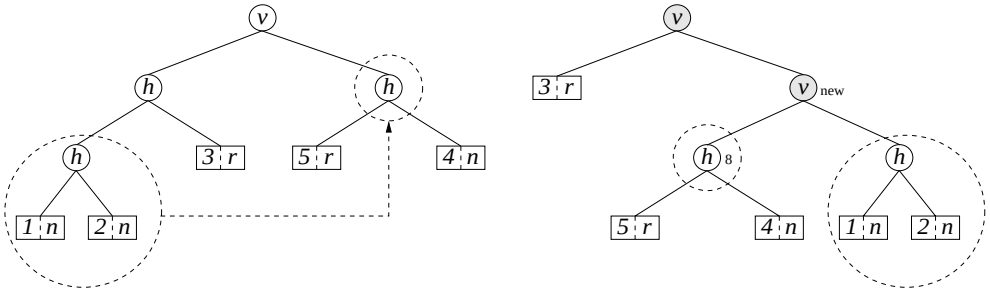
v) Mutation: Mutation changes the structure of a fixed slicing tree. If a tree was selected for mutation, one of three different mutation types is applied. Exchange mutation swaps two randomly selected subtrees (see Figure 5(a)). Insertion mutation attaches a randomly chosen subtree at a randomly selected insertion node of the tree (see Figure 5(b)). Rotation mutation randomly selects a subtree whose corresponding task arrangement is rotated by 90° left or right or 180° (see Figure 5(c)).

4 Scheduling task rearrangements

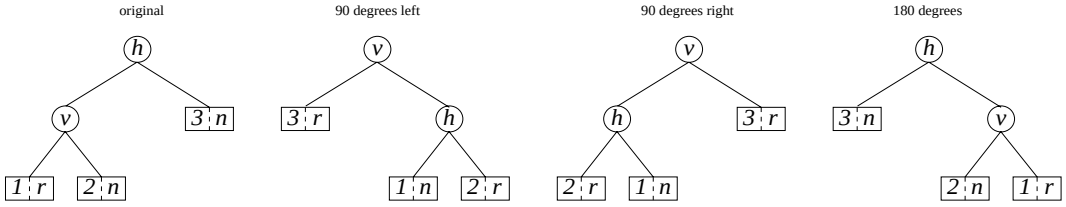
We assume the time to load a task is proportional to its area. The choice of tasks to move therefore fixes the time needed to complete the rearrangement. We assume a task may continue executing until it is suspended prior to moving and that a task is resumed as soon as it has been reloaded. If its destination is not free when it is reloaded, the tasks occupying the destination are immediately suspended and removed.



(a) Exchange mutation



(b) Insertion mutation



(c) Rotation mutation

Figure 5: Mutation operations

In this work, we distinguish between the minimum possible cost of moving a task, and the actual cost of moving it. The minimum cost is the time needed to save and reload the task, which is unavoidable. However, the actual cost needs to account for the time a task is suspended while other tasks are being reloaded. The difference between the actual and minimum costs represents a schedule delay that is to be minimized for all tasks. The problem of scheduling FPGA task rearrangements to realize this goal is NP–complete [4]. Further heuristics are therefore needed. First we describe an approximation algorithm and a genetic algorithm for scheduling rearrangements in arbitrary order. Then we describe a method that does not delay the moving tasks more than the minimum, if they are to be orderly compacted.

4.1 Arbitrary rearrangements

4.1.1 Search Tree

The problem of optimally scheduling the tasks can be viewed as a search for an optimal path in a state–space tree [4]. Each node represents the choice of a task to place into the final arrangement next, and a path from the root to a leaf represents the sequence in which tasks are chosen to be placed. A depth–first search heuristic that uses a local cost estimator to determine which node to expand next can be used to find a near–optimal path. Here we measure the cost of a node v as the maximal time a suspended task is delayed on a minimal-cost path from the root passing through node v to a leaf. For the first part, from the root to v, the maximum delay of the tasks already moved is known. For the second part, an estimator is used that ignores all executing tasks and determines the maximum amount by which the suspended tasks could be delayed. Since the executing tasks are ignored, this causes no additional tasks to be suspended. It can be shown that the optimal solution for this case is achieved when the suspended tasks are scheduled in nondecreasing $r(t) + s(t)$ order, where $r(t)$ is the time when task t was removed from the array and $s(t)$ is the size of t. Clearly, this estimator will never over-estimate the maximum delay of a relocated task. Different depths of lookahead can be used when deciding which task is scheduled next. For this paper we choose a lookahead of depth two, i.e. at a node

all possibilities for the next two scheduled tasks are evaluated by the estimator. We also constrain the method to place the waiting task first of all [1]. With a lookahead of depth two, this scheduling method needs $O(n^4 \log n)$ time.

4.1.2 Genetic Algorithm

The second approach to solve the task scheduling problem employs a GA. For our GA — called S-GA — we represent a schedule by a string of integers corresponding to the task indices arranged in the order they are moved. For initialization, a set of randomly generated permutations of these indices is used. The mutation operator exchanges two randomly chosen task indices in the gene string. For crossover we use order crossover which is suited to genes that represent permutations. Order crossover chooses a pair of cut points at random and combines two parental gene strings by keeping the substring between the cut points and adding the missing genes in the order they appear in the other parent (see Figure 6).

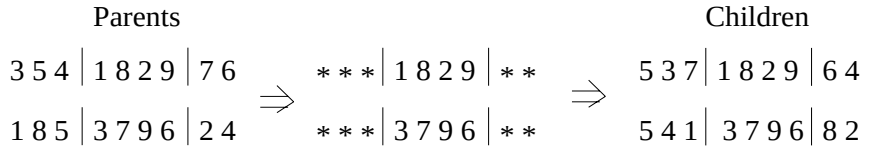


Figure 6: Order crossover

4.2 Ordered compaction

If tasks are moved as they are discovered in a depth–first traversal of the visibility graph of the executing tasks, they are moved to free destinations, and therefore do not intersect or suspend further executing tasks [3]. Tasks are not delayed more than the minimum because they are moved as soon as they are suspended. Although the waiting task is allocated last of all, the rate at which waiting tasks can be allocated is unaffected. Clearly, ordered compaction can be scheduled in $O(n)$ time.

[1] Empirical evidence suggests maximum schedule delays incurred by the heuristic are usually less than twice those of optimal solutions [5]. Theoretical bounds have not yet been established.

5 Performance assessment

A series of experiments was conducted to assess the performance of the methods with synthetic task sets. For each experiment, sets of 10,000 tasks characterized by 4 independently chosen uniformly distributed random variables were generated. Two of these variables, representing the task row and column sizes, were permitted to range from 1 cell to a specified common maximum task side length. A variable representing the tasks' service periods was allowed to range from 1 to 1,000 time units, and the intertask arrival period was chosen between 1 time unit and a specified maximum intertask arrival period. These 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 (respectively, 5 runs for the GA).

For the GA approach with R-GA and S-GA we maintained a population size of 40 over 60 generations for the R-GA and population size of 70 over 2000 generations for the S-GA. For a new generation 10 individuals of the old generation were replaced by new individuals. Mutation probability was 0.5 (with 1/3 chance for each mutation type in the R-GA).

Note that the computation times of the suggested methods (local repacking, ordered compaction, GA) are not included in the delays discussed in this section. One of the reasons for this is that such methods may be possible to be executed in the background [11].

Figure 7 compares the performance of the three approaches local repacking, ordered compaction and GA (combination of R-GA and S-GS) for a configuration delay of 0.001 time units per cell. The benefit of reallocating tasks was gauged by also examining the performance of the first fit allocation method [18], which does not move the tasks once placed. The results were obtained by varying the maximum intertask arrival period while the maximum task side length and configuration delay per cell were kept fixed.

Figure 7(a) shows the effect of varying intertask arrival period on the mean allocation

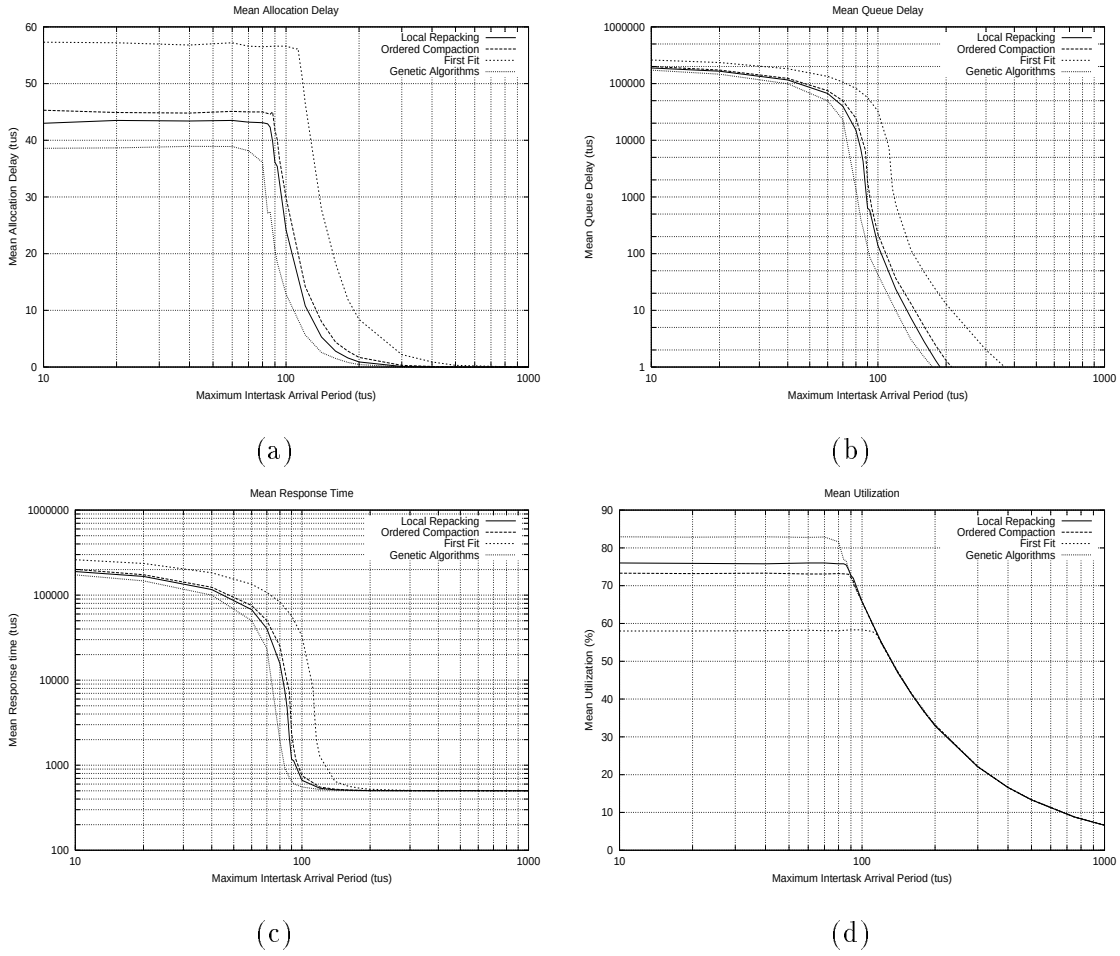


Figure 7: Effect of varying the task load on allocation performance. (a) Mean allocation delay, (b) mean queue delay, (c) mean response time, (d) mean utilization.

delay, i.e. the time between the allocation start and load start. In the left part of the curves the FPGA was saturated while tasks arrived faster than they could be allocated. However, performance differences between the methods are caused by their differing abilities to make or find space for the new task waiting at the head of the queue. Comparing the different approaches, the GA is in the lead, followed by local repacking, ordered compaction and first fit. The corresponding values of the mean allocation delay are approximately 38.6, 43.5, 44.9, and 57.2. The benefit of partially rearranging the tasks placed on the FPGA disappeared when tasks arrived infrequently enough for them to be accommodated immediately and the FPGA came out of saturation. The rearrangement heuristics enter the unsaturated region earlier than first fit. For instance, the mean allocation delay falls below one time unit at maximum inter–task arrival periods of approximately 175, 200, 300, and 400 time units for GA, local repacking, ordered compaction and first fit respectively. Finally, no rearrangements are necessary because tasks can be loaded as soon as they reach the head of the queue.

Figure 7(b) shows the effect of varying task load on mean queue delay, i.e. the time between task arrival at the tail of the queue and allocation start. Figure 7(c) shows the effect of varying task load on mean response delay, i.e. the time between task arrival at the tail of the queue and execution stop. Both figures are similar to figure 7(a) with the exception that the curves are not constant in the saturation region, since queue delays decrease as intertask arrival times rise.

In Figure 7(d) the mean utilization is measured for different task load levels. In the saturated region the GA is superior with a constant utilization level of approximately 82.3% followed by local repacking with 75.9%, ordered compaction with 73.2% and first fit with only 58.0%. Since the rearrangement approaches leave the saturated region earlier, their utilization values drop earlier but they always remain at least as high as the values of first fit.

In Figure 8 the effect of a varying configuration delay per cell from 0.004 to 2.2 on the allocation and execution delays are shown. The range of values corresponds to increasing the mean configuration delay per task from approximately 1 to 600 time units. Two

different levels of system load were examined. At a maximum inter–task arrival period of 40 time units the system has to cope with a heavy load, whereas a maximum inter–task arrival period of 120 time units corresponds to a system leaving the saturated region.

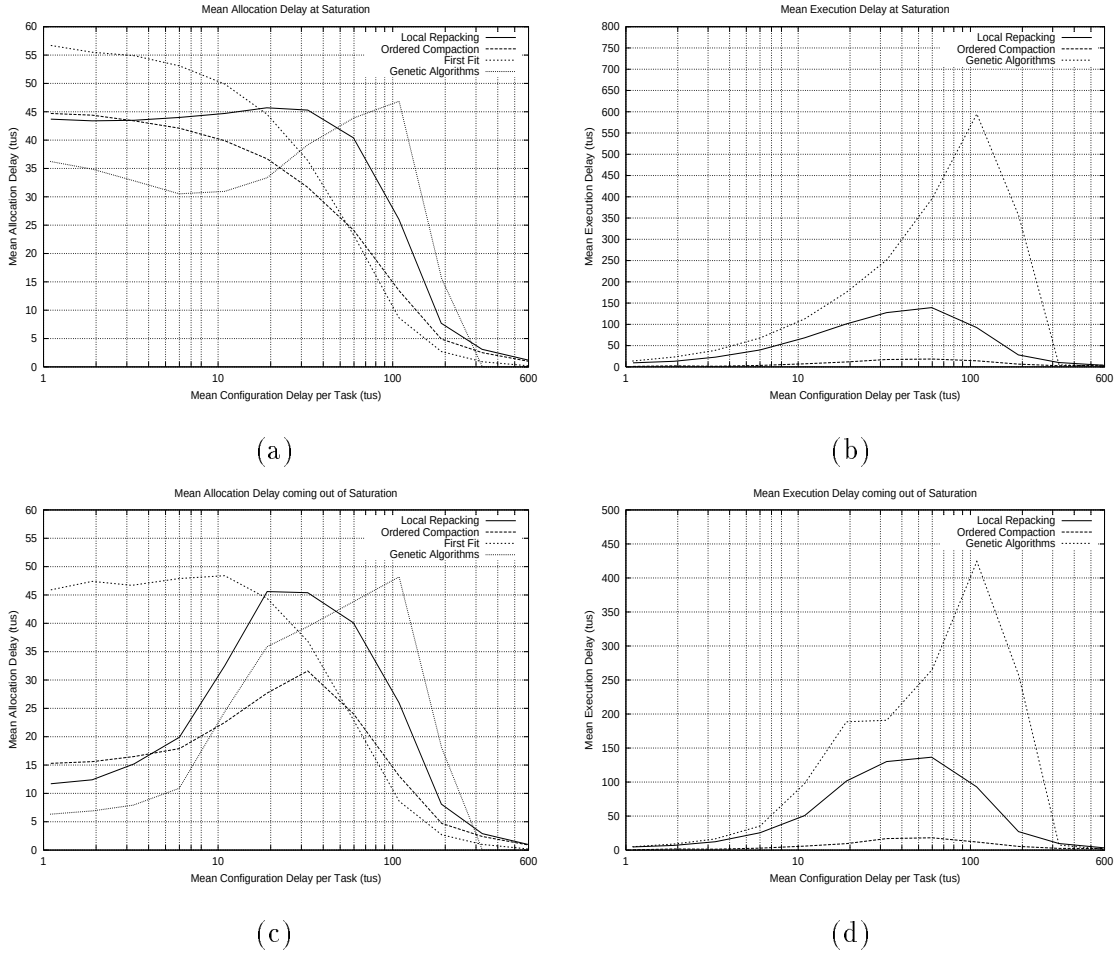


Figure 8: Effect of mean configuration delay. (a) Mean allocation delay at saturation, (b) mean execution delay at saturation, (c) mean allocation delay of a system coming out of saturation, (d) mean execution delay of a system coming out of saturation.

Figure 8(a) shows that at saturation the heuristics start with mean allocation delays which are similar to the values in Figure 7(a). As the mean configuration delay increases, ordered compaction achieves better allocation delays than local repacking. Later GA reaches a local optimum at the mean configuration delay of approximately 7 time units per task. This minimum is yet to be explained. At the beginning GA is superior. At mean configuration delays of 13, 20 and 50 time units per task it commences performing worse

than ordered compaction, first fit and local repacking respectively. The global maximum is encountered at a mean configuration delay of approximately 120 time units per task. From a mean configuration delay longer than 55 time units per task first fit performs better than all arrangement approaches. This behaviour can be explained as follows:

- The longer the mean configuration delay, the more tasks can finish their executions while a task is loaded, and before the next task is tried to be allocated. Consequently, it is more likely that first fit finds an allocation site for the new task.

- Partial rearrangement is used if first fit fails. But while executing a rearrangement, the I/O ports of the FPGA are blocked, i.e. the next request can not be satisfied. The longer the mean configuration delay, the longer the I/O ports are blocked. While the rearrangement takes place, some tasks might have regularly finished their executions, thereby freeing enough space for the next task. In other words, rearrangement retards the allocation of new tasks, whereas first fit could find an allocation site earlier.

As a consequence, the use of partial rearrangement should be restricted to applications where the mean configuration delays are low compared to the execution times. However, from a configuration delay of 320 time units per task the GA shows a very strange behaviour. The values of the mean allocation delay fall below the corresponding values of first fit. Finally, they drop to 0 time units.

In Figure 8(b) the mean execution delays of ordered compaction and local repacking increase until a mean configuration delay of 60 time units per task is reached. The mean execution delay of the GA approaches its global maximum at 120 time units per task. After the global maxima are passed the mean execution delays drop with an increasing mean configuration delay. Two factors might explain the shape of the curves:

- A rise in the mean configuration delay leads to an increasing execution delay, because rearrangements take longer.

- As the mean configuration delay increases, more and more tasks can be allocated

without allocation delays. Consequently, less tasks are rearranged and the mean execution delay falls.

Just as the mean allocation delay, the mean execution delay for the GA drops to 0 time units at a configuration delay of 320 time units per task. For first fit no curve is shown, since no tasks are rearranged by this approach. On average ordered compaction performs significantly better than local repacking. Local repacking in turn performs significantly better than the GA. The higher the mean execution delay of an algorithm the more often a task is moved, or larger tasks are more likely to be rearranged. Another factor is the chosen scheduling method and the objective of the scheduling problem. Since the S–GA minimizes the mean execution delay, the large gap between local repacking and the GA is more likely to be caused by a higher amount of rearranged cells.

Figure 8(c) shows the mean allocation delay of a system coming out of saturation. In contrast to first fit, the curves for local repacking, ordered compaction and the GA obviously changed their shapes. The crossing points of first fit and the rearrangement algorithms are approximately the same as before (cmp. Figure 8(a)). Therefore, the mean configuration delay at which first fit becomes better is likely to be independent of the system–load level. The GA performs best of all at configuration delays up to 8 time units per task and at configuration delays greater than 320 time units.

The similarities between the plots in figures 8(b) and (d) are noticeable. But the mean execution delays in the medium–load situation are slightly smaller than the values at saturation. At saturation more tasks are rearranged causing a rise in the mean execution delay.

6 Concluding remarks

When tasks arrive more quickly than they can be processed, partial rearrangements can reduce queue delays significantly. As a consequence, tasks are completed earlier, the utilization of the hardware is improved, and the system is more resilient to saturation. Current FPGA technology supports task movement by reconfiguration. When the mean

time to reconfigure a task is small compared to the mean processing time, this approach is adequate.

Areas for further investigation include elucidating the hardware support necessary for on–chip task movements, developing algorithms for arbitrary on–chip task rearrangements, designing algorithms that avoid relocating tasks too often, and developing techniques for decentralized or autonomous garbage collection to further reduce overheads.

References

[1] Atmel: 'AT6000 FPGA Configuration Guide.' Document 0436B, Atmel, 1997

[2] Brebner, G.: 'A Virtual Hardware Operating System for the Xilinx XC6200', Hartenstein, R.W. and Glesner, M. (Eds.), Proc. 6th International Workshop on Field–Programmable Logic (FPL'96), LNCS 1142, Springer–Verlag, Berlin, Germany, 1996, pp. 327 – 336

[3] Diessel, O. and ElGindy, H.: 'Run–Time Compaction of FPGA Designs', Luk, W. and Cheung, P.Y.K. and Glesner, M. (Eds.), Proc. 7th International Workshop on Field–Programmable Logic and Applications (FPL'97), Springer–Verlag, Berlin, Germany, 1997, pp. 131 – 140

[4] Diessel, O. and ElGindy, H.: 'Partial FPGA Rearrangement by Local Repacking.' Technical Report 97–08, Department of Computer Science and Software Engineering, The University of Newcastle, 1997

[5] Diessel, O.: 'On Scheduling Dynamic FPGA Reconfigurations — A Partial Rearrangement Approach.' PhD Dissertation, Department of Computer Science and Software Engineering, The University of Newcastle, 1998

[6] Dillien, P. and Phillips, I.: 'ASIC design flexibility with ERAs,' *Electronic Product Design*, 1989, **10** (10) pp. 29 – 34

[7] Eggers, H.; Lysaght, P.; Dick, H. and McGregor, G.: 'Fast Reconfigurable Crossbar Switching in FPGAs', Hartenstein, R.W. and Glesner, M. (Eds.), Proc. 6th Internatio-

nal Workshop on Field–Programmable Logic (FPL'96), LNCS 1142, Springer–Verlag, Berlin, Germany, 1996, pp. 297 – 306

[8] Eldredge, J.G. and Hutchings, B.L.: 'Density enhancement of a neural network using FPGAs and run–time reconfiguration', Buell, D.A. and Pocek, K.L. (Eds.), Proc. IEEE Workshop on FPGAs for Custom Computing Machines, IEEE Computer Society, Los Alamitos, CA, 1994, pp. 180 – 188

[9] Li, K. and Cheng, K.H.: 'Complexity of Resource Allocation and Job Scheduling Problems on Partitionable Mesh Connected Systems', Proceedings 1st IEEE Symposium on Parallel and Distributed Processing, IEEE Computer Society, Los Alamitos, CA, 1989, pp.358 – 365

[10] Mühlenbein, H. and Voigt, H.-M.: 'Gene pool recombination in genetic algorithms', Osman, I.H. and Kelly, J.P. (Eds.), Proc. Metaheuristics Int. Conf., Kluwer Academic Publishers, Norwell, 1995

[11] Schmidt, B.: 'FPGA Task Arrangement with Genetic Algorithms.' Diploma thesis, Institute of Applied Computer Science and Formal Description Methods, University of Karlsruhe, 1999

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

[13] Teich, M.; Fekete, S. and Schepers, J.: 'Compile-Time Optimization of Dynamic Hardware Reconfigurations', Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), Las Vegas, U.S.A., 1999

[14] Villasenor, J.; Jones, C. and Schoner, B.: 'Video Communications Using Rapidly Reconfigurable Hardware,' *IEEE Transactions on Circuits and Systems for Video Technology*, 1995, **5** (6), pp. 565 – 567

[15] Vuillemin, J.E.; Bertin, P.; Roncin, D.; Shand, M.; Touati, H.H. and Boucard, P.: 'Programmable Active Memories: Reconfigurable Systems Come of Age,' *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1996, **4** (1), pp. 56 – 69

[16] Wirthlin, M.J. and Hutchings, B.L.: 'Sequencing Run–Time Reconfigured Hardware with Software', ACM Fourth International Symposium on Field Programmable Gate Arrays (FPGA'96), ACM Press, New York, NY, 1996, pp. 122 – 128

[17] Xilinx: 'XC6200 Field Programmable Gate Arrays.' Xilinx, Inc., 1997

[18] Zhu, Y.: 'Efficient Processor Allocation Strategies for Mesh–Connected Parallel Computers,' *Journal of Parallel and Distributed Computing*, 1992, **16** (4), pp. 328 – 337