Partial Rearrangements of Space–shared FPGAs (Extended Abstract)

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

Abstract. Dynamically reconfigurable field–programmable gate arrays (FPGAs) appear to be highly suited to embedded high performance computing applications. They naturally support array–based concurrent computations, which are often needed to meet performance requirements; applications can be rapidly prototyped, thereby reducing the time to market; and, perhaps most interestingly, they can be reconfigured in system to reduce total hardware requirements, or to support new applications in the future.
How to use a single FPGA chip or system to support multiple simultaneous real–time circuits is currently under investigation. Such systems should execute arbitrary sequences of temporally and spatially varying tasks without degrading performance. Unfortunately, on–line allocation is by nature sub–optimal. Resources consequently become fragmented, which possibly delays reconfigurations and thus reduces utilization. We propose rearranging a subset of the executing tasks to alleviate this problem. We describe and evaluate methods for overcoming the NP–hard problems of identifying feasible rearrangements and scheduling the rearrangements when 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 normally.

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, and memory. Embedded applications that have successfully exploited this feature to conserve hardware include an image processing system, a reconfigurable crossbar switch, and a postscript driver. Successful designs for cryptographic applications, video communications, and neural computing, attest to the suitability of the architecture for high performance array–based computations. An

additional attraction of FPGAs is that they facilitate rapid prototyping by allowing designs to be conceived, implemented, tested, and modified under one roof.

As more ambitious systems are developed, it is conceivable that it becomes possible and desirable for related or even disparate embedded functions to share a single hardware platform. 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 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 would allow 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. We describe two methods by which feasible rearrangements, ones that allow the waiting task to be accommodated as well, may be identified. We examine the cost of rearranging a set of tasks by reloading their configuration bit streams, and present techniques for scheduling the task movements so as to minimize delays to the moving tasks. The complexity and performance of the methods are briefly reviewed before concluding with some final remarks.

2 The techniques

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 tasks so as to minimize the delays to executing tasks. 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 problem of identifying the best rearrangement is thus linked by feedback to the problem of scheduling the rearrangement.

The following assumptions are made. Tasks 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. The time to load or configure a task is assumed to be proportional to its area.

2.1 Identifying feasible rearrangements

The problem of deciding whether or not a waiting task can be accommodated on an FPGA is NP–complete [3]. Heuristic solutions are therefore sought. In the following, two solutions, which we call local repacking and ordered compaction, are presented.

Local Repacking The local repacking method [1] 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 having the potential to accommodate the waiting task because they contain sufficient free cells in total. A depth–first search of the tree allows promising sub–arrays to be identified and evaluated. A repacking of those tasks both partially and wholly contained within the sub–array is then attempted using a two–dimensional strip–packing method with good absolute performance bounds [4]. If the resulting packing represents a feasible rearrangement of the tasks, movement of the tasks can be scheduled in order to evaluate the cost of the rearrangement.

Ordered compaction The ordered compaction heuristic [2] places the waiting task at a favourable location, and moves those tasks that initially occupy the site off to one side. Ordered compaction therefore has the effect of moving the executing tasks that are to be compacted closer together while preserving their relative order. Without loss of generality, consider ordered compaction to the right. It can be shown that in order to minimize the time to complete a compaction it is best to 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. The feasibility of a site can then be decided by searching a visibility graph that is defined over the executing tasks. An optimal site minimizes the total area of moving tasks.

2.2 Scheduling task rearrangements

Since the time to reload an individual task is proportional to its area, the choice of tasks to move fixes the time needed to complete the rearrangement. We assume a task may continue executing until it is suspended prior to moving. The task is then resumed as soon as it has been reloaded. If its destination is not free when it is moved, the tasks initially occupying the destination are immediately suspended and removed. 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 scheduling delay that is to be minimized for all tasks. The problem

of scheduling FPGA task rearrangements to realize this goal is NP–complete [1]. Further heuristics are therefore needed. We first describe an approximation algorithm for scheduling rearrangements with arbitrary overlaps between the initial and final arrangements. Then we describe a method that does not delay the moving tasks more than the minimum if they are to be orderly compacted.

Arbitrary rearrangements The problem of optimally scheduling the tasks can be viewed as a search for an optimal path in a state–space tree. Each node represents the choice of 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 simple local cost estimator to determine which node to expand next can be used to find a near–optimal path. While the actual delay to tasks already moved is known, the delay to tasks that have not yet been moved is approximated by determining the maximum delay to the suspended tasks were they scheduled optimally and were they not to cause any additional suspensions. This method can be constrained to place the waiting task first of all.

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. 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.

3 Performance assessment

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. Ordered compaction, on the other hand, needs $O(n^3)$ time. Local repacking requires $O(n^3 \log n)$ time to produce a schedule, whereas an ordered compaction can be scheduled in $O(n)$ time.

An experimental assessment of the performance of the methods with simulated task sets indicates partial rearrangements are of significant benefit (allocation delays were reduced by up to 24%) when the mean task configuration delay is a small fraction (<1%) of the mean service period and when tasks arrive more quickly than they can be processed. See Figure 1. Local repacking appeared to be slightly better able to find feasible rearrangements than ordered compaction when tasks were small. This situation reversed as task sizes grew, indicating better packing methods are then needed. Both methods became ineffective with modest increases in the configuration delay. The benefits of local repacking were overwhelmed by delays to moving tasks at configuration delays of less than 5% of the service period. Since ordered compaction delayed moving tasks less, it was capable of sustaining a benefit at configuration delays as high as 10% of the service period.

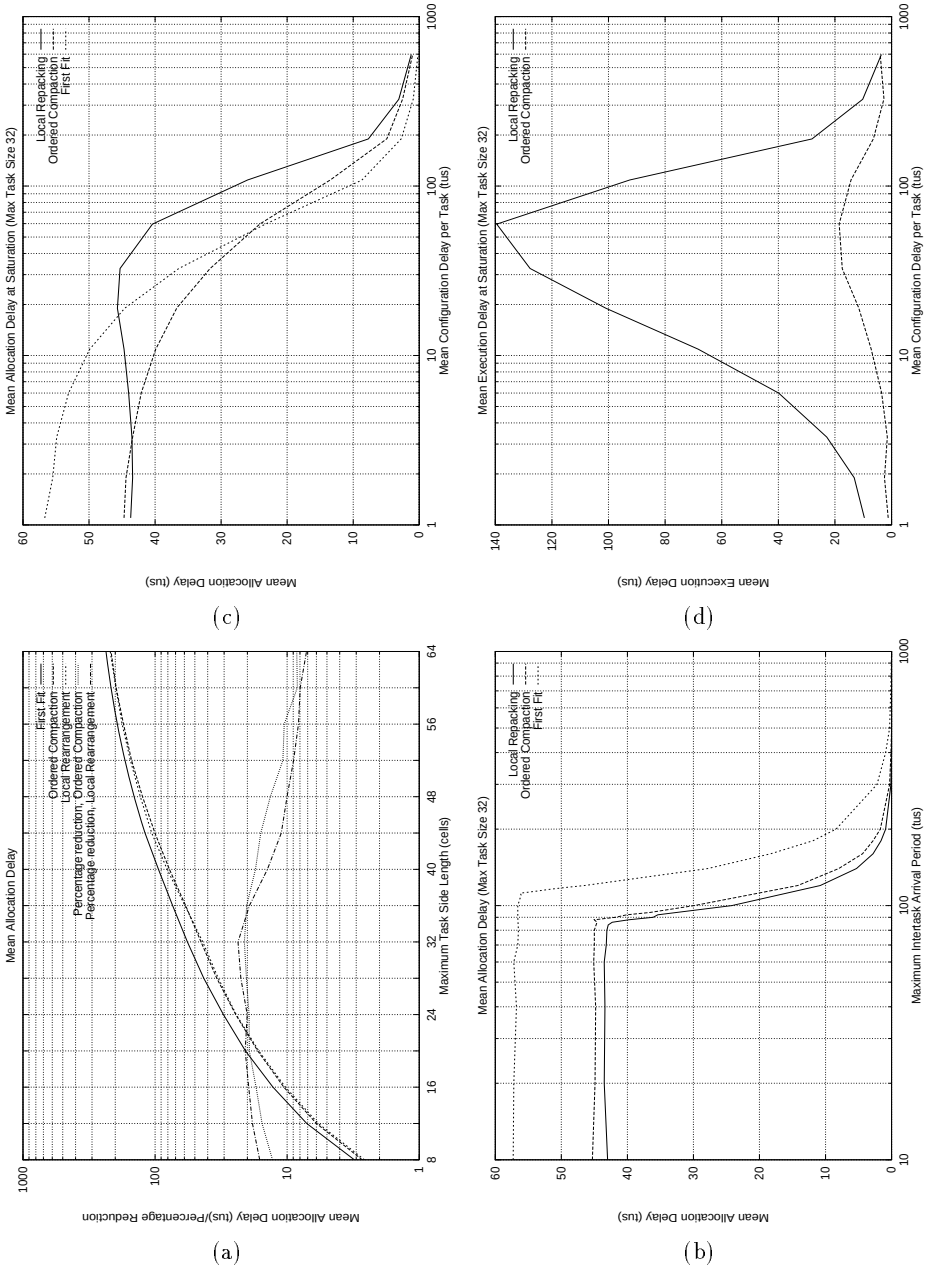

Fig. 1. (a) Effect of varying the maximum task size, (b) task load, and (c) configuration delay on allocation performance. (d) Effect of configuration delay on execution delays. Task side lengths, service periods, and intertask arrival periods were uniformly distributed with a minimum of 1. The maximum service period was fixed at 1,000 time units (tus). A 64×64 cell FPGA was simulated.

4 Concluding remarks

When tasks arrive more quickly than they can be processed, and the time to load a task is small compared to its processing time, 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. Of the two methods discussed, ordered compaction appears to perform more effectively as the cost of reloading tasks increases because the rearrangements can be scheduled with less delay to the executing tasks.

The techniques described above do not consider the service needs or deadlines of real–time tasks. However, if this information is known, it is not difficult to determine whether a rearrangement schedule allows individual tasks to meet their deadlines.

Areas for further investigation include designing an algorithm that avoids relocating tasks several times. Overcoming the I/O bottleneck of reloading tasks by moving them using on–chip resources is also under investigation. It is hoped that such techniques will facilitate decentralized or autonomous garbage collection to further reduce overheads.

References

1. O. Diessel and H. ElGindy. Partial FPGA rearrangement by local repacking. Technical report 97–08, Department of Computer Science and Software Engineering, The University of Newcastle, Sept. 1997. Available by anonymous ftp: `ftp.cs.newcastle.edu.au/pub/techreports/tr97-08.ps.Z`.
2. O. Diessel and H. ElGindy. Run–time compaction of FPGA designs. In W. Luk, P. Y. K. Cheung, and M. Glesner, editors, *Field–Programmable Logic and Applications, 7th International Workshop, FPL'97 Proceedings*, pages 131 – 140, Berlin, Germany, 1997. Springer–Verlag.
3. K. Li and K. H. Cheng. Complexity of resource allocation and job scheduling problems on partitionable mesh connected systems. In *Proceedings 1st IEEE Symposium on Parallel and Distributed Processing*, pages 358 – 365, Los Alamitos, Ca, 1989. IEEE Computer Society.
4. D. D. K. D. B. Sleator. A 2.5 times optimal algorithm for packing in two dimensions. *Information Processing Letters*, 10(1):37 – 40, Feb. 1980.

This article was processed using the LaTeX macro package with LLNCS style