On Scheduling Dynamic FPGA Reconfigurations

Oliver Diessel[1] and Hossam ElGindy[2]

[1] School of Computer and Information Science, University of South Australia, Mawson Lakes SA 5095, AUSTRALIA
[2] Department of Electrical and Computer Engineering, The University of Newcastle, Callaghan NSW 2308, AUSTRALIA

Abstract. The ability to partially reconfigure dynamically reconfigurable Field–Programmable Gate Arrays (FPGAs) at run–time allows them to 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. However, reloading tasks face an I/O bottleneck that must be eliminated if partial rearrangements are to be practical for short–lived tasks. Techniques for copying tasks to their destinations over on–chip routing resources are therefore developed and assessed.

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 paper appears in Kenneth A Hawick and Heath A James, editors. *Proceedings of the Fifth Australasian Conference on Parallel and Real–Time Systems (PART'98)*, pages 191 – 200, Singapore, 1998. Springer–Verlag.

this feature to conserve hardware include an image processing system [15], a reconfigurable crossbar switch [8], and a postscript driver [11]. Successful designs for cryptographic applications [14], video communications [13], and neural computing [9], 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, 7]. 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. 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 two methods by which feasible rearrangements, ones that allow the waiting task to be accommodated as well, may be identified. 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. Unfortunately, when several tasks are to be moved at once they face an I/O bottleneck reconfiguring. To overcome this problem, we propose making use of on–chip resources to move tasks in parallel. Although infeasible at present, we simulate pipelining task movements over nearest neighbour links in order to demonstrate the performance benefit compared with that of reloading. We conclude with a summary and directions for further investigation.

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 scheduling strategies employed depend upon the methods available to move the tasks. Thus the problem of identifying the best rearrangement is linked by feedback through the schedule to the underlying hardware and its capabilities. Current FPGA architectures allow tasks to be moved by reloading them. Simulation results indicate that significant reductions in allocation delays are possible when the FPGA is saturated with work and the time to load a task is relatively short. However, the reloading tasks face an I/O bottleneck that must be eliminated if partial rearrangements are to be practical for short–lived tasks. Techniques for copying tasks to their destinations over on–chip routing resources are seen as one way out of the dilemma.

The following assumptions are made. A space–shared dynamically reconfigurable FPGA is a rectangular array of configurable logic and routing resources that may be partitioned among multiple independent tasks [1, 16]. 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. We therefore do not examine the interesting problem of rerouting I/O to a task when it is moved.

2.1 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 [10]. Heuristic solutions are therefore sought. In the following, we present two solutions which we refer to as local repacking and ordered compaction.

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 having the potential to accommodate the waiting task by virtue of the total number of free cells they contain. 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 bin packing method with good absolute performance bounds [12]. 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 [5] 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 sliding 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 suffices 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.

2.2 Scheduling task rearrangements I — Reloading tasks

We assume the time to load or configure 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.

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. 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 [4]. 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 [1] 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 assuming they did not cause any additional suspensions when placed. This method can be constrained to place the waiting task first of all.

[1] Empirical evidence suggests maximum schedule delays incurred by the heuristic are usually less than twice those of optimal solutions [3]. Theoretical bounds have not yet been established.

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 [5]. 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.

2.3 Scheduling task rearrangements II — Moving tasks on–chip

The main drawback to performing task rearrangements by reloading is the I/O bottleneck at the chip boundary. Schedule delays are introduced because tasks are forced to be reloaded sequentially. Larger chip sizes or larger numbers or sizes of moving tasks would exacerbate the problem. We therefore consider moving the tasks on–chip as a possible method for overcoming this drawback. For example, the FPGA routing resource might be harnessed to move the configuration and state for multiple cells at a time and several tasks could use the available bisection width to move in parallel. Reductions in schedule delays to individual tasks and reductions in the schedule length could lead to additional performance gains.

To investigate this idea further we assume each FPGA logic element can store the configuration and state (collectively known as task element) for two tasks at a time. The time needed for a cell to switch contexts between task elements is assumed to be a constant and is assumed to be equivalent to the time needed to configure the cell. We assume the routing resource can be switched in constant time so as to form a circuit for routing a task element from its source to its destination. We consider a model in which the time to move a task element over a circuit from source to destination is a constant.

Unfortunately, the complexity of scheduling even linear array task movements over a one–dimensional array so as to minimize execution delays appears to be NP–hard [6]. Although we have found approximation algorithms with reasonable performance bounds for the ordered compaction of one–dimensional tasks, we have not yet found an effective algorithm for the more complicated problem of minimizing the delay to two–dimensional tasks when they are to be rearranged arbitrarily. However, the potential benefits of performing task movements on–chip are illustrated by the use of nearest neighbour links to pipeline the ordered compaction of FPGA tasks.

Ordered Compaction Given a set of tasks to be orderly compacted to the right, compaction over nearest neighbour links proceeds as follows. The tasks to be compacted are simultaneously halted and switched out of context. Cells containing task elements that are to be moved then send them to their right neighbours. Cells receiving a task element from the left check whether it has reached its destination and pass it onto the right if not. These steps are repeated until all task elements reach their destination. When a task arrives at its destination, the task elements are switched back into context to resume execution.

In order to implement the proposed method efficiently, several hardware enhancements to current FPGAs are required. First, a mechanism for efficiently

halting and resuming a subset of the executing tasks that does not affect the remaining tasks is needed. Second, the pipelining of task elements over nearest neighbour links needs to be supported. It should be possible to instruct the cells in specified regions of the FPGA to pass task elements from left to right, and to instruct them to stop doing so at the appropriate time.

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.

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 period 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.

Fig. 1 compares the performance of the local repacking and ordered compaction heuristics when tasks are moved by reloading. The benefit of reallocating tasks was gauged by also examining the performance of the first fit allocation method [17], which does not move the tasks once placed.

Fig. 1(a) shows the effect of varying the task load on the mean allocation delay. 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. In the left part of the graph 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 room for the task at the head of the queue. 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.

Fig. 1(b) depicts the effect on the mean allocation delay of varying the maximum task side length when the FPGA was saturated with work. The intertask arrival period was fixed at 1 time unit for this experiment. The benefit of partially rearranging the tasks under these conditions was found to be as high as 24%. The superior performance of local repacking when tasks are small reflects the benefit of collecting free space in two dimensions. The superiority of ordered

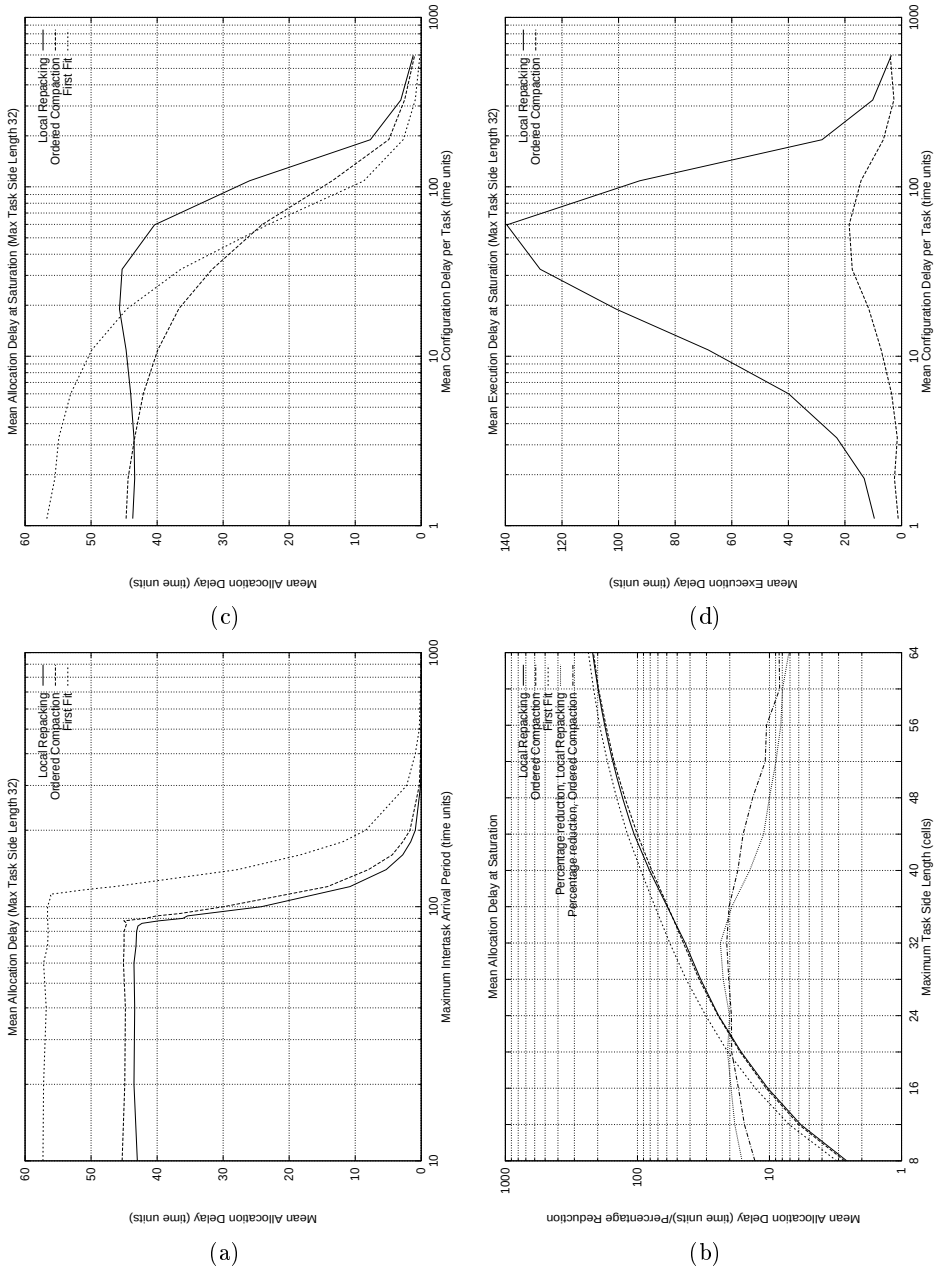

Fig. 1. Moving tasks reloaded. (a) Effect of varying the task load, (b) maximum task size, and (c) configuration delay on allocation performance. (d) Effect of configuration delay on execution delays.

compaction when tasks are large highlights the need for better two–dimensional packing heuristics.

In Fig. 1(c) the effect on the mean allocation delay of varying the configuration delay per cell is plotted. For this experiment, the maximum task side length was fixed at 32 cells and the intertask arrival period was fixed at 1 time unit. The plot shows that both methods become ineffective with modest increases in the configuration delay due to the I/O bottleneck when tasks are reloaded. An examination of the mean execution delay to tasks (Fig. 1(d)) indicates that 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 delays moving tasks less, it was capable of sustaining a benefit at configuration delays as high as 10% of the service period.

Fig. 2 illustrates the benefit of moving tasks over nearest neighbour links. In these experiments, the ordered compaction heuristic was used to find partial rearrangements of the tasks, which were executed by simulating the pipelining of task element movements over nearest neighbour links on the chip. The time needed to move a task element was considered to be equal to the time needed to configure a cell were it loaded from off the chip. Because this approach allows multiple task elements and indeed multiple tasks to be reconfigured simultaneously, schedule lengths were reduced and the mean execution delay to tasks became negligible (see Fig. 2(b)). The next rearrangement could therefore commence sooner, and space became available more frequently due to tasks finishing earlier. Interestingly, benefits over first fit were obtained at very high link delays.

4 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. However, the I/O bottleneck imposed by reconfiguration needs to be overcome for partial rearrangements to be practical for short–lived tasks. Moving the tasks on–chip results in improved performance but introduces additional scheduling complexity and the need for additional hardware support.

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, Aug. 1997.

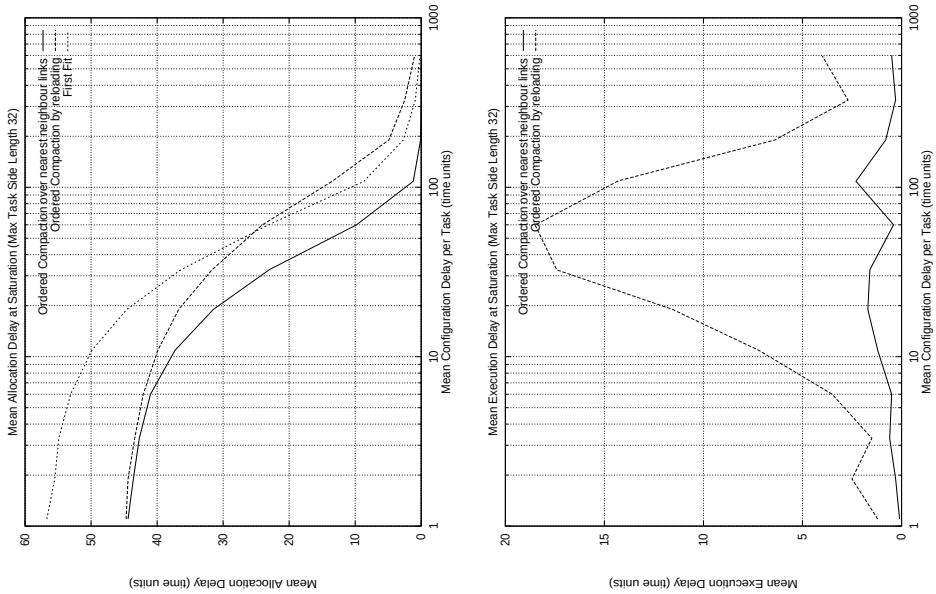

Fig. 2. Tasks moved over nearest neighbour links. (a) Effect of varying the link delay (equal to the configuration delay per cell) on allocation performance. (b) Effect of link delay on execution delays.

2. G. Brebner. A virtual hardware operating system for the Xilinx XC6200. In R. W. Hartenstein and M. Glesner, editors, *Field–Programmable Logic: Smart Applications, New Paradigms and Compilers, 6th International Workshop, FPL'96 Proceedings*, pages 327 – 336, Berlin, Germany, Sept. 1996. Springer–Verlag.
3. O. Diessel. *On Scheduling Dynamic FPGA Reconfigurations — A Partial Rearrangement Approach.* PhD thesis, Department of Computer Science and Software Engineering, The University of Newcastle, Jan. 1998. Available by anonymous ftp: `ftp.cs.newcastle.edu.au/pub/theses/phd/odiessel98.ps.Z`.
4. 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`.
5. 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.
6. O. Diessel, H. ElGindy, and B. Beresford-Smith. Partial task compaction reduces queuing delays in partitionable–array machines. In *Proceedings of the Third Australasian Conference on Parallel and Real–Time Systems*, pages 186 – 194, Brisbane, Australie, Sept. 1996. Griffith University.
7. P. Dillien and I. Phillips. ASIC design flexibility with ERAs. *Electronic Product Design*, 10(10):29 – 34, Oct. 1989.
8. H. Eggers, P. Lysaght, H. Dick, and G. McGregor. Fast reconfigurable crossbar switching in FPGAs. In R. W. Hartenstein and M. Glesner, editors, *Field–Programmable Logic. Smart Applications, New Paradigms and Compilers. 6th International Workshop, FPL'96 Proceedings*, pages 297 – 306, Berlin, Germany, 1996. Springer–Verlag.
9. J. G. Eldredge and B. L. Hutchings. Density enhancement of a neural network using FPGAs and run–time reconfiguration. In D. A. Buell and K. L. Pocek, editors, *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines*, pages 180 – 188, Los Alamitos, CA, 1994. IEEE Computer Society.
10. 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.
11. S. Singh, J. Patterson, J. Burns, and M. Dales. PostScript rendering with virtual hardware. In W. Luk, P. Y. K. Cheung, and M. Glesner, editors, *Field–Programmable Logic and Applications, 7th International Workshop, FPL'97 Proceedings*, pages 428 – 437, Berlin, Germany, 1997. Springer–Verlag.
12. 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.
13. J. Villasenor, C. Jones, and B. Schoner. Video communications using rapidly reconfigurable hardware. *IEEE Transactions on Circuits and Systems for Video Technology*, 5(6):565 – 567, Dec. 1995.
14. J. E. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. H. Touati, and P. Boucard. Programmable active memories: Reconfigurable systems come of age. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 4(1):56 – 69, Mar. 1996.
15. M. J. Wirthlin and B. L. Hutchings. Sequencing run–time reconfigured hardware with software. In *FPGA'96 1996 ACM Fourth International Symposium on Field Programmable Gate Arrays*, pages 122 – 128, New York, NY, Feb. 1996. ACM Press.

16. Xilinx. XC6200 Field Programmable Gate Arrays. Technical report, Xilinx, Inc., Apr. 1997.
17. Y. Zhu. Efficient processor allocation strategies for mesh–connected parallel computers. *Journal of Parallel and Distributed Computing*, 16(4):328 – 337, Dec. 1992.