Opportunities for Operating Systems Research in Reconfigurable Computing

Oliver Diessel and Grant Wigley
Advanced Computing Research Centre
School of Computer and Information Science
University of South Australia
Mawson Lakes SA 5095
{o.diessel,g.wigley}@unisa.edu.au

Abstract

Reconfigurable computing involves adapting hardware resources to the specific needs of applications in order to obtain performance benefits. This emerging architectural paradigm holds some promise for delivering significant speedups to compute–bound applications. However, many challenges need to be overcome before reconfigurable computing becomes mainstream. Significantly, supporting applications design and providing a convenient run–time environment create opportunities to propose and investigate new ways of managing chip resources. In examining the potential benefits of providing operating systems support for reconfigurable processors, this paper identifies opportunities for the development of policies for dynamic hardware management and strategies for communicating design ideas to the run–time management system.

1 Introduction

Reconfigurable computers typically consist of a workstation host loosely coupled via a general purpose bus to a reconfigurable logic processor, currently a Field Programmable Gate Array (FPGA). Current design methods result in the production of one or more FPGA configurations together with host software to access and control the FPGA. The host operating system typically provides no more than device driver support for accessing the reconfigurable logic processor. Although this approach allows raw access to the device for maximum performance, the approach is unproductive since designers need to be intimately familiar with low–level details of the reconfigurable device that, for lack of suitable abstractions, influence the design process at all levels. In addition, portability of the application is severely limited.

Much of the work on the run–time management of reconfigurable systems considers the problem of managing the time evolution of a single application. However, there is increasing interest in using reconfigurable computers for horizontally and vertically integrated application domains such as image and signal processing. With this shift, it is important that run–time management of the resource be consistent, efficient, and fair. We therefore believe it is timely to consider the possibility of providing operating systems support for reconfigurable logic.

Operating systems support for reconfigurable logic becomes necessary when one desires a higher level interface to the resource, as might occur when many different circuits are to be instantiated over time or the users do not wish to deal with low–level architectural details. Run–time support is also necessary when one requires a virtual or general model of a FPGA that is to be independent of the circuits that are to be instantiated on it. When it is necessary to share FPGA resources among multiple tasks that may be unrelated, owned by different clients, and simultaneously active, the allocation and scheduling of resources needs to be handled by an independent and fair agent. Support is also needed when the resource a client wishes to use is not under its direct control, such as when it is remotely networked.

There are several arguments against developing and implementing operating systems support for FPGAs. The first is that it would necessarily impact on performance. For a technology whose present applications niche is relatively small, the cost of a more usable and secure resource may be so high as to make use of FPGAs uneconomical. Second, there may very well be a concomitant loss in flexibility, and designers who desire complete control over the functionality of a device may loose out. Loss of some flexibility may also impact upon the feasibility of implementing some applications on FPGAs. However, the loss in performance and flexibility are compensated for in many ways.

The potential benefits of providing operating systems support for FPGAs include:

- greater convenience through abstraction, virtualization, and generalization (whether the resource is embedded, attached, or integrated with microprocessors),
- support for multi–tasking,
- improved system performance and fault tolerance, and
- integration of reconfigurable devices into the global computational pool.

Obtaining such benefits may necessitate the redesign of traditional electronic design automation and synthesis tools, the development of scheduling systems that will allow resources to be shared in space and time, and the development of new FPGA architectures that will allow high level functions such as task placement, repartitioning and migration to be handled by the chip itself.

We aim to clarify the role of operating systems as we see them impacting on reconfigurable computing systems of today and the medium term future. To this end, we first provide an overview of current reconfigurable computing technology before describing how operating systems may affect and support the life–cycle of a single task on a single user system. This section is also intended to familiarize the reader with the main challenges faced by the FPGA applications designer. In many ways it is precisely these activities that an operating system would ideally enhance. We then expand our view to include shared systems that are intended to support multiple concurrently executing tasks. The presentation initially focuses upon issues relating to the management of tasks and then shifts to the management of the resource. We conclude by summarising what we believe to be interesting and challenging areas for further research identified in the paper.

2 Reconfigurable Computing Architecture

A reconfigurable computer contains configurable logic resources, commonly field–programmable gate array (FPGA) chips, that may be adapted to process certain classes of computations at high speed. FPGAs are two–dimensional grids of configurable logic cells that are capable of implementing limited logic functions and are interconnected by a programmable routing matrix. High performance is achieved by embedding application specific digital circuits in the logic cell matrix of the FPGA, thereby allowing parallel execution streams and eliminating the need to fetch and decode instructions at run time. While not as fast as application specific integrated circuits, FPGAs and reconfigurable computers have the advantage of being reprogrammable, some within microseconds.

Several reconfigurable computing models have been proposed and reached various stages of development. The earliest and most successful of these views the reconfigurable logic as a coprocessor, and is often implemented as a board attached to the system bus of a sequential host processor (see, for example, [21]). This approach has the advantage of simplicity, but it suffers from low bandwidth between the host and coprocessor. It also introduces the so–called co–design problem — deciding which part of the application to implement in hardware (on the coprocessor), and which part to do in software (on the host). To overcome these problems, there have been several efforts to integrate configurable logic arrays onto conventional processor dies [22, 11]. There is also some interest in the development of systems composed entirely of reconfigurable logic and memory [5, 17].

A variety of FPGA models have also been developed. One of the most significant recent developments is the ability to partially configure a portion of an FPGA chip while allowing the remainder of the chip to continue to operate without interruption [23]. Along with the concept of reusing resources by swapping active portions of a circuit onto the device, this development has spawned interest in sharing a single chip among multiple tasks at the same time. An alternative approach with a similar goal is to provide the array with memory to store multiple contexts that can be switched in and out on a cycle by cycle basis through a context switching instruction [3].

A major stumbling block with reconfigurable computing is the difficulty of programming reconfigurable computers. We lack consistent and easy to use abstractions for describing reconfigurable computing applications, applications developers need to be highly skilled in digital circuit and parallel algorithms design, and the few tools available were initially developed for a much less dynamic design domain in which users were guaranteed exclusive access to chip resources. To complicate matters, application designers are often also required to manage their application's execution because reconfigurable logic resources are poorly integrated into host operating systems. Many coprocessing systems provide no more than a device driver with which to communicate with the board controller.

3 Designing FPGA tasks

The designer of FPGA–based applications is faced with many decisions. These include, but are not limited to, decisions about the method of specifying the application (design capture), where to allocate the parts of a computation (partitioning and placement), how to allow the parts to communicate (routing), how to reuse resources to compute the various parts (swapping), and how to schedule the use of those resources (control).

3.1 Design capture

The FPGA designer must contend with considerably more degrees of freedom than the conventional programmer. To begin with, designs (in this case circuits) may be specified using schematic capture, hardware description languages such as VHDL or Verilog, or a programming language. Each have their merits and their use is therefore commonly combined, but unfortunately none of them are effective for mapping applications to the limited hardware resource typical of current reconfigurable computers. Conventional design capture tools are not able to partition an application into circuit components that will fit onto the resource, or schedule the allocation and reconfiguration of resources so as to minimize execution delays. The traditional approach is to perform these functions for a given machine under the assumption that it will be dedicated to the application being designed and that run–time conditions are known and static. However, if the machine is to be shared among multiple tasks or users, or it is to be fault tolerant, then the application will need to cope with changing run–time conditions. This need forces the operating system to support activities that were traditionally handled off–line by the FPGA designer.

3.2 Partitioning

To illustrate this point, consider the partitioning problem. This arises in several ways. First, a designer must decide which part of an application is best allocated to an FPGA and what should sensibly be done in software. This is known as the hardware–software partitioning problem. It is not always obvious how this should be done, and it is conceivable that automatic techniques will do a better job than humans can when they are not fully informed of the available hardware. Typically, the designer must specify precisely which part is to be executed where, and then design the circuits for the hardware part in addition to the programs for the software part. In a networked environment the designer may not know what resources are available. Moreover they may change with time as network resources and run–time parameters change. It thus becomes desirable to allow the system to decide where to execute what and generate/execute the appropriate instructions.

A second type of partitioning that is often necessary is that of breaking a large task graph down into components that can fit onto available FPGA resources at the same time. Another approach is to partition the circuit so as to minimize the reconfiguration time [20]. Automated techniques are typically static and employed early in the design phase — certainly not at run time when conditions may necessitate repartitioning or resizing to optimize the use of available resources.

3.3 Placement

Operating systems may also have a role to play in task placement. Traditional FPGA design involves the use of automatic place and route tools to find a static placement of logic elements and routing of interconnections. In doing so, the designer fixes the location of the design during the construction process. This approach has the advantage of allowing the design to be loaded quickly, but it suffers from inflexibility should the region designated for the task be faulty, or occupied, or otherwise unavailable.

The disadvantage of hard placement can be overcome by binding the circuit to the hardware at run time. How much of the configuration data can be generated prior to load time, and how much needs to be done to place the task at run time are open questions. For example, if the FPGA chip is homogeneous and the task is not to be resized, a straightforward address translation should suffice. But if re-routing is necessary, perhaps because some resources are not available at the chosen location, the cost of run–time instantiation could increase considerably. In any case, the FPGA designer should not need to worry about designing the systems to support such flexibility. Moreover, when the FPGA is shared by multiple users, such services need to be controlled by the resource manager in the common interest of all users. This function must therefore be taken over by the operating system.

3.4 Routing

With arbitrary run–time task placement comes the problem of run–time routing. Run–time routing may not be needed within the task, as noted above. However, I/O may need to be routed to the border of the chip or to dependent tasks located elsewhere on the chip. Dynamic routing algorithms that allow communications to be re–routed as tasks are placed and removed can also provide fault tolerance. Such techniques are not needed while FPGA systems are designed with static task placement into known environments.

3.5 Scheduling

The problems of partitioning, placing and routing the task components are closely related to the problem of scheduling the task graph. After specifying the resource requirements and interdependencies of task graph nodes, an FPGA designer or automated design system attempts to allocate and partition the potentially large graph onto a limited physical resource. The partitions then need to be scheduled in order to correctly complete the computation in minimum time. If the FPGA is partially reconfigurable, the amount of commonality between partition configurations may further influence the reconfiguration time and thus the desirable order in which reconfigurations should be performed. The FPGA scheduling of arbitrary computations is therefore very complex and should be supported as much as possible by the tools and operating systems.

3.6 Swapping

The swapping of task components onto and off an FPGA chip is usually handled by the designer who decides upon the conditions that give rise to reconfigurations. This is acceptable when designs are static, but if we have the operating system participating in the dynamic partitioning, allocation, and scheduling of tasks then it is also desirable that it provide a means for task components to inform it when they are done, and which other components they would like to have loaded to complete their processing. These ideas follow the principles of virtual memory handling, but new mechanisms are required to handle circuit objects.

Many reasons exist for needing new mechanisms to handle circuit objects like pages in a virtual memory system. First, there is a need to know which partitions to invoke and in what order, depending upon run–time conditions. Second, it may be necessary to store state if the current partition is to be reinvoked. How to do this efficiently needs to be determined. Third, mechanisms may need to be provided to spool data between partition invocations. Fourth, run time conditions may prevent partitions from being loaded into suitable locations, so how should exceptions be handled? In addition, the operating system must be able to swap tasks onto the FPGA without unduly affecting the other concurrently executing tasks on the FPGA. The differing resource needs for each partition also need to be allocated and scheduled before loading the design onto the FPGA surface.

4 Multi–tasking FPGAs

Many reconfigurable computing models include the possibility of having multiple tasks. With FPGA chips becoming larger and with ever more resource potentially lying idle, there is a need to maintain a high level of utilization of the chips by sharing the resource in both space and time dimensions. These techniques are currently supported in hardware by partial reconfigurability and context switching respectively.

In this section we first examine the additional operating system functions needed to support the partitioning of FPGA resources among multiple tasks that are possibly unrelated and owned by different users. Then we consider the role of the operating system from the perspective of the FPGA resource when multiple tasks are concurrently active.

4.1 Task sizing

Before designs can be partitioned, a few issues relating to sizing need to be addressed. It is necessary to decide whether to fix the size of the partitions as this will directly impact on the partitioning and scheduling strategies. If the partition size is chosen to be fixed there will be a tradeoff between the size and the performance of the partition. If the partition size is too small, then the designs may not easily be partitioned into them. If variable partition sizes are permitted, the operating system may be able to resize a task to fit into a region of unallocated space or to allow other critical tasks the necessary resources to successfully complete. This method will maximize the space utilization of the FPGA at the cost of scheduling complexity and performance loss.

While some methods for parametric (based on aspect ratio) instantiation of FPGA tasks are known (see, for example, [12, 15]), these are static and performed at load time at the latest. Methods for dynamically resizing such tasks once they have begun have not been investigated. A framework for resizing tasks should identify which tasks can be resized at run time as well as the impact on their performance.

4.2 Task placement

In a multi–tasking system, task placement must be handled by the operating system since only it knows the location of free cells.

Constraining partition sizes to be fixed simplifies the partitioning problem, provides location independence, and improves the overall system throughput since any task partition can be loaded wherever one is removed.

If task partitions are a fixed size the placement strategy is simplified since partitions can be paged onto the FPGA surface. It suffices to provide a wiring harness for I/O to each page and to maintain an allocation table for managing the free pages [2, 16]. Tasks may need to communicate with each other, thereby making it desirable to co–locate them in order to reduce communication delays, and the lengths of routes required. The wiring harness provided to support I/O to fixed size partitions can be designed to support arbitrary simultaneous task to task communications.

Allowing task partitions to vary in size allows the internal fragmentation that occurs in fixed size partitions to be reduced at the cost of managing the free space, increasing the scheduling complexity, and solving a less constrained dynamic routing problem. It would be possible to adopt strategies employed in multiprocessors to reduce the fragmentation experienced with fixed partition sizes.

4.3 Location independence

It is desirable to provide location independence so that tasks may be placed wherever free space is available, and so that tasks can be moved, if necessary. Tasks might be moved to collect free space, provide fault tolerance, improve communications, and support load balancing in distributed, networked devices. With fixed partition sizes and a rigid task harness it should be easier to support location independence and task movements. FPGA architectures that support efficient task movements on chip are yet to be developed, however some suggestions have been advanced [17]. Managing the movement of variable sized tasks has been considered in [4].

4.4 Swapping, caching and pre–loading tasks

Multi–tasking may also be supported by time–sharing the FPGA resource. If so, it is necessary to decide how context switches are to be supported from an architectural as well as an operating systems perspective. Many FPGA devices support multiple contexts by providing on–chip memory for storing inactive configurations and state (see, for example, [3]) but time–sharing can also be managed in software by swapping contexts off–chip (see, for example, [10]). When multiple contexts are supported in hardware, the active context is usually selected by a globally broadcast context instruction. Overheads are small with this method, and the potential exists to pass data in place between the configurations. Pre–loading configurations into the area occupied by an executing task is also feasible.

4.5 Input and output

FPGAs supporting multiple tasks may need to handle multiple concurrent I/O streams. Since FPGAs have limited pin and routing resources, multiple concurrent communications may lead to contention that would need to be resolved by the operating system. In addition, with the operating system responsible for the run–time placement of tasks, it also needs to ensure I/O can be routed to the border and determine acceptable port locations for interfacing signals to off–chip wires.

The Virtual Wires project at MIT has developed techniques for time multiplexing the use of pins [1]. Internal wire resources are typically statically allocated and are at best space–shared by segmenting their lengths. It may be desirable to allow these to be multiplexed as well. One possibility is to provide redundant links and implement slightly more sophisticated switches that would allow simple message routing on an as needs basis. See, for example, the reconfigurable multiple bus (RMB) proposal [6].

System I/O resources need managing just as user I/O resources do. This need becomes evident when several tasks request reconfigurations that overlap in time. The chip configuration bus then needs to be shared.

4.6 Intertask communications

Tasks needing to communicate with each other constrain the operating system to place them favourably with respect to each other and find appropriate routes for signals to flow between them. These problems are harder to deal with when task sizes are not fixed and tasks are placed so as to achieve maximum packing density. The solution suggested by [17] is to provide a separate cellular automaton–based routing plane for providing autonomous operating systems functions such as intertask communications. With more coarse–grained allocation units at fixed locations, it should be possible to support total interconnection through the use of crossbar switches, or RMB networks.

4.7 Allocation and scheduling

The goals of the system need to be decided. Are they to maximize utilization subject to task deadlines, or is the response time of tasks to be minimized? Presumably the scheduling objectives will be determined by the applications area. However, the common thrust will be to make best use of limited array hardware (cells, wires, pins and memory) in meeting the computational objectives of multiple simultaneous tasks. The central question is: how should the resources be shared? Present day systems are either shared in space or time, whereas the optimum may involve a mixture of both approaches much like gang scheduling schemes for multiprocessors [7]. We may wish to pack multiple tasks into the one time slice (context) and pack different sets of tasks into subsequent time slices. The contexts might then be alternated to allow the FPGA resource to be used efficiently. Tasks that cannot co–exist within a slice because they would contend for resources can be separated into different slices in order to satisfy their competing needs. Benefits similar to those of multi–threading can be obtained when resources that are tied up by a task that is idle, waiting for I/O say, are employed for some of the waiting period to execute other tasks in different time slices [13].

Important issues to consider are: how to minimize the overheads, how to be fair, how to ensure that deadlines can be met, and how to ensure that no task suffers too many delays? The FPGA architecture will need to help the operating system support multiple simultaneous I/O streams. It may also have to support clocking individual tasks at different rates. There should also be ways of detecting and recovering from faults. And the systems should clean up discarded resources (provide garbage collection facilities). Effective abstractions for modelling the hardware are also needed — how does the hardware appear to the user and to the operating system?

4.8 Security

It is desirable that task configurations be safe, i.e., not impact on other tasks or compromise the functioning of the system. There is also the need to restrict access to acceptable users. These two requirements suggest operating a virtual machine environment similar to the Java Virtual Machine, which is being investigated by several researchers (see, for example, [14]). A user would submit hardware–independent configurations that could be checked for safety. This approach might also provide location independence, because the virtual machine would necessarily perform technology mapping.

5 Research Plans

In order to study and develop operating systems support for reconfigurable computing, we believe it is of fundamental importance to decide whether the partitioning policy allows for fixed or variably sized partitions. Not only does the answer to this question determine suitable abstractions for reconfigurable computing, it influences the formulation of partitioning strategies, of scheduling and placement policies, effective tools design, and appropriate reconfigurable computing architectures. We would like to know under what conditions one method is superior to the other; what are the factors that influence the outcome? Is there a relationship between architecture or application domain and the partitioning policy? Is it always possible to partition an application onto a given reconfigurable logic resource? Is it feasible to do partitioning at run time?

To answer these questions, we are investigating the tradeoffs and intend performing a cost benefit analysis on the SPACE.2 reconfigurable computing platform [8]. The relatively large FPGA array available to us allows us to experiment with both policies. Initial experimentation will focus on how to manage a shared FPGA array while a particular application's demand on contiguous area grows.

There are several directions in which future work may take us. The first direction attempts to answer questions about scheduling reconfigurable computing tasks. If partitions are fixed in size, we would like to know whether the problem of minimizing response time is equivalent to multiprocessor scheduling, and if so, can we use similar scheduling techniques? With respect to multitasking systems, we are interested in knowing how competition for scarce resources influences the problem, and what the effect of deadlines is. We would also be interested in knowing the impact of allowing partition sizes to vary.

To support configuration swapping, we are interested in how to inform the operating system when swaps are required and which configurations to swap? We are interested to know whether it is possible to eliminate the need for the application designer to specify a control algorithm.

If variable partition sizes are to be allowed, we would need to know how to support location independence. It would also be possible to investigate dynamic approaches to routing IO to the border or between communicating tasks.

Of overall interest is an answer to the question of how much computation can be done at compile time, and what can we afford to do at run time?

Acknowledgements

We thank David Kearney, Bernard Gunther, Jihan Zhu, and Hossam ElGindy for helpful discussions and comments on this work.

References

[1] Jonathan Babb, Russell Tessier, and Anant Agarwal. Virtual wires: Overcoming pin limitations in FPGA–based logic emulators. In Duncan A Buell and Kenneth L Pocek, editors, *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines*, pages 142 – 151, Los Alamitos, CA, April 1993. IEEE Computer Society Press.

[2] Gordon Brebner. The swappable logic unit: a paradigm for virtual hardware. In Pocek and Arnold [18], pages 77 – 86.

[3] Andre DeHon. DPGA–coupled microprocessors: Commodity ICs for the early 21st Century. In Duncan A Buell and Kenneth L Pocek, editors, *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines (FCCM'94)*, pages 31 – 39, Los Alamitos, CA, April 1994. IEEE Computer Society Press.

[4] Oliver Diessel and Hossam ElGindy. On scheduling dynamic FPGA reconfigurations. 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.

[5] Adam Donlin. Self modifying circuitry — A platform for tractable virtual circuitry. In Hartenstein and Keevallik [9], pages 199 – 208.

[6] Hossam ElGindy, Arun Somani, Heiko Schröder, Hartmut Schmeck, and Andrew Spray. RMB — A reconfigurable multiple bus network. In *Proceedings. Second International Symposium on High–Performance Computer Architecture*, pages 108 – 117, 1996. Available by anonymous ftp:
`ftp.cs.newcastle.edu.au/pub/reconfig/papers/rmb.ps`.

[7] Dror G Feitelson. Packing schemes for gang scheduling. In *IPPS'96 Workshop on Job Scheduling Strategies for Parallel Processing*, Los Alamitos, CA, April 1996. IEEE Computer Society Press.

[8] Bernard K Gunther. SPACE 2 as a reconfigurable stream processor. In Nalin Sharda and Audrey Tam, editors, *Proceedings of PART'97 the 4th Australasian Conference on Parallel and Real–Time Systems*, pages 286 – 297, Singapore, September 1997. Springer–Verlag.

[9] Reiner W Hartenstein and Andres Keevallik, editors. *Field–Programmable Logic and Applications, From FPGAs to Computing Paradigm, 8th International Workshop, FPL'98 Proceedings*, volume 1482 of *Lecture Notes in Computer Science*, Berlin, Germany, 1998. Springer–Verlag.

[10] Gunter Haug and Wolfgang Rosenstiel. Reconfigurable hardware as shared resource in multipurpose computers. In Hartenstein and Keevallik [9], pages 149 – 158.

[11] John R Hauser and John Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In Pocek and Arnold [18], pages 12 – 21.

[12] Jonathan Hogg, Satnam Singh, and Mary Sheeran. New HDL research challenges posed by dynamically reprogrammable hardware. In *Proceedings, Third Asia Pacific Conference on Hardware Description Languages (APCHDL96)*, 1996.

[13] Jack Jean, Karen Tomko, Vikram Yavagal, Robert Cook, and Jignesh Shah. Dynamic reconfiguration to support concurrent applications. In Pocek and Arnold [19], pages 302 – 303.

[14] Eric Lechner and Steven Guccione. The Java environment for reconfigurable computing. In Wayne Luk, Peter Y K Cheung, and Manfred Glesner, editors, *Field–Programmable Logic and Applications, 7th International Workshop, FPL'97 Proceedings*, volume 1304 of *Lecture Notes in Computer Science*, pages 284 – 293, Berlin, Germany, September 1997. Springer–Verlag.

[15] W Luk, S Guo, N Shirazi, and N Zhuang. A framework for developing parametrised FPGA libraries. In Reiner W Hartenstein and Manfred Glesner, editors, *Field–Programmable Logic: Smart Applications, New Paradigms and Compilers, 6th International Workshop, FPL'96 Proceedings*, volume 1142 of *Lecture Notes in Computer Science*, pages 24 – 33, Berlin, Germany, September 1996. Springer–Verlag.

[16] Pedro Merino, Juan Carlos López, and Margarida Jacome. A hardware operating system for dynamic reconfiguration of FPGAs. In Hartenstein and Keevallik [9], pages 431 – 435.

[17] Kouichi Nagami, Kiyoshi Oguri, Tsunemichi Shiozawa, Hideyuki Ito, and Ryusuke Konishi. Plastic cell architecture: Towards reconfigurable computing for general–purpose. In Pocek and Arnold [19], pages 68 – 77.

[18] Kenneth L Pocek and Jeffrey M Arnold, editors. *The 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'97)*, Los Alamitos, CA, April 1997. IEEE Computer Society Press.

[19] Kenneth L Pocek and Jeffrey M Arnold, editors. *The 6th Annual IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'98)*, Los Alamitos, CA, April 1998. IEEE Computer Society Press.

[20] Nabeel Shirazi, Wayne Luk, and Peter Y K Cheung. Automating production of run–time reconfigurable designs. In Pocek and Arnold [19], pages 147 – 156.

[21] Jean E Vuillemin, Patrice Bertin, Didier Roncin, Mark Shand, Herve H Touati, and Philippe Boucard. Programmable active memories: Reconfigurable systems come of age. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 4(1):56 – 69, March 1996.

[22] Ralph D Wittig and Paul Chow. OneChip: An FPGA processor with reconfigurable logic. In Kenneth L Pocek and Jeffrey M Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'96)*, pages 126 – 135, Los Alamitos, CA, 1996. IEEE Computer Society Press.

[23] Xilinx. *XC6200 Field Programmable Gate Arrays.* Xilinx, Inc., April 1997.