UNSW SYDNEY

Chip-based reconfigurable task management

Author/Contributor:
Brebner, Gordon; Diessel, Oliver

Publication details:
Field-Programmable Logic and Applications, 11th International Conference, FPL 2001 Proceedings
pp. 182-191
3540424997 (ISBN)

Event details:
11th International Conference on Field-Programmable Logic and Applications
Belfast

Publication Date:
2001

Publisher DOI:
http://dx.doi.org/10.1007/3-540-44687-7_19

License:
https://creativecommons.org/licenses/by-nc-nd/3.0/au/
Link to license to see what you are allowed to do with this resource.

Downloaded from http://hdl.handle.net/1959.4/39669 in https://unsworks.unsw.edu.au on 2023-02-08

Chip-based Reconfigurable Task Management

Gordon Brebner

Division of Informatics
University of Edinburgh
Mayfield Road
Edinburgh EH9 3JZ
United Kingdom

Oliver Diessel

School of Computer Science & Engineering
University of New South Wales
Sydney
NSW 2052
Australia

Abstract. Modularity is a key aspect of system design, particularly in the era of system-on-chip. Field-programmable logic (FPL), particularly with the rapid increase in programmable gate counts, is a natural medium to host run-time modularity, that is, a dynamically-varying ensemble of circuit modules. Prior research has presumed the use of an external processor to manage such an ensemble. In this paper, we consider on-chip management, implemented in the FPL itself, based upon a one-dimensional allocation model. We demonstrate an algorithm for on-chip identification of free FPL resource for modules, and an approach to on-chip rearrangement of modules. The latter includes a proposal for a realistic augmentation to existing FPGA reconfiguration architectures. The work represents a key demonstration of how FPL can be used as a first-order computational resource, rather than just as a slave to the microprocessor.

1 Introduction

Field-programmable logic (FPL) continues to grow in importance as a digital implementation medium in markets that are driven by diminishing lead times, a hunger for performance, rapidly changing standards and the need to be capable of rapidly differentiating or personalising products. It has become impractical to design whole systems from scratch, so a modular design approach is used instead. Common functions are identified and parameterized, and then a system is composed of many modules, some of them interacting, others operating quite independently. This methodology suits the nature of FPL which, as a general-purpose digital circuit implementation medium, is a natural platform for hosting the variety of circuit modules that comprise a system. Moreover, it is useful that only those modules that need to be active at some time need be configured on the reusable resource.

This paper appears in Gordon Brebner et al., editors, *Field-Programmable Logic and Applications, 11th International Conference, FPL 2001 Proceedings*, volume 2147 of Lecture Notes in Computer Science, pp. 182 – 191, Berlin, Germany, 2001. Springer-Verlag.

This motivates the need for allocation of the logic resource amongst the various modules and for a mechanism to recycle or reclaim and reallocate logic resources, as computational requirements dictate. It is also conceivable (although few examples exist so far) that FPL-based systems can be shared in time and/or space amongst several concurrent independent users or tasks.

Most SRAM-based field-programmable gate array devices are composed of a two-dimensional grid of configurable logic cells embedded in a hierarchical routing network. In general, this structure does not easily lend itself to a systematic, modular view of circuits that can be pursued both at design and execution time. Aside from basic wiring considerations, just the fact that the resource is two-dimensional is a major complication for run-time module management. One approach to handling this is to superimpose a one-dimensional management strategy. With such a (column-oriented) scheme, individual modules occupy the complete height of the FPGA but have a variable width. The regions of the array associated with different modules can be easily separated and managed independently. Several such one-dimensional or striped FPGA systems and devices have been proposed (for example, DISC [15], GARP [9], PipeWrench [13], even Xilinx Virtex chips). As new modules arrive, it is then simply necessary to find a region of sufficient free columns. I/O can generally be readily provided at pins adjacent to the tasks needing it at the top and bottom edges of the array, and inter-module communication can be provided by abutting modules or by routing signals via an interconnecting bus.

To obtain good utilisation from one-dimensional FPGA-based systems, effective and efficient allocation methods are needed to cope with arbitrary arrival and departure sequences for differently sized modules. Traditionally this is done by an allocator executing on an attached host [1–3]. In this paper, however, we discuss on-chip module management methods that do not require an external host, being implemented on the array itself. This eliminates a computational load and communication bottleneck from an auxiliary processor and also provides a decentralised means of scalably allocating FPGA resources.

In high-load applications or environments, utilisation can be improved by defragmenting or partially rearranging the executing modules to collect sufficient free columns to allocate to an incoming module. The results of previous experiments suggest that rearranging modules by moving them on-chip provides the greatest boost to utilisation by minimising reconfiguration overheads [6]. In this paper, we therefore present on-chip methods both for identifying space for arriving modules and also for compaction of the space used by existing modules. In the latter case, we suggest a specific and realistic architectural enhancement that facilitates the efficient on-chip movement of modules.

2 One-dimensional Task Model

We take a high-level view of the capability and operation of reconfigurable systems. Since we are interested not just in the static circuitry requirements of modules, but also their dynamic computational functions, we shall refer to *tasks*

from now on. There are various published examples of the use of tasks in various guises, and their management at run time. The desire to reallocate FPL resources during operation has a number of origins, including the desire to redistribute the components that are to be computed between hardware and software in response to performance objectives.

In this paper, we consider a system model that does not require a controlling host. Traditional host-based CCMs are not excluded from consideration — it is just that the host is no longer required for detailed task management. In a 'closed' system, for which the set of tasks and the possible sequences of operation are known, an optimal bespoke on-chip controller can be constructed to manage the task reconfigurations. This has been investigated in the past by several researchers [11, 14, for example]. We consider 'open' systems here, seeking a general dynamic solution to task management.

Since the approach is one-dimensional, each task will be allocated the entire height of the array, and as many contiguous columns as are needed to lay out the circuit. The array is thus partitioned into vertical slices of variable width corresponding to the various tasks, and so the model can be seen as one-dimensional with variable-width contiguous blocks either allocated to tasks or free.

The block of logic resource used by each task is composed of three parts, layered vertically, as illustrated in Fig. 1. Some of the logic associated with the

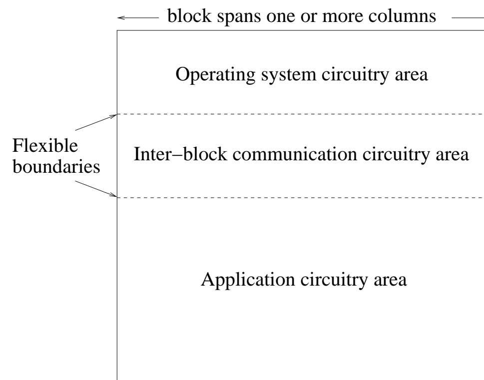

Fig. 1. Basic structure of a task block (in practice, application circuitry area would be much larger relative to the others)

task will be devoted to system functions such as assisting in the management of tasks. A second component is envisaged to be associated with communications between the task and other tasks or its environment (devices, host or network). The third, and typically most substantial, part comprises the computational circuitry for performing the task. As with all system overheads, there is a need to keep the space associated with the first two components small and to ensure that their operation does not reduce the performance of the task significantly.

Tasks are typically configured by loading a configuration bitstream onto the FPGA. Usually such streams are loaded sequentially so, short of exploiting compression techniques, the amount of data that must be loaded for a task is assumed to be proportional to the area of the task. In the one-dimensional model, the area of the task is proportional to its width.

As tasks arrive and leave, fragmentation of the free space occurs, and consolidation of the free space by rearranging the executing tasks is desirable. Host-based task rearrangement methods rely on moving executing tasks by reloading them at their new location, and methods that minimize the (additional) configuration delays to the moving tasks are therefore sought. Previous attempts have focused on finding feasible rearrangements that minimize the delay to individual tasks. Here, we describe a one-dimensional simplification of a two-dimensional method known as ordered compaction [5]. In this method, a subset of the executing tasks is squashed together towards one side of the array until there is no free space left between them. Their relative order is preserved, and a single free block is created adjacent to the compacted tasks. When carried out by an external host, each task is moved in sequence, with the reconfiguration of each task in turn carried out serially. Compton *et al* proposed a novel adaptation of the Xilinx 6200 series FPGA that allows a column (using our task arrangement) of configuration bits to be relocated in a constant number of steps [4]. Their structure supports the relocation of a single task under host control in time proportional to its width. We improve on this using an on-chip method that enables tasks to be moved in parallel, thereby reducing overheads and improving the efficacy of compaction.

3 On-Chip First-Fit Allocation

In this section, we present an on-chip implementation of a search algorithm that finds a free block of a required column width, if such a block is available. An integer value is input to the circuit, to specify the number of columns required, and an integer value is output from the circuit, indicating the column where a block was found or equal to zero if no block was found. The basic circuitry used is an inverted variant of the string pattern matching circuitry that has been studied by numerous researchers since the advent of programmable logic technology [10, 8, for example]. The variance arises from the fact that the pattern is shifted past the 'text', in this case, the pattern string representing a unary encoding of the column width required (e.g., 11111 for five columns), and the text string representing the current status of the array, having length equal to the array width, with 1's denoting used columns (occupied by tasks) and 0's denoting unused columns (free). In our design, the text string bits are not stored explicitly, instead being encoded by circuitry variants.

The matching circuitry is contained in the system part of each column of the array as illustrated in Fig. 2. The pattern string is shifted through this circuitry from the rightmost column to the leftmost column (the direction of the data flow being arbitrary, but in this case implying finding the first fit from the right-hand

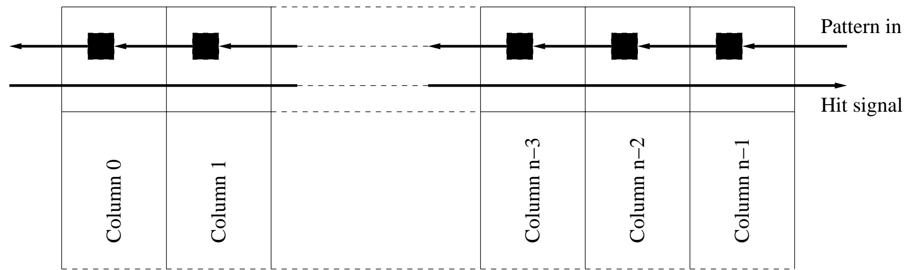

Fig. 2. Matching circuitry distributed over array columns

side of the array). A 'hit' signal line runs from left to right, the output value at the right being zero until a pattern match is detected, that is, a suitable free block has been detected. Each column's circuitry is dynamically configured with one of two layouts, depending on whether the column is used or unused, as shown in Fig. 3. The reconfiguration occurs when a block is allocated or deallocated.

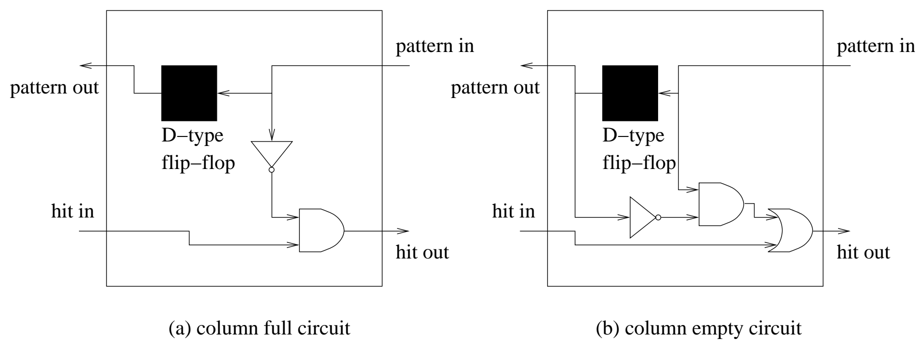

Fig. 3. Per-column matching circuitry variants

The operation of the pattern-matching circuitry involves a very simple control circuit, to be located in a system circuitry area immediately to the top-right of the array area available for allocation. This control circuit contains a counter wide enough to count to the maximum number of columns available for allocation (note that it is allowable for this maximum, and hence the size of the counter to be dynamically reconfigurable). The operation of the control circuit is as follows:

1. The counter is set to the number of columns sought as part of the search request.
2. A single zero is shifted into the pattern chain.
3. While the counter is non-zero, a one is shifted into the pattern chain, and the counter is decremented.
4. Decrement the counter.

5. While the hit line is zero **and** the counter is non-zero, a zero is shifted into the pattern chain and the counter is decremented.

On completion, if the hit line is zero, no available block has been found. Otherwise, the counter indicates the column number (labelled from zero at the left end of the array) of the rightmost column of the block found. The basic run time of this circuitry is $O(k+n)$, where k is the size of the required block, and n is the number of columns in the array.

Note that this control circuit is in the simplest form possible. To improve the running time, some optimisations could be made at the expense of more circuitry. First, Step 2. can be omitted if a zero was shifted in as the last pattern bit of the previous match. Second, Step 5. could be shortened, in the case where no match is found, if the counter is only decremented as far as the number of columns sought minus the maximum number of columns. (In fact, if the counter was equipped with testing for other than zero contents, Step 4 could be eliminated.) The run time of the optimised circuitry would be improved to $O(n)$.

However, we believe that it is appropriate to use the simplest possible circuit because the matching algorithm can run in parallel with both the task circuits present on the array and any agent requesting the search. In this sense, the run time is 'free', at the expense of a small amount of programmable logic resource.

4 On-Chip Ordered Compaction

The circuitry described in the previous section provides an efficient means of finding free blocks when they exist. In general, the total number of columns required might be available, but not in a contiguous block. This means that two enhancements are necessary. First, the matching process must be extended to find non-contiguous columns and to indicate where they are located. Second, a compaction process must be introduced to create a block of the required size. Due to limitations of space, we just outline the necessary extension to the matching process, since the more profound implications arise from the compaction process.

The basic modification to the matching process is that the pattern string is shortened by one bit each time it passes through an unused column. This can be implemented simply by having the circuitry at an unused column forward a zero rather than a one, just when it sees the leading one of the pattern (i.e., a one which was preceded by a zero). A hit will have occurred when the pattern has been reduced to a single bit, and enters an unused column. This causes a hit signal to be propagated back to the right-hand side. It indicates that there are sufficient free columns to form the required block size. The run time of this matching process is the same as for the contiguous case above.

Unlike the contiguous-space case, where the location of the free block can immediately be inferred when the hit signal is received, the situation is more complex here. While it is easy for an external controller to maintain a bitmap of the free and occupied columns, or for the array to provide one by shifting it out to the right, we consider a purely internal mechanism for free space compaction that is targeted at the underlying configuration mechanisms of typical FPGAs.

Fig. 4 shows an example of what might be required, where three free sub-blocks must be combined to produce a free block. The essential operation is to

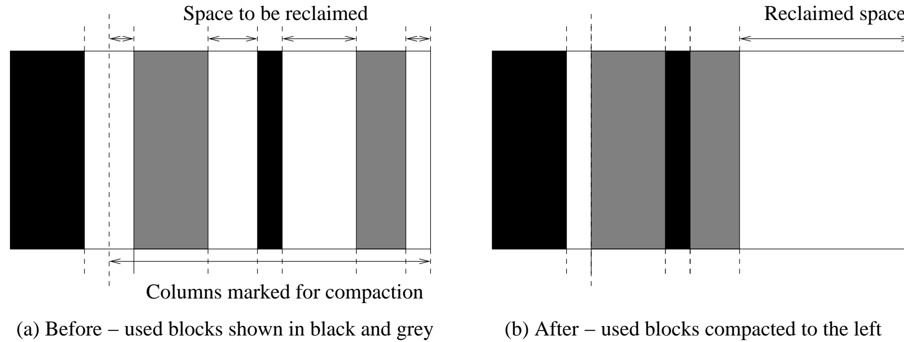

Fig. 4. Example of compaction requirement

shift the necessary number of rightmost used blocks to the left, thereby creating a large enough free block. It is easy to envisage how the required shifting of configuration information can be achieved by sequentially reloading configuration bitstreams using an external agent or, more adventurously, an internal agent on the array.

However, we wish to point out a more efficient possibility, targeted at exploiting the nature of contemporary FPGA technologies, but requiring minor physical changes to the device architecture. Our suggestion therefore represents a possible and feasible future option for FPL.

The basic idea is to map two-dimensional block shifts onto one-dimensional configuration data shifts. Reflecting upon the past and continuing architectural heritage of FPGA devices, this should be convenient since in many architectures configuration is carried out by passing a serial bitstream through a shift register formed over all cells of the array. This persists in features such as the partial reconfiguration capability of the Xilinx Virtex family, where each column still has a shift register nature.

Here, we postulate the existence of (additional) configuration shift registers spanning FPGA rows and advocate the addition of extra gating along the register, in order to allow partial shifts within the register, as required for block compaction. The structure of a typical row shifter needed to support our on-chip compaction mechanism is depicted in Fig. 5. Assuming the required block width is k, the algorithm for identifying and compacting the tasks on chip is as follows:

1. The head of the search string sets up a 'shift mask' by marking all columns as it travels left until a hit is generated or all columns have been searched. This mask covers the columns in the space to be reclaimed.
2. If no hit is generated, the mask is cleared and we are done (the pending task cannot be accommodated).

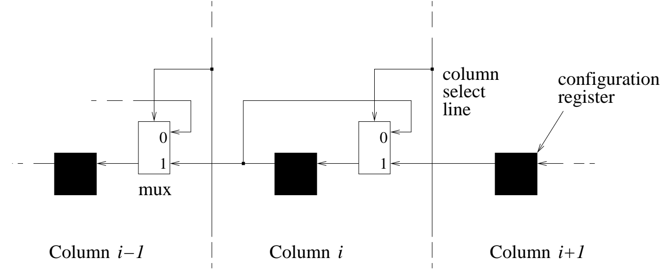

Fig. 5. Configuration shift register in a typical row.

3. For k cycles do:
   (a) If the leftmost column of a moving task coincides with the leftmost column of the shift mask, then clear the fragment of the shift mask covered by that task (it has reached its destination).
   (b) Shift columns one place left into all columns covered by the shift mask (the mask bits drive the column select lines shown in Fig. 5).

Step 1. requires $O(n)$ cycles in the background as previously discussed. Step 2. can be completed in a single cycle if a long line is dedicated to the operation, or also takes $O(n)$ cycles operating in the background. Step 3(a). can be accomplished in a single cycle with appropriate propagation from the left end of the mask to the right end of the task. Step 3(b). requires a single cycle for all columns shifting in parallel. Note that the used/unused nature of the column, which is implicit in part of the operating system circuitry for the task, is shifted with the task. The algorithm thus needs $O(n + k)$ cycles in total.

Considering the run time of the on-chip compaction process, the major benefit derives from the fact that all necessary blocks can be shifted in parallel, in contrast to a conventional sequential approach. The time required for shifting is upper-bounded by the *width* of the free block requested. This time can be contrasted with a sequential approach, where the required time would be equal to the sum of the *areas* of the blocks moved. The latter would be smaller in some cases, for example where one or more small blocks are being shifted relatively long distances. However, a particular benefit of the proposed scheme is that it has a run time directly related to the size of request made and that is independent of the current array usage.

If we assume the FPGA device is provided with column-oriented configuration shift registers to facilitate rapid one-dimensional task configuration, the enhancement suggested above requires the addition of a wire segment and a 2-input multiplexer between each horizontal pair of registers, as well as a select line and select logic for each column. The cost of this additional hardware could be reduced by forming a single snake over all columns of the array and simply providing column select logic to support task compaction. Compaction would

then take time proportional to the *area* of the incoming task, and moving tasks would be delayed by the area of the free space to their left that is to be reclaimed. It should be noted that these delays are also bounded by the area of the incoming task and that the greater delays associated with this scheme can significantly erode the benefits of compaction. Instead, we sought to minimize the delays associated with task rearrangement by making full use of the available on-chip bandwidth for moving tasks in parallel.

Since information about which columns are used and unused resides at the appropriate physical positions in the array, there is no need for an external agent, nor the major complication of indirecting shift operations through any superimposed layers of mapped access to the configuration memory.

Aside from moving configuration information for blocks, there are some other task management issues to be considered. Where inter-block communication circuitry is present, this must be adjusted appropriately. However, depending upon the inter-task communications method adopted, this could be as simple as eliminating the wire segments spanning empty columns. Also, to allow other blocks and external interfaces to continue operation while a block is being moved, there must be buffering of data sent to the block [7]. Our solution bounds the maximum buffer space needed by a task during its lifetime since, at most, it moves from one end of the array to the opposite end. We can therefore determine whether a task should be admitted to the system based on the available buffer space and its maximum possible requirement.

5 Conclusions and future work

In this paper, we have built on two threads of prior research related to the management of tasks on a reconfigurable resource. The first thread, pursued by both authors amongst others, has concerned use of a separate processor resource to perform management of a dynamically-varying collection of tasks. The second thread has concerned the use of reconfigurable logic itself to oversee a static collection of tasks with static scheduling. This work, by focusing on the more challenging aspect of each thread, has demonstrated that reconfigurable logic can be used as a first-order computational resource. We have also proposed a realistic extension to the reconfiguration hardware of FPGAs that would enable very efficient partial reconfiguration geared towards task management.

Various challenges to building on this initial basis lie ahead. First, we will be applying the technique in a practical case study, that of autonomously managing a collection of reconfigurable function units made available to a microprocessor core. Second, we will be investigating the extension of the one-dimensional task management strategies to two-dimensional strategies, seeking to find ways of efficiently implementing proven software algorithms in reconfigurable logic.

More profoundly, we are also interested in what exercises like this, where programmable logic is the first-order computational mechanism, reveal about the computational power of this medium. In particular, we are exploring the relationship between the reconfigurable mesh (RMESH) theoretical model [12]

and the capabilities of FPGAs. For instance, in contrast to our FPGA algorithm, a word-model RMESH algorithm for finding a suitable free block requires just a constant number of steps. We intend investigating practical techniques to overcome the differences between the models.

In summary, we intend our work to illuminate reconfigurable computing at several levels, including: physical architectures; dynamic modularisation; and computational models.

References

1. K. Bazargan, R. Kastner, and M. Sarrafzadeh. Fast template placement for reconfigurable computing systems. *IEEE Design and Test of Computers*, 17(1):68 – 83, Jan.–Mar. 2000.
2. G. Brebner. A virtual hardware operating system for the Xilinx XC6200. *Proc. 6th International Workshop on Field Programmable Logic and Applications*, Springer LNCS 1142, 1996, pages 327–336.
3. J. Burns, A. Donlin, J. Hogg, S. Singh, and M. de Wit. A dynamic reconfiguration run-time system. *Proc. 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines*, IEEE, 1997, pages 66 – 75.
4. K. Compton, J. Cooley, S. Knol, and S. Hauck. Abstract: Configuration relocation and defragmentation for reconfigurable computing. *Proc. 8th Annual IEEE Symposium on FPGAs for Custom Computing Machines*, IEEE, 2000, pages 279 – 280.
5. O. Diessel and H. ElGindy. Run–time compaction of FPGA designs. *Proc. 7th International Workshop on Field Programmable Logic and Applications*, Springer LNCS 1304, 1997, pages 131 – 140.
6. O. Diessel and H. ElGindy. On scheduling dynamic FPGA reconfigurations. *Proc. Fifth Australasian Conference on Parallel and Real–Time Systems*, Springer, 1998, pages 191 – 200.
7. H. ElGindy, M. Middendorf, H. Schmeck, and B. Schmidt. Task rearrangement on partially reconfigurable FPGAs with restricted buffer. *Proc. 10th International Workshop on Field Programmable Logic and Applications*, Springer LNCS 1896, 2000, pages 379 – 388.
8. B. K. Gunther, G. J. Milne, and V. L. Narasimhan. Assessing document relevance with run–time reconfigurable machines. *Proc. 4th Annual IEEE Symposium on FPGAs for Custom Computing Machines*, IEEE, 1996, pages 10 – 17.
9. J. R. Hauser and J. Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. *Proc. 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines*, IEEE, 1997, pages 12 – 21.
10. E. Lemoine and D. Merceron. Run time reconfiguration of FPGA for scanning genomic databases. *Proc. 3rd Annual IEEE Symposium on FPGAs for Custom Computing Machines*, IEEE, 1995, pages 90 – 98.
11. P. Lysaght, G. McGregor, and J. Stockwood. Configuration controller synthesis for dynamically reconfigurable systems. *IEE Colloquium on Hardware–Software Cosynthesis for Reconfigurable Systems*, IEE, 1996, pages 1 – 9.
12. R. Miller, V. K. Prasanna-Kumar, D. I. Reisis, and Q. F. Stout. Parallel computations on reconfigurable meshes. *IEEE Transactions on Computers*, 42(6):678 – 692, June 1993.

13. H. Schmit. Incremental reconfiguration for pipelined applications. *Proc. 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines*, IEEE, 1997, pages 47 – 55.
14. N. Shirazi, W. Luk, and P. Y. K. Cheung. Run–time management of dynamically reconfigurable designs. *Proc. 8th International Workshop on Field Programmable Logic and Applications* Springer LNCS 1482, 1998, pages 59 – 68.
15. M. J. Wirthlin and B. L. Hutchings. Sequencing run–time reconfigured hardware with software. *Proc. Fourth International ACM Symposium on Field Programmable Gate Arrays*, ACM, 1996, pages 122 – 128.