

# Efficient Self-Reconfigurable Implementations Using On-Chip Memory\*

Sameer Wadhwa and Andreas Dandalis

University of Southern California  
{sameer, dandalis}@halcyon.usc.edu

## 1 Self-Reconfiguration

The limited I/O bandwidth in reconfigurable devices results in a prohibitively high reconfiguration overhead for dynamically reconfigured FPGA-based platforms. Thus, the full potential of dynamic reconfiguration can not be exploited. Usually, any attainable speed-up by executing an application on hardware is diminished by the reconfiguration overhead. The self-reconfiguration concept aims at drastically reducing the reconfiguration overhead by performing dynamic reconfiguration on-chip without the intervention of an external host. Thus, using self-reconfiguration, a configurable device can alter its functionality autonomously. Implementations based on self-reconfiguration promise significant speed-up compared with conventional approaches [7, 8].

Self-reconfiguration was first introduced in [4, 5]. In [7, 8] self-reconfiguration was proposed to be realized by altering the configuration bit-stream, that is, on-chip logic accesses and alters the configuration bit-stream to reconfigure the device. Compared with conventional implementations, significant speed-up was achieved for string matching and genetic programming problems [7, 8]. However, the proposed approach in [7, 8] can be realized only using multi-context configurable devices that allow on-chip manipulation of the configuration bit-stream. In state-of-the-art FPGAs, direct manipulation of the configuration bit-stream can only be performed by an external host. Moreover, the complexity depends on the structure of the configuration bit-stream and the on-chip configuration mechanism, and has not been analyzed thus far.

## 2 Our Approach

Our goal is to realize self-reconfiguration efficiently using state-of-the-art FPGAs. Since on-chip manipulation of the configuration bit-stream is not allowed, our key idea is to abstract the dynamic nature of a computation to embedded data memory (which is accessible on-chip). The dynamic nature of a computation corresponds to the dynamic features of its implementation, that is, features that are likely to be altered at runtime. Hence, instead of implementing logic that alters the configuration bit-stream, we implement logic that can control its functionality on-the-fly by altering on-chip data memory.

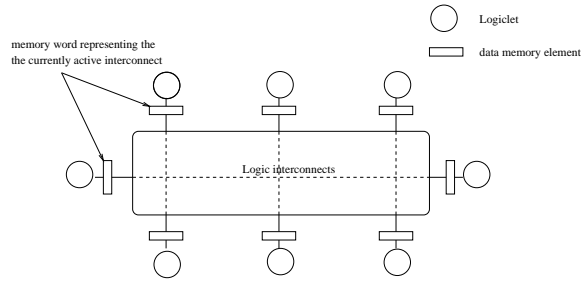
---

\* This research was performed as part of the MAARCII project. This work is supported in part by the DARPA Adaptive Computing Systems program under contract no. DABT63-99-1-0004 monitored by Fort Huachuca and in part by the National Science Foundation under grant no. CCR-9900613.

Based on our ideas, we demonstrate efficient self-reconfigurable implementations for string matching, shortest path, and genetic programming.

A self-reconfigurable implementation is problem-specific and consists of self-reconfigurable logic and the corresponding control circuit. The functionality of self-reconfigurable logic can be altered on-the-fly. The control circuit orchestrates the alteration of the underlying functionality.

Self-reconfigurable implementations can be abstracted as a set of *logiclets* and a programmable interconnection network as shown in figure 1. The *logiclets* are primitive logic elements. For example, a 16-bit arithmetic component can be realized as two 8-bit *logiclets* connected to each other. The functionality of a *logiclet* can also be determined by a memory-based look-up table (e.g. Finite State Machine). Thus, the functionality of an implementation can be altered by modifying the interconnection among *logiclets* and/or by altering the functionality of individual *logiclets*.



**Fig. 1.** Conceptual representation of logiclets connected by an addressable interconnect

In a programmable interconnection network, each interconnection can be set “active” or “inactive” depending on the computation requirements. Hence, based on the run-time parameters, a new permutation of “active” interconnections can be derived. Such a permutation can be represented as a bit-pattern (interconnect address) as shown in figure 1. The bit-pattern can be stored in either embedded memory blocks or distributed memory. For example, if a distributed memory is used, each logiclet is associated with a memory element to store the “active” interconnect address. This “active” interconnect leads to the next logiclet in the function sequence. A number of such interconnects are configured on the device at compile-time and one of them is tagged as “active” during self-reconfiguration. As we demonstrate in the applications section, this functionality can be easily realized using a multiplexer.

On the other hand, if shared memory is used, an interconnection is represented as a memory address. As a result, *logiclets* can exchange data by sharing the same memory address. As we demonstrate in the applications section, this functionality can be implemented using memory elements.

Self-reconfiguration is orchestrated by a control circuit. The control circuit is problem-specific and determines the permutation of “active” interconnections as well as the functionality of RAM-based *logiclets*. Thus, the underlying functionality is altered by modifying the bit-patterns stored in on-chip memory. The complexity of the control circuit is problem-specific and depends on the amount of dynamic modifications that are supported by the implementation. It is important to note that our approach to self-reconfiguration is significantly different from the approach adopted in [8, 7]. In [8, 7], the logic structures are adapted on the device to achieve self-reconfiguration. On the

other hand, in our approach, pre-compiled logic structures are controlled on the device to self-reconfigure the logic functionality.

### 3 Application Demonstration

In this section, self-reconfigurable implementations for string matching, shortest path, and genetic programming are demonstrated. The implementations are based on the approach described in Section 2 and are realized using the Xilinx Virtex series of FPGAs.

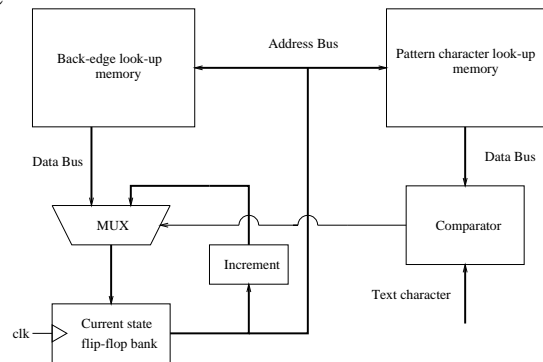
#### 3.1 String Matching

The string matching problem consists of finding all occurrences of a pattern in a text. In our implementation, we consider the KMP string-matching algorithm [2]. The algorithm begins by constructing an optimized finite state automata specific to the input pattern. The optimization involves constructing pattern-specific back-edges in the finite state automata. Then, the finite state automata performs the string matching on the input text.

The pre-processing phase can be realized efficiently using self-reconfiguration. As described in [8], the back-edges are constructed by using an OR-gate grid. By altering the configuration bit-stream, the OR-gate grid is adapted to the input pattern. On the contrary, in our implementation, the back-edges are abstracted as a look-up table. The look-up table is realized using the embedded RAM blocks of the Virtex FPGAs. Each state of the automaton corresponds to a memory address. Thus, a back-edge construction can be easily realized by altering the data contents of the corresponding address. In Figure 2, a high-level view of our implementation is shown.

Unlike the self-reconfigurable implementation in [8], our implementation can be realized using state-of-the-art FPGAs. The back-edge representation in our design results in simple control circuit for realizing self-reconfiguration. The simplicity of the control circuitry leads to reconfiguration time at least as fast as it is claimed in [8]. In addition, the clock rate achieved by our implementation outperforms the one achieved by [8]. This is because the OR-gate grid delay in [8] appears in the critical path of the design. On the contrary, in our design, look-up-table access occurs in parallel with character comparison and does not affect the critical path.

For a pattern size of six characters, our implementation achieved a clock rate of  $110MHz$  as opposed to  $15MHz$  in [8]. However, the performance analysis in [8] was



**Fig. 2.** Our proposed self-reconfigurable KMP implementation

based on Xilinx 6200 series FPGAs. To make a fair comparison, we analyzed the performance of [8] on Xilinx Virtex series FPGAs and our proposed implementation still achieves a clock rate atleast twice as fast. Finally, our implementation requires less hardware area since it replaces the OR-gate grid by embedded RAM blocks and requires only one one comparator (in [8] the number of comparators required is proportional to the length of the input pattern).

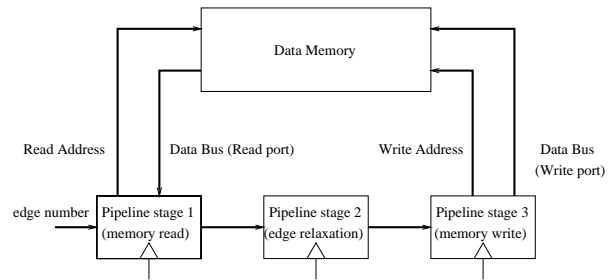
### 3.2 Single-Source Shortest Path

The single source shortest path problem is a classical combinatorial problem that arises in numerous applications. For a given weighted, directed graph and a source vertex, the problem consists of finding the shortest paths from the source to all the vertices of the graph. In our implementation, we use the Bellman-Ford algorithm to solve the shortest path problem. The Bellman-Ford algorithm [2] relaxes the weights of the edges in an iterative fashion until the shortest paths to all the vertices of the graph are computed.

The key aspect of an FPGA-based implementation for solving the shortest path problem, is the time required to adapt the hardware to an input graph instance. For example, in [1], a very efficient implementation can be derived for a given graph instance by exploiting the massive parallelism inherent in the Bellman-Ford algorithm. However, a prohibitively high mapping time is also required to derive an efficient implementation. As a result, any consequent speed-up by executing Bellman-Ford using FPGAs is diminished by the reconfiguration overhead. In [3], a domain-specific approach was introduced that reduced significantly the reconfiguration overhead. Consequently, compared with software-based implementations, constant speed-up was achieved. In both [1] and [3], a host machine was used to adapt the hardware to the input graph instance by altering the configuration bit-stream.

Similarly to [1, 3], our implementation is also based on the Bellman-Ford algorithm but it does not require the intervention of a host machine to adapt to a graph instance. The dynamic characteristics of a graph can be efficiently represented by its adjacency matrix and stored in on-chip memory. The relaxation of the edges can be achieved by reading and updating the memory contents repeatedly according to the Bellman-Ford algorithm. In Figure 3, the proposed implementation is shown (the control circuitry is not included).

Our implementation requires at most as much reconfiguration time as in [3]. Furthermore, it requires less hardware area and can scale more efficiently than the implementation demonstrated in [3]. Regarding execution time on hardware, our implementation also outperforms the one proposed in [3] since it requires  $O(n^2)$  less number of clock cycles while achieving a faster average edge relaxation rate. In Table 1, the indicated data correspond to a graph with



**Fig. 3.** Our proposed self-reconfigurable Bellman-Ford algorithm implementation

the indicated data correspond to a graph with

weights of 16-bit precision. The average edge relaxation time for [3] is based on an implementation using Xilinx XC6200 FPGAs. However, we have implemented the same design as in [3] using Xilinx Virtex FPGAs, and our proposed implementation still achieves an average edge relaxation rate at least twice as fast.

Implementation	Avg. edge relaxation time	Number of clock cycles	Area
[3]	66.67ns	$O(n(n+e))$	$O(n+e)$
This paper	16.7ns	$O(ne)$	$O(e)$

**Table 1.** Performance Comparison with [3]

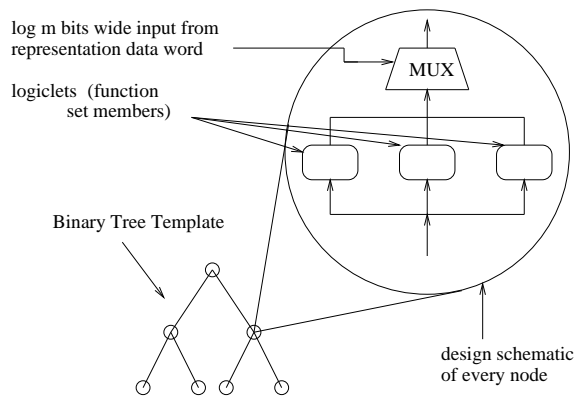
### 3.3 Genetic Programming

Genetic Programming (GP) realizes a learning system by employing the Darwinian evolution principles to evolve a population of computer programs. GP consists of an evolution and a fitness evaluation phase that are executed repeatedly. The fitness evaluation phase decides which programs survive while the evolution phase evolve the survived programs. The fitness evaluation phase is the most computationally intensive phase in GP. By mapping the fitness evaluation phase onto reconfigurable hardware, significant speed-up is possible compared with software-based implementations [6].

In [7], self-reconfiguration was exploited to demonstrate a design where both the evolution and the fitness evaluation phases are executed on the reconfigurable device. Since both the phases of Genetic Programming algorithm were executed on the device, the reconfiguration overhead due to the limited I/O bandwidth was effectively eliminated from the critical path to the solution. The programs are represented as binary trees of fixed interconnection. During the evolution phase, the configuration bit-stream data corresponding to each tree-node is modified according to the evolution directives. As a result, the functionality of each tree-node can be switched based on a pre-defined function set.

In our implementation, each node of the binary tree is based on the conceptual model shown in figure 1. The binary tree-nodes are realized as shown in figure 4. The function set members for a GP application are realized as *logiclets*. Furthermore, each function set member is represented by a data word of length  $\log m$  where  $m$  is number of members in the set. Each binary tree can thus be

represented as a data word of length  $n \log m$  where  $n$  is the number of nodes in the tree. This data word representation (i.e. bit-pattern) is stored in distributed memory and



**Fig. 4.** Our proposed self-reconfigurable tree template and node implementation for Genetic Programming

is used to alter the functionality of the tree-nodes. Thus, self-reconfiguration is realized by modifying the bit-pattern in accordance with the evolution semantics.

Each tree-node in our design is more complex and requires more hardware area than in [7]. However, in our design, self-reconfiguration can be achieved by using state-of-the-art FPGAs and does not rely on specific device architectures. Finally, a preliminary performance analysis indicates that our implementation can be as fast as the one demonstrated in [7].

## 4 Conclusions

In this paper, we proposed an approach to realize efficient self-reconfigurable implementations using state-of-the-art FPGAs. Our key idea is to abstract the dynamic nature of a computation. Using our approach, we demonstrated self-reconfigurable implementations for string matching, shortest path and genetic programming. Our implementations outperformed the contemporary implementations for string matching and shortest path while performing at least as well as the contemporary implementation for genetic programming.

The USC MAARCII project (<http://maarcII.usc.edu>) is developing novel mapping techniques to exploit dynamic reconfiguration and facilitate run-time mapping using configurable computing devices and architectures. The goal is to alleviate the long mapping time required by conventional CAD tools. Computational models and algorithmic techniques based on these models are being developed to exploit self-reconfiguration using FPGAs.

## 5 Acknowledgement

We would like to acknowledge the continued guidance and support of our research advisor, Prof. Viktor K. Prasanna, towards this work.

## References

1. J. Babb, M. Frank, and A. Agarwal. Solving graph problems with dynamic computation structures. In *SPIE'96: High-Speed Computing, Digital Signal Processing, and Filtering using Reconfigurable Logic*, 1996.
2. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, 1993.
3. A. Dandalis, A. Mei, and V. K. Prasanna. Domain specific mapping for solving graph problems on reconfigurable devices. In *sixth IEEE Reconfigurable Architectures Workshop*, April 1999.
4. A. Donlin. Self modifying circuitry - a platform for tractable virtual circuitry. In *Eighth International Workshop on Field Programmable Logic and Applications*, 1998.
5. P. C. French and R. W. Taylor. A self-reconfiguring processor. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 50–59, April 1993.
6. J.R. Koza, F.H. Bennett III, and J.L. Hutchings. Evolving sorting networks using genetic programming and the rapidly reconfigurable xilinx 6216 field-programmable gate array. In *FPGA'98 Sixth International Symposium on Field Programmable Gate Arrays*, February 1998.
7. R. P. S. Sidhu, A. Mei, and V. K. Prasanna. Genetic programming with self-reconfigurable fpgas. In *International Workshop on Field Programming Logic and Applications*, August 1998.

8. R. P. S. Sidhu, A. Mei, and V. K. Prasanna. String matching on multicontext fpgas using self-reconfiguration. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 217–226, Monterey, CA, February 1999.