# Configurable Hardware Solutions for Computing Autocorrelation Coefficients: a Case Study

*to be presented at FPGA2005*

J. E. Rice
University of Lethbridge
Dept. of Math & Computer Science
Lethbridge, Alberta, Canada
j.rice@uleth.ca

K. B. Kent
University of New Brunswick
Faculty of Computer Science
Fredericton, New Brunswick, Canada
ken@unb.ca

## ABSTRACT

There are many computationally intensive problems in the area of digital design and logic synthesis. Some of these have no "good" solution; that is, simply by their definition they have exponential run-times. In order to overcome this, we examine the possibility of a configurable hardware solution to speed up one such problem. The computation of the problem is carried out on a Field Programmable Gate Array (FPGA), where the problem is encoded in such a way that within certain parameters, the design of the solution need not be changed for working with a variety of benchmark circuits. This saves considerably on compilation and configuration times. The use of configurable hardware, however, still allows for configuration in situations where the parameters change significantly enough to require an altered approach. We examine two implementations of the problem, which in this case consists of computing the autocorrelation coefficients for a Boolean function.

## Categories and Subject Descriptors

B.7.1 [**Hardware**]: Integrated CircuitsTypes and Designs

## General Terms

Design

## 1. INTRODUCTION

### 1.1 Configurable Hardware

One technique for accelerating computation is to introduce configurable hardware solutions. This has been done for the computation of various problems such as image compression [3], string matching [5], and problems such as the Hamiltonian cycle problem [10]. In this work we apply this technique to the computation of a mathematical transform known as the *autocorrelation transform.* By making use of hardware to perform all or part of the algorithm it is possible to not only speed up the computations, but also to do some of them in parallel. The application of configurable hardware to this problem was first introduced in [6].

### 1.2 Application

Boolean functions are functions for which the inputs and the output(s) are restricted to the Boolean domain. If a transform such as the autocorrelation transform is applied to the output vector of a Boolean function, the result is a representation of the function in a non-Boolean domain. This representation is generally referred to as the autocorrelation coefficients of the function.

The autocorrelation coefficients provide a measure of the function's similarity to itself, shifted by a given amount. This is also called the cross-correlation, or convolution function. The autocorrelation coefficients have been used in various areas including optimisation and synthesis of combinational logic [12], variable ordering for ROBDDs [8], and to compute the estimate $C(f)$ of a function's complexity [12, 4]. However, the commonly used methods of computation are exponential in the number of inputs to the function(s), which somewhat limits their practical use. Prior work [6] demonstrated that there is value in implementing this particular problem in hardware in order to overcome this problem of exponential computation requirements; however, the choice of algorithm, architecture, and other variables play a large role in the degree of success.

### 1.3 The Approaches

In this paper we compare two approaches to the problem. The first approach, based on the initial work in [6], uses an instance-specific approach based on a BDD representation of the input. Additional work is introduced that improves resource usage. The second approach is a parameter-specific approach, also attempting to maximise resource usage.

## 2. BACKGROUND

### 2.1 The Autocorrelation Transform

The autocorrelation function is defined as follows [4]:

$$B^{f\,f}(u) = \sum_{v=0}^{2^n-1} f(v) \cdot f(v \oplus u) \tag{1}$$

The superscripts $f$ $f$ are generally omitted. Values for $u$ range from 0 to $2^n - 1$ where $n$ is the number of inputs to the Boolean function $f(X)$. Clearly there are $2^n$ possible coefficients $B(u)$ that may be computed for a given function. The brute-force application of this equation, even for a single value of $u$ requires an exponential number of operations. However, as discussed in the following sections there are various techniques that may be employed that reduce the computation requirements to a certain extent.

For multiple-output functions a second step must be performed to combine the autocorrelation function for each of the individual functions into the *total autocorrelation function*. Only single-output functions are considered in this paper.

## 2.2 Approach 1

In the first approach a Binary Decision Diagram (BDD) [2, 1] is used in the representation of the function for which the coefficients are to be computed. This technique is based on a software technique introduced in [7]. The use of BDDs significantly reduces the amount of processing required, simply due to the fact that most Boolean functions have a fairly compact BDD representation. This has the added advantage in this work of ensuring that the problem is compact enough to be implemented on our chosen device.

The theory behind an instance-specific approach is that for some classes of problems it may be possible to utilise a configurable device not for a general problem, but rather for a specific instance of a problem. For certain applications with desirable characteristics this can achieve a high performance increase [10]. Some characteristics that make the calculation of a problem suitable for this computation technique are that:

1. the computation is intensive and time consuming; this is a required characteristic needed to offset the penalty for generation of the hardware design, and

2. the computation is relatively self-contained. This is suitable to allow the generated hardware design to be compact enough to fit within an available re-programmable device.

A software application, using a generic circuit solution, is then used to create a circuit description to compute the solution for a specific input case. Both the generic circuit and the instance-specific circuit are specified in some manner that is accepted by the traditional CAD design tools being used, for example VHDL or verilog.

One problem that may arise in the circuit generation lies in how to choose amongst various levels of parallelism. The level of parallelism must be based both on the problem and on the characteristics of the FPGA. Certain amounts of parallelism will be inherent to the chosen approach; however, the tools used in the generation of the instance-specific circuit should be aware of the target environment and how the resulting circuit will be placed and routed. At the early stages in the design flow this information is not known and therefore must be predicted with a reasonable level of accuracy in order to maximise the usage of the available resources.

## 2.3 Approach 2

The second approach is based on a significantly different algorithm, also introduced in [7]. This technique requires that the input function be pre-processed, in software, such that the input to the FPGA is a list of disjoint cubes describing the on-set of the problem. The reader is directed to [11] for one algorithm for the computation of disjoint cube lists. The hardware is then used to implement a series of comparisons of the disjoint cubes, each of which compute a contribution to the overall coefficient values.

Unlike the above approach, the second approach can best be described as *parameter-specific*. Rather than requiring a new circuit for every new instance of the problem, we design a circuit that may be used for many instances of the problem, as long as they fall within certain parameters. One advantage over the above approach is that configuration is required far less often. However, the circuit is – of necessity – not optimally designed for any particular instance.

## 3. CIRCUIT ARCHITECTURES

## 3.1 Instance-Specific Circuit Architecture

The hardware architecture used in approach 1 is described in detail in [6]. Briefly, the hardware architecture for this solution consists of three main components: a function component, the calculator, and the controller. This architecture is depicted in Figure 1. The function component contains the logic function for which the autocorrelation coefficients are being computed. Two instances of this component are used, one for each function calculation that is required for each summation. This component contains an embedding of the BDD representation of the logic function. The most important aspect of this is that it provides a means of storing the function that, in general, will not require exponential size. The calculator is responsible for performing the logic

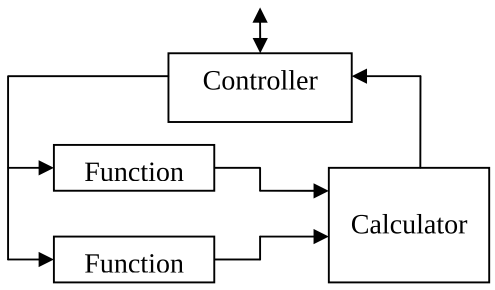

**Figure 1: Architecture of the instance-specific design used in approach 1.**

exclusive-or and summations of the autocorrelation function on inputs that are received from the function component(s). The controller process, as the name suggests, controls the overall computation process and the storing of its results. The controller is also responsible for sending inputs to each of the functions for computation.

## 3.2 Parameter-Specific Circuit Architecture

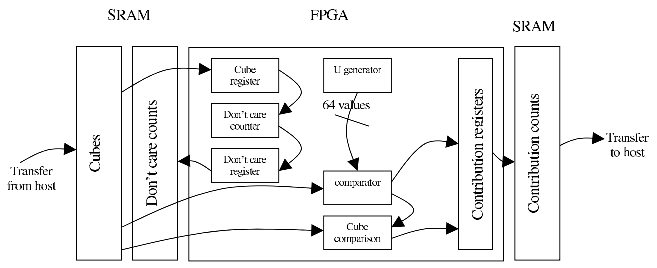

**Figure 2: Architecture of the parameter-specific design used in approach 2.**

Briefly, the process for computing $B(u)$ is as follows:

- for each cube in the disjoint cube list
  - compute cube $\oplus$ $u$
  - search for the new cube or one containing it in the cube list
  - if found add 2 to the sum register as the contribution to the coefficient

Complete details are given in [7].

As shown in Figure 2, a daughter-board with on-board SRAM for storage of the input function was utilised. One of the keys to this algorithm is that the number of don't cares in each cube must be counted. This is done by the hardware solution, and is optimised by counting both halves of each cube word in parallel. We should note that there is a limit of one memory access to the SRAM per clock cycle; thus the design was optimised to minimise SRAM accesses and store any intermediate results on-chip.

## 3.3 Space Utilisation

As was found in both approaches, computation of the autocorrelation transform is a highly parallel problem. Many of the tasks performed in the process are not dependent on their ordering; thus there is the flexibility to add a certain degree of parallelism to the solutions. However, in each approach, it was necessary to balance the addition of parallel computing components with the additional complexity and overhead such additions required.

In approach 1, the architecture currently calculates one term of the summation each clock cycle. Given no restriction on design space, an obvious enhancement to the architecture would be to replicate the function components. This replication would allow for multiple terms of the summation to be computed in parallel during one clock cycle. Changes to the controller and calculator in support of these replications should require minimal design space. The function component, however, is more demanding on design space. The size requirements of this component is dependent on the size of the BDD representation of the logic function. The problem is that to create the design, some estimate of the space requirements is required during the generation of the design so that the optimal amount of parallelism can be built in to the design.

In the parameter-specific design a slightly different approach to the addition of parallelism was used. As shown in Figure 2, the $u$ generator generates 64 values in parallel, which are then passed in one clock cycle to the comparator. The comparator is designed to have 64 comparator sub-components in order to support this. Thus for the computation of a single coefficient there is no real advantage, but for computations of more than one coefficient up to 64 can be performed in parallel.

## 4. EXPERIMENTAL RESULTS

The results given are for a series of single-output benchmarks from the ISCAS 89 set with a maximum of 32 inputs, as shown in Table 1. This is due to the limitations of a 32-bit

| function | inputs | BDD size (nodes) |
|---|---|---|
| 9symml | 11 | 25 |
| cm152a | 11 | 16 |
| co14 | 14 | 27 |
| ex10 | 5 | 6 |
| ex20 | 5 | 11 |
| ex30 | 5 | 10 |
| life | 9 | 26 |
| majority | 5 | 8 |
| max46 | 9 | 75 |
| mux01 | 21 | 33 |
| ryy6 | 16 | 21 |
| sym10 | 10 | 31 |
| xor5 | 5 | 6 |

**Table 1: The functions used in these experiments and their sizes in terms of inputs and BDD nodes.**

word size and 219 cubes inherent to the second approach. Extensions to larger functions are discussed in Section 5.

## 4.1 CLB Utilisation

### *4.1.1 Approach 1*

In approach 1 each test utilised a varying amount of parallelism: 2 function computations (one summation term) in parallel, 4, 6, 8 and so on with the maximum amount computed in parallel being 14. For each test the percentage of CLBs utilised was measured, along with the minimum period (giving a maximum possible frequency) and the speed of computation, calculated based on the frequency. These experiments were conducted using the Xilinx Foundation 3.1 tool set targeting the XC 4062XLA FPGA. Table 2 presents the complete results for one of the benchmarks, ryy6. The first column lists the number of parallel computations that were attempted. As illustrated by this table, it is possible to maximise CLB usage through additional function components; however, this has the result of lowering the clock speed of the device. This drawback was not encountered in the second approach. Additionally, the highest amount of parallelism does not necessarily lead to the fastest computation time. Finally, these results, while indicative of the general results for other benchmarks, do not necessarily hold for all benchmarks since this is an instance-specific approach and the circuit will vary for each instance of the problem. This is illustrated by the results shown in Table 3, which shows the amount of parallelism (column 2) resulting in the fastest computation for each of the benchmarks used when

|  | CLB Usage | Computation time (sec) | Min. period (ns) | Max. freq. (MHz) |
|---|---|---|---|---|
| 2 | 31% | 100.17 | 23.323 | 42.876 |
| 4 | 35% | 52.132 | 24.276 | 41.193 |
| 6 | 38% | 34.511 | 24.106 | 41.483 |
| 8 | 42% | 33.038 | 30.769 | 32.500 |
| 10 | 46% | 32.828 | 38.217 | 26.166 |
| 12 | 49% | 31.814 | 44.444 | 22.500 |
| 14 | 53% | 34.905 | 56.889 | 17.578 |

**Table 2: Complete CLB usage and timing results for the benchmark ryy6 using the instance-specific approach.**

| function |  | CLB Usage | Computation Time | Max. Freq. (MHz) |
|---|---|---|---|---|
| 9symml | 8 | 43% | 17.944 ms | 36.522 |
| cm152a | 12 | 50% | 298.35 ms | 27.268 |
| co14 | 10 | 49% | 1820.5 ms | 29.491 |
| ex10 | 8 | 40% | .06962 ms | 36.769 |
| ex20 | 8 | 40% | .07025 ms | 36.438 |
| ex30 | 10 | 44% | .07187 ms | 28.493 |
| life | 10 | 46% | 18.104 ms | 28.959 |
| majority | 12 | 47% | .06767 ms | 25.220 |
| max46 | 10 | 61% | 19.268 ms | 27.210 |
| mux01 | 10 | 48% | 31188 sec | 28.203 |
| ryy6 | 12 | 49% | 31.184 sec | 22.500 |
| sym10 | 8 | 43% | 74.331 ms | 35.267 |
| xor5 | 10 | 44% | 3556.7 ns | 28.791 |

**Table 3: Results showing the fastest computation times and corresponding amount of parallelism for approach 1 (instance-specific).**

utilising the instance-specific approach. Figure 3 shows a graph relating the percentages of overall CLB usage, performance, and clock rates to the amount of parallelism introduced. This seems to indicate that the range of 8 to 12 function components in parallel will provide an increase in performance, depending on the function. This corresponds to usage of the device ranging from 43% to 61% of the available CLBs. For most functions, roughly 50% usage is likely to provide the best performance. The overall maximum clock rate for the generated circuits began to degrade rather quickly beyond the threshold of 6 function components. An important contributing factor to the performance is not just the number of CLBs used, but also the density of the circuit. Increasing the level of parallelism beyond 12 consistently begins a decrease in overall performance. It was also observed that increasing the number of function components to above 14 resulted in the circuit failing to synthesise. This is attributed to the density of the routing necessary with respect to the controller and calculator components. Regardless of the number of CLBs available, beyond this threshold the synthesis tools cannot place and route the circuit. To demonstrate the importance of place and route in the maximisation of resource usage, the overall circuit design is modified to relieve some of the contention of routing at the controller component. To achieve this design goal, the calculator component is modified to sum several functional results instead of just a single sum. While this increases the incoming communication at the calculator, it

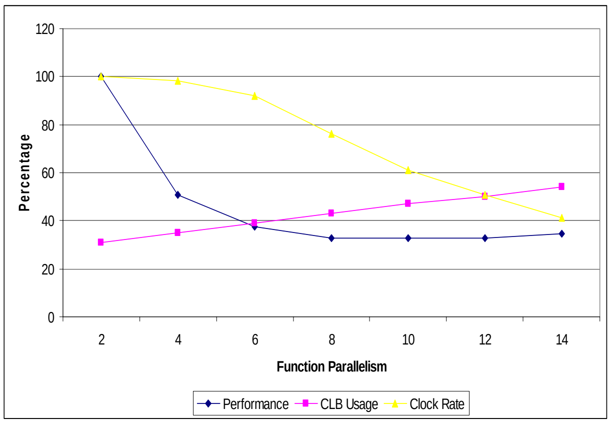

**Figure 3: Percentages of overall CLB usage, performance, and clock rates in relation to the amount of parallelism for approach 1.**

reduces the number of calculator components in the overall design, thus reducing the amount of communication with the controller . Table 4 shows results for these higher performance circuits, utilising greater levels of parallelism than was achievable with the original design (column 1). Clearly the additional parallelism still affects the clock rate, but as reflected in the speed-up in the computation speed the advantages of the added parallel computations outweigh this disadvantage up to a certain point.

|  | CLB Usage | Computation Time (sec.) | Min. Period (ns) | Max. Freq. (MHz) |
|---|---|---|---|---|
| 16 | 60% | 19.866 | 37.003 | 27.025 |
| 18 | 57% | 17.958 | 37.629 | 26.575 |
| 20 | 68% | 27.716 | 64.531 | 15.496 |
| 24 | 82% | 24.414 | 68.212 | 14.66 |

**Table 4: Results for ryy6 when using improved circuit design for approach 1.**

### 4.1.2 *Approach 2*

For approach 2 the Xilinx Virtex 812E chip was targeted with Xilinx ISE version 5.2i used for place and route. This chip contains 18,816 4 input LUTs as CLBs (approximately 200,000 logic gates). The design was implemented using a variety of options, namely a varying number of bits for storage of each cube and a varying number of coefficient being computed in parallel. Table 5 gives the resulting CLB usage, broken down by logic and routing requirements. It should be noted that for approach 1 all results were obtained by simulation, while in approach 2 the results were obtained by actual execution on the targeted device.

### 4.1.3 *Comparisons*

With the improved design for approach 1, it was possible to utilise up to 82% of the CLBs for a particular design. With the best combination of variables for approach 2, 78% of the

| Cube Bits | parallel coeffs | LUTs for logic | LUTs for routing | LUT Usage |
|---|---|---|---|---|
| 32 | 64 | 13933 | 891 | 78% |
| 26 | 64 | 12696 | 696 | 71% |
| 21 | 64 | 11722 | 502 | 62% |
| 15 | 64 | 10554 | 307 | 57 % |
| 10 | 64 | 9588 | 176 | 51% |
| 32 | 32 | 7412 | 477 | 41% |
| 10 | 32 | 5119 | 114 | 27% |
| 32 | 1 | 956 | 81 | 5% |

**Table 5: Space usage of the Xilinx Virtex 812E chip for various scenarios of the second (parameter-specific) approach.**

CLBs were utilised. The CLB utilisation therefore seems to be comparable for the two approaches.

## 4.2 Timing Comparisons

Comparing the timing results of the various approaches for computing the autocorrelation coefficients is not a trivial matter; in approach 1 there is a considerable amount of overhead in the generation for each instance's design and in the download and configuration of the device, while in approach 2 this time applies to each set of parameters that is used, and there is also preprocessing of the input function to be considered. *espresso -Ddisjoint* [9] is used to create the disjoint cube list, which required less than 1 second in nearly all cases. Additionally, the cube lists must be prepared as 32 bit words for the DMA transfers; this required less than 0.1 seconds in all cases. If we assume that all preprocessing and solution generation/configuration is previously done, then Tables 7 and 6 provide a comparison of approach 1, sequential and parallel versions of approach 2, and two software solutions implemented on an Intel Pentium 4 and SUN SPARC. The software solutions are based on approach 2, utilising a disjoint cube list.

| | approach 1 best result (varied) | approach 2 64 parallel (26 MHz) | approach 2 no parallel (26 MHz) |
|---|---|---|---|
| 9symml | 0.01794 | 0.2940 | 1.0069 |
| cm152a | 0.2984 | 0.2680 | 0.3181 |
| co14 | 1.821 | 0.2490 | 0.4480 |
| ex10 | 0.00006962 | 0.2650 | 0.3024 |
| ex20 | 0.00007025 | 0.2680 | 0.2934 |
| ex30 | 0.00007187 | 0.3010 | 0.2968 |
| life | 0.01810 | 0.2760 | 0.6843 |
| majority | 0.00006767 | 0.2680 | 0.3102 |
| max46 | 0.01927 | 0.2750 | 0.3349 |
| mux01 | 31188 | 24.5830 | 309.2230 |
| ryy6 | 31.814 | 2.1330 | 32.8776 |
| sym10 | 0.07433 | 1.1740 | 27.9384 |
| xor5 | 0.3557 | 0.2700 | 0.3058 |

**Table 6: Times in seconds to compute all $2^n$ coefficients for each of the various hardware approaches.**

From the tables several interesting results are obtained. First, for all of the benchmarks with the exception of ex20, the Intel Pentium 4 outperformed the Sun SPARC using the same software implementation. When comparing the two versions of approach 2, with and without parallelism, the parallel version outperformed the sequential version in every benchmark except ex30. This is attributed to the small size of the benchmark as demonstrated by the extremely low computation times required for all implementations. In such a small time frame, the overhead of performing the parallel computations outweighs the gains in performance.

| | Intel Pentium 4 (2.66GHz) | SUN SPARC (500MHz) |
|---|---|---|
| 9symml | 0.4160 | 6.2250 |
| cm152a | 0.0400 | 0.1350 |
| co14 | 0.1050 | 1.2030 |
| ex10 | 0.0001 | 0.0040 |
| ex20 | 0.0170 | 0.0030 |
| ex30 | 0.0001 | 0.0020 |
| life | 0.1790 | 3.2750 |
| majority | 0.0001 | 0.0020 |
| max46 | 0.0430 | 0.3940 |
| mux01 | 440.4620 | 7837.2870 |
| ryy6 | 15.6440 | 338.1470 |
| sym10 | 14.1680 | 383.2700 |
| xor5 | 0.0001 | 0.0150 |

**Table 7: Times in seconds to compute all $2^n$ coefficients for each of the various software implementations.**

In comparisons of the results from the various hardware and software implementations we can see that it is possible to achieve a performance gain of up to 18 times through the use of parallelism in hardware. However, as above, given a small benchmark, the parallel version can provide a performance decrease. For the ex2 benchmark software outperformed hardware by a factor of approximately 18.

Comparing the two hardware approaches is very interesting. The parameter-specific version provides more "consistent" results while the instance-specific version provides a great deal of speed-up in some cases while a significant performance in some cases, most notably for the benchmark mux01. One must take into consideration here the underlying approach; it is likely that the BDD approach is simply not feasible for this particular benchmark. Indeed, in comparisons between software and hardware for these tests we must take into account that the underlying approach of either a BDD or a cube-list will perform better for some benchmarks and worse for others. In these experiments the instance-specific BDD-based approach outperformed the parameter-specific cube-list approach for 8 out of the 13 benchmarks. These correspond loosely to the smaller of the benchmarks, in terms of numbers of inputs variables. BDD size does not, however, seem to be a factor. This can be seen in the fact that for sym10 the BDD has 31 nodes and the BDD approach still out-performs the other approaches. On the other hand the BDD for mux01 has 33 nodes and the BDD approach is significantly slower than all the other implementations.

## 5. CONCLUSION

This paper reports the results of implementing a problem used in logic synthesis, the computation of a logic function's autocorrelation transform. Two configurable hardware approaches are used and are compared to software implementations on two common platforms.

It is clear that the use of configurable hardware can provide some speed-up in the computation of this problem. Further work is required to identify which approach, instance-specific or parameter-specific, would be most beneficial and furthermore, which underlying algorithm is best suited to benchmark in question as this clearly has an effect in the performance of each of the approaches. Additionally, we have demonstrated that the addition of parallelism in each approach can lead to speed-up of the computation, but that there is a limited amount that may be added, beyond which the additional complexity of the circuit outweighs the advantage of the added circuitry.

As mentioned earlier, the techniques used are currently limited to fairly small functions, and only to single-output functions. Computation for multiple-output functions is currently being investigated, and would require some modification of each of the techniques. However, the parallelism currently implemented could be rededicated to the computation for multiple outputs, and would be a relatively straightforward modification. Approach 1 is limited only by the the size of the BDD for the logic function, and so this computation technique can be extended to larger functions if they have relatively small BDD sizes. Approach 2 is currently limited by a 32 bit word size for storing cubes, but this could be modified to allow larger word sizes, and if necessary, multiple downloads to the daughter-board's SRAM from the host computer. Thus neither of these current limitations present a major drawback to the hardware techniques.

The major drawback, at least in the case of approach 1, is the time required to generate an instance circuit and then to configure the target device. This is a drawback of any instance-specific approach, and requires that the instance of the application being targeted be utilised often enough that the overhead is offset by the time saved in using a hardware approach. Approach 2 still requires this overhead, but the additional time requirements are offset even further due to the fact that this is a parameter-specific approach, and therefore more instances can be solved without having to configure the device. In fact, all of the sample benchmarks investigated in this work were solved by one set of parameters – the main restriction being the 32-bit word size. One approach to this problem is to create many BIT (configuration) files for the problem,and have a preprocessor select the appropriate configuration for the instance function. Fewer resources would then be required for smaller input sizes which results in an increase in the clock rate. This idea could be extended so that the host workstation records the current FPGA configuration. With knowledge of the configuration we can reduce runtime by $\approx 0.250$ seconds. This would be significant if we only needed to calculate one coefficient, and would make much better use of the configurable aspects of the target FPGA.

## 6. FUTURE WORK

An interesting line of research begun in this paper is that of the development of heuristics to "foresee" the optimal amount of parallelism to introduce to a solution design, as indicated by the experiments run with approach 1. Some knowledge of the solution design is clearly required, as shown by the improvements in Table 4. Additionally, knowledge of the specific instance solution is required; for instance, with a high number of inputs to the BDD representing the logic function, additional CLBs must be utilised for routing purposes only. When the number of CLBs utilised for logic exceeds 50% of the total resources, routing becomes much more difficult and adversely affects the performance.

Additional work is currently investigating these factors with the goal of developing heuristics for predicting optimal parallelism in instance-specific configurable solutions.

## 7. ADDITIONAL AUTHORS

Additional authors: T. Ronda (University of Lethbridge, email: `troy.ronda@uleth.ca`) and Z. Yong University of New Brunswick, email: `b15v3@unb.ca`)

## 8. REFERENCES

[1] S. Akers. Binary Decision Diagrams. *IEEE Trans. on Comp.*, C-27(6):509–516, June 1978.

[2] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. on Comp.*, C-35(8):677–691, Aug. 1986.

[3] J. Heron, D. Trainor, and R. Woods. Image Compression Algorithms Using Re-configurable Logic. Technical report, Virtual Computer Corporation, year unknown. downloaded from www.vcc.com.

[4] M. Karpovsky. *Finite Orthogonal Series in the Design of Digital Devices*. John Wiley & Sons, 1976.

[5] H. Lee and F. Ercal. RMESH Algorithms For Parallel String Matching. In *Proceedings of the 3rd International Symposium on Parallel Architectures, Algorithms and Networks (I-SPAN'97)*, pages 223–226, 1997.

[6] J. Rice and K. Kent. Using Instance-Specific Circuits to Compute Autocorrelation Coefficients. In *Proceedings of the First Annual Northeast Workshop on Circuits and Systems (NEWCAS)*, 2003.

[7] J. E. Rice and J. C. Muzio. Methods for Calculating Autocorrelation Coefficients. In *Proceedings of the 4th International Workshop on Boolean Problems, (IWSB P2000)*, pages 69–76, 2000.

[8] J. E. Rice, J. C. Muzio, and M. Serra. The Use of Autocorrelation Coefficients for Variable Ordering for ROBDDs. In *Proceedings of the 4th International Workshop on Applications of th e Reed-Müller Expansion in Circuit Design*, 1999.

[9] R. Rudell. Espresso minimization tool man pages.

[10] M. Serra and K. Kent. Using FPGAs to Solve the Hamiltonian Cycle Problem. In *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, 2003.

[11] M. Thornton and L. Shivakumaraiah. Computation of Disjoint Cube Representations Using a Maximal Binate Variable Heuristic. In *Proceedings of the IEEE Southeastern Symposium on System Theory*, pages 417–421, 2002.

[12] R. Tomczuk. *Autocorrelation and Decomposition Methods in Combinational Logic Design.* PhD thesis, University of Victoria, 1996.