Journal of VLSI Signal Processing Systems, 1, 1–15 (March, 2001)
© March, 2001 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Quantitative Analysis of FPGA-based Database Searching

N. SHIRAZI
Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124-3400, USA

D. BENYAMIN
Department of Electrical Engineering, UCLA, 56–125B, Engr. IV Bldg., Los Angeles, CA 90095, USA

W. LUK
Department of Computing, Imperial College, 180 Queen's Gate, London SW7 2BZ, UK

P.Y.K. CHEUNG
Department of Electrical Engineering, Imperial College, Exhibition Road, London SW7 2BT, UK

S. GUO
Philips Semiconductors, 811 E. Arques Ave. MS31, Sunnyvale, CA 94088-3409, USA

Editors: W. Burleson and N. Shanbhag

Abstract. This paper reports two contributions to the theory and practice of using reconfigurable hardware to implement search engines based on hashing techniques. The first contribution concerns technology-independent optimisations involving run-time reconfiguration of the hash functions; a quantitative framework is developed for estimating design trade-offs, such as the amount of temporary storage versus reconfiguration time. The second contribution concerns methods for optimising implementations in Xilinx FPGA technology, which achieve different trade-offs in cell utilisation, reconfiguration time and critical path delay; quantitative analysis of these trade-offs are provided.

1. Introduction

As the volume of information stored in databases continues to expand, fast database searching has become an important and necessary activity. For this reason, two database searching algorithms have been mapped to the Splash 2 FPGA-based custom computing machine [1]. These algorithms are a text searching algorithm that can process 20 million characters per second, and a fingerprint matching algorithm that can search through 250,000 fingerprint records per second. FPGA-based search engines have also been reported for biological databases [3], [8].

This paper reports two contributions to the theory and practice of search engines based on hashing techniques, which have been used in the two database searching algorithms on Splash 2 described above. Our first contribution concerns technology-independent optimisations involving run-time reconfiguration of the hash functions for such search engines. A quantitative framework is developed for estimating design trade-offs, such as the amount of temporary storage versus reconfiguration time. Our second contribution concerns methods for optimising implementations in Xilinx FPGA technology, which achieve different trade-offs in cell utilisation, reconfiguration time and critical path delay. Quantitative analysis of these trade-offs are provided.

There are several reasons for selecting database searching as our case study. First, for many applications including signal processing and computer vision, a database search facility often constitutes an essential component such that extracted data features can be matched against known ones. Second, some of our observations and analyses for database search, such as those in Section 4 on technology-independent analysis, can be adapted to cover other computations. Third, database searching is simple enough on the one hand to illustrate the principles involved, while complex enough on the other hand to provide a feel for how real systems may behave.

The paper is organised as follows. Section 2 motivates the use of FPGAs in implementing hash functions. Section 3 covers technology-independent optimisations involving run-time reconfiguration of such hash functions. Section 4 describes an analytical model for estimating design trade-offs. Section 5 presents methods for optimising implementations in Xilinx FPGA technology, while Section 6 contains quantitative analysis of the resulting trade-offs. Concluding remarks are provided in Section 7.

2. Hash Functions in FPGAs

The text searching application on Splash 2 tests a stream of words for inclusion and/or exclusion in a dictionary, given a predetermined list of keywords. The words are streamed through a series of FPGAs, each configured to implement a different hash function. These hash functions are set up to use a single bit on each attached memory module to represent the inclusion of a word in the search list.

The database search facility that we have developed is based on the version implemented on Splash 2 described above. This design performs a database search by using a hash function to map a word to a pseudo-random value. This value references a lookup table, indicating whether a given word is in the user dictionary. The lookup tables are generated by passing the user dictionary through the same hash functions that are used at run time.

FPGA-based computing platforms are well suited for this type of application. A hash function and lookup tables can be more efficiently implemented in an FPGA with an attached memory, as opposed to a general-purpose processor. The efficient implementation of the hash function is due to: (1) the size of the hash function not always being 32 or 64 bits in size, and (2) the irregular bit-level operations that are performed.

For instance, the core of the hardware implementation of a hash function is a barrel shifter. Barrel shifters of any size can be implemented efficiently using FPGAs. While on a microprocessor, barrel shifting operations are restricted to the size of its ALU – often 16 or 32 bits.

The hash function computation involves computing the XNOR or XOR of each bit of the input word with the current hash value according to the hash function mask, and then performing an n-bit circular shift. Figure 1 shows what happens when the word "the" is passed through a 22-bit hash function. These hash functions are set up to use a single bit to represent the inclusion of a word in the search list. A 22-bit hash function will be used in our examples in Section 5, since 2^{22} bits of memory has been shown to be sufficient for many uses [1]. The English language, for example, has around 2^{18} words, and 2^{22} bits of memory would allow a sparse scattering of words throughout the memory address space. Search facilities based on hash functions have shown to be effective for a variety of exact data matching.

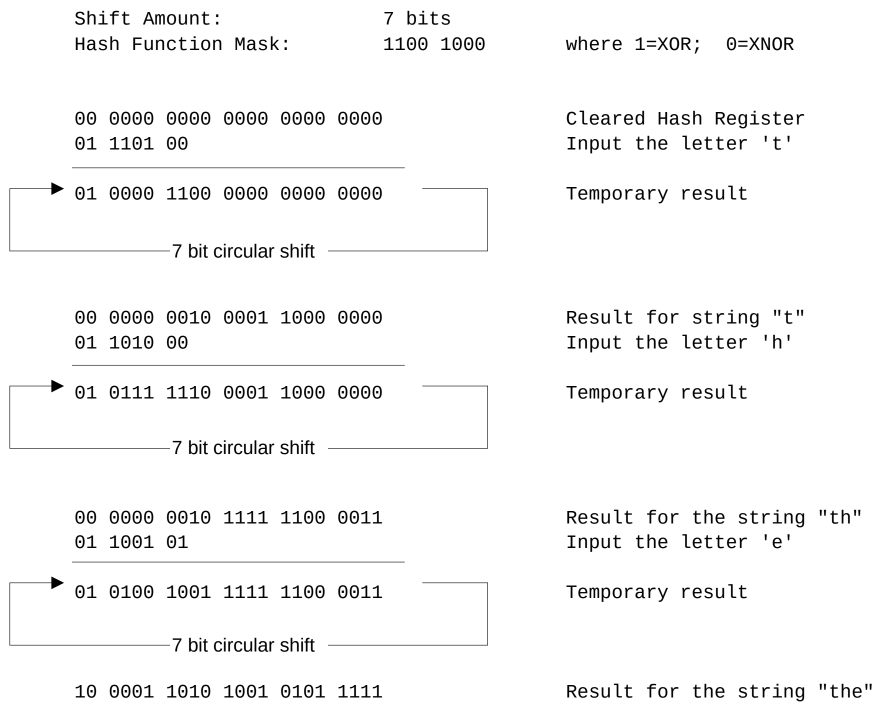
Shift Amount: 7 bits
Hash Function Mask: 1100 1000 where 1=XOR; 0=XNOR
00 0000 0000 0000 0000 0000 Cleared Hash Register
01 1101 00 Input the letter 't'
01 0000 1100 0000 0000 0000 Temporary result
7 bit circular shift
00 0000 0010 0001 1000 0000 Result for string "t"
01 1010 00 Input the letter 'h'
01 0111 1110 0001 1000 0000 Temporary result
7 bit circular shift
00 0000 0010 1111 1100 0011 Result for the string "th"
01 1001 01 Input the letter 'e'
01 0100 1001 1111 1100 0011 Temporary result
7 bit circular shift
10 0001 1010 1001 0101 1111 Result for the string "the"

Fig. 1. A 22-bit hashing example of the string “the”.

3. Database Searching

This section describes the use of run-time reconfiguration to optimise FPGA-based search engines. The system concerned should be partially and incrementally reconfigurable: the smaller the size of the reconfigured region, the lower the reconfiguration time.

It has been observed that cascading independent hash functions has the effect of reducing the probability of a false match [1]. The cascade can be implemented in two ways. The first implementation is a pipeline where multiple hash functions operate in parallel. While the parallelism is attractive, a pipeline with a fixed number of stages can have complications: when one of the stages detect a false match, the remaining pipeline stages will have to be skipped and become redundant. This requirement complicates the control of the pipeline, and the implementation may not be space efficient because of skipping.

The second implementation involves sequential execution of the hash functions. Only one hash function is used at one time; the results of each match is put into a temporary storage while the hardware is reconfigured to implement the next hash function (Figure 2).

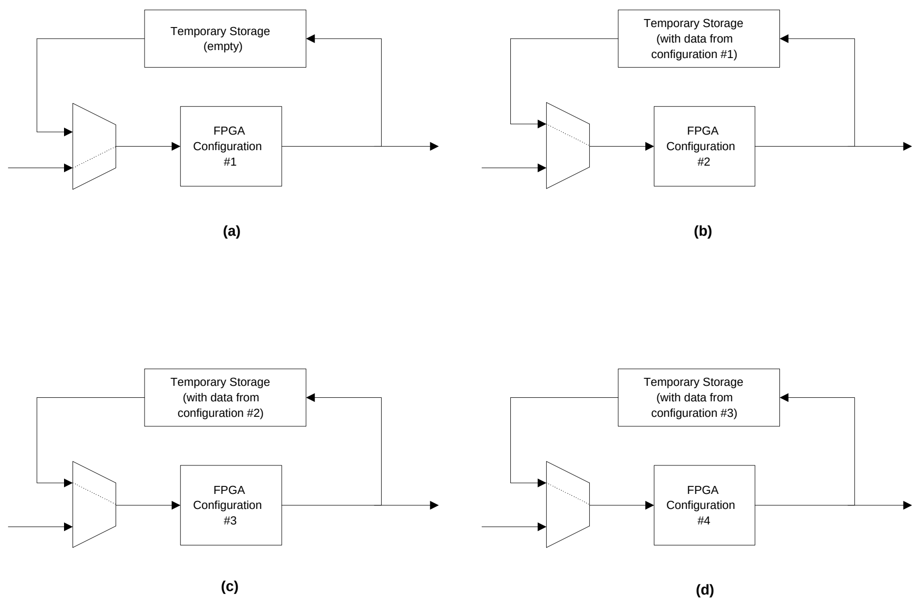
Temporary Storage (empty)
FPGA Configuration #1
(a)
Temporary Storage (with data from configuration #1)
FPGA Configuration #2
(b)
Temporary Storage (with data from configuration #2)
FPGA Configuration #3
(c)
Temporary Storage (with data from configuration #3)
FPGA Configuration #4
(d)

Fig. 2. Reconfigurable implementation of the sequential method. Each FPGA configuration implements a particular hash function, so that the input is passed through four hash functions before producing the output.

Note that the design operates in two modes: the "input" mode and the "feedback" mode. In the input mode shown in Figure 2(a), the hardware is connected to the external input and data pass through the first hash function implemented in the FPGA. Results are stored in the temporary storage, and the design then operates in the feedback mode so that the data can be processed by the second hash function, and so on, as shown in Figure 2(b)–(d). The design reverts to the input mode when existing data have been processed by all the appropriate hash functions.

The sequential method has the advantage that it may result in shorter execution time, since output can be produced as soon as possible and additional hash functions are not required to test for a false match. It also requires less resources than the pipeline method, since only one hash function is implemented in the FPGA. For these reasons, the sequential method has been chosen for our implementation.

Another reason for changing the hash functions at run time is to adapt to different computational loads to achieve the highest performance. Section 6 contains the analysis of a situation when different designs are used to minimise execution time, depending on the number of words to be searched and the amount of temporary storage available.

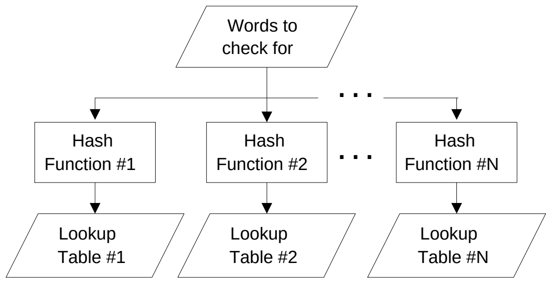

Fig. 3. Off-line generation of lookup tables from different hash functions.

It should be noted that the hash functions and the lookup tables are both produced off-line (Figure 3). Each hash function is implemented by an FPGA configuration, generated using design tools from the FPGA vendor and from recent research [6]. The configurations can then be loaded onto the FPGA, following the steps in Figure 2. The flowchart in Figure 4 shows in greater detail what happens during execution, when a set of input words are passed through one or more hash functions to check against the entries stored in the lookup tables.

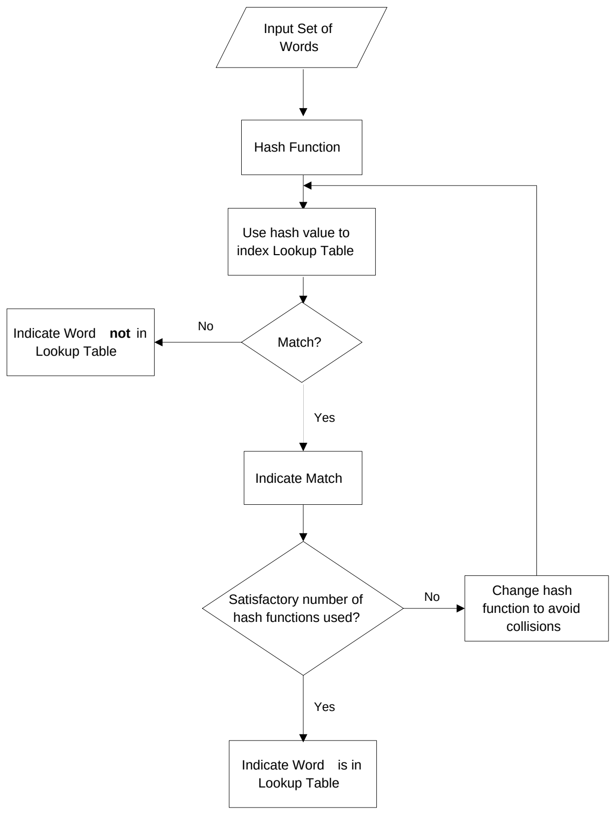
Input Set of Words
Hash Function
Use hash value to index Lookup Table
Match?
No
Indicate Word **not** in Lookup Table
Yes
Indicate Match
Satisfactory number of hash functions used?
No
Change hash function to avoid collisions
Yes
Indicate Word is in Lookup Table

Fig. 4. Flowchart showing the execution steps. The hash function and the lookup tables are generated off-line.

Run-time reconfiguration can be used to alter the hash function parameters and to switch between different hash functions. The two hash function parameters that can be altered at run time are the mask and shift values.

An example reconfiguration state machine for the sequential method is shown in Figure 5. The hash function parameters in each state are given. This diagram shows that the design will revert to its initial state when a false match occurs.

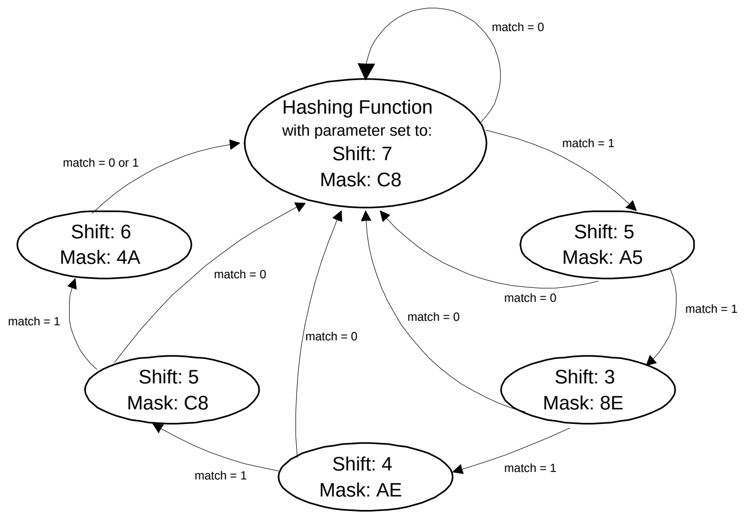

Fig. 5. Example of a reconfiguration state machine for the hash function used in the database searching application. Note that the initial state is at the top of the diagram.

Different hash functions will be optimised individually by partial evaluation [6] to take advantage of their constant coefficients. Since the hash functions are usually known at compile time, the combined reconfiguration method [9], which involves incrementally reconfiguring from one processing state to another, can be used.

A single bit of temporary storage is required for each input word to indicate whether the word is a match or not. If the hash function can itself be implemented as a pipeline, then the virtual pipeline method [7], which involves overlapping computation and reconfiguration, can be used in implementing the run-time reconfiguration.

The temporary storage can be implemented in many ways. If a large amount of temporary storage is required, then external memory can be used; otherwise on-chip registers or embedded memories within the FPGA may be sufficient. New FPGAs, such as Xilinx Virtex devices, contains various types of configurable memories which can be used in implementing temporary storage.

Our database engine, however, has been implemented on a platform [5] involving Xilinx 6200 FPGAs. The platform contains a Xilinx 6216 or a Xilinx 6264 device and four 8-bit wide memories organised into two banks. Each bank of memory can be accessed from either of the two separate address busses (Figure 6), and each of the four memories can be controlled individually. This memory architecture allows multiple modes of operation to be set-up by selecting multiplexers and bus switches for flow control in the desired manner; it has proved convenient in experimenting with temporary storage for the database search application covered by this paper.

A framework will be introduced in the next section for quantitatively assessing various design trade-offs, such as the amount of temporary storage versus the length of reconfiguration time. Device-specific aspects will be covered in Sections 5 and 6.

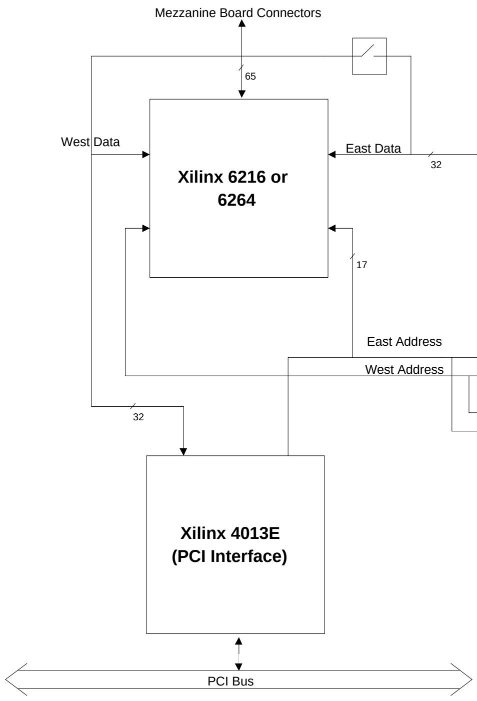

Fig. 6. Xilinx 6200 PCI system.

4. Technology-Independent Analysis

Under what circumstances is it worth using run-time reconfiguration within an application? We attempt to answer this question by quantifying the trade-offs between implementing the database searching application by using multiple FPGAs or a single partially reconfigurable FPGA. For this analysis, the design parameters that we will address are total execution time, FPGA area within a single FPGA or multiple FPGAs, and the amount of memory required for temporary storage to support data recirculation [7].

For this application, the input data set are divided into distinct parts to minimize the amount of temporary storage, and also to control the frequency of reconfiguration. We divide the total number of words, w, into l subsets of words, so each subset contains w/l words. This data subset is processed using a particular hash function, and one bit per word is used to indicate if a match has occurred. The indicator bit is stored along with the corresponding word in temporary memory. The temporary data are recirculated and processed by the next hash function, and the match indicator bit is updated on each iteration. This cycle is repeated until all the hash functions have been used. Once the cycle is completed, all temporary data are discarded and the next data subset is processed. The frequency of reconfiguration is controlled by the size of the data subset, because reconfiguration occurs once after each data subset has been processed.

In the equation describing the total execution time, we do not take into account the possibility that, if a match does not occur, the cycle is terminated and the remaining hash functions in the sequence are not used. Instead, we assume the worst case and execute all the hash functions in the reconfiguration state diagram.

The total execution time to process one subset of data, T_{subset}, is the sum of the reconfiguration time, T_{config}, and the processing time, T_{proc},

$$T_{subset} = T_{config} + T_{proc} \quad (1)$$

The configuration time is a product of the number of cycles required for reconfiguration, N_{config}, and the speed the FPGA can perform a single reconfiguration cycle, t_{config},

$$T_{config} = N_{config} \times t_{config} \quad (2)$$

Note that N_{config} can take into account device-specific optimizations, such as wildcarding in Xilinx 6200 devices.

The time to process a subset of data is a function of the size of the data set, w/l, the number of cycles needed to calculate the hash value, and the critical path of the hash function circuit t_{proc}. The number of cycles required to calculate the hash value of a word is determined by the number of cycles to access the lookup table in memory, m, and the average number of characters per word, c. The equation for the processing time T_{proc} is expressed as:

$$T_{proc} = \frac{w}{l}(m + c)t_{proc} \quad (3)$$

From Equation 1, the total execution time for one subset of data is given by:

$$T_{subset} = N_{config} \times t_{config} + \frac{w}{l}(m + c)t_{proc} \quad (4)$$

Since the execution order of the hash functions is not important, a number of them can execute as a reconfigurable pipeline to emulate the virtual pipeline of hash functions. The total execution time varies with the number of hash functions, h, and the number of hash functions that are executed in parallel, p. The ratio h/p is the number of hash functions that are being computed at one time. This ratio, together with the number of data sets to be processed l and the execution time for a single subset of data (Equation 4), result in the equation for the total execution time, T_{total}:

$$T_{total} = \frac{hl}{p}\left(N_{config} \times t_{config} + \frac{w}{l}(m + c)t_{proc}\right) \quad (5)$$

where h denotes the number of hash functions
p denotes the number of pipeline stages
l denotes the number of subsets of the complete input data set
N_{config} denotes the number of cycles needed for reconfiguration
t_{config} denotes the duration for a single reconfiguration cycle
w denotes the number of words
m denotes the number of cycles needed to access a lookup table in memory
c denotes the average number of characters per word
t_{proc} denotes the critical path of a hash function

Equation 5 is only valid for $w > 1$, $l \leq w$, $l > 0$ and $p \leq h$.

It would be useful to provide a measure of the amount of time spent on reconfiguration compared to the total time T_{total}. Assuming that the configuration cycle time t_{config} is the same as the execution cycle time t_{proc}, this measure is given by the ratio:

$$\frac{T_{config}}{T_{total}} = \frac{N_{config}}{N_{config} + w(m + c)/l} \quad (6)$$

The amount of temporary memory storage needed is based on the number of words in a subset of data, and the average number of characters per word of the subset. Also, an additional bit is added per word to indicate if a match has occurred. The total number of bits needed for temporary storage is given by:

$$Mem = \frac{w}{l}(b \times c) + \frac{w}{l} \quad (7)$$

where b is the number of bits per character, typically of value 7 or 8. The total circuit area is determined by the number of hash functions that are computed in parallel, p, and the size of each hash function, a,

$$Area = p \times a \quad (8)$$

Although Equations 5, 7 and 8 have been derived for the text searching application, they can be adapted for other applications. For instance, Equation 5 can be used for analysing another application, say F, provided that the term $(m + c)$ is replaced by a term denoting the number of cycles to compute F.

As shown in Figure 1, the circular shifter is the primary component in hash function computations. We therefore use the above equations in Section 6 to estimate the speed and size of shifter implementations for Xilinx 6200 FPGAs presented in the next section. Their efficiency will be calculated according to the functional density metric proposed by Wirthlin and Hutchings [10]: see Equation 9 below.

$$D_{Area} = \frac{1}{(Area)(T_{total})} \tag{9}$$

Since we are also interested in characterising the efficient use of temporary memory, an additional metric, given by Equation 10, is obtained by replacing the area term in Equation 9 by the amount of temporary memory used:

$$D_{Mem} = \frac{1}{(Mem)(T_{total})} \tag{10}$$

5. Device-Specific Mapping

The reconfiguration time of a circuit can be greatly reduced by taking into account the mechanism used to program an FPGA, as well as device-specific optimizations that may be available. This section introduces several shifter designs that can be used in one stage of a hash-function pipeline for a database search facility. The designs are implemented in Xilinx 6200 FPGA technology which supports partial run-time reconfiguration.

The Xilinx 6200 FPGA has a facility called 'wildcarding' that allows simultaneous configuration of multiple FPGA cells with the same data. Wildcarding can be performed on all 64 rows of the chip to allow an entire column to be configured in one configuration write cycle. However, wildcarding can only be performed on 4 columns at a time, therefore only 4 cells in a row can be configured at one time. Due to this limitation, it is advantageous to align components that are going to be reconfigured vertically along the columns of the FPGA in order to take maximum advantage of wildcarding.

A compact variable circular shifter has been implemented in Xilinx 6200 technology (Figure 7) [4]. The component is made up of an array of multiplexors that circularly shift the data n-bits; this design uses only the four nearest neighbour connections available in each FPGA cell.

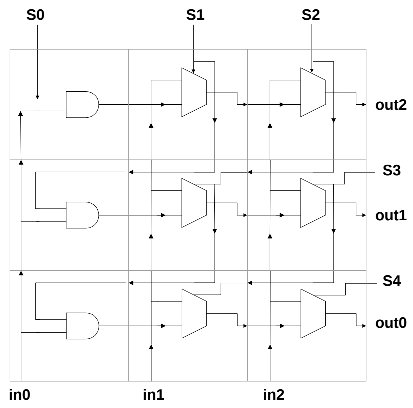

Fig. 7. Circuit diagram of a 3-bit Variable Circular Shifter. S0, S1, S2, S3 and S4 are signals controlling the shift amount.

The clock period for a 22-bit variable circular shifter is 105 ns; this large critical path delay is due to the control lines routed diagonally across the array to each of the multiplexors. The counter-flowing data organization makes this design difficult to be pipelined. For FPGAs with more abundant and faster routing resources, higher performance can be achieved with this implementation.

Without taking routing into account, this component requires n^2 FPGA cells, or 484 cells in the case of a 22-bit shifter. If routing is taken into account, the area of the 22-bit variable circular shifter grows to 638 cells. Since the component is constructed as a variable circular shifter, it only takes a single cycle to change the shift amount. The relevant statistics are summarised in the top row of Table 1.

Table 1. Trade-offs between different implementations of a 22-bit circular shifter.

Circuit	Clock Period t_{proc} (ns)	Area including routing (FPGA cells)	Reconfiguration Time N_{config} (number of cycles)
Variable	105	638	1
Fixed	29.5	88	245
Square Hybrid	37	484	44
Rhombus Hybrid	55	484	4

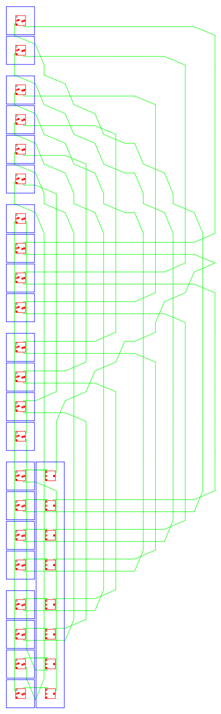

Fig. 8. Fixed Circular Shift Component on a Xilinx 6216 device. Note the abundance of random routing to perform the shift.

Three different implementations of the circular shifter have been developed to reduce their critical path delay. The first method uses constant propagation techniques. The variable n-bit circular shifter is converted to an n-bit constant circular shifter, by treating the variable shift amount as a constant. The propagation of this constant converts the array of multiplexors into simple wire connections.

For example, a 22-bit fixed circular shifter with a 7-bit shift amount has been designed by propagating the constant shift value into the circuit (Figure 8). The size of the component is reduced from 638 cells to 88 cells and its critical path from 105 ns to 29.5 ns. However, this shifter requires much random routing to perform the shift, and hence the time needed to reconfigure it is large.

The second method, known as the square hybrid method, is a hybrid between the variable implementation and the fixed version of the circular shifter. A two-dimensional array of buffers are used to control the routing of the component. The input data enter the component from the bottom and perform a corner turn and exit from the right. The shift amount is determined by the location of the corner turn in the array of FPGA cells; the corner turn is implemented by diagonally connecting the bottom inputs to the outputs on the right at different positions, resulting in two diagonally-placed reconfigurable regions (Figure 9). The reconfiguration time for this method is $2n$ cycles, where n is the size of the circular shifter, since it takes two clock cycles to make a new connection and remove the previous one.

The third method, known as the rhombus hybrid method, is similar to the square hybrid method except that the layout has been rearranged to enable fast partial reconfiguration. As previously discussed, the Xilinx 6200 FPGA supports wildcarding more effectively along a column of the chip rather than a row. The layout of the square hybrid implementation has been skewed into a rhombus shape, so that the previous diagonal reconfiguration regions are aligned into two columns (Figure 10). The number of cells used in the rhombus hybrid method is the same as that for the square hybrid method. However, due to its irregular shape, it is harder to take advantage of the unused triangular-shaped areas on either side of the component. Also, the input data have to propagate diagonally across the component; since the FPGA only has Manhattan-style routing, this increases the critical path.

The advantage of using the rhombus hybrid method is that the reconfiguration time is reduced to a constant value regardless of the size of the component, thanks to wildcarding. Wildcarding can be performed since the reconfiguration involves changing the reconfigurable region to identically configured cells. This reconfigurable region consists of buffers that either perform a corner turn or pass data from the left to the right. As shown in Table 1, the reconfiguration time for the rhombus hybrid method is an order of magnitude less than that for the square hybrid method, with the same circuit area and a modest reduction in clock speed.

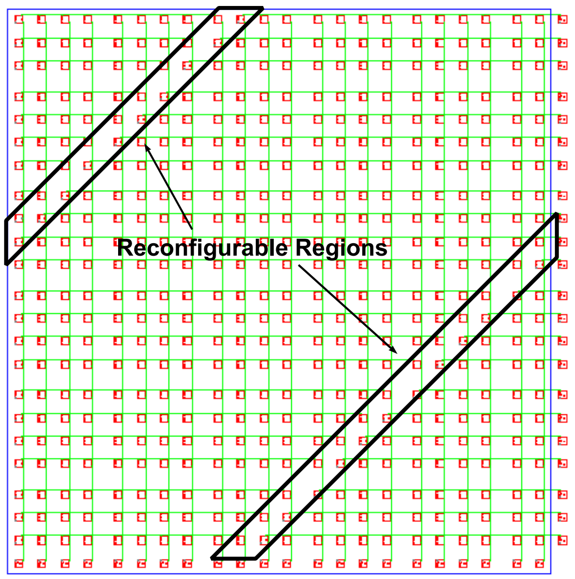

Fig. 9. Square Hybrid Implementation of a Fixed Circular Shift Component on a Xilinx 6216 device. The reconfigurable regions are enclosed by the two boxes.

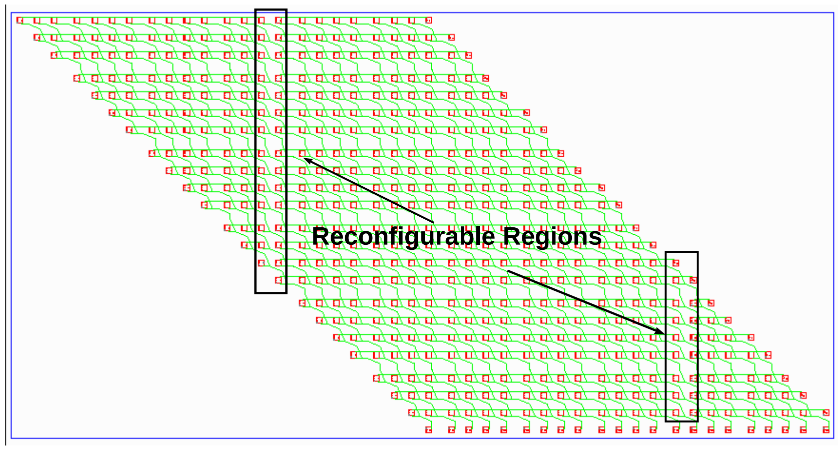

Fig. 10. Rhombus Hybrid Implementation of a Fixed Circular Shift Component on a Xilinx 6216 device. The reconfigurable regions are enclosed by the two boxes.

6. Device-Specific Design Analysis

Three test cases are used to illustrate the trade-offs between execution time and the amount of temporary storage required. The first test case assumes that there is a large amount of temporary storage available. The second test case examines the other extreme case and assumes that only a very small amount of temporary storage is available. In reality, there is usually a moderate amount of temporary storage available, either off chip or within the FPGA, and the third case examines this possibility.

Case #1. If a large amount of temporary storage is available, then we do not need to subdivide the input data set. Since the input data set is not subdivided, the amount of reconfiguration is minimized and reconfiguration only occurs after the complete data set has been processed. To specify this case, the value of l is set to 1 in Equation 5 in Section 4. This value indicates that the input data set is one complete set. A typical number of words to be searched is 10×10^6, hence $w = 10 \times 10^6$, and the average number of characters per word is set to 5, so $c = 5$. Since it takes one cycle to access memory, $m = 1$. The number of hash functions used to statistically ensure that false matches do not occur is 8, so $h = 8$, and we set $p = 1$ which indicates sequential execution of hash functions.

Using these parameters, along with the area and speed data given in Table 1, we calculate the total execution time, functional densities and the percentage reconfiguration time with respect to the number of cells and the amount of temporary storage for each of the circuits described in the preceding section. The results are shown in Table 2.

Table 2. Case #1: Total Execution Time, Functional Densities and Percentage Reconfiguration Time when $l = 1$.

Circuit	T_{total} (sec)	D_{Area} $(1/(cells \times sec))$	D_{Mem} $(1/(bits \times sec))$	T_{config}/T_{total} (%)
Variable	50.4	0.31×10^{-4}	0.48×10^{-10}	1.7×10^{-6}
Fixed	14.16	8.03×10^{-4}	1.72×10^{-10}	4.1×10^{-4}
Square Hybrid	17.76	1.16×10^{-4}	1.37×10^{-10}	7.3×10^{-5}
Rhombus Hybrid	26.4	0.78×10^{-4}	0.92×10^{-10}	6.7×10^{-6}

Since reconfiguration time is very small compared to execution time, the circuit with the smallest critical path, the Fixed Circular Shift Component, is the fastest design. In this case, the critical path delay is the dominating factor in the overall computation time. The Fixed Circular Shift Component also has the maximum functional density with respect to both area and temporary storage size of the four different implementations. The disadvantage of using this method is that, in this case, 48 megabytes of temporary storage are required.

Case #2. If limited temporary storage is available, frequent reconfiguration minimizes the amount of temporary storage needed. In this case, we examine the extreme case by reconfiguring to the next hash function after every word. This is done by dividing the input data set into the smallest possible unit, a single word, by setting $l = w$. When $l = w$, Equation 7 is reduced to $Mem = (b \times c) + 1$, and therefore only 41 bits needs to be stored between reconfigurations. This can be achieved using a single register on the FPGA, and off-chip temporary storage is not required.

Table 3. Case #2: Total Execution Time, Functional Densities and Percentage Reconfiguration Time when $l = w$.

Circuit	T_{total} (sec)	D_{Area} $(1/(cells \times sec))$	D_{Mem} $(1/(bits \times sec))$	T_{config}/T_{total} (%)
Variable	58.8	2.7×10^{-5}	4.15×10^{-4}	14
Fixed	592	1.9×10^{-5}	0.412×10^{-4}	98
Square Hybrid	148	1.4×10^{-5}	1.65×10^{-4}	88
Rhombus Hybrid	44	4.7×10^{-5}	5.54×10^{-4}	40

Using the same parameters in Case #1, we calculate the total execution time, functional densities and percentage reconfiguration time with respect to the number of cells and the amount of temporary storage for each of the circuits (Table 3). In this case, since reconfiguration time is large compared to processing time, the reconfiguration time of the circuit is the dominating factor in the overall execution time; in particular it can be seen from Table 3 that the Fixed method involves the largest percentage reconfiguration time as well as the largest total execution time.

An implementation that involves the minimum amount of run-time reconfiguration may be expected to have the fastest overall computation time. However, this is not the case in this example, since the Rhombus Hybrid method involves a larger percentage reconfiguration time than the Variable method, but it still has the overall fastest execution time. The small reconfiguration time of the Rhombus Hybrid method is due to the optimizations discussed in Section 3.

Table 4. Case #3: Total Execution Time, Functional Densities and Percentage Reconfiguration Time when $w/l = 16$.

Circuit	T_{total} (sec)	D_{Area} $(1/(cells \times sec))$	D_{Mem} $(1/(bits \times sec))$	T_{config}/T_{total} (%)
Variable	50.9	0.31×10^{-4}	2.99×10^{-10}	1
Fixed	50.3	2.26×10^{-4}	3.03×10^{-10}	72
Square Hybrid	25.9	0.79×10^{-4}	5.88×10^{-10}	31
Rhombus Hybrid	27.5	0.75×10^{-4}	5.54×10^{-10}	4

Case #3. A more realistic scenario occurs when there is limited amount of temporary storage and the optimal implementation has to be found. For example, if there is enough left-over chip area for 100 bytes of temporary storage on the FPGA, we need to know which one of the four circuits would have the fastest execution time. The same parameters in Case #1 yield a value of l to be approximately $1/16^{th}$ of w. Again, total execution time, functional densities and percentage reconfiguration time for each of the circuits are calculated and are shown in Table 4. We find that the circuit with the fastest execution time, given these parameters, is the Square Hybrid Circular Shift circuit. It involves the second largest percentage reconfiguration time, showing once more that large amount of reconfiguration does not necessarily increase the total execution time.

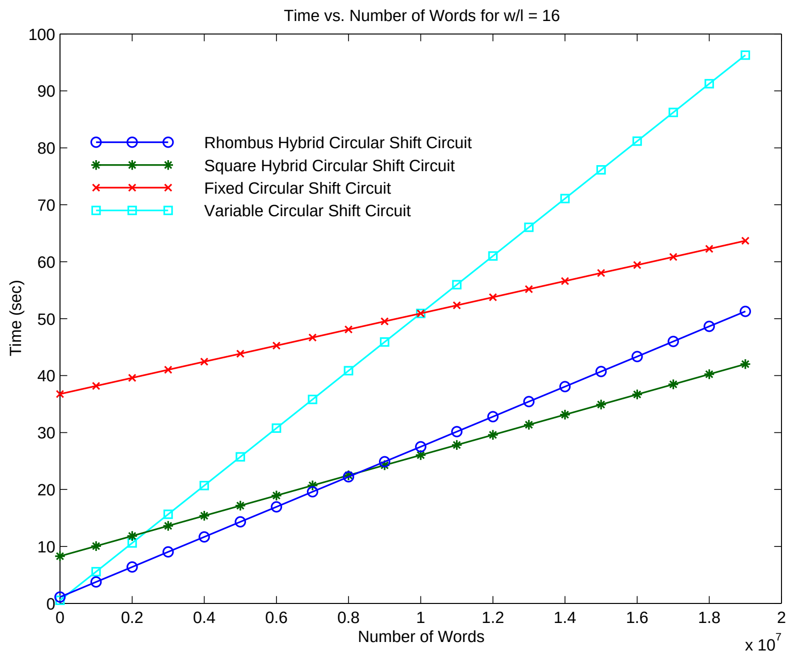

Fig. 11. Graph of Total Execution Time (T_{total}) versus Number of Words (w) for $w/l = 16$.

For $w/l = 16$, a plot of execution time versus the number of words is shown in Figure 11. If data partitioning is kept at $w/l = 16$, and fewer than 10^7 words are processed, we note from Figure 11 that the Rhombus Hybrid method is the circuit with the fastest execution time. In this case we can reconfigure the FPGA to use this circuit until the cross over point is reached where the Square Hybrid circuit is the faster circuit.

The test cases do not cover the use of multiple FPGAs executing the text search in parallel as a pipeline. However, by changing the value of p, this case can be explored using the same method described in this section.

Equations 5, 7 and 8 enable us to find the circuit which is the most appropriate under given constraints, such as the availability of FPGA resources, by quantifying the trade-offs between the amount of temporary storage and execution time for this application. Other applications such as image processing can also be explored using these methods.

7. Concluding Remarks

This paper describes how run-time reconfiguration can be used to optimise database search engines. Both technology-independent and device-specific aspects are covered by our framework, which supports quantitative analysis of design trade-offs in performance and resource usage.

Current and future work includes generalizing our analysis techniques to cover other applications and devices. Our approach can also be extended to exploit contextual information in domains such as multimedia processing, and to take advantage of various circuit optimisations such as on-line arithmetic [8].

Acknowledgements

The authors are indebted to John Gray, Douglas Grant, Hamish Fallside, Tom Kean, Steve McKeever, Stuart Nisbet, Bill Wilkie and the anonymous reviewers for their constructive comments. The support of Xilinx Inc., the UK Overseas Research Student Award Scheme and UK Engineering and Physical Sciences Research Council (Grant number GR/24366, GR/54356 and GR/59658) is gratefully acknowledged.

References

1. D. Buell, J. Arnold and W. Kleinfelder, *Splash 2, FPGAs in a Custom Computing Machine*, IEEE Computer Society Press, 1996.
2. S. Churcher, T. Kean and B. Wilkie, "The XC6200 FastMap processor interface," in *Field Programmable Logic and Applications*, W. Moore and W. Luk (eds.), LNCS 975, Springer, 1995, pp. 36–43.
3. E. Lemoine and D. Merceron, "Run time reconfiguration of FPGAs for scanning genomic databases," *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, IEEE Computer Society Press, 1995, pp. 90–98.
4. W. Luk, S. Guo, N. Shirazi and N. Zhuang, "A Framework for developing parametrised FPGA libraries," in *Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, LNCS 1142, Springer, 1996, pages 24–33.
5. W. Luk, N. Shirazi and P.Y.K. Cheung, "Modelling and optimising run-time reconfigurable systems," *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, IEEE Computer Society Press, 1996, pp. 167–176.
6. W. Luk, N. Shirazi and P.Y.K. Cheung, "Compilation tools for run-time reconfigurable designs," *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, IEEE Computer Society Press, 1997.
7. W. Luk, N. Shirazi, S.R. Guo and P.Y.K. Cheung, "Pipeline morphing and virtual pipelines," *Field Programmable Logic and Applications*, W. Luk, P.Y.K. Cheung and M. Glesner (eds.), LNCS 1304, Springer, 1997, pp. 111–120.
8. E. Mosanya and E. Sanchez, "A FPGA-based hardware implementation of generalized profile search using online arithmetic," *Proc. ACM/SIGDA Int. Symp. on FPGAs*, ACM Press, 1999, pp. 101–111.
9. N. Shirazi, W. Luk and P.Y.K. Cheung, "Run-time management of dynamically reconfigurable designs," *Field Programmable Logic and Applications*, R. W. Hartenstein and A. Keevallik (eds.), LNCS 1482, Springer, 1998, pp. 59–68.
10. M.J. Wirthlin and B.L. Hutchings, "Improving functional density through run-time constant propagation," *Proc. ACM Int. Symp. on FPGAs*, ACM Press, 1997, pp. 86–92.