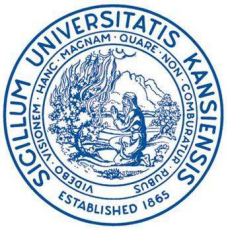
Design and Evaluation of a Scalable and Portable, Reconfigurable Computing Implementation of BLAST

Parag Beeraka



Submitted to the Department of Electrical Engineering & Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master's of Science

Thesis Committee:

Dr. Ron Sass: Chairperson

Dr. David Andrews

Dr. Xue-wen Chen

Date Defended

2006/04/19

The Thesis Committee for Parag Beeraka certifies
That this is the approved version of the following thesis:

Design and Evaluation of a Scalable and Portable, Reconfigurable Computing Implementation of BLAST

Committee:

_____

Chairperson

_____

_____

_____

Date Approved

Abstract

Basic Local Alignment Search Tool (BLAST) is a standard computer application that molecular biologists use to search for sequence similarity in genomic databases. This thesis describes a FPGA-based hardware implementation of the BLAST application. The main objective of this document is to explore the feasibility of using this new technology to realize a scalable, portable and cost-effective FPGA-based accelerator for the BLAST Algorithm. Since it is not practical to map the entire application to hardware, a profile study was conducted that identifies the computationally intensive part of BLAST. This computationally intensive critical segment has been designed and implemented in the FPGA while the rest of the application runs on a PowerPC processor in the FPGA. The concepts of portability, scalability and cost-effectiveness of the implementation are also demonstrated from the results obtained.

Contents

List of Figures

Chapter 1

Introduction

In the late 1970's, the fields of computer and information science and biology came together to give rise to the new field of bioinformatics [10]. Bioinformatics uses computers to advance the scientific understanding of living systems and it does so more accurately and more productively than what was previously possible (via manual techniques). The discipline is increasingly playing a key role in fields as varied as:

- Molecular biology
- Genomics
- Functional genomics
- Systems biology
- Protein design and engineering
- Pharmaceutical development
- Medicine
- Ecology

- Population genetics
- Agriculture

With the advent of bioinformatics, scientists have the possibility of studying the genome in a number ways by applying already established analytical and theoretical techniques, such as mathematical models and computational simulation, to digital representations of the genome. These computational techniques aid the process of genome sequencing and similarity searches. The main focus of this thesis is to improve the computational aspect of such tools that would enhance the productivity of scientists and biologists.

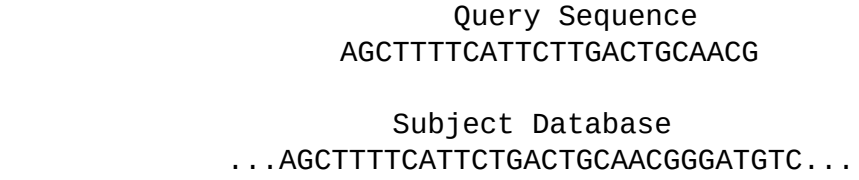
For example, after sequencing the DNA of an organism, a scientist interested in learning the function of a particular gene might start by checking to see if its sequence is similar to any other gene of a data look for this new information helps scientists and biologists discover the function of a new gene, reveal gene functions, uncover relationships between organisms, and other related scientific activities by comparisons of sections of sequence. Prior to the development of the field bioinformatics or computational biology, biologists used to manually compare the genes (queries) with databases (sequences) of genes. This is indeed is a very time consuming process and more over the probability of human error is very high. The sequence of gene determines its function, so finding similar sequences provides clues to a new sequence's function. Moreover, the initial assembly of a new sequence relies on a large number of similar comparison operations.

BLAST is an acronym for Basic Local Alignment Search Tool [6]. Although BLAST originated at the Washington University in St. Louis, its development continues at various institutions, both academically and commercially. BLAST is one of the tools used by scientists in order to find regions of local similarity between sequences i.e.

the comparison operation described above. It compares nucleotide or protein sequences to sequence databases and calculates the statistical significance of matches. The results obtained from a BLAST search have a well-defined statistical interpretation, making real matches easier to distinguish from random background hits. BLAST uses a heuristic algorithm which seeks local as opposed to global alignments and is therefore able to detect relationships among sequences which share only isolated regions of similarity. However various reports and journal papers [37] establish the fact that these results are appropriate for a scientist to determine the significance of the search. Since BLAST algorithm is based on heuristics, it is not the most accurate tool in similarity searching, but it still remains as the most widely used tool among biologists because of the nature of the results that it produces.

In order to know the importance of BLAST in a practical perspective, let us assume that various biologists and scientists discovered few new microbes from various field experiments. In order to find out the genes present in the new microbes, one can run a BLAST similarity search on the newly found genes and various known genome databases Figure 1.1 is an example illustration the problem described above. In the Figure 1.1 , the query can be considered as a part of an unknown gene found from the field experiments and the subject database as part of a genome database.

In Figure 1.1, the first example compares a query sequence of length 24 which might be a part of the unknown gene sequences of the microbes found from the field experiments with a part of the subject sequence of length 23. From the results produced by BLAST, it can be interpreted that there is no perfect match between the query sequence and the subject sequence, but there is an identity match of approximately 95% with a score of 32.2 given a gap of ’1’ between the two sequences. Moreover, the Expect value of 0.005 indicates that a sequence with a similar score is unlikely to appear in the



(a)

```
Score = 32.2 bits (16), Expect = 0.005
Identities = 23/24 (95%), Gaps = 1/24 (4%)
Strand = Plus / Plus
Query: 1  agcttttcattcttgactgcaacg 24
          |||||||||||| ||||||||||
Sbjct: 1  agcttttcattc-tgactgcaacg 23


Score = 22.3 bits (11), Expect = 9.9
Identities = 11/11 (100%)
Strand = Plus / Minus
Query: 3    tgactgcaacg 13
            ||||||||||
Sbjct: 6627 tgactgcaacg 6637
```

(b)

Figure 1.1. Example Query - Subject Database Comparison

subject sequence. So based on these results obtained from BLAST, the scientists gather enough information in order to proceed in the right direction about the new microbes. From the example mentioned above, it is also clear that BLAST can be used to infer functional and evolutionary relationships between sequences as well as help identify members of gene families.

BLAST software is open source and runs on various high end machines to low level desktop machines which makes the tool more powerful. However, since speed is vital in making the algorithm more practical, most medium to large bioinformatics laboratories also include special-purpose computing machines to run applications like BLAST. As one would expect, a special-purpose equipment with custom hardware which is much faster than BLAST running on a general-purpose computer. Various labs use these type of machines [41] when turn-around time is important or, when they want to manually prioritize their jobs. However, the issue with BLAST now is the time to search, the database is essentially proportional to the size of the subject database. Therein lies an important problem, even though Moore's Law has processor performance doubling every 18 months (a compound annual growth rate of 59%), biological databases are growing even faster. Figure 1.2 shows the growth rates of several key indicators since 1994 on a semi-log graph. The data points come from GenBank [21], a public collection of sequenced genomes. A line fitted to this data shows a compound annual growth rate of 77%. Also important to note is the growth rate of I/O subsystem (disk and interface). The most aggressive estimates [11] suggest a 10% compound annual growth rate in performance while others [26] suggest a more modest 6% growth rate. Regardless, these trends have an important consequence in answering the same question (is this gene similar to any known gene?) will take longer every year. In other words, the problem size is growing faster than single processor performance and much faster than

I/O subsystems.

If Moore's Law does not provide enough compute power, one naturally would seek a parallel solution. However simply using multiple computers (perhaps organized as a cluster) presents its own problems. For example as Figure 1.2 shows, I/O performance is an issue. So a cluster must choose between expensive I/O subsystems on every node or a shared network file system that is generally an order of magnitude slower than a local disk. Second problem is cost, although commodity cluster nodes are getting less expensive, every node has a set of basic costs: local hard drive, local RAM, power supply, case etc. While cost gets multiplied with the number of nodes, but they do not directly contribute to the solution.

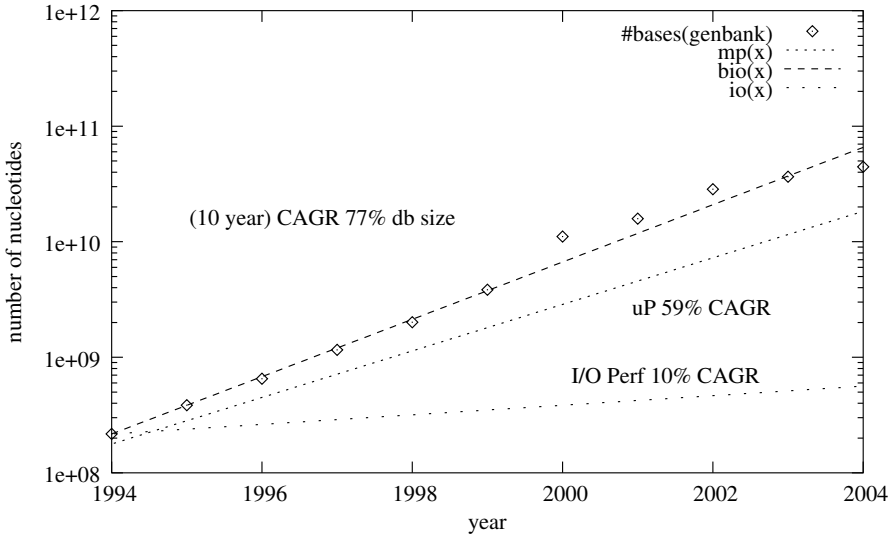


Figure 1.2. Compound annual growth rate of database problem size, single processor performance, and I/O subsystem performance

So the question is can the problem expressed be dealt in an effective way by FPGA's. The term *effective* in the above sentence includes the terms *scalable* and *cost-effective* as its main objective. Reconfigurable Computing (RC) is an area of computing where in FPGA's are used as computational devices. FPGA's are devices with a sea of logic

gates and interconnects between the gates. In this field of computing the programming is done at the gate level usually using a hardware description language (HDL) rather than at the machine instruction level by high level or low level languages. FPGA's also advance by Moore's law in area and clock frequency of the die but unlike FPGA's, ordinary microprocessors use the transistors as memory units instead of using them as computational units. But in the case of FPGA's the more the number of transistors available more is the potential for the number of computational units [45]. Moreover research has also demonstrated that FPGA's are well-suited to processing bioinformatics applications in general [36] [38] [30]. However, BLAST specifically has not been addressed.

This brings us to the central question of the work, if BLAST is so critical to scientists and biologists and given all the advantages of using FPGA's over microprocessors, *Is a scalable and cost-effective Reconfigurable Implementation of BLAST feasible which can aid scientists and biologists in a more productive way?*. Implicit in the term *scalable* is also how well the design can scale as technology progresses over the years according to Moore's Law which continues to give double the number of transistors every 18 months. And implicit in the term *cost-effective* is the measure of the price of computing power which also means the amount of performance obtained per dollar ($) when compared to other implementations over various other architectures available.

And the answer for the question is explained in the subsequent chapters that follow with Chapter 2 providing the context for the work by describing related work, a general background on similarity searches and FPGA's in regard to Reconfigurable Computing. In chapter 3, the current BLAST application is analyzed and explained. The computationally demanding parts of the application are identified and targeted for hardware implementation. The general hardware design is explained. Later on in chapter 3, the

resulting performance of the basic design is shown. Specifically, the scalability and cost-effectiveness of the design are analyzed. From the analysis of results, conclusions along with a brief description of future work is explained in chapter 5.

In order to answer the above question and add knowledge to the body of science, these are the list of contributions as part of the thesis.

- Repeated and verified the profiling characteristics as specified in *Design and Implementation of Open source FPGA-based accelerator for BLAST* [33].
- Extended the profiling characteristic to several machines with different processors, memory, disk and network interfaces.
- Helped to quantify the I/O bound characteristic of BLAST by running it across different file systems with different bandwidths.
- Ported BLAST to Xilinx ML-310 Platform FPGA development board and characterized the essential things needed to do a port
  - Developed a hardware design which is scalable and cost-effective .
  - Ported Linux to a Platform FPGA .
  - Developed a device driver that accesses the hardware from the software .
  - Developed a patch to the BLAST software needed to invoke the hardware .
  - Implemented and measured the pros & cons of using the On - Chip BRAM's as a cache in order to speed up BLAST .
  - Proved the concept of portability, by a dummy port to a Xilinx ML - 403 development board .

Chapter 2

Background

This chapter provides information on two aspects of the general problem. First we provide a brief introduction to the general problem and current software solutions. Second, we describe FPGA's and Reconfigurable Computing in general. Together we provide a context for the work described in the next chapter.

2.1 Bioinformatics and Similarity Searching

2.1.1 Genetics and Sequencing

Instructions that provide almost all of the information necessary for a living organism to grow and function are in the nucleus of every cell, so a cell is usually described as the smallest living organism. These instructions tell the cell what role it plays in a body. The instructions are in the form of a molecule called deoxyribonucleic acid, or DNA. DNA consists of two long, twisted chains made up of nucleotides which act as a main building block. The bases in DNA nucleotides are adenine, thymine, guanine, and cytosine which are represented by the letters A,T,C,G respectively and they can be strung together in billions of ways.

```
A single strand of DNA is made of letters:

        ATGCTCGAATAAATGTGAATTTGA

These letters make words:

        ATG CTC GAA TAA ATG TGA ATT TGA

These words make sentences :

        < ATG CTC GAA TAA > < ATG TGA ATT TGA >
```

Figure 2.1. Sample DNA Sequence

As shown in Figure 2.1 sentences which constitute pure data of a DNA sequence make genes and the scientific study of genes is called genetics. So, every organism, including humans, has genomes that contains all of the biological information needed to build and maintain a living example of that organism. Genes tell the cell to make other molecules called proteins and proteins are required for the structure, function, and regulation of the body's cells, tissues, and organs.

Much effort in the field of genetics continues to be spent locating genes and moreover the rate of accumulation of sequence data is exponentially growing. This has been partly due to the fact that the technology to carry out DNA sequencing has rapidly advanced. With the technology available to date, the entire job can be carried out by robots - from an input of tissue, the robots can automatically extract the DNA, amplify regions of interest, and prepare sequence cocktails. These are then loaded onto the gels of automatic sequencing machines. These machines will run the gels, a laser scans the gels and calculates the DNA sequence, finally the sequence is automatically entered into a computer, and the computer will automatically assemble the fragments and may also do some preliminary analysis. In addition, the number of laboratories that

routinely sequence DNA has also increased.

Today's genetics investigates various aspects at the genome level and bioinformatics has made the job of genetics quite easier since its advent. Since the sequence of DNA encodes the necessary information for living things to survive, determining a sequence is therefore useful in research into why and how organisms live, as well as in applied subjects. For example if it is said that a particular organism has 'X' percent of it's DNA with humans then the number indicates the percentage of DNA identical within the two which indirectly indicates that that particular organism has 'X' amount of that particular functionality similar to human beings. So if the same described above is related to an unknown gene sequence, various scientists and biologists would compare the unknown sequence against various other sequences in order to determine the functionality of the new gene or at-least could determine a hypothesis based on the results obtained. But this is not simply a case of finding an identical match or not simply a case of finding a slight mismatch. Indeed it's a case of finding enough of a match in part of the sequence, such that the partial match is statistically unlikely to happen by any chance. So the way sequence matching is done is by sequence alignment. So in terms of the process of sequence alignment all the possible alignments are looked at and a score is assigned to each of them and finally the alignment with the best score or above a certain a threshold are usually the ones taken into consideration.

Since the above process is highly compute bound, various search algorithms have been defined and various software tools have been designed so that all the sequence matching can be done by computers instead of biologists.

2.1.2 Database search algorithms and Tools

The most important Database search algorithms present are

- Smith - Waterman Algorithm
- FASTA
- BLAST

2.1.3 Smith Waterman Algorithm

The Smith-Waterman (SW) algorithm [39] is a search algorithm developed by T. F. Smith and M. S. Waterman and the algorithm implements a technique called dynamic programming, which takes alignments of any length, at any location, in any sequence, and determines whether an optimal alignment can be found. The algorithm compares two sequences by computing a distance that represents the minimal cost of transforming one segment into another. Two elementary operations are used: substitution and insertion/deletion, also called a gap operation. Through series of such elementary operations, any segments can be transformed into any other segment. The smallest number of operations required to change one segment into another can be taken into as the measure of the distance between the segments. Based on these calculations, scores or weights are assigned to each character-to-character comparison: positive for exact matches/substitutions, negative for insertions/deletions. Scores are added together and the highest scoring alignment is reported. The scoring scheme in this algorithm is based on exact matches and gaped matches. An ungapped scoring scheme has two values, one positive value for a match, and a negative penalty value for a mismatch. A gaped scoring scheme has three values, a positive match score, a negative gap initiation score, and a negative gap extension score (with the extension score smaller in magnitude than the initiation score) . Instead of looking at an entire sequence at once, the algorithm compares multi lengthed segments, looking for whichever segment maximizes the scoring measure. It is superior to other algorithms because it searches a larger field of possibil-

ities, making it a more sensitive technique; however, individual pair-wise comparisons between letters slows the process down significantly because these computations require execution time in the order of quadratic time.

Choose two strings $S1$ and $S2$ of length l_1 and l_2 . To identify common subsequences, the SW algorithm computes the similarity $H(i, j)$ of two sequences ending at position i and j of the two sequences $S1$ and $S2$.

The computation of $H(i, j)$ for $1 \leq i \leq l_1, 1 \leq j \leq l_2$. is given by the following recurrences.

$$H(i,j) = max\{0, E(i,j), F(i,j), H(i-1,j-1) + Sbt(S1_i, S2_j)\}$$
$$E(i,j) = max\{H(i,j-1) - \alpha, E(i,j-1) - \beta\}$$
$$F(i,j) = max\{H(i-1,j) - \alpha, F(i-1,j) - \beta\}$$
$$H(i,j) = max\{0, E(i,j), F(i,j), H(i-1,j-1) + Sbt(S1_i, S2_j)\}$$

where Sbt is a character substitution cost table. Initialization of these values are given by $H(i, 0)$ = $E(i, 0)$ = $H(0, j)$ = $F(0, j)$ = 0 for $0 \leq i \leq l_1, 0 \leq j \leq l_2$. Multiple gap costs are taken into account as follows: α is the cost of the first gap; β is the cost of the following gap. This type of gap cost is known as affine gap penalty and there exists a different type of gap penalty called linear gap penalty where α = β. For linear gap penalties the above recurrence relations can be simplified to

$$H(i,j) = max\{0, H(i,j-1) - \alpha, H(i-1,j) - \alpha, H(i-1,j-1) + Sbt(S1_i, S2_j)\}$$
$$for\{1 \leq i \leq l_1, 1 \leq j \leq l_2.\}$$

$$H(i,0) = H(0,j) = 0 \textit{ for } \{0 \leq i \leq l_1, 0 \leq j \leq l_2.\}$$

Each position of the matrix H is a similarity value. The two segments of $S1$ and $S2$ producing this value can be determined by a back tracing procedure.

2.1.4 FASTA

FASTA [17] is a search algorithm developed by David J. Lipman and William R. Pearson in 1985. The FASTA algorithm is a heuristic approximation to the Smith-Waterman algorithm. The heuristics used by FASTA allows it to run much faster that the Smith-Waterman algorithm but at the cost of some sensitivity. The first step in the FASTA algorithm is to divide the query sequence into its constituent overlapping words of length which is two for proteins or six for nucleic acids. Then as each sequence is read from the database it is also divided into its constituent overlapping words. These two list of words are compared to find the identical words in both sequences. Indeed this comparison can be viewed as a set of dot plots, with the query as the vertical sequence and the group of sequences to which the query is being compared as the different horizontal sequences. An initial score is computed based on the number of identities concentrated within small regions of the dot plot. If this initial score is high enough, then a second score is computed by evaluating which of the initial identities can be joined into a consistent alignment using only gaps of less than some maximum length. Finally, if the secondary score is high enough then a Smith-Waterman alignment is performed within the region of the dot plot defined by the concentrated identities and this is the final score based on which matches are determined. Thus, significant speedups observed in a FASTA search relative to a full Smith-Waterman search are due to the prior restriction in alignment space.

2.1.5 BLAST

The BLAST algorithm is used as the most popular search algorithm for searching queries against biological sequence databases [25,6]. After being released by the Washington University at St. Louis, the BLAST algorithm has resided and further developed by The National Center for Bioinformatics Information (NCBI) and is available at this website **http://www.ncbi.nim.nih.gov/BLAST/**. The approach used by the BLAST algorithm is to first identify short segments with high-scoring alignments without gaps, and then to extend each such local alignment as far as possible in both directions, with or without gaps, so long as the score resulting after each new extension remains sufficiently large. The method then evaluates the statistical significance of all such high-scoring matches and reports as hits only those that satisfy a pre-selected threshold of significance. More precisely, BLAST begins by dividing the input query sequence into all possible contiguous sub-sequences (called "words") of length w (called the "word length"). The value of w depends on the type of sequence involved. For a given database sequence, BLAST searches for sub-sequences that exactly match one of the words. When such a match is found, the search process is suspended, and BLAST tries to extend the sub-sequence match in both directions (possibly introducing gaps), so long as the score for the extended match does not decrease significantly from the score for the original word match. Ultimately, the extension process terminates either when the end of one of the sequences is reached, or when the score has diminished sufficiently. If the score at that point is high enough, then the extended match is tentatively included in the hit list and the search for word matches resumes. When all word matches have been processed for a given database sequence, that sequence is discarded, and the algorithm starts on the next sequence in the database. At the end of the entire process, BLAST reports the hit list along with various overall statistics. The results

obtained from a search of BLAST has various parameters which have very important significance. The various parameters are

- Score :- Score or bit score is a value calculated from the number of gaps and substitutions associated with each aligned sequence. The higher the score, the more significant the alignment. Each score links to the corresponding pairwise alignment between the query sequence and subject sequence

- E Value :- E Value (Expect Value) describes the likelihood that a sequence with a similar score will occur in the database by chance. The smaller the E Value, the more significant the alignment. For example, a E value of e-117 is a very low E value meaning that a sequence with a similar score is very unlikely to occur simply by chance.

- Strand :- Any DNA molecule that is double-stranded means genes may occur on either strand. The two strands are the plus strand and the minus strand. The minus strand is the reverse complement of the plus strand. If the similarity between the query sequence and the subject database is on the same strand, it is a given strand value of *plus/plus*. If the minus strand of the query sequence is similar to the database sequence, it is a given a *plus/minus* .

Karlin-Altschul [37] statistics have been extrapolated to the task of assessing the significance of hit scores obtained from comparisons of biological sequences in BLAST. Moreover, the implementation of BLAST from NCBI can perform five different types of similarity searches, corresponding to different combinations of sequence types in the input queries and databases. Table 2.1 shows the various available search types within BLAST. All these are specified as a command line parameter when running BLAST.

Table 2.1. Various search options in BLAST

Search Name	Query Type	Database Type	Translation
blastn	Nucleotide	Nucleotide	None
tblastn	Peptide	Nucleotide	Database
blastx	Nucleotide	Peptide	Query
blastp	Peptide	Peptide	None
tblastx	Nucleotide	Nucleotide	Query and Database

The BLAST algorithm is an improvement over the similar FASTA algorithm by offering advantages such as speed, sensitivity, matches having an estimate of statistical significance.

2.2 FPGAs and Reconfigurable Computing

2.2.1 Introduction to FPGA's

A field programmable gate array (FPGA) is a general-purpose integrated circuit that is programmed by the designer rather than the device manufacturer. Unlike an application-specific integrated circuit (ASIC), which can perform a similar function in an electronic system, an FPGA can be reprogrammed by downloading a configuration program called a *bitstream*, even after it has been deployed into a system. Much like the object code for a microprocessor, a bitstream is the product of compilation tools that translate the high level abstractions produced by a designer into something equivalent but low level and executable. Over the last three decades, FPGA's have grown from simple logic components, through moderate prototyping platforms and more recently, as complete System on a chip (SoC) components. One of the greatest advantages with FPGA's is that they can be used as custom hardware avoiding the initial costs, fabrication costs and fabrication time.

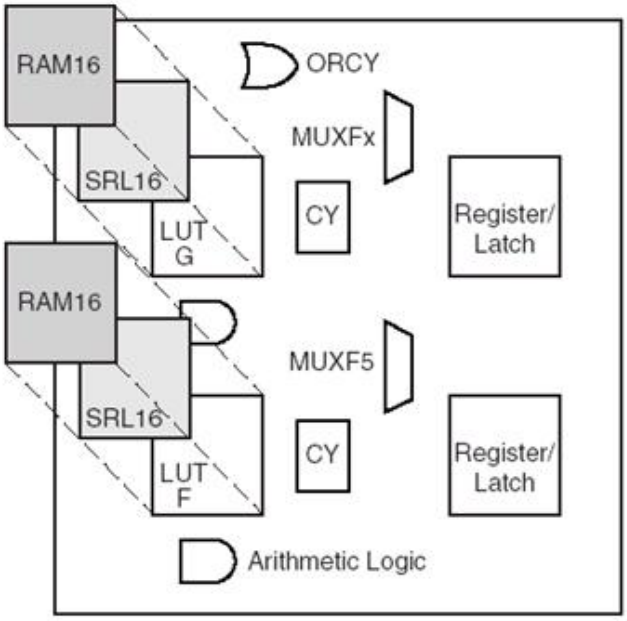


Figure 2.2. Virtex Family FPGA Logic slice

A simple FPGA fabric consists of an array of configurable logic blocks (CLBs) attached by a programmable interconnect. Digital circuits are mapped to the CLBs which consist of logic slices which consists of look-up tables (LUTs) and flip-flops (FFs). Each logic slice as shown in Figure 2.2 contains two 4-input lookup tables (LUTs), two configurable D-flip flops, multiplexers, dedicated carry logic, and gates used for creating slice based multipliers. Each LUT can implement an arbitrary 4-input Boolean function. Four inputs is a good size for a look-up table as suggested by various studies, trading utility (complexity of a block) against utilization (what fraction ends up in use) [2, 3] [15]. Coupled with dedicated logic for implementing fast carry circuits, the LUTs can also be used to build fast adder/subtracters and multipliers of essentially any word size. In addition to implementing Boolean functions, each LUT can also be configured as a 16x1 bit RAM or as a shift register.

In addition to logic slices, current generation FPGAs include additional diffused

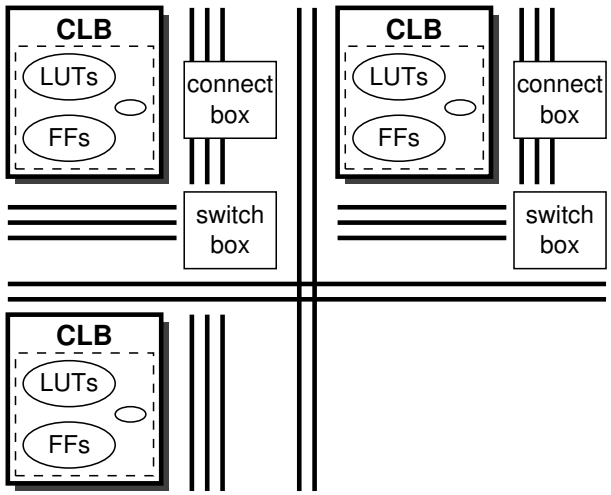


Figure 2.3. An abstract view of a Field Programmable Gate Array (FPGA)

hardware resources beneficial for embedded systems. For example the Xilinx XC4FX140 which is a product of the latest 90 nm CMOS technology features various dedicated digital signal processing 18-bits multipliers and accumulators which are called as DSP slices, dual port BLOCK RAM's which can be used for storing few kilobytes of data, Digital Clock Managers, 2 Power-PC RISC Processors, 10/100/1000 Ethernet MAC Blocks, and Rocket IO Serial Transceivers which can be used to provide high-speed connections for communication between FPGA's and inter-module communications. Moreover with the advance of Moore's Law, FPGA's are also increasing in total capacity and speed which gives the users more number of computational units and is shown in Figure 2.4 [4].

2.2.2 Programming an FPGA

In current practice, hardware descriptive languages (HDL) and schematics are widely used to implement applications on the FPGAs. Figure 2.5 is a pictorial representation of the design flow that usually occurs with FPGA's.

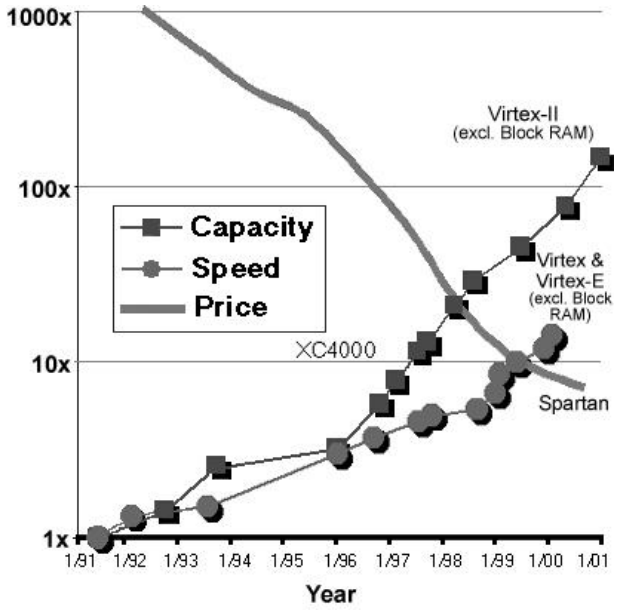


Figure 2.4. FPGA's due to Moore's Law

Several HDL languages like VHDL (Very High Speed Integrated Circuit Hardware Description Language), Verilog, JHDL, SystemC, Streams - C, HandelC etc exist where in the application can be specified and this stage is usually called the Design Entry stage. After this stage, the design is verified for it's functionality through a Simulation process. After the Simulation process the design is converted to a form of representation called the netlist which is the complete representation of the logic in terms of basic gates (AND,OR,XOR,NOT). After this process the design is mapped which is mapping the above obtained netlist to the actual Configurable Logic Blocks (CLB) and Input/Output Blocks (IOB) available in the device that has been targeted. After the design has been mapped the next stage in the process is called Place and Route where in the design that has been mapped is physically mapped to the device's logic cells based on the timing and layout requirements. After these steps, a timing simulation is performed and the design is modified so that the best possible timing is obtained. After the re-design, the

design is again sent through the process of converting the design into a netlist, MAP and then Place and Route. After the final Place and Route the design is converted to a configuration file called a BIT file which defines the behavior of the FPGA that has been targeted. The BIT file obtained can be downloaded into the FPGA and verified for functionality.

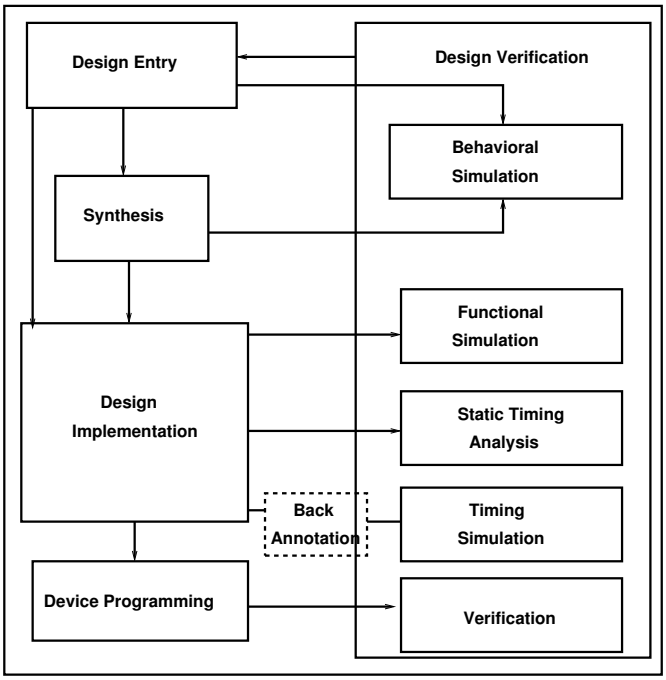


Figure 2.5. FPGA Design Flow

2.2.3 Hybrid CPU/FPGA Architecture's

Hybrid CPU/FPGA architecture's are the first of its kind from Xilinx,Inc which are also called as Platform FPGA's which are the latest FPGA's with processors embedded (Hard Cores) in the FPGA fabric apart from the vast number of freely available logic gates. The processors inside the Platform FPGA's are IBM PowerPC 405's which implement the standard RISC style architecture and are based on the Core-Connect Ar-

chitecture [27] from IBM and are implemented as Hard Cores inside the FPGA. This level of integration allows various Intellectual Property (IP) cores to be attached to the processor and the cores are also easily accessible through the Core Connect Architecture that is provided as a Intellectual property core (Soft Core). The Core Connect provides three bus standards as a means of communication between the PowerPC and other cores. The three bus standards are Processor Local Bus (PLB) , On-Chip Peripheral Bus (OPB) and the Device Control Register (DCR) bus. The processor local bus (PLB) is used to connect processor cores to the system main memory and other high-speed devices. The OPB bus is dedicated for connecting slower on-chip peripheral devices indirectly to the CPU. The OPB bus supports variable size data transfers and as well as flexible arbitration protocols. Both the PLB and OPB buses have their own bus arbiters, and the two buses are interconnected by at least one bridge (PLB2OPB Bridge or OPB2PLB Bridge). Various intellectual Property (IP) Cores (Soft Cores) are also available in order to interact with various standard peripherals in the FPGA such as the DDR SDRAM (Double Data Rate - Synchronous Dynamic Random Access Memory) , EEPROM (Electrically Erasable Programmable Read-Only Memory), PCI (Peripheral Component Interconnect), RS232 UART (Universal Asynchronous Receiver/Transmitter). In addition to the peripheral and utility Intellectual Property cores, an interface called the Intellectual Property Interface (IPIF) is available in the form of a soft core which allows any Intellectual Property (IP) Core to connect to either of the buses. The IPIF is decomposed into two layers to allow easy migration of peripherals or IP cores to each of the different system buses in the Core Connect Architecture . The first layer provides an interface facility to be used between the IP core and the IPIF. The second layer is a bus specific portion, and interfaces the IPIF to one of the buses. These interface modules allow to greatly accelerate the process of connecting

pre-existent IP, or creating a new IP in a system. The IPIF provides two different types of attachment to an IP core: a slave and a master attachment. With the master attachment, user cores have the ability to initiate bus transactions. Moreover, bus arbitration logic is also included within the master attachment. However it is the user core's responsibility to re-arbitrate or abort the bus and switch the data bus between the slave and master modes.

2.2.4 Reconfigurable Computing

Reconfigurable Computing (RC) [13] [12] started of during the late 1960's but was still a research field until the late 1980's because of lack of availability of suitable hardware. But with the advent of Field Programmable Gate Array Technology (FPGA), the field of reconfigurable computing got a boost since FPGA's provided a reconfigurable platform and gave a broader meaning to the field. The main feature of Reconfigurable Computing is the ability of the hardware to reconfigure based on various functions. Although FPGA's provided a full reconfiguration of the chip since its ingression until recently, due to increase in technology various FPGA's now even support partial reconfiguration which means that a portion of the device can be altered even though when the FPGA is actually running.

When Reconfigurable Computing was in its initial development stages, the cost of FPGA hardware and Reconfigurable cards were very costly, but as years passed by and with the advancement of Moore's Law which gave more transistors per die, FPGA's and Reconfigurable Computing boards have become a lot cheaper. Moreover the introduction of FPGA's with processors embedded in it became a stepping stone to the field of Reconfigurable Computing. For example, today a Reconfigurable Computing Mother board with a Xilinx Virtex II Pro FPGA which houses around 100,000 free logic gates,

two PowerPC processors and the ability to house a DDR SDRAM, a Compact Flash Card and various other peripherals costs around $250 as compared to $6000 in the year 1998 which housed a Xilinx XC4085 FPGA with only 10,000 logic gates and with minimal peripheral support.

2.3 Related Work

Since BLAST is an open source tool and many scientists and biologists favor it for various research purposes, various groups in industry and academia tried to port BLAST to different types of architectures in various ways.

Among the various implementations present, WU-BLAST [22] is a copy-righted implementation based on the BLAST algorithm developed by researchers at Washington University, St. Louis which provides more sensitive, selective and rapid similarity searches of protein and nucleotide sequence databases by using extra command line options than the original BLAST and yields different results when compared to the original BLAST from NCBI. Moreover, few scripts are provided in-order to automate the process of converting people using BLAST from NCBI to use WU-BLAST.

Among other implementations from the academia world include BLAST++ [46], an implementation from the researchers at Nanyang Technology University where the implementation has a capability of processing multiple queries with the same database. In this implementation several individual queries are transformed as a single large virtual query and a look-up table is built in order to compare the virtual query to the database. The researchers also have developed a FPGA hardware implementation of Smith - Waterman Algorithm [39] [34] where in the dynamic programming part in the algorithm has been mapped to Processing Elements (PE's) which are implemented as logic resources inside an FPGA.

On the industry side, initial attempts were made by Silicon Graphics,Inc (SGI) to implement BLAST on SGI machines during the late 90's and was called as High Throughput BLAST (HT-BLAST) [9]. In the implementation of High-Throughput BLAST the parallelism inherent in the BLAST source using pthreads was replaced by their own parallel constructs in-order to make it run faster on SGI machines. Later on, efforts were made by TimeLogic which has two versions of BLAST namely GeneBLAST [44] and Decypher BLAST [43] which are extensions to the original BLAST Algorithm which provide more statistical information than the original BLAST. However these versions of BLAST are not compatible to run on all platforms and have been optimized to run on a single platform called Decypher which have FPGA's and are available from TimeLogic [42].

Few other efforts include an implementation called TurboBLAST [8] which is a version of BLAST from TurboGenomics,Inc, a parallel implementation to run on networked clusters of heterogeneous PC's, workstations, and Macintosh computers. The implementation co-ordinates multiple versions of unmodified BLAST running on various machines by using Turbohub which is a execution engine for Parallel and Distributed Java applications.

Apart from various copy righted implementations, research efforts were also made in the Open Source community to implement BLAST on various different architectures. An open source implementation of BLAST that is available is mpiBLAST [16] which has been developed by researchers at the Lawrence Livermore National Labs (LLNL) is based on Message Passing Interface (MPI) [18, 20] [24] and was designed to target cluster machines which have large number of high end processors. mpiBLAST partitions the database among all the cluster nodes in-order to speed up the process of execution by reducing the amount of disk I/O bandwidth. However the disadvantage

it has is even though it has many processors running, it has no ability to partition the query.

Apart from development in the research and industry community, efforts were also made to come up with text books such as *BLAST* [48] and *Introduction to Bioinformatics* [29] from Oreilly Publishers. The textbook *BLAST* explains how exactly the BLAST algorithm works and it also clearly explains how to get more results by defining the significance of various command line parameters that are available. The book also specifies the significance of the statistics obtained by the BLAST algorithm and various ways to interpret them. The textbook *Introduction to Bioinformatics* gives a brief description of the BLAST Algorithm and the way it works.

In this chapter, the related background information for the thesis statement made in the chapter 1 have been specified and also the work done by various other academic and commercial institutions have been specified.

Chapter 3

Design and Implementation

To establish the fact whether a scalable and cost-effective reconfigurable implementation of BLAST is feasible or not, a prototype was designed, implemented and analyzed. To figure out the computational intensive part in the BLAST software tool, experiments were setup by enabling one of the standard Unix profiling options. After determining the most computationally intensive sub-routine in the BLAST software tool, the software was finely profiled in order to determine the lines of code that were most computationally intensive inside the sub-routine. Moreover to prove that the above determined lines of code were the most time consuming segments in the software, the same test was repeated on various machines with different architectures, memory and network interfaces. After performing the initial set of profiling experiments, BLAST was cross compiled for the appropriate PowerPC architecture in the target hardware using a cross compiler developed. The same profiling tests were also conducted on the target hardware after developing a base system. After porting BLAST to the PowerPC processor, hardware was designed as an Intellectual Property (IP) core to replace the most computational intensive part of the BLAST software as a Master - Slave peripheral which could be attached to the On - Chip Peripheral (OPB) Bus. The

hardware developed was tested for functionality using a standalone system. After the initial test with a standalone system, a Linux character device driver was developed in order to test the hardware design developed in a complete Linux based system environment. After testing the hardware design, appropriate changes were made to the BLAST source code to invoke the new hardware designed. After the initial round of tests, in order to take advantage of the on chip memory available in the FPGA, the hardware design was changed to obtain speed up by saving clock cycles by having a lookup to a BRAM instead of external memory. After the various tests on the development board for both the hardware designs developed, the scalability issue in relation to portability and cost effectiveness was addressed. Since the FPGA was abundant in resources and only 50% of the FPGA was occupied for the initial systems developed, the cores developed were replicated and subsequent changes were made to the hardware and software to take advantage of the numerous cores present in the system.

3.1 Partitioning

3.1.1 The Target : Xilinx ML - 310

The target hardware selected for experiments is a development board from Xilinx which hosts a Virtex II - Pro XC2VP30 FPGA which is an embedded platform for accelerated system development. In addition to more than 30,000 logic cells, over 2,400 Kb of BRAM, and dual PPC405 processors available in the FPGA, the ML310 provides on-board Ethernet MAC/PHY, DDR memory, multiple PCI slots, and standard PC I/O ports within an ATX form factor board. An integrated System ACE CF controller is deployed to perform board bring-up and to load applications from the included Compact-Flash card and the complete system is shown in Figure 3.1.

The Virtex II - Pro FPGA from Xilinx,Inc is a platform FPGA and has two embed-

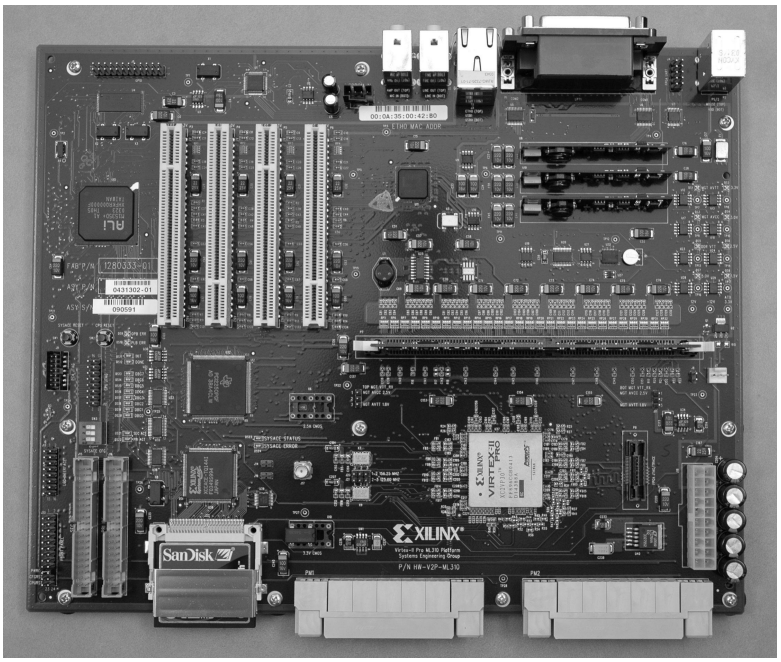


Figure 3.1. Xilinx ML - 310 Board Diagram
Source : http://www.xilinx.com/products/boards/ml310/current/index.html

ded PowerPC 405 processors in it. Apart from the PowerPC processors it also has full duplex serial transceivers which can provide speeds ranging from 622 Mbps to 3.125 Gbps. Virtex II - Pro FPGA's are built on a 130nm, using a 9 layer copper process technology and it has about 30000 logic cells available which can be programmed to implement any logic of the user's choice. They are indeed good supplements for various existing embedded systems because of their higher performance and lower power consumption. Figure 3.2 is an internal block diagram of the Virtex II - Pro FPGA. The PowerPC processors are based on the Harvard Architecture and support the Core-Connect Bus Architecture from IBM [27] and also are capable of running the Linux operating system and thereby a Linux kernel 2.4.26 from Montavista [32] with PCI support was ported on to the PowerPC processor inside the FPGA. The Virtex - II Pro FPGA also has support for various commonly used devices like the IDE and USB ports

through the PCI Bus and ALi South Bridge. Apart from running the standard peripherals, user-defined cores can also be run on the Xilinx Virtex - II Pro FPGA and be accessed from the PowerPC processor. The user-defined core to be accessed by the PowerPC processor in the FPGA should be attached to either the Processor Local Bus (PLB) or the On-Chip Peripheral Bus (OPB) buses available in the FPGA. In order to generate such a system, a software called Embedded Development Kit (EDK) or Xilinx Platform Studio (XPS) [40] from Xilinx is usually used.

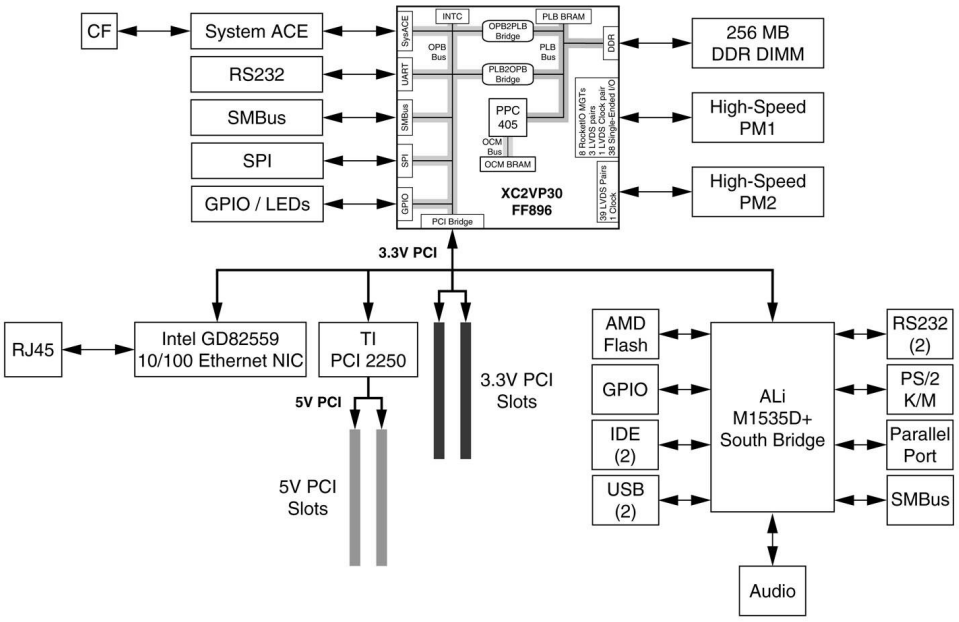


Figure 3.2. Xilinx ML - 310 Block Internal Block Diagram
Source : http://www.xilinx.com/products/boards/ml310/current/index.html

3.1.2 Profiling BLAST

The first step in order to answer the central question of the work is to find the most computationally intensive segment in BLAST. For this purpose, BLAST software was profiled and indeed finely profiled in order to determine the most computationally in-

tensive part in the software. In order to do this, BLAST source code was downloaded from the NCBI ftp site **ftp.ncbi.nlm.nih.gov**, both the source code and the BLAST executables are available for download from this web site. The various flavors of BLAST available which can be catered according to user requests are :-

- blastn - Compares nucleotide queries against nucleotide sequences
- blastp - Compares protein queries against protein sequences
- blastx - Compares translated queries against protein databases
- tblastn - Compares protein queries against translated databases
- tblastx - Compares translated queries against translated databases

The NCBI 2.2.6 version which is the latest version of BLAST source code consists of about 1500 source files spread out in about 15 directories approximately. After unzipping the source obtained, all the profiling tests were run on an Intel Based Machine in spite the final target being a PowerPC Machine. The Intel machine used for initial profiling was running RedHat Enterprise Linux version 3.0. The Unix profiling tool *gprof* was used in order to determine the profiling information.

To run the BLAST program, the software needs three arguments to be given :-

1. The flavor of BLAST program to execute (e.g : blastp, blastn)
2. A database which is a large collection of known genetic sequences
3. A query which is a generally a short sequence of which is unknown

As explained above, BLAST code base supports various flavors or various programs. Among the various programs available from the BLAST source code, **blastn** which compares nucleotide queries against nucleotide database of sequences was selected to

do the initial profiling. A sample query sequence of size 560Kb was selected to perform the initial profiling tests. While BLAST has the ability to create local BLAST databases from any FASTA formatted protein or nucleotide sequences, one of the NCBI databases was downloaded for profiling. Several database files can be downloaded from the BLAST database FTP directory available at **ftp://ftp.ncbi.nih.gov/blast/db/**. A relatively small database file called ecoli.nt.Z which represents the bacterium Escherichia coli and around 1.3 MB in size was downloaded for initial tests from the above mentioned website. This is a FASTA formatted file of nucleotide sequences which is also compressed. Once uncompressed the database was formatted using the **formatdb** program which comes with the BLAST software on download using the following the steps :-

- formatdb -i ecoli.nt -p F -o T

After formatting the database, the following steps were followed in order to enable profiling :-

- The compiling option **'-pg'** was added as one of the options to **'gcc'** in the top level Makefile
- After enabling the option specified above, the BLAST software is compiled to obtain the binaries.
- The binary **blastall** was run with the following options
  blastall -p blastn -i my_query -d ecoli.nt -o results
- Running the BLAST software produces a file called **gmon.out**
- The profiling tool **gprof** is now run on the binary **blastall** in order to obtain the profiling results.

As the source code is not modified and since the total computation is run in software or on the general purpose processor available, hence this run is aptly named as Software-BLAST or SW-BLAST.

Table 3.1 represents the initial profiling results obtained for SW-BLAST. Based on the statistics obtained from the initial profiling as shown in Table 3.1, the function called **BlastNtWordFinder** was determined to be the most computationally intensive segment in the BLAST software.

Table 3.1. Profiling BLAST

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
85.71	0.42	0.42	1170	0.36	0.36	BlastNtWordFinder
4.08	0.44	0.44	16	1.25	1.57	readdb_get_sequence
2.04	0.45	0.01	41795	0.00	0.00	BlastNtWordExtend
2.04	0.46	0.01	1202	0.01	0.01	readdb_get_sequence_ex
2.04	0.47	0.01	34	0.29	0.29	ALIGN_EX
2.04	0.48	0.01	24	0.42	0.42	RebuildDNA_4na
2.04	0.49	0.01	1	10.00	10.00	mb_make_mod_lt
0.00	0.49	0.00	7610	0.00	0.00	Nlm_SwitchUint4
0.00	0.49	0.00	7425	0.00	0.00	Nlm_StringCmp
0.00	0.49	0.00	6766	0.00	0.00	AsnFindBaseType
0.00	0.49	0.00	6230	0.00	0.00	AsnDeBinDecr
0.00	0.49	0.00	5826	0.00	0.00	s_MemAllocator
0.00	0.49	0.00	5196	0.00	0.00	Nlm_MemFree
0.00	0.49	0.00	5163	0.00	0.00	AsnDeBinScanTag
0.00	0.49	0.00	4679	0.00	0.00	Nlm_MemGet
0.00	0.49	0.00	4085	0.00	0.00	SeqPortGetResidue
0.00	0.49	0.00	3802	0.00	0.00	AsnFindBaseIsa
0.00	0.49	0.00	3733	0.00	0.00	ObjMgrGet
0.00	0.49	0.00	3493	0.00	0.00	BlastHitListPurge
0.00	0.49	0.00	2918	0.00	0.00	NlmRWunlockEx
0.00	0.49	0.00	2869	0.00	0.00	Nlm_StringMove
0.00	0.49	0.00	2506	0.00	0.00	NlmTlsGetValue
0.00	0.49	0.00	2426	0.00	0.00	AddXMLname
0.00	0.49	0.00	2204	0.00	0.00	AsnBinReadId
0.00	0.49	0.00	2204	0.00	0.00	AsnBinReadVal
0.00	0.49	0.00	2204	0.00	0.00	AsnReadId

These profiling tests were also run on various other databases and sequences to establish the fact that this behavior found is universal for all the queries and databases and not just the sequence and database taken into consideration for initial testing. After obtaining the profiling results, the results were also compared to the work done in [33] in order to check for consistency of the results obtained. The **BlastNtWordFinder** was further analyzed and about 30 lines of code were separated which were found to be the most critical part inside the function **BlastNtWordFinder** and were named as **critical_code** and further profiling established the fact these 30 lines of code were the most computationally intensive segment. Table 3.2 gives the profile information after changes to the BLAST source code.

Apart from running the test on just the test machine specified above the version of SW-BLAST software was indeed profiled on various high end machines and low end machines from different vendors and board level architectures. Sample configuration of the machines chosen to use were :-

- Dual Intel Xeon 32 bit Processors running at 2.8Ghz with 2GB of RAM
- Quad Intel Xeon 64 bit Processors running at 3.2GHz with 4GB of RAM
- Dual Intel Pentium III 32-bit Processors running at 1.2GHz with 1GB of RAM
- Dual Intel Pentium III 32-bit Processors running at 550MHz with 512MB of RAM
- AMD Athlon 32-bit Processor running at 1000MHz with 1GB of RAM

After running the same test on all the machines specified above **critical_code** was determined to be the most computationally intensive segment in the BLAST software and roughly amounted to about 80% of the computation time. Further more analysis was

Table 3.2. Profiling BLAST with critical code

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
82.94	0.42	0.42	1170	0.36	0.36	critical_code
4.08	0.44	0.44	16	1.25	1.57	readdb_get_sequence
3.28	0.42	0.42	1170	0.36	0.36	BlastNtWordFinder
2.04	0.45	0.01	41795	0.00	0.00	BlastNtWordExtend
2.04	0.46	0.01	1202	0.01	0.01	readdb_get_sequence_ex
2.04	0.47	0.01	34	0.29	0.29	ALIGN_EX
2.04	0.48	0.01	24	0.42	0.42	RebuildDNA_4na
2.04	0.49	0.01	1	10.00	10.00	mb_make_mod_lt
0.00	0.49	0.00	7610	0.00	0.00	Nlm_SwitchUint4
0.00	0.49	0.00	7425	0.00	0.00	Nlm_StringCmp
0.00	0.49	0.00	6766	0.00	0.00	AsnFindBaseType
0.00	0.49	0.00	6230	0.00	0.00	AsnDeBinDecr
0.00	0.49	0.00	5826	0.00	0.00	s_MemAllocator
0.00	0.49	0.00	5196	0.00	0.00	Nlm_MemFree
0.00	0.49	0.00	5163	0.00	0.00	AsnDeBinScanTag
0.00	0.49	0.00	4679	0.00	0.00	Nlm_MemGet
0.00	0.49	0.00	4085	0.00	0.00	SeqPortGetResidue
0.00	0.49	0.00	3802	0.00	0.00	AsnFindBaseIsa
0.00	0.49	0.00	3733	0.00	0.00	ObjMgrGet
0.00	0.49	0.00	3493	0.00	0.00	BlastHitListPurge
0.00	0.49	0.00	2918	0.00	0.00	NlmRWunlockEx
0.00	0.49	0.00	2869	0.00	0.00	Nlm_StringMove
0.00	0.49	0.00	2506	0.00	0.00	NlmTlsGetValue
0.00	0.49	0.00	2426	0.00	0.00	AddXMLname
0.00	0.49	0.00	2204	0.00	0.00	AsnBinReadId
0.00	0.49	0.00	2204	0.00	0.00	AsnBinReadVal
0.00	0.49	0.00	2204	0.00	0.00	AsnReadId

performed by obtaining databases and queries from Future Systems Group which is a group that specializes in bioinformatics at Oak Ridge National Labs (ORNL), Tennessee, Knoxville. The databases obtained were being used as a set of benchmarks by the group at ORNL in order to determine few characteristics by the biologists about the BLAST software. The analysis was done in order to determine whether BLAST was compute bound or I/O bound. A set of 20 queries and two databases were compared against each other to analyze the results using a Perl script and Table 3.3 shows the results obtained. As shown in Table 3.3 even though Moore's Law gives double the number of transistors every 18 months, the amount of time taken to run the BLAST software doesn't increase linearly either with the database residing on the local drives or mounted through Network File System (NFS).

Table 3.3. Profiling BLAST on various High-end and Low-end machines

Mount type	500MHz	1000MHz	2400MHz	3200MHz
NFS	152m	48m	77m	17m
local drive	125m	39m	27m	13m

(N/A in the above table indicates that the tests described could not be run on the particular machine because of the intensity of the databases and low capability of the machine)

This tests were a strong motivation that substantial speedups can be obtained by accelerating the **critical_code** section inside the **BlastNtWordFinder** function in the BLAST software using Reconfigurable Computing.

3.1.3 Base System Platform

One of the strong reasons that the ML-310 development board from Xilinx was chosen as the target hardware is because it had PowerPC processors which could run the ordinary BLAST software and the most computationally intensive part in the BLAST

software can be transformed as a user core and could be run on the FPGA slices. In order to run any experiments on the target hardware specified in subsection 3.1.1, a Base System Platform as shown in Figure 3.3 has to be built and Xilinx Embedded Development Kit software was used to generate a Base System Platform which (BSP) consisted of

- PowerPC Processor
- Block-RAM to store the instructions and data for the processor
- DDR-RAM core on the PLB bus
- PCI Core on the OPB Bus
- PCI Bridge Core on the OPB Bus
- Serial Port Core on the OPB Bus
- Compact Flash core on the OPB Bus

Apart from the standard peripheral cores, even user-cores can be added to the Base System Platform. Few other essential elements were also added to system which include :-

- Linux kernel (2.4.26) to run on the PowerPC processor
- Linux Device Drivers for the IP cores
- An IDE Hard drive connected to the PCI bus through Ali South Bridge

The following procedure is used in order to generate a Base System Platform :-

- Initially the **Base System Wizard** in the Xilinx Embedded Development Kit software is invoked in order to create a base system with standard peripheral components.

- After selecting all the standard peripherals mentioned above, an option by name **Create / Import Peripheral Wizard** in Xilinx Embedded Development Kit can be used to generate a user-core as a Master - Slave interface with an Interconnect Peripheral Interface (IPIF). The Interconnect Peripheral Interface bridges the user-core with one of the buses that the core is attached to.

- The files obtained from the above step are modified according to the user requirements.

- After modifications, the user-core is stimulated using Mentor Graphics [23] Modelsim Tool [31] along with Xilinx's Bus Functional Model in order to establish the correct functionality of the core.

- The user core is then added to one of the Buses available in the FPGA and synthesized to a bit-stream along with the Base system generated using the Xilinx ISE Design Tool Suite and Embedded Development Kit.

- The base system used can be either a standalone system or else a system which has the ability to run the Linux Kernel by changing an option in **Software Platform Settings**

  – In case the system built is a standalone system, then the bit-stream generated is combined with an elf file which is a binary for a C program that runs on the PowerPC processor in order to test the user-core to generate a new bit-stream which can be downloaded using the JTAG interface using a Parallel

Cable.

– In case the system built is a Linux based system, the bit-stream generated is combined with the Linux kernel to create an AceFile which can be loaded onto one of the partitions in the Compact Flash.

– After booting up in Linux, then a Linux Device driver is used as a loadable module in order to test the user-core.

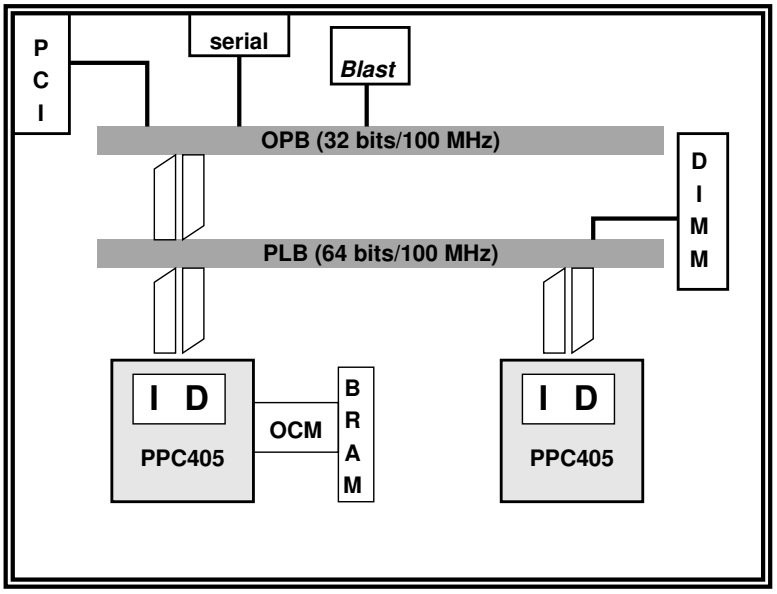


Figure 3.3. ML-310 Base System Platform

3.2 BLAST Intellectual Property (IP) Core

3.2.1 Implementation Overview

Initially after building the base system, a bit file was generated by the Xilinx Tool set and by combining the bit-file with a Linux Kernel cross compiled for PowerPC,

an ace-file was generated. The ace-file was loaded onto the Compact Flash card in the ML - 310 Development board and the BLAST source was cross compiled to the PowerPC Architecture and was run on the ML-310 development board to check for compatibility of the BLAST software. After an initial run, the profiling was enabled for the cross compiled version of BLAST and profiling results obtained were checked for consistency. After these phase of tests, it was determined to build the hardware for the most computationally intensive part.

In order to implement the BLAST core in the FPGA, the **critical_code** function which was determined as the most computationally intensive segment in the BLAST software from the profiling results in subsection 3.1.2 was analyzed and its functionality was determined. The detailed functionality of the function is as follows: All the databases available consist of sequences and sequences are further sub-divided into sub-sequences and all the sub-sequences use the **BlastNtWordFinder** to determine the hit information. Inside the **BlastNtWordFinder** function, the hit information for all the sub-sequences is determined using a lookup linked list which is built using the query sequence when the BLAST software is invoked using the query. So, words of 16 bits or 8 characters are formed from the subject sequence by traversing it in four letter hops. If the count of occurrences of the word in the query sequence is zero, the current word does not occur in the query and is discarded. If the count is non zero then each offset is retrieved from the linked list. For each offset in the query sequence the subject and query words are extended to the left and right. If the comparison routine generates a high score, then the subject and the query words, as well as their offsets, are passed to another routine called the **BlastNtWordExtend** where further search is performed. The task of indexing the newly created words from the subject sequence into the lookup linked list and retrieving the query offsets of these words is implemented in

the new **critical_code** function.

Since the functionality of **critical_code** function was determined, an equivalent digital design was made in order to replace the **critical_code** function. The digital circuit design was implemented using VHDL (Very High Speed Integrated Circuits Hardware Descriptive Language) as an Intellectual Property (IP) core and is shown in Figure 3.4 in a top level view and is explained in detail in subsection 3.2.2. The circuit design was made in order to be portable and scalable to various different architectures, so that this particular implementation could be ported to many architectures like the Cray XD -1 [14] [1], SRC - 7 [35] or development boards with Xilinx Virtex 4 Multi-Platform FPGA's [19] without much redesign. Apart from replacing the **critical_code** function, the lookup linked list which the function **critical_code** uses to determine the lookup's as described above was converted to a lookup table which would ease the hardware design in order to do the lookup's and the structure of the table is explained in detail in subsection 3.2.3. In the present hardware design implemented, the BLAST application is first started by invoking the **blastall** command on the PowerPC processor in the ML-310 Development board. As the execution reaches the **critical_code** function, a Linux character device driver initializes the BLAST user peripheral core in the hardware using the *open* system call. After performing the initializations, the virtual address of the BLAST hardware core is determined by using *io-remap* function call by giving the physical address of the hardware core. By using the patch for the BLAST source code, the lookup table is built using the query sequence and the lookup table is loaded onto to the DDR-RAM which is attached to the Peripheral Local Bus through the *write* system call. After loading the lookup-table into the DDR-RAM, each subsection of the subject sequence is transferred to the BLAST hardware core on the FPGA by using the *write* call where in the core identifies all the basic eight letter or 16 bit hits between this

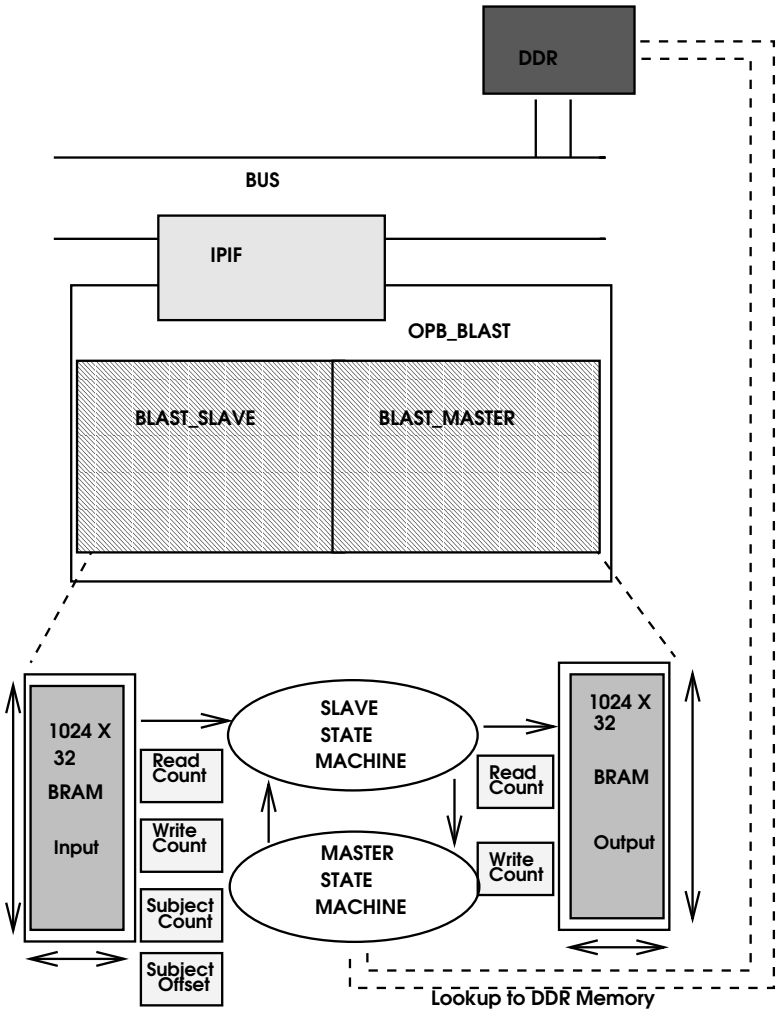


Figure 3.4. Hardware BLAST on OPB bus

subsection and the query sequence. After the transfer the character device driver on the other hand waits for hit information to be obtained back from the FPGA and uses the *read* system call to perform the operation. And this operation is continued until all the sequences in the sequence database have been processed. After receiving the control back from the FPGA, the remaining part of software execution is again resumed on the PowerPC processor. The character device driver was carefully designed in such a way that the hardware core never stalls waiting for data from the processor and also an efficient polling scheme was chosen to implement the character device driver. Since this run has both hardware and software involved, it is called as Hardware-BLAST or RC-BLAST which includes the modified BLAST source code as an Open Source patch where the computationally intensive segment of the sequence matching code is moved onto the FPGA hardware, the hardware user core and the Linux character device driver.

3.2.2 Hardware Implementation

The BLAST hardware core design as shown in Figure 3.4 is a master-slave peripheral connected through an Intellectual Property Interface (IPIF) to the On-Chip peripheral Bus (OPB) . The PowerPC processor and the core communicate with an OPB2PLB Bridge and a PLB2OPB Bridge. The master-slave design consists of two BLOCK RAM's of size 1024 X 32 bits which act as Input and Output FIFO's in the slave design. It also has registers to monitor the read and write counts for the two FIFO's by the character device driver. The BLOCK RAM's were chosen to act as FIFO's because they are built in to the FPGA and provide faster access to the data and do not use any of the Configurable Logic Blocks (CLB's) available in the FPGA. And also the concept of FIFO can be translated to any other architecture with very minimal design modifications. Initially, when the point of execution reaches the **critical_code**

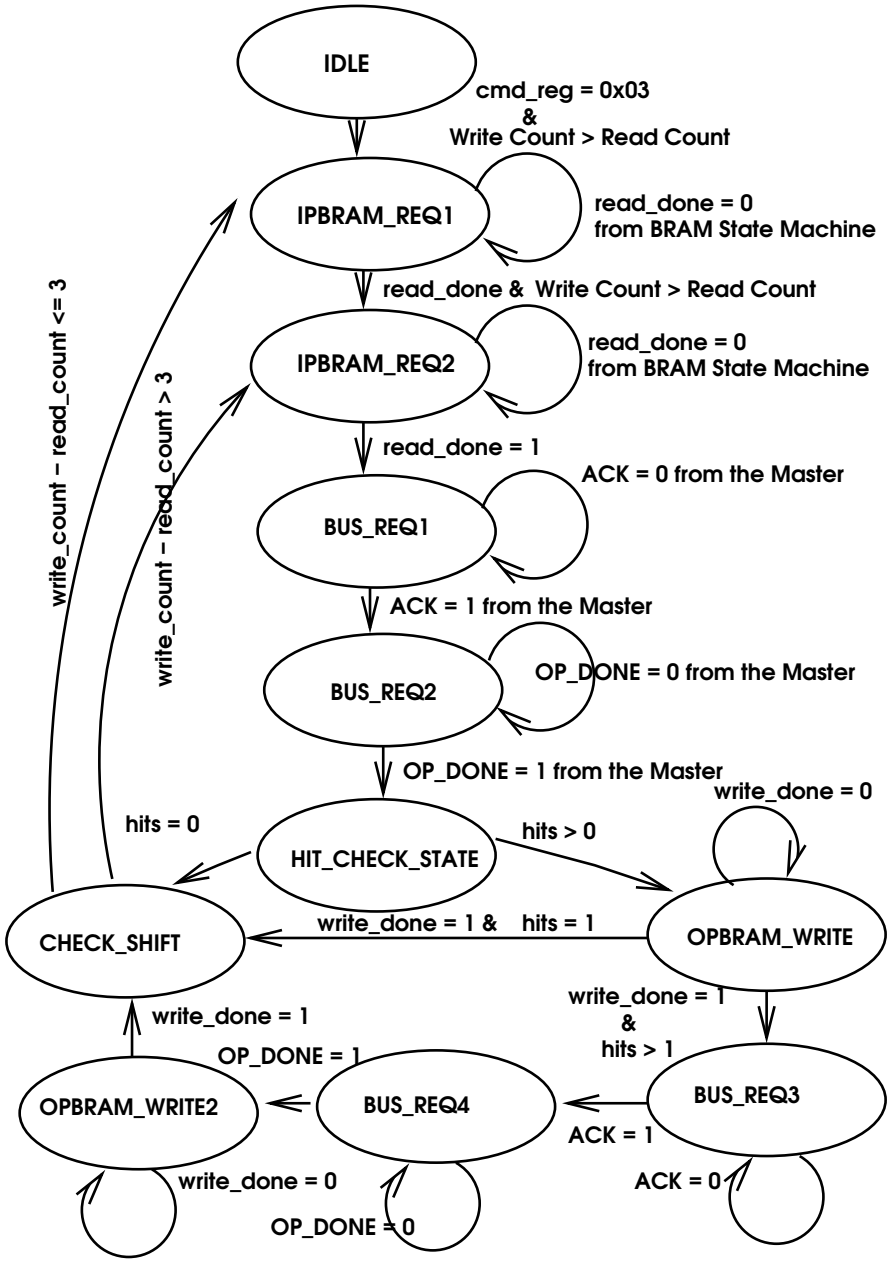


Figure 3.5. Hardware BLAST Slave State Machine

function, the device driver fills in the sub-sequences into the Input FIFO and writes the write count to the Input FIFO read count register. Upon seeing a value greater than the Input FIFO's write count which is initially zero the Slave start Machine starts execution. The slave state machine reads in data from the Input FIFO by invoking the BRAM state machine which is shown in Figure 3.7 and then passes it to the master state machine. The job of the master state machine is to do a lookup to the lookup-table in the DDR-RAM and then write back the data obtained from the lookup back to the slave and is shown in Figure 3.6. The slave state machine reads back the hit information and writes back the appropriate hit information and offset information to the Output FIFO using the BRAM State Machine. The character device driver reads in the values from the Output FIFO and the above mentioned process continues until all the subject sequences in the sequence database are processed. The complete flow of the slave state machine is shown in Figure 3.5.

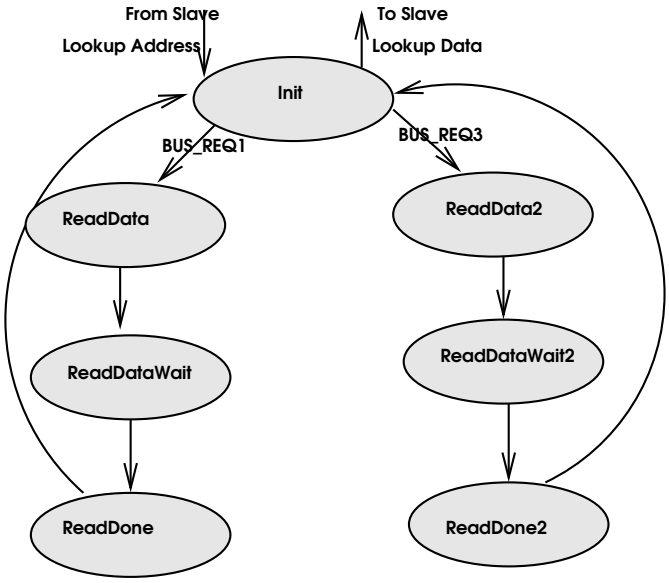


Figure 3.6. Hardware BLAST Master state machine

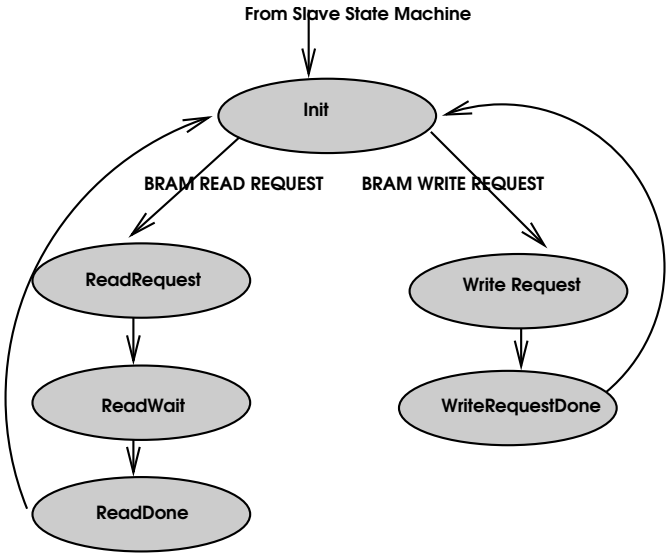


Figure 3.7. Xilinx Block RAM State Machine

3.2.3 Lookup Table

The lookup table is one of the most critical elements in the hardware design of the BLAST Algorithm and is translated from a linked list to a table. The lookup table is generated by the software and is loaded onto the DDR-RAM. It is based on the lookup table that the number of hits is determined and moreover it is also very easy to port the lookup table to any FPGA development board or a FPGA based system since it just takes 512KB of memory. In the format of the lookup table, the first column of each row in the lookup table is a word, which determines if the hits are greater than 3 or not and is constructed from consecutive 8 characters in the query sequence. The second column for each row has a 2 bit count which indicates the number of hits that particular word has and the rest of the columns have the offsets of each appearance of the word in the query sequence. Since there are 8 characters or 16 bits in each row, by applying the maximum number of combinations, the lookup table is 2 power 16 rows in length. The lookup table configuration is shown in Figure 3.8. However in the present version

of the hardware design made, the hardware cannot handle the situation if the hits are greater than 3.

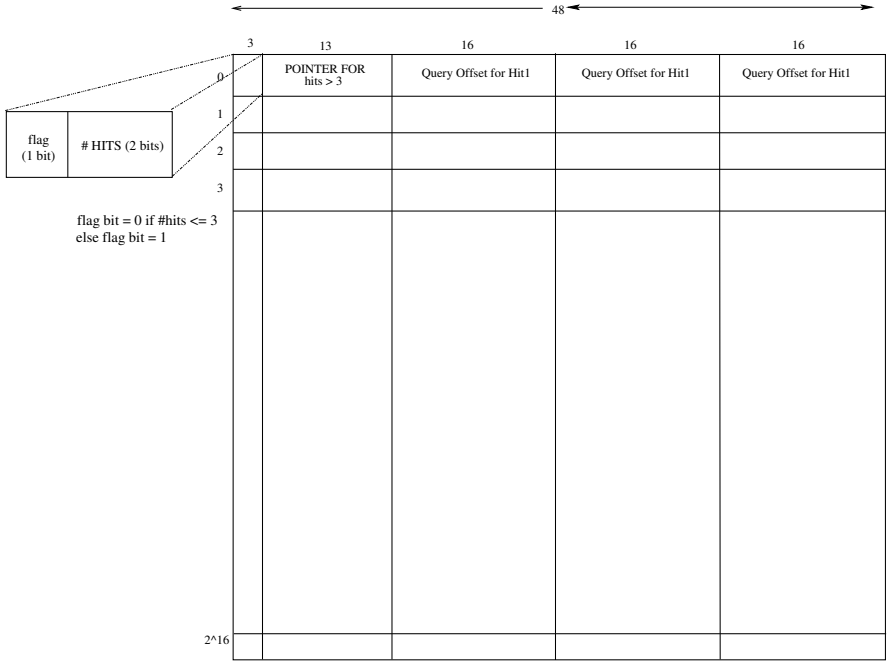


Figure 3.8. Hardware BLAST Lookup table

3.3 Enhancements to BLAST Hardware

3.3.1 On-Chip Caching of Partial Look-up

Taking advantage of On-Chip memory instead of using external memory by going off-chip is always considered to be a good design decision in the field of Hardware Design. So in order to achieve this particular advantage, the on-chip BRAM's can be utilized in order to achieve performance. In the design described in the previous sections, the lookup to the external memory is made to all the subsequences of the subject sequence, and the number of hits is partly determined by the first bit in the lookup table. So if the first bit in the lookup table can be cached to the on-chip BRAM's,

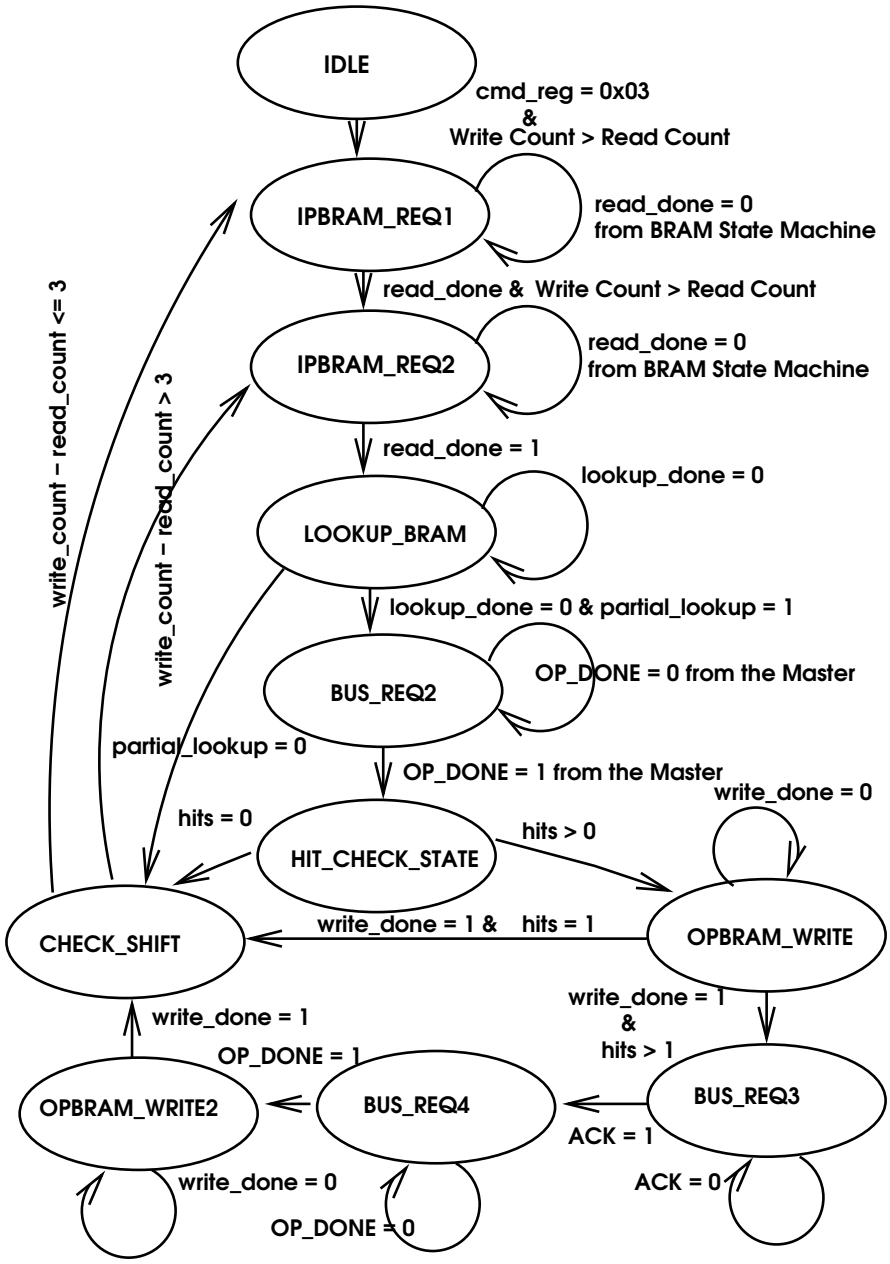


Figure 3.9. Hardware BLAST Slave State Machine for On-Chip Partial Lookup

the decision whether to access the external memory can be made based on the bit in the BRAM which would save a large number of clock cycles and improve performance. As per the original slave state machine as shown in Figure 3.5 the state BUS_REQ1 is the state where the slave hands-off control to the Master in order to determine the lookup. This particular state has been replaced by a lookup to a BRAM and so based on the content of the BRAM, the decision whether to access the external memory or not is made and the corresponding state machine is shown in in Figure 3.9. This design decision has both its inherent advantages and also disadvantages. The advantages it has is, it saves a couple of clock cycles by not allowing to access to the external memory if the value that has been looked up is a *'1'*. However in case the lookup value in the BRAM's is a *'0'* then it adds additional 2 cycle delay of accessing the BRAM before accessing the external memory.

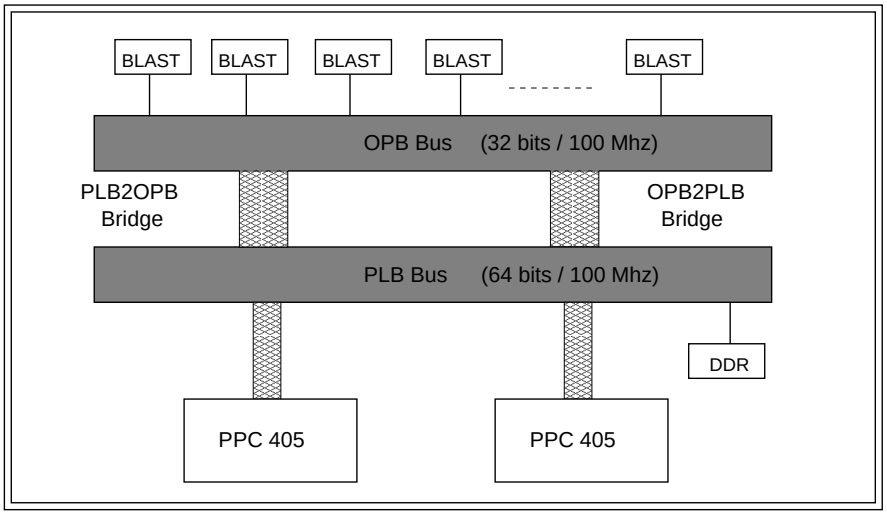


Figure 3.10. Scalable Design

3.3.2 Scalable Design

This section discusses about the scalable design implementation in order to optimize the resources available in the hardware. The reasons for implementing the scalable design are:

- Speed
- Cost
- Power
- Moore's law

As the database sizes are exponentially increasing and as shown clearly in Figure 1.2 they no longer fit in the main memory and an access to the disk has to be made in order to access the database which indeed rapidly decreases the program's speed of execution because of the lesser disk I/O bandwidth when compared to processor speed. However major bioinformatics labs whose main concern is the job turn-around time deploy large clusters which cost millions of dollars, but small bioinformatics labs who cannot afford such big clusters, settle down with ordinary machines. So cost is a major factor which hinders the progress of various biologists in small bioinformatics labs. So one of the goals that the thesis addresses is the fact that even though the approach and the design may not be compared with the speed of running BLAST on a million dollar machine, but if it can give substantial results which can be compared to a million dollar machine, then the approach can be suggested to biologists in small bioinformatics labs. So since Moore's law gives us double the number of transistors every 18 months, for FPGA's it is more number of Configurable Logic Block's which the user can use as computational units. And after the design of a single BLAST core unit on a ML-310 development

board, the BLAST hardware core occupies just 4% of the XC2VP30 FPGA which is approximately 800 slices and draws just 15mW of power. About 50% of the chip is occupied by the base system to run the Linux kernel on the FPGA, so the remaining slices in the FPGA can be utilized to fit in more units and can be made to run in parallel while drawing just few Milli watts of power which is very less when compared to adding desktop processors to run more BLAST match units giving speedup at no cost and is shown in Figure 3.10. The same procedure was also adapted to incorporate more BLAST units with support for on-chip caching of partial lookup and all the statistics are explained in detail in chapter 4.

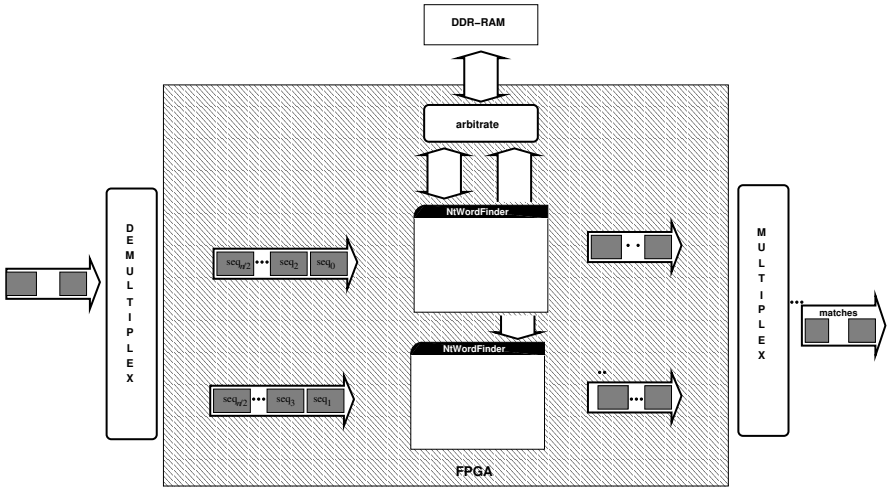


Figure 3.11. Scalable Design for reducing Latency

3.3.2.1 Latency and Throughput

As described previously that, BLAST is more I/O bound, the design can be further extended in order to reduce the latency and increase the throughput and by changing few design decisions in software, the hardware can be optimized for either of the characteristics. A scalable design will be more efficient if the available bandwidths are uti-

lized to their maximum extent. As mentioned in chapter 2 earlier any subject sequence database consists of many independent sections of subject sequences. Blast application performs a match operation between one section of subject sequence and the query and generates the match results. After generating results with the first section, it performs the match operation between the query sequence and the second section of the subject sequence and so on. At any instant a Blast match unit in hardware performs the match operation between one section of subject sequence and the query sequence. By having multiple Blast match units instantiated in the FPGA hardware, match operations between multiple sections of the subject sequence and the query sequence can be performed in parallel. This scalable model utilizes all the bandwidth and space available to its maximum extent to increase the performance of the Blast application.

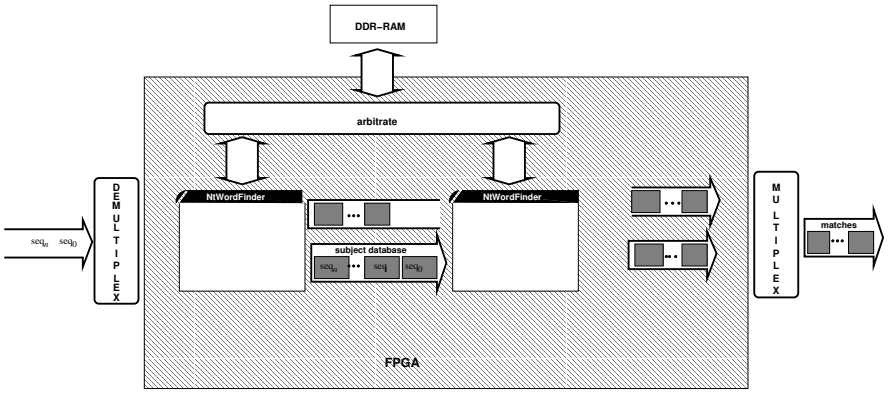


Figure 3.12. Scalable Design for increasing throughput

The Figure 3.11 shows one implementation of the scalable design where in the latency of the BLAST algorithm can be reduced. The Figure 3.11 shows a case in which two Blast match units are instantiated into the FPGA hardware. Along with the match units, a demultiplexer and a multiplexer are also added to the design in the software in order to partition the subject database into subsequences and feed the hardware in order to produce correct results. The different sub-sequences of the subject sequence in

the input stream are demultiplexed properly so that each sub-sequence is fed to a different match unit in the hardware. The output hit information produced by each match unit is multiplexed onto a single output stream and execution of the rest of the BLAST Algorithm continues. The DDR Controller on the PLB Bus arbitrates the address requests targeted to DDR-RAM and are arbitrated by writing different base address to do the lookup's to different blast match units and retransmit the data returned from DDR-RAM to the correct match unit.

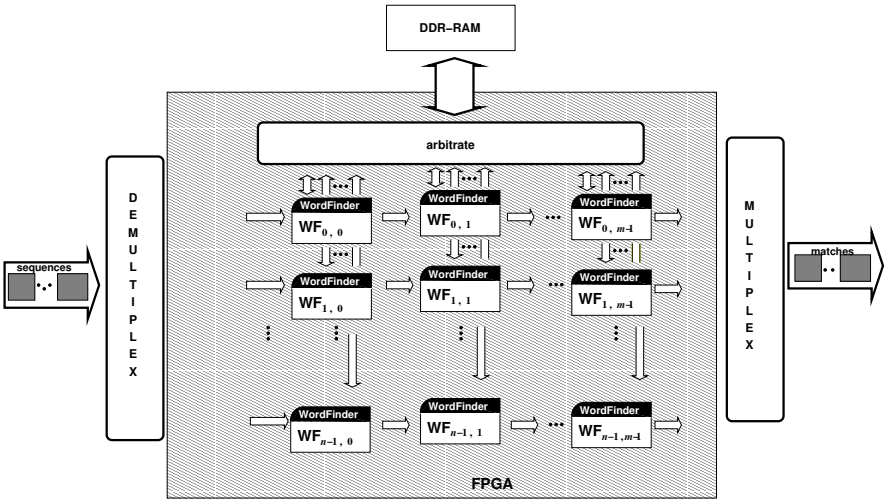


Figure 3.13. Scalable Design

This scalable design model with few modifications can also be used to increase the throughput of the BLAST Algorithm also by performing the match operation between the same subject sequence database and multiple query sequences at the same time. Figure 3.12 shows the design in order to achieve high throughput. The figure shows two Blast match units instantiated in the FPGA hardware. Each Blast match unit in this design generates hit information between the same section of the same subject sequence and different query sequences. The multiplexer in software multiplexes the hit information from different match units. However, the number of different query se-

quences is limited by the number of query lookup tables that can be put in the available DDR-RAM or the amount of resources in the FPGA.

Combining the above two scalable designs described which can reduce the latency and increase the throughput, a generalized design decision was made in order to combine both the designs into a single design of the BLAST Algorithm. The model shown in Figure 3.13 is a generalized model with $n \times m$ match units performing match operations between n different subject sequences and m different query sequences. The maximum scalability of this design is mainly limited by the number of resources available in the FPGA and the number of lookup tables that the DDR RAM can hold.

In this chapter, all the work performed in order to answer the thesis question made in chapter 1 are explained with designs.

Chapter 4

Analysis

In order to answer the thesis question made in chapter 1, a number of experiments were performed and analysis was done as part of the results obtained from the experiments. In the chapter, the emphasis is made on the term cost-effectiveness, since the term *cost-effectiveness* is one of the major aspects of the thesis statement. The term cost-effectiveness implicitly means how effective is the design in terms of cost in comparison to the cost of a typical desktop processor or cost per node of a large cluster. The chapter analyzes all the results obtained in terms of cost-effectiveness with regard to scalability, performance and portability which are it's indirect contributors.

Initially a standalone system was built with a single RC-BLAST core. After verifying for functionality in a standalone system, the core was added to the base system which had the ability to run a Linux kernel with the additional cores specified in chapter 3 and all the results described in the sections below are as result of this complete system.

4.1 Scalability

Scalability is the ability to increase units and continue to effectively increase the performance. With respect to FPGA's, scalability depends on the amount of logic slices the core consumes and the amount of performance obtained out of it. In terms of RC-BLAST, since the Intellectual property (IP) core for RC-BLAST implemented in the FPGA takes only 4% of the amount of logic resources available, RC-BLAST IP cores could be replicated in order to fill up the additional resources available. So thereby a platform FPGA is giving the ability to incorporate more cores at no extra cost. So thereby the term *scalable* is considered as an indirect contributor to the term *cost-effectiveness* . As it can be seem from Table 4.1, the FPGA XC2VP30 that has been chosen to implement RC-BLAST reaches a slice usage of 95% with 8 RC-BLAST cores. The same procedure was also adapted to build a system with the RC-BLAST core with support for On-Chip Caching of the Partial Look-up. The resource usage for the system with support for On-Chip caching of Partial Look-up is shown in Table 4.2. The Table 4.2 clearly shows that the FPGA XC2VP30 runs out of resources with regards to the amount of Block RAM's consumed in regard to the core design for On-Chip Caching of Partial Look-up. Figure 4.1 shows the amount of resources that one core of RC-BLAST consumes in the XC2VP30 FPGA and Figure 4.2 shows the amount of resources that one RC-BLAST core with support for On-Chip caching of Partial Look-up consumes. So having the ability to instantiate more cores in the system thereby means that all the cores instantiated could run in parallel thus providing significant speed up at no extra cost.

Table 4.1. Statistics for the system with various number of cores of RC-BLAST

Number Of Cores	Slices	% of slices consumed	BRAM's	% of BRAM's consumed	Power Consumption (mW)
One	7349	53%	36	25%	2338.81
Two	8150	59%	40	29%	2351.45
Four	9752	71%	48	36%	2363.27
Eight	13119	95%	56	41%	2376.63

Table 4.2. Statistics for the system with various number of cores of RC-BLAST with support for on chip caching of partial look-up

Number Of Cores	Slices	% of slices consumed	BRAM's	% of BRAM's consumed	Power Consumption(mW)
One	7700	56%	68	50%	2353.94
Two	8816	64%	104	76%	2366.44

```
-------------------------------------------------------------------
Logic Utilization:
  Number of Slice Flip Flops:      907  out of  27,392   3.2%
  Number of 4 input LUTs:          996  out of  27,392   3.5%

Logic Distribution:
  Number of occupied Slices:       801  out of  13,696   5.8%

Total Number 4 input LUTs:         1104 out of 27,392    8.0%
  Number used as logic:             996
  Number used as a route-thru:      111

  Number of Block RAMs:              4  out of   136      2.9%
-------------------------------------------------------------------
```

Figure 4.1. Amount of Logic resources used for one RC-BLAST core

```
-------------------------------------------------------------------------
Logic Utilization:
  Number of Slice Flip Flops:      1049  out of  27,392   3.8%
  Number of 4 input LUTs:          1568  out of  27,392   5.7%

Logic Distribution:
  Number of occupied Slices:       1116  out of  13,696   8.14%

Total Number 4 input LUTs:         1679 out of 27,392     6.1%
  Number used as logic:            1568
  Number used as a route-thru:      111

  Number of Block RAMs:             36  out of   136      26.4%
-------------------------------------------------------------------------
```

Figure 4.2. Amount of Logic resources used for one RC-BLAST core with On-Chip caching of Partial Look-up

4.2 Power

One of the major issues also that the thesis addresses is amount of power consumption that the design implemented takes since power, static and dynamic, have become a major issue for building large parallel systems. Table 4.1 and Table 4.2 also show the amount of power that the system dissipates with the addition of various number of RC-BLAST cores. From Table 4.3 and Table 4.4 that show the amount of power consumption for one core of RC-BLAST in the system and it is clear that adding one core of RC-BLAST to the system would increase the power consumption of the system by just around $12mW$. This clearly addresses the issue of scalability of the system since, when a bigger FPGA is obtained the cores can be simply added to the system with a lot lesser power consumption as opposed to adding a cluster of desktop processors where in the power consumption increases by around $100W$ each which is very high. A plot comparing the power dissipation of desktop processors versus FPGA's for various number of BLAST cores is shown in Figure 4.3 and it is very clear from the plot that FPGA's consume very less power with regard to processors and are at a very

advantageous position over the desktop processors.

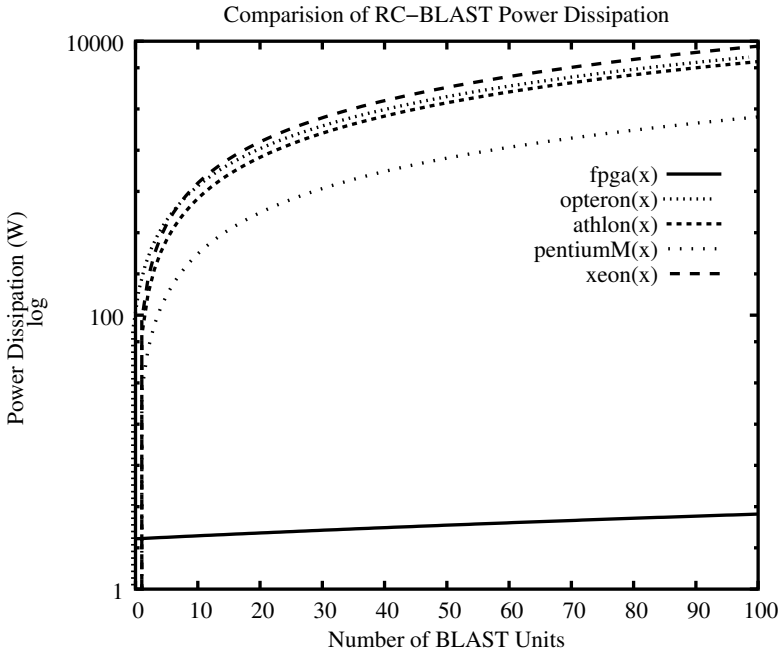


Figure 4.3. Power Dissipation Statistics of RC-BLAST

Table 4.3. Power statistics for one core of RC- BLAST

Vcc	Dynamic/ Quiescent	Volts(V)	Current(mA)	Power(mW)
Vccint	Dynamic	1.5	8.42	12.63

4.3 Price

This section describes the effectiveness of the design in terms of *price*. Price implicitly means the amount of dollars spent in order to run the complete system. From section 4.1, it is clear that each Xilinx ML - 310 Development board which has a XC2VP30 FPGA can hold 8 RC-BLAST cores and the approximate cost of the development board is around $1000. Assuming the present prices of Desktop processors,

Table 4.4. Power statistics for one core of RC- BLAST with On-chip caching of Partial Look-up

Vcc	Dynamic/ Quiescent	Volts(V)	Current(mA)	Power(mW)
Vccint	Dynamic	1.5	8.33	12.49

an average node for a cluster costs around $450, thereby adding each desktop processor will enable to run more software versions of BLAST. An XC2VP100 FPGA from Xilinx, which is one of the largest and costliest FPGA's available in the market can accommodate up to 24 RC-BLAST cores. Based on the present estimates, Figure 4.4 is a plot that compares the number of BLAST cores obtained as the cost increases for various hardware systems described above. It is clear from the plot that initially, the regular desktop processors have an intrinsic advantage over the FPGA's, but as the price increases, the FPGA solution presented overcomes all the advantages that the desktop processors have and prove to be cost-effective providing more number of RC-BLAST cores at a lesser cost.

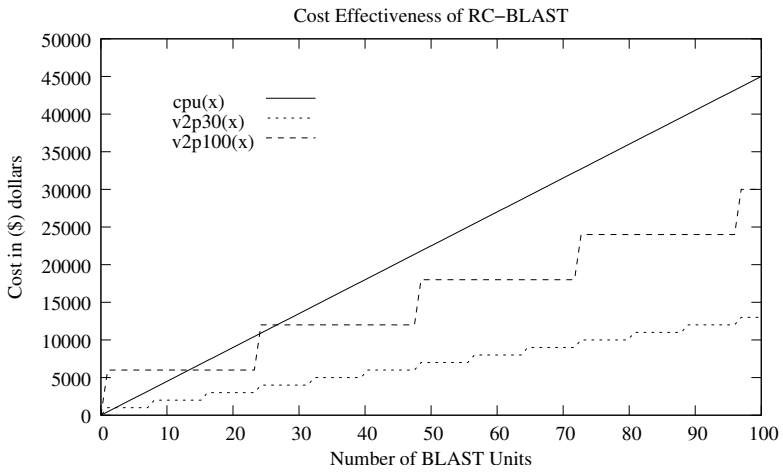


Figure 4.4. Scalability of RC-BLAST

4.4 Performance

Performance is also an important aspect that has to be taken into consideration. Since obtaining more number of cores at a lower price with a lot lesser power consumption is of not much use unless there is enough performance obtained with the solution presented. So in order to examine the performance of the FPGA based version of BLAST, the number of lookup's that can be performed per second was considered as a good measure and is plotted for a query and a database in Figure 4.5. In Figure 4.5 "S/W on xc2vp30" indicates the time taken to perform the number of lookup's on the PowerPC processor inside the FPGA and "H/W + S/W on xc2vp30" indicates the time taken to run the same number of lookup's, with BLAST running on the Power PC processor and the computationally intensive part running on the logic slices of the FPGA. In order to have fair comparison, the same set of queries and databases were also run on a Pentium 4 2.2 GHz Machine and an average of all the runs was taken and it can be observed from Figure 4.5 that the H/W and S/W implementation on the FPGA can perform almost the same number of lookup's/sec as a typical desktop processor.

So according to the results obtained in the above sections, it can be easily be stated that with bigger FPGA's with more number of logic gates, more would be the number of RC-BLAST cores that could fit in an FPGA with lot of less power consumption along with speed-up at a very low cost.

4.5 Portability

The issue of portability is very important since the design has to meet various future technologies that are going to be available in the near future. The present design can be targeted to any platform FPGA from Xilinx, Inc with very minimal effort by simply us-

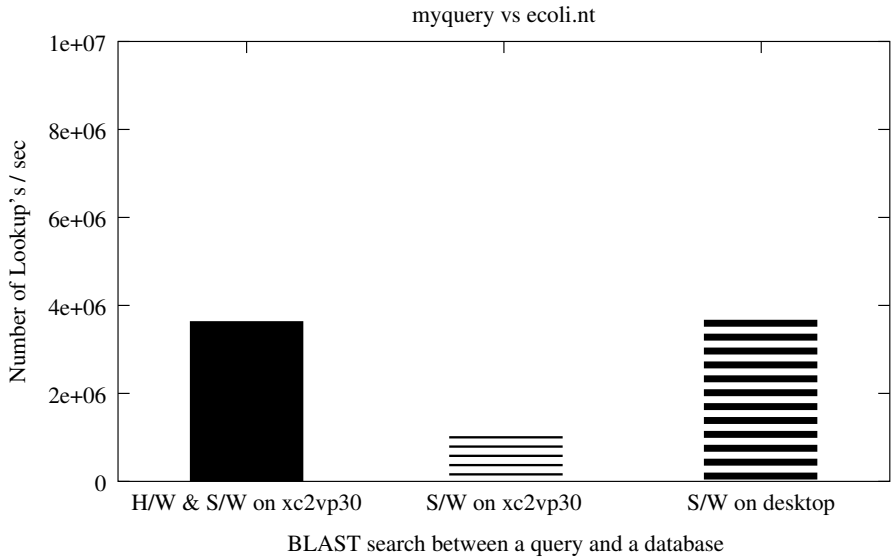


Figure 4.5. Performance of RC-BLAST

ing the Xilinx Platform Studio and the Embedded Development Kit (EDK). With more effort, the standard vendor tools could be used on other Xilinx based boards. However in order to implement the design for various other commercial Platform FPGA's with different Bus Architectures, the design has to be undergo minimal changes in the code by replacing the IPIF Interface present in the logic with the appropriate Bus Interface for the Platform FPGA chosen to use. Thereby in order to prove the concept of portability, the RC-BLAST core designed was also synthesized to a bit-file for a Xilinx ML-403 platform which hosts a Virtex 4 FPGA (XC4VFX12) using the Xilinx EDK software and the resource usage is shown in Figure 4.6.

This chapter analyzes all the results obtained as part of the experiments performed in order to answer the various aspects of scalability, portability and cost-effectiveness according to the thesis question made in chapter 1.

```
-------------------------------------------------------------------------
Logic Utilization:
  Number of Slice Flip Flops:       1,134 out of  10,944   10.3%

Logic Distribution:
  Number of occupied Slices:        971    out of   5,472   17.7%
-------------------------------------------------------------------------
```

Figure 4.6. Amount of Logic resources used for one RC-BLAST core on a Xilinx Virtex 4 FX - 12 device

Chapter 5

Conclusion

The goal of the thesis work was to answer the question *Is a scalable and cost-effective Reconfigurable Implementation of BLAST feasible which can aid scientists and biologists in a more productive way?* Based on the experiments and analysis performed on the results obtained, the answer to the thesis question is *Yes, a scalable and cost-effective Reconfigurable Implementation of BLAST is feasible which can aid scientists and biologists in a more productive way.*

In order to answer the above question , these were the list of contributions as part of the thesis.

- Repeated and verified the profiling characteristics as specified in *Design and Implementation of Open source FPGA-based accelerator for BLAST* [33].
- Extended the profiling characteristic to several machines with different processors, memory, disk and network interfaces.
- Helped to quantify the I/O bound characteristic of BLAST by running it across different file systems with different bandwidths.

- Ported BLAST to Xilinx ML-310 Platform FPGA development board and characterized the essential things needed to do a port
  - Developed a hardware design for RC-BLAST which is scalable and cost-effective.
  - Ported Linux to a Platform FPGA .
  - Developed a device driver that accesses the hardware from the software .
  - Developed a patch to the BLAST software needed to invoke the hardware .
  - Implemented and measured the pros & cons of using the On - Chip BRAM's as a cache in order to speed up BLAST .
  - Proved the concept of portability, by a dummy port to a Xilinx ML - 403 development board .

This work apart from just the implementation also showed a portable, scalable and a cost effective solution with a lot of lesser power consumption for BLAST. Moreover the source files for both hardware and software have been made Open-source under the GNU Public License, so that many other researchers at various academic institutions and industry can modify the source according to their interface and board requirements and port it to their platforms with minimal modifications.

However, this work mainly concentrates on the **blastn** component of the BLAST program which performs matching operations on nucleotide sequences and can be easily extended to other flavors of BLAST as mentioned in chapter 3.

From the implementation statistics shown in chapter 3, it is identified that the BLAST heuristic is no longer compute bound, but an I/O bound problem. Future work will focus on handling the I/O bound problem of the application in a more effective way. This can be achieved in different ways.

- Porting the FPGA based BLAST Algorithm to a Cray XD-1 [14] where the memory bandwidth problem is of very less concern because of the architecture of Cray XD-1. The architecture of Cray XD-1 uses a Rapid Array Interconnect and Rapid Array Transport Interface in-order to reduce the memory bandwidth problem to a large extent. Also porting the present implementation to an SGI - Altix [5] super computing machine whose main purpose is to deal with memory intensive applications.

- Port FPGA based BLAST implementation to a Reconfigurable Cluster with SATA drives [7] with FPGA's and use Aurora Protocol [47] for transmission of sequence and query data over Rocket I/O Transceivers [28] between FPGA's.

- Port FPGA based BLAST to a Virtex IV device with a APU interface which disables the extra latency obtained by the OPB and PLB buses in a Virtex II Pro FPGA.

Appendix A

Protocol Component Declarations

In this section, the vhdl entity declarations for the components that were used in the design of the **BLAST match unit** are listed. As specified in the chapter 3, the design is attached to the On-Chip Peripheral bus as a Master-Slave user peripheral core. In-order to simplify the process of attaching a user core to a Core-Connect bus like the On-Chip Peripheral Bus, the user core makes use of a portable, predesigned bus interface (called the IP Interface, IPIF) that takes care of the bus interface signals, bus protocol, and other interface issues. The IPIF presents an interface to the user's core called the IP Inter-Connect (IPIC). Any user core that is designed with an IPIC has the advantage that it is portable and can be easily reused on different processor buses by changing the IPIF to which it is attached.

Various entities in the **BLAST match unit** design are as follows :

- The top-level entity with the IPIF and IPIC interface and master - slave components for RC - BLAST to be compatible to run on the On-Chip Peripheral Bus :-

```
entity OPB_BLAST is
```

```
-- Generic Declarations for the Blast Match Unit
generic
(
  C_OPB_CLK_PERIOD_PS   : integer   := 10000;
  C_IP_MASTER_PRESENT   : integer   := 1;
  C_DEV_BLK_ID          : integer   := 0;
  C_DEV_BURST_ENABLE    : integer   := 0;

  C_BASEADDR            : std_logic_vector(0 to 31) := X"93000000";
  C_HIGHADDR            : std_logic_vector(0 to 31) := X"930FFFFF";
  C_DEV_MIR_ENABLE      : integer                   := 0;
  C_OPB_AWIDTH          : integer                   := 32;
  C_OPB_DWIDTH          : integer                   := 32;
  C_FAMILY              : string                    := "virtex2p"
);
port
(
  --Global ports:
  OPB_Rst       : in  std_logic := '0';
  OPB_Clk       : in  std_logic := '0';

  Interrupt         :  out std_logic;
  IP2Bus_IntrEvent :  in  std_logic_vector(0 to 0);

  --OPB ports master/slave:
  OPB_ABus      : in  std_logic_vector(0 to C_OPB_AWIDTH-1);
  OPB_DBus      : in  std_logic_vector(0 to C_OPB_DWIDTH-1);
  OPB_BE        : in  std_logic_vector(0 to C_OPB_DWIDTH/8-1);
  OPB_RNW       : in  std_logic;
  OPB_select    : in  std_logic;
  OPB_seqAddr   : in  std_logic;

  --OPB slave output ports:
  Sln_DBus      : out std_logic_vector(0 to C_OPB_DWIDTH-1);
  Sln_errAck    : out std_logic;
  Sln_retry     : out std_logic;
  Sln_toutSup   : out std_logic;
  Sln_xferAck   : out std_logic;

  --OPB master output ports:
  Mn_ABus           : out std_logic_vector(0 to C_OPB_DWIDTH - 1 );
  Mn_request        : out std_logic;
  Mn_busLock        : out std_logic;
```

```
    Mn_select          : out std_logic;
    Mn_RNW             : out std_logic;
    Mn_BE              : out std_logic_vector(0 to C_OPB_DWIDTH/8 - 1 );
    Mn_seqAddr         : out std_logic;

    --OPB master input ports:
    OPB_MnGrant        : in  std_logic := '0';
    OPB_xferAck        : in  std_logic := '0';
    OPB_errAck         : in  std_logic := '0';
    OPB_retry          : in  std_logic := '0';
    OPB_timeout        : in  std_logic := '0';

    -- Other control ports:
    Freeze             : in  std_logic := '0'


  );
end entity OPB_BLAST;
```

- Slave Interface of the BLAST match unit :- The slave interface is the main match unit component which receives the subject sequence data from the device driver and writes to the FIFO's. The slave interface also has various other registers which are necessary to debug while running the design on the hardware. After the hardware is done with running it's function, the hardware signals a **DONE** signal to the device driver which resumes the ordinary execution of the processor. The slave interface is also responsible for writing the hit information back to the Device Driver based on the subject sequence data and the query.

```
entity blast_slv is
  port (

------------------------------------------------------------------------------
-- Signals from the IPIF
------------------------------------------------------------------------------

    Bus2IP_Reset          :     in  std_logic;
    Bus2IP_Addr           :     in  std_logic_vector(0 to 31);
    Bus2IP_Clk            :     in  std_logic;
```

```
    Bus2IP_RdReq          :     in  std_logic;
    IP2Bus_RdAck          :     out std_logic;
    Bus2IP_WrReq          :     in  std_logic;
    IP2Bus_WrAck          :     out std_logic;
    Bus2IP_Data           :     in  std_logic_vector(0 to 31);
    Bus2IP_Reg_RdCE       :     in  std_logic_vector(0 to 3);
    Bus2IP_Reg_WrCE       :     in  std_logic_vector(0 to 3);
    IP2Bus_Data           :     out std_logic_vector(0 to 31);
    IP2Bus_Data_Sel       :     out std_logic;

-------------------------------------------------------------------------------
-- Signals between the Master and the Slave
-------------------------------------------------------------------------------

    bus_request           :     out std_logic;
    bus_request2          :     out std_logic;
    bus_req_ack           :     in  std_logic;
    bus_req_ack2          :     in  std_logic;
    bus_rel_done2         :     in  std_logic;
    bus_rel_done          :     in  std_logic;
    lookup_reg            :     out std_logic_vector(0 to 18);
    lookup_reg_plusfour   :     out std_logic_vector(0 to 18);
    lookup_data           :     in std_logic_vector(0 to 31);
    lookup_data2          :     in std_logic_vector(0 to 31);
    ddr_addr              :     out std_logic_vector(0 to 31);
    master_status_reg     :     in std_logic_vector(0 to 7);
    read_count            :     in std_logic_vector(0 to 31)
    );
end entity blast_slv;
```

- Master Interface of the BLAST match unit:- The master interface of the BLAST match unit component is used to read the look-up table entries that are stored in the DDR Memory based on the subject sequence data. The Master Interface has a state machine that is invoked through the Slave State Machine, and obtaining the required look-up table value, the control is reverted back to the Slave State Machine which continues processing.

```
entity blast_msc is
  generic
```

```
(
  C_OPB_AWIDTH                : INTEGER := 32;
  C_OPB_DWIDTH                : INTEGER := 32
);

port
(
-------------------------------------------------------------------------------
-- Signals from the IPIF
-------------------------------------------------------------------------------

  Bus2IP_Clk          : in  std_logic;
  Bus2IP_Data         : in  std_logic_vector(0 to C_OPB_DWIDTH-1);
  Bus2IP_Freeze       : in  std_logic;
  Bus2IP_MstRdAck     : in  std_logic;
  Bus2IP_MstWrAck     : in  std_logic;
  Bus2IP_MstRetry     : in  std_logic;
  Bus2IP_MstError     : in  std_logic;
  Bus2IP_MstTimeOut   : in  std_logic;
  Bus2IP_MstLastAck   : in  std_logic;
  Bus2IP_Reset        : in  std_logic;
  IP2Bus_Addr         : out std_logic_vector(0 to C_OPB_AWIDTH - 1);
  IP2Bus_Data         : out std_logic_vector(0 to C_OPB_DWIDTH -1 );
  IP2Bus_MstBE        : out std_logic_vector(0 to C_OPB_DWIDTH/8 - 1);
  IP2Bus_MstRdReq     : out std_logic := '0';
  IP2Bus_MstWrReq     : out std_logic := '0';
  IP2Bus_MstBurst     : out std_logic := '0';
  IP2Bus_MstBusLock   : out std_logic := '0';

-------------------------------------------------------------------------------
 -- Signals between the Master and the Slave --
-------------------------------------------------------------------------------

  bus_request         : in  std_logic;
  bus_req_ack         : out std_logic := '0';
  bus_request2        : in  std_logic;
  bus_req_ack2        : out std_logic := '0';
  master_status_reg   : out std_logic_vector(0 to 7);
  lookup_data         : out std_logic_vector(0 to 31);
  lookup_data2        : out std_logic_vector(0 to 31);
  lookup_reg          : in  std_logic_vector(0 to 18);
  lookup_reg_plusfour: in std_logic_vector(0 to 18);
  read_count          : out std_logic_vector(0 to 31);
  ddr_addr            : in  std_logic_vector(0 to 31);
```

```
    bus_rel_done2      : out std_logic :='0';
    bus_rel_done       : out std_logic :='0'
  );
end entity blast_msc;
```

- Entity declaration for the FIFO :- The declaration of the FIFO has been designed by concatenating two Block RAM's available in all the Xilinx FPGA's. The same entity is indeed used for both the Input and the Output FIFO's. The FIFO's have a capacity to hold 1024 values and each value can be a 32-bit value.

```
entity bram is
 port (
        clk         : in std_logic;
        bram_addra  : in std_logic_vector(0 to 9);
        bram_addrb  : in std_logic_vector(0 to 9);
        bram_dia    : in std_logic_vector(0 to 31);
        bram_dib    : in std_logic_vector(0 to 31);
        bram_dopa   : in std_logic_vector(0 to 3);
        bram_dopb   : in std_logic_vector(0 to 3);
        bram_wea    : in std_logic;
        bram_web    : in std_logic;
        bram_clka   : in std_logic;
        bram_clkb   : in std_logic;
        bram_ssra   : in std_logic;
        bram_ssrb   : in std_logic;
        bram_ena    : in std_logic;
        bram_enb    : in std_logic;
        bram_doa    : out std_logic_vector(0 to 31);
        bram_dob    : out std_logic_vector(0 to 31);
        bram_dipb   : out std_logic_vector(0 to 3);
        bram_dipa   : out std_logic_vector(0 to 3));

end entity bram;
```

Appendix B

Device Driver declarations

In this section, the device driver function declarations for the **BLAST Match Unit** are specified. The device driver makes use of standard Linux System Calls in-order to access the FPGA based Hardware.

```
/*
 ============================================================
                        rcblast-drv
 ------------------------------------------------------------
*/

MODULE_DESCRIPTION("Device Driver for RC-BLAST core");
MODULE_LICENSE("GPL");

EXPORT_NO_SYMBOLS;

static int ninstances = 1; /* Number of match instances */


/*-------------------------------------------------------------------------
 Register Address declarations for the RC-BLAST Match unit Hardware
-------------------------------------------------------------------------*/

#define MSWSTART        0x00010000 /* Master Start */
#define SLWSTART        0x00020000 /* Slave Start */
#define DDRBASE         0x00030000 /* DDR Base Address register */
```

```
#define CMD             0x00040000 /* Command register */
#define OUTRDC          0x00050000 /* Output fifo read count */
#define INWRC           0x00060000 /* Input fifo write count */
#define INRDC           0x00070000 /* Input fifo read count */
#define OUTWRC          0x00080000 /* Output fifo write count */
#define SLSTAT          0x00090000 /* slave status */
#define MSSTAT          0x000A0000 /* master status */
#define DONE            0x000B0000 /* done register */
#define SOFF            0x000C0000 /* subject offset register */
#define SUBJSIZE        0x000D0000 /* size of the subject sequence register *
#define OUTFIFO         0x000E0000 /* Output Fifo */
#define INFIFO          0x000F0000 /* Input Fifo */

#define INSIZE          1024       /* num. of 32-bit entries */
#define OUTSIZE         1024       /* num. of 32-bit entries */

/*-----------------------------------------------------------------------
 Prototypes and global variables
-----------------------------------------------------------------------*/

/*-----------------------------------------------------------------------
Sub-routine for allocating Virtual Memory Space in the Linux Kernel
-----------------------------------------------------------------------*/

static unsigned char *alloc ( unsigned long , unsigned long , char *) ;

/*-----------------------------------------------------------------------
Sub-routine for Initializing the Hardware.
-----------------------------------------------------------------------*/

static int rcblast_open(struct inode*,struct file*) ;

/*-----------------------------------------------------------------------
Sub-routine for reading back the hit information from the Hardware.
-----------------------------------------------------------------------*/

static ssize_t rcblast_read(struct file*,char *,size_t,loff_t *);

/*-----------------------------------------------------------------------
Sub-routine for writing to the Hardware.
-----------------------------------------------------------------------*/

static ssize_t rcblast_write(struct file*,const char *,size_t,loff_t *);
```

```
/*-------------------------------------------------------------------------
Sub-routine for writing the Look-up table to the DDR - RAM.
-------------------------------------------------------------------------*/

static int write_lut(struct file *,const char *,size_t,loff_t *);

/*-------------------------------------------------------------------------
Sub-routine for writing the subject database to the FIFO's.
-------------------------------------------------------------------------*/

static int write_fifo(struct file *,const char *,size_t,loff_t *);

/*-------------------------------------------------------------------------
Constant Declarations of the physical address space of the hardware
-------------------------------------------------------------------------*/

static unsigned long    base1 = 0x93000000 ;
static unsigned long    len1 = 0x00100000 ;

/*-------------------------------------------------------------------------
Structure declaration of File Operations for RC-BLAST
-------------------------------------------------------------------------*/

static struct file_operations rcblast_fops = {
        owner : THIS_MODULE,
        open :          rcblast_open,
        read :          rcblast_read,
        write :         rcblast_write,
        mmap :          rcblast_mmap,
};
```

References

[1] Cray XD 1. White paper - closing the gap between peak and achievable performance in high performance computing, 2004.

[2] Sangiovanni-Vincentelli A, El Gamal A., and Rose J. Synthesis method for field programmable gate arrays. 81:1057–1083, Jul 1993.

[3] Elias Ahmed and Jonathan Rose. The effect of LUT and cluster size on deep-submicron FPGA performance and density. In *FPGA*, pages 3–12, 2000.

[4] Peter Alfke. Evolution and revolution: Recent progress in field-programmable logic, 2001.

[5] SGI® Altix®. URL: http://www.sgi.com/products/servers/altix/.

[6] S. Altschul, W. Gish, W. Miller, E. W. Myers, and D. Lipman. A basic local alignment search tool. In *J.Mol.Biol.215,3,403-310.*, 1990.

[7] Serial ATA. URL: http://www.serialata.org.

[8] R. D. Bjornson, A.H.Sherman, S.B.Weston, N. Willard, and J.Wing. Turboblast : A parallel implementation of blast built on the turbohub. In *Proceedings of the*

Third IEEE International Workshop on High Performance Computational Biology, April 2003.

[9] Nick Camp, Haruna Cofer, and Roberto Gomperts. High throughput - blast, 1998.

[10] Jacques Cohen. Bioinformatics - an introduction for computer scientists. *ACM Comput. Surv.*, 36(2):122–158, 2004.

[11] K. Coloma, A. Choudhary, A. Ching, W. K. Liao, S. W. Son, M. Kandemir, and L. Ward. Power and performance in i/o for scientific applications. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium IPDPS*, April 2005.

[12] K. Compton and et al. An introduction to reconfigurable computing.

[13] K. Compton and S. Hauck. Reconfigurable computing: A survey of systems and software, 2000.

[14] Cray XD-1 Super Computer. URL: http://www.cray.com/products/xd1.

[15] Jason Cong and Yuzheng Ding. Combinational logic synthesis for lut based field programmable gate arrays. *ACM Trans. Des. Autom. Electron. Syst.*, 1(2):145–204, 1996.

[16] Mpiblast Aaron Darling. The design, implementation, and evaluation of.

[17] FASTA. Fasta. URL: http://www-nbrf.georgetown.edu/pirwww/search.fasta.html.

[18] Message Passing Interface Forum. MPI: A message-passing interface standard, 1994.

[19] Xilinx Virtex-4 FPGA's. URL: http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/index.htm.

[20] Al Geist, William Gropp, Steve Huss-Lederman, Andrew Lumsdaine, Ewing L. Lusk, William Saphir, Tony Skjellum, and Marc Snir. MPI-2: Extending the message-passing interface. In *Euro-Par, Vol. I*, pages 128–135, 1996.

[21] GenBank®. the nih genetic sequence database. URL: http://www.ncbi.nlm.nih.gov/Genbank.

[22] W Gish. Wu - blast. URL: http://blast.wustl.edu.

[23] Mentor Graphics. URL: http://www.mentor.com.

[24] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[25] D. Gusfield. *Algorithms on strings, Trees and Sequences*. Cambridge University Press, NewYork, 1997.

[26] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996.

[27] IBM. Ibm coreconnect architecture®. URL: http://www-03.ibm.com/chips/products/coreconnect.

[28] Xilinx Inc. Multi - giga bit serial i/o transceivers, December 2004. From webpage at http://www.xilinx.com/esp/networks_telecom/optical/collateral/serial_io.pdf.

[29] Robert Jones. *Introduction to Bioinformatics*. Oreilly Publishers, 2004.

[30] Pierre Marchal. Field-programmable gate arrays. *Commun. ACM*, 42(4):57–59, 1999.

[31] Modelsim. URL: http://www.model.com.

[32] Montavista. URL: http://www.mvista.com.

[33] Krishna Muriki. Design and implementation of open source fpga-based accelerator for blast. Master's thesis, Clemson University, December 2004.

[34] Tim Oliver, Bertil Schmidt, and Douglas Maskell. Hyper customized processors for bio-sequence database scanning on fpgas. In *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 229–237, New York, NY, USA, 2005. ACM Press.

[35] SRC-7 OVERVIEW. URL: http://www.srccomp.com/SRC7.htm.

[36] Kiran Puttegowda, William Worek, Nicholas Pappas, Anusha Dandapani, Peter Athanas, and Allan Dickerman. A run-time reconfigurable system for gene-sequence searching. In *Proceedings of the 16th International Conference on VLSI Design*, 2003.

[37] Karlin. s and Altschul S.F. Methods for assessing the stastical significance of molecular sequence features by using general scoring schemes. *P.A.N.S*, 87:2264–2268, 1993.

[38] Reetinder P. S. Sidhu, Alessandro Mei, and Viktor K. Prasanna. String matching on multicontext fpgas using self-reconfiguration. In *FPGA '99: Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 217–226, New York, NY, USA, 1999. ACM Press.

[39] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[40] Platform Studio and the EDK. URL: http://www.xilinx.com/ise/embedded_design_prod/platform_studio.htm.

[41] DeCypher Time Logic. Recent product citations. url:http://www.timelogic.com/decypher_citations.html.

[42] TimeLogic. URL: http://www.timelogic.com.

[43] TimeLogic. Decypherblast. URL: http://www.timelogic.com/decypher_blast.html.

[44] TimeLogic. Geneblast - an intron spanning extension to the blast algorithm.

[45] Keith Underwood. Fpgas vs. cpus: trends in peak floating-point performance. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 171–180, New York, NY, USA, 2004. ACM Press.

[46] Hao Wang, Twee Hee Ong, Beng Chin Ooi, and Kian Lee Tan. Blast++: A tool for blasting queries in batches. In *Proceedings of the Third IEEE International Workshop on High Performance Computational Biology*, April 2003.

[47] Inc. Xilinx. Aurora link layer protocol. From webpage at http://www.xilinx.com/products/design_resources/conn_central/grouping/aurora.htm.

[48] Mark Yandell, Ian Korf, and Joseph Bedell. *BLAST*. Oreilly Publishers, 2003.