Software and hardware approaches to data compression of IEEE 64-bit floating point data

by

Rashmi Arun Patankar

A Thesis Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE

Department: Electrical Engineering and Computer Engineering
Major: Computer Engineering

Iowa State University
Ames, Iowa
1994

To

Mummy, Daddy and Ayon

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation and thanks to my advisor, Dr. John L. Gustafson for all the assistance, guidance and support offered throughout this research. My sincere thanks to Dr. Charles T. Wright and Dr. Doug Jacobson for their suggestions and for being on my committee. I am indebted to Dr. John Moulder from the Center for Non-destructive Evaluation, US Department of Energy for being a great source of inspiration. Finally, I thank my family and friends for their constant support and encouragement throughout this research.

CHAPTER 1. INTRODUCTION

1.1 Synopsis

Various theories of coding have been proposed to achieve data compression. The choice of the technique to be employed is made based on the nature of the application. In applications where a slight loss of data in the original message does not cause noticeable change in the output, the linear algorithm transformation techniques have proven to be useful. However, if a complete integrity of the message is to be maintained and no loss in the original data is allowed, other techniques can be employed. From the foregoing discussion, it will be noted that the Huffman Coding proves to be an efficient data compression technique for lossless transmission. The work towards this thesis studies two approaches for the compression: hardware and software.

1.1.1 Software

The Huffman code is a variable length code. This code has the unique property of being self-delimiting; there is no need to specify the end of one code and the beginning of the next. A binary search tree is traversed by the bits in the Huffman sequence and the value of the first leaf node encountered is output at the receiving end. Thus, the original sequence is obtained and the complete input sequence is coded.

1.1.2 Message Passing

Every message-passing multicomputer supports a set of inter-node communication methods. The set of inter-node communication methods distinguishes one multicomputer from another. The iWARP, INMOS T800 transputer, iPSC/860, and nCUBE are a few examples of message passing machines.

The performance obtained by a software method for message passing was measured on the nCUBE 2 processor. All memory is distributed in the nCUBE architecture. The information between processors is shared in the form of messages explicitly by communication across I/O channels. The nCUBE typically takes 0.10 milliseconds to start and 2 microseconds for every successive byte.

Latency of the data is measured. Conceptually, it can be defined as the time elapsed between the instant at which the data is sent at the source and the instant when it is received at the destination, including compression and decompression time. The latency time for different lengths of message was noted and a graph of latency verses message length plotted to observe the percent increase in latency with increase in message length.

1.1.3 Hardware

A high-level synthesis approach to realize the behavior of a hardware system has been adopted. The behavioral description of the hardware is given in Verilog Hardware Description Language and the theoretical latency times for different message lengths obtained. Finally, a graph of latency versus message length is plotted to compare the theoretical latency times with the actual latency times obtained in the software case.

The second chapter discusses the various compression techniques followed by a brief overview to the communication technique employed in the nCUBE processor in chapter three. The fourth chapter provides an introduction to the high-level synthesis technique of realising the timings. The behavioral simulation of the hardware was performed on the Olympus synthesis tool, hence an overview on Olympus is provided. Finally, chapters five and six give the implementation details and the results and conclusions repectively.

CHAPTER 2. LITERATURE SURVEY

2.1 Data Compression

Data compression plays a vital role in the field of Digital Communication. Most high speed data communication systems employ one or more data compression techniques.

Speedup can be achieved by reducing the number of bits to be transferred. In cases where there is a large amount of repetition in sequences or a predictable occurrence of bits at regular intervals, data compression helps achieve a great amount of speedup.

Data compression techniques are chosen based on the nature of the input sequence that is to be compressed. The performance that can be achieved from a technique depends upon the length of the message and the nature of the data. Hence, it is left to our discretion to choose an optimal compression technique in order to achieve the best possible performance.

Compression can be performed such that there could be a loss or no loss of data received after compression. In certain applications, a slight loss of data does not cause a significant difference in the data received. Hence it is advantageous to use an efficient coding by allowing for a small loss.

Compression results in the reduction of transmission costs. There is an increase in the operational efficiencies, as more information can be transferred over the data link per unit time interval. Transmission cost is reduced by reducing the amount of data to be physically transferred, thereby making it possible to use a lower speed data link, saving the cost of using a high speed data link.

There are many compression techniques, such as Statistical Encoding, Null Suppression, Bit Mapping, Run Length coding, Half-Byte Packing, Diatomic Encoding, Pattern Substitution and Relative encoding [4] and [3].

2.1.1 Null Suppression

The input data stream has a repeated number of nulls or zeros. Hence, a number of nulls can be replaced by a few characters specifying the number of repetitions and the point of occurrence, thus avoiding the the necessity to transmit an excessive amount of data. The longer the repetition, the higher the speedup.

2.1.2 Bit Mapping

The input data has a large number of specified data types which may be digits or a specific character such as blanks. A bit map is used to indicate the presence or absence of data characters.

2.1.3 Run Length

This technique saves transmitting excess data after the particular repeating sequence reaches a certain pre-established level of occurrence. The coding is performed at run length after the pre-established level of occurrence is obtained.

2.1.4 Half-Byte Packing

This is a variant of the Bit Mapping technique. The original sequence is split into sets of four bits. Each of the four bits is assigned a code.

2.1.5 Diatomic Encoding

Here one character substitutes for two. The bit structure of the special character represents the encoded pair and this provides a fifty percent reduction in data size. To implement this technique, one needs to know the input data sequence beforehand.

2.1.6 Pattern Substitution

This is a variant of diatomic coding where a special character is substituted with a predefined character pattern. This finds extensive use when transmitting program listings and other types of data files containing repetitions. A pattern table gives the set of list arguments and function values which represents the compressed value of a particular argument.

2.1.7 Relative Encoding

In cases where the input data contains a sequence of sequences that vary very slightly with relation to their preceding sequences, the compression allows all the sequences except the first to be represented as the difference function where the difference is within a pre-set value.

This technique is used in telemetry compression, where for example there are numerous space probes that transmit readings at pre-defined intervals of time. They contain a sequence of numeric fields that in turn have sub-sequences or runs of numerics which vary slightly with respect to each other.

2.1.8 Statistical Encoding

All the coding techniques enumerated above assume character codes to be of a fixed size. The size is a constant throughout the coding sequence. Statistical encoding techniques are unique, in that, they employ variable length codes. They account for the probabilities of occurrences of single characters and groups of characters. The shorter length codes are assigned to the sequences with higher probability and the longer length codes to the sequences of lower probability. On the whole there is a great amount of reduction in the number of bits as compared to the original sequence. Examples of Statistical coding are the Huffman Coding technique and Arithmetic coding [2].

2.1.9 Linear Orthogonal Transform

Fourier Transforms and other transform techniques can be applied to achieve data compression in cases where there is no stringent requirement that the data be received with no loss. In applications such as image compression a slight amount of data loss does not make any significant change in the quality of the picture received. It may be possible that by reducing a large amount of data to be transmitted, the picture quality is still retained.

The transform coefficients are calculated for each character in the input data. This results in a one-to-one correspondence of the frequency component with the input data. The Fourier coefficients in the range of frequencies between direct current and about 20 kHz contains the maximum amount of information. The coefficients that lie above 20 kHz or the audible range contribute very little to the strength of the signal and are negligible. These coefficients are therefore ignored and only the values between DC and 20 kHz are transmitted. This results in a large savings of transmitting data at a negligible cost of the quality of data received. But, in situations where no loss is desired, this method is not acceptable.

2.2 Information Theory

If a system transmits at n discrete levels every λ second intervals, the number of different signal combinations in T seconds is $n^{\frac{T}{\lambda}}$. Information is proportional to the length of transmission time. Therefore, the information transmitted in T seconds is $(\frac{T}{\lambda})\log n$. In base 2 systems:

$$H = \frac{T}{\lambda}\log_2 n \tag{2.1}$$

Here, information is transmitted in the bit form. System Capacity, C, refers to the maximum amount of information transmitted per second and is called bits per second.

$$C = \frac{H}{T} = (\frac{1}{\lambda})\log_2 n \tag{2.2}$$

The probability of occurrence of any combination of characters or any single character is P, where $P =$ number of occurrences of the sequence of characters or character/ sum of the probabilities. If the sequences are all equally probable and there are n combinations of sequences, $P = \frac{1}{n}$. Information contained by each sequence in time (H_1) is:

$$H_1 = \log_2 n = -\log_2 P \;\; bits \tag{2.3}$$

$$\tag{2.4}$$

For t time periods;

$$H = tH_1 = -t\log_2 P \ \ bits\ in\ t\ periods \tag{2.5}$$

$$t = \frac{T}{\lambda} \ \ where\ the\ interval\ is\ \lambda. \tag{2.6}$$

$$H = -\frac{T}{\lambda}\log_2 P = (\frac{T}{\lambda})\log_2 n \ \ bits\ in\ T \tag{2.7}$$

2.2.1 Huffman Coding

Huffman coding is a statistical encoding data compression technique. The coding can be performed on any kind of data. Among all other statistical coding techniques, this method is the most advantageous as it reduces the number of bits to a large extent as it makes use of the shortest length codes. However, it requires time to build the statistical tables needed to establish the one-to-one mapping of the original code and the Huffman code [5].

It has a unique property of being self-delimiting. Despite the fact that it is a variable length code, it does not require any commas or delimiters to specify where one code ends and where the next begins. This is because of its prefix property. No short code group duplicates as the beginning of the longer code group.

This code is implemented by using a tree structure called the binary tree [1]. The tree is traversed such that when a zero is encountered, it traverses the left branch of the tree and when a one is encountered, it traverses the right branch of the tree. An example of the binary tree is given in Figure 2.1

The moment a leaf node is encountered, the value in the value field of the leaf node is read and the tree is traversed from the root node over again until the next leaf node is encountered.

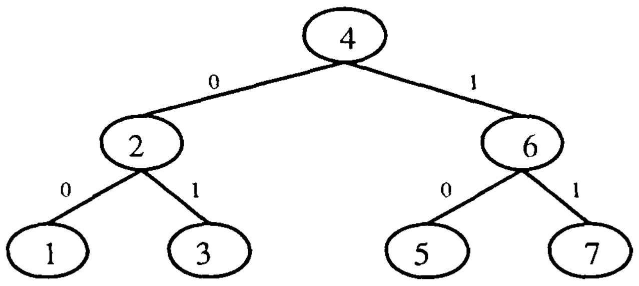

Figure 2.1: BINARY TREE REPRESENTATION

Hence this technique is very useful and efficient. It is used in applications where a lossless transmission is desired. As there is no ambiguity about the coded sequence and since every sequence has a unique leaf node, the transmission is safest in terms of recovering the data at the receiving end exactly in the same way as it was sent from the transmitting end. There is no error in the decoded message and it is very reliable. The binary input sequence decides the path traversed down the tree.

The disadvantage of using this code is that as the number of various sequences in the input sequence increases the length of the unique Huffman codes associated with each of the sequences increases very rapidly.

In case the input data is such that each of the various sequences occurs with equal probability, the coding technique does not prove to be most efficient as there may be an overall gain in the number of bits in the coded sequence. This defeats the purpose of data compression.

The number of bits required to encode a letter using the Huffman technique is determined from the following formula:

$$b = f(\log_2 P) \tag{2.8}$$

where P = probability that the particular sequence or character occurs, and $f(x)$=

the smallest integer greater than or equal to x.

To implement this code, there is a need for *a priori* information of the frequency distribution of each of the sequences of characters or single characters.

2.2.2 Adaptive Huffman Coding

Adaptive coding helps achieve a higher-order modeling at no cost of increasing the statistics. This is done by adjusting the Huffman tree on the fly, on the basis of the earlier data and having no information on the future data sequence. Only the nature of the data earlier to the present instant of observation is used to make changes to the Huffman tree. Adaptive coding can be visualized as consisting of two algorithms, the initialization and the update.

Both the compressor and de-compressor initially have identical models to encode and decode. Thus, when the compressor writes the first encoded symbol or sequence, the de-compressor is capable of interpreting it. Just after writing the first symbol, the compressor updates its original model. This allows the process to be adaptive. The character just encoded is observed and the frequency and the corresponding encoded data sequence is updated.

In the method of modifying the Huffman tree to consider the occurrence of any new characters, the *sibling property* is used. Every internal and leaf node in the Huffman tree is assigned a weight. Each of the nodes has a sibling when the nodes are listed in weights of ascending order. Binary trees comply with the *sibling property.* The node is assigned numbers from left to right beginning with the row of nodes with the maximum depth and moving up. It is the *sibling property* that gives the information on the changes that need to be incorporated to the Huffman tree while updating the counts. Updating involves two steps:

First the count of the leaf node is incremented, and then the tree is traversed upward toward the parent node. The weight of the parent is the sum of the weights of the child nodes, therefore incrementing its weight by one sets it to the right weight. The average number of increment operations to be performed for the weights is dependent upon the number of bits that encode a symbol. Second, if the sibling property is violated on account of the increment; that is, if the node that is incremented has the same weight as the next higher node in the list, there will be no Huffman tree. Then the incremented node is swapped with the preceding node whose weight is less than or equal to the weight of the current node until the node that precedes the node is of higher weight.

In the case of an adaptive process, where the symbols that occur in the message are unknown, one way of encoding is to have each of the possible 256 bytes entered in the table with a count of one. During encoding each message is a byte long. But this amounts to a large waste of coding capacity because for short messages, the extra unused codes negate the compression effect. Instead, an empty table is used at the beginning and symbols are added only after they are encountered. But the first time, there is no symbol in the table, and in order to circumvent this difficulty, an escape symbol is sent that indicates that the current state needs to be "escaped", that is, preceded by a special character not used as data. The decoder automatically knows that the next symbol is encoded in a different context.

The counter in the compressor is incremented with every compression. The count may at some point exceed the capacity of the counters in the tree. This will result in an overflow. To counteract this, the Fibonacci sequence is used. The weight at the root node of a Huffman tree equals $fib(i)$, which implies that the longest code possible for that tree is $i-1$. Once the value is reached, the counters are rescaled by dividing by some fixed number (usually two). Division may result in fractional numbers. But as integers are considered, the fractional part is truncated, causing imbalances.

The scaling of the tree periodically results in a certain amount of loss in the accuracy. On the other hand, the advantage is that higher compression ratios can be gotten as there is a constant updating of the sequences or characters that are coded. Hence, the latest statistics of data is considered making the situation more realistic. The overall effect is a high compression ratio.

2.2.3 Static Huffman Coding

This coding assumes probabilities prior to the coding process. The probabilities remain the same and do not change at any time during the coding process. Hence the name Static has been given to it. The codes of the Huffman tree are the values stored in the leaf nodes of the tree. These are prefix codes. In effect, the Huffman encoding and decoding is nothing but the prefix encoding and decoding. It is very easy and straightforward to implement. Since the codes are a constant based on the pre-defined probabilities, there is a fixed Huffman tree structure that is maintained throughout the coding process. It cannot be altered any time during computation. This may not be the most optimal coding technique in case of a completely random data sequence. The probability that any particular sequence occurs is totally random, hence to assign a particular probability upon a part of the sequence may not yield the best results. A constant monitoring of the probabilities of occurrence of various sequences is to be maintained. This coding technique gives best performance when the input data is not changing very drastically and the ratio of probabilities of each of the sequences is maintained within a particular range.

2.2.4 Dynamic Huffman Coding

If there is no *a priori* information available of the data to be compressed, the static encoding technique cannot be applied. In such circumstances, the input data is completely processed once to get the probabilities of occurrences of the entire set of sequences or characters. After assigning the suitable codes based on the statistics derived after processing the data, the actual encoding process begins.

At first, all the sequences or characters are considered to be equi-probable. Each of the sequences or characters has a counter associated with it. On every occurrence, the associated counter is incremented until the complete data has been scanned. Every time the encoder transmits a code word for a particular input pattern, a new tree is constructed based on the probabilities of the patterns so far encountered. In the same way, the decoder constructs a new tree every time it decodes a particular pattern. Thus identical trees are maintained both at the decoder and encoder end in lock-step. This approach is best when the length of the input stream approaches infinity. But, if $| S |$ is large, the method is inefficient as the tree needs to be modified every time a character in the input stream is encountered.

2.2.5 Higher Order Huffman Codes

Here the Huffman codes are based on the statistics of higher order. The trees are constructed by considering k-tuples of elements or patterns as a new alphabet whose size is $| S |^k$. Each block of k-elements from the input set is assigned a Huffman code on the basis of the statistics of k-tuples. The probability of an element occurring is calculated on the basis of the probabilities of the $k-1$ elements that precede it. This is referred to as the k^{th} order statistics. The Huffman codes are constructed for $| S |^{k-1}$ separate static or dynamic Huffman trees.

The disadvantage in adopting the above method is the fact that the data structure requires a space of the order of $\Theta(| S |^k)$. For example, when $k = 2$ is considered, the storage size results in 2^{16} multiplied by the constant. In order to overcome the space limitation, only those parts of the data structure that are used most frequently are retained.

To implement a second-order probability static Huffman code, $| S |$ first-order codes are considered. To reduce the space for an integer $m > 1$, the $m-1$ most frequent codes are retained. For patterns that are not covered by the $m-1$ codes, the pattern's code word is transmitted by appending a raw bit code. This results in a space requirement of $\Theta(| S | m)$. The space reduced is proportional to the value of $| S |$.

2.2.6 Multi-Group Huffman Coding

This technique helps in cases where there is more than one source that outputs the patterns to be coded. This may result in a series of letters, followed by a series of numbers and this may in turn be followed by a series of spaces. Thus, the data are of different types. Each of the types is coded as a separate binary tree. The code to be decoded is searched in the binary tree. If the binary tree contains that code, then its equivalent code is output. If the binary tree does not contain that particular code, then the fault value of the binary tree is output. This is an indication that the particular code does not exist in the binary tree and a different tree needs to be searched. Every binary tree has an extra value associated with it called the fault value. This is output in case where the particular code searched does not exist in the tree [5].

CHAPTER 3. nCUBE 2 PROCESSOR

3.1 Introduction

The nCUBE processor is a series parallel processor that can support a hypercube array of 2^{13} or 8192 individual processors. Every node is comprised of an nCUBE 2 processor having memory ranging from 1 to 64 Megabytes.

The block diagram of the nCUBE processor is shown in Figure 3.1 .The processor can address memory ranging from 1 to 64 Megabytes per node [12].

The communication channels make use of hardware cut-through routing technique to communicate without being initiated by the CPU. When a channel is available or when an error occurs during data communication, interrupt signals provide the relevant indication. The external interrupt facility aids the nCUBE 2 processor to issue interrupts or be interrupted by other processors.

3.2 Communication Methods

To be scalable to a large number of processing nodes and to be able to support multiple levels and forms of parallelism, it is required that parallel machines like nCube be multicomputer architectures having networking support and very minimal internode communication latencies. The internode communication latencies are the yard-stick in determining the performance of a parallel machine. Hence, parallel

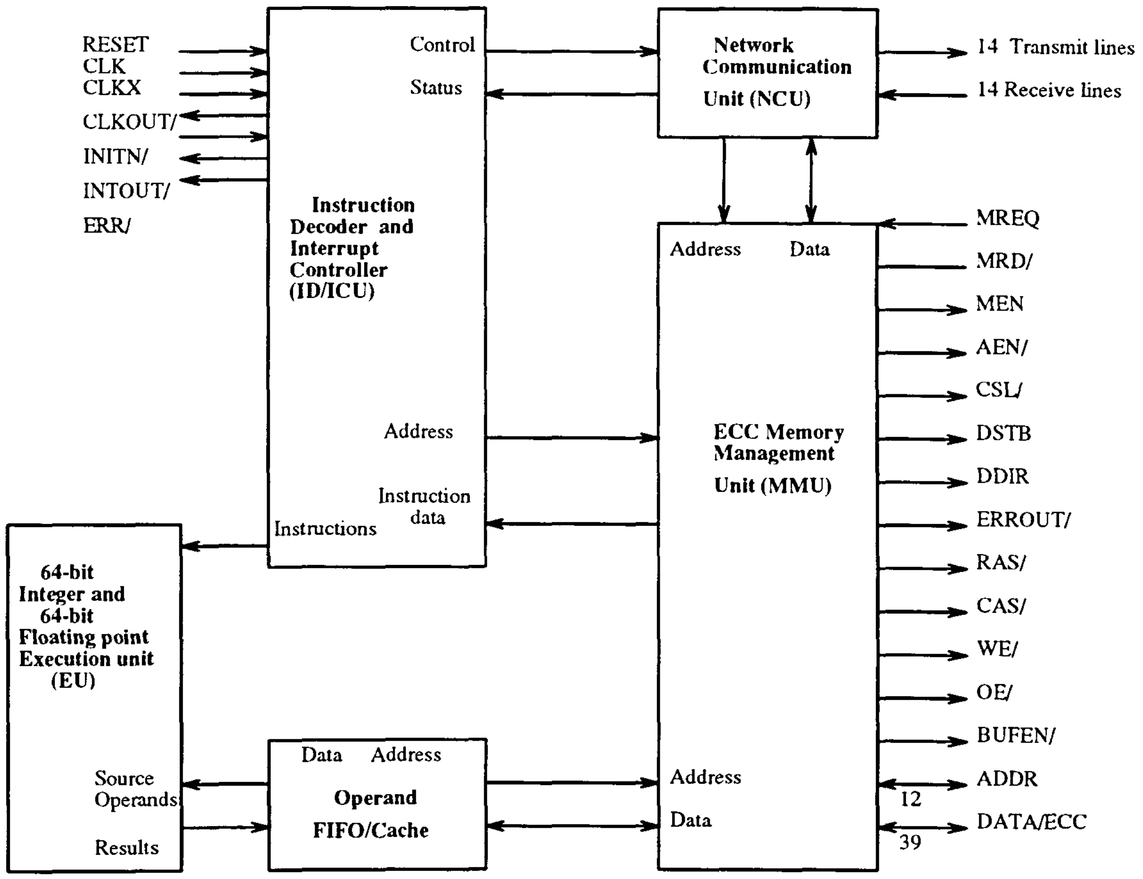

Figure 3.1: nCUBE 2 PROCESSOR BLOCK DIAGRAM

applications are best in situations where the message is routed through fewer number of nodes. The ratio of *computation time* to that of *communication time* is the performance metrics of an algorithm.

In the nCUBE 2 processor, the memory is distributed. Messages are communicated as information explicitly between processors. Typically, on the nCUBE-1 the message takes 0.35 milliseconds to start and continues at 2 microseconds per byte.

The communication overhead is a significant portion of the parallel execution time, it therefore requires that the inter-processor communication time be reduced.

The inter-processor time is made of two components, the *message startup time* and the actual DMA *transfer time.* The hardware of the communication channels of every node can operate simultaneously with the processor itself and with each other until the memory bus bandwidth gets saturated. The communication time can be reduced to a large extent by data reorganization and computation. DMA channels are controlled by the software running on the processor. The overhead added formed by the software reduces the parallelism in communication. The *message startup time* is predominant in the parallel overhead.

There are different sets of inter-node communication methods [11]. The set of inter-node communication method employed distinguishes one multicomputer from another. There are five aspects that make the distinction between the sets of inter-node communication.

(1) Communicating nodes

A node can directly communicate with many other nodes which could be a restricted set of nodes or an arbitrary node in the system.

(2) Connection setup

A connection is defined as the physical routing path on the network that assists one or more communications required by the application during the process of program execution.

(3) Routing path selection

There could be more than one physical routing path connecting any two nodes in the system. The routing path can be either deterministic or adaptive as desired. In the deterministic routing, the route is decided by the source and destination address of the connection. In adaptive routing, the route is decided by the source node that initiates the communication or by the network to avoid congested nodes.

(4) Network flow control

A message may not be forwarded to the next node if the next node is busy. Network flow control is responsible for providing methods that help avoid network queue overflows and underflows in the event that messages get blocked.

- Naive

The message transmission is initiated only after the connection has been completely established and the acknowledgment from the destination node has been received.

- Store-and-forward

The entire packet is stored in the system memory of the intermediate node before sending it forward to a selected neighboring node when it is not busy. This method is simple to implement since only two nodes are considered at any given time. The drawback with the method is that, it requires a lot of memory and bandwidth of the intermediate nodes resulting in increased communication delays.

- Virtual cut-through

Immediately upon receiving the header of the message the, intermediate node directs the packet to the next node if the selected output channel is free. The packet is buffered only if the output channel is busy.

- Wormhole

This routing technique preserves the blocked message in the network till the output channel is available. The channels along the route of the blocked message are not available for transmission of other messages, but they are available in virtual cut-through routing. It can be seen that the Wormhole routing method completely avoids the overhead of buffering packets in the system memories of intermediate nodes. The message is serialized into a sequence of parallel data units called flow control digits (flits). *Flits* are smallest units of information that a queue accepts or rejects. The node notes the contents of the header flit(s) of the message and decides the next routing channel. It then forwards the message down the channel.

There is a possibility that a flits of the same message may spread over several nodes during the communication process. The communication consists of a set of queues one per channel. The queue of the input channel is connected to the output channel and establish a connection over which message is routed in a pipeline manner till the complete message is passed. Connections can be broken and new ones established. As there is nor routing information in all flits of the message, it is required that they remain in adjacent channels of the network and are not interrupted with flits of other messages.

- Network Latency

The latency in Wormhole routing is the same as in the case of virtual cut-through.

$$T_f D + L/B = (L_f/B)D + L/B$$

T_f is the delay of individual routing nodes on the path, D the number of nodes on the path and L/B the time required for message of length L to traverse through the channels of bandwidth B and L_f the length of every flit.

$L_f << L$ implies that the distance D does not affect the network latency. The network latency depends on the message length L and the number of communication channels traversed, D.

Wormhole routing has low latency and uses small amount of dedicated buffers at every node. This is why it is employed in the nCUBE-2 processor.

(5) Buffering at end nodes

In certain communication strategies, messages are buffered in the system memory at either the sending or the receiving end or both ends of the node. Some of the schemes are enumerated.

Packets are communicated through system buffers. While sending the packets, they are copied to the system buffer prior to sending similarly, they are copied to the user space before being read. This is station-to-station communication.

Packets are directly communicated between user spaces. It is called door-to-door communication. The latency involved is less than the case of station-to-station communication.

Application programs communicate individual data items as and when they are computed called program-to program or systolic communication. The latency involved here is minimum since the communication is effected immediately upon computation. The disadvantage of using this method is the need for synchronization of the sending and receiving programs.

In the nCUBE 2 processor, the communicating nodes are chosen arbitrarily, the connection setup is dynamic circuit -SW, the routing path selection is deterministic, network flow control uses Wormhole technique and the buffering technique at the end nodes is station-to-station type.

The communication in nCUBE is initiated in a user program by making calls to the nCUBE communication library.

Message is sent using *nwrite()* and received using *nread()*. The parameters used in referencing *nwrite()* and *nread()* are processed by the library call for error and validity. The library call executes a trap instruction that directs the program to execute the corresponding trap handling routine in the Vertex operating system. The *psend()* trap routine buffers the message, creates a header with the information available from the user and the system. The channel through which the message is to routed is then determined and the message is put in the queue for the corresponding channel. The receive trap routine is examined for valid messages in the "received messages" queue for that process and copies in into the specified buffer in the parameter of the call. The sending and receiving operations are asynchronous and are performed by the input and output message ready interrupts for that channel. The channels are established asynchronously and independently of the CPU.

When the DMA has sent everything, Output message ready interrupt is generated. The CPU then is informed that the DMA channel is ready for transmitting more messages. The interrupt service routine handling the output message ready interrupt ensures that all the queued messages are sent through the channel.

At the end of transmission, the input message ready interrupt for that channel is generated indicating to the CPU that the DMA channel is ready for receiving messages. The interrupt handler initializes the DMA input channel to receive the next header packet.

Messages are considered to be in two parts: the header message packet and the data message packet. The first 32 bits of the header is the destination address and destination mask. The rest of the header message packet and data message packet is encapsulated by hardware as data packets have 32 bits of data information in each packet. The address packet is routed along the channels determined as follows: Each processor compares the masked node address with its own processor ID and forwards the packet along the port that differs in the address by the first bit position. The message is sent on the port with a higher number than on which it was received.

After a communication path is setup, it remains until the EOT packet clears the path for other messages. The Vertex operating system sends an EOM packet after every message packet has been sent and finally sends an EOT packet after the transmission.

3.3 Communication

Communication between processors is through messages. Message Passing is executed in three steps. The first is Path creation, the second is Data Transmission, and finally Path removal. Each of the three steps is executed by every processor in the path that the message takes to reach the given destination from the given source.

Cut-through routing is used for routing data between nodes. This hastens the message passing operation. There is no storage of the message in the memories of the nodes in the routing path and the software is executed at the initiating and receiving nodes (Source and Destination nodes).

The Network Communication Unit is made of three layers: the interconnect layer, routing layer and the message layer.

Interconnect layer is vital in giving the necessary hardware for creating the physical communication links. The distinct features of the interconnect layer are enumerated:

Message layer provides services for reliable and efficient point-to-point data transfer between processors. The features of the layer are as enumerated : Interrupt driven communications reduces the load on the CPU.
DMA capability provides concurrent message transmission independent of the CPU activity.
Message transmission without path removal provides transmission of non-contiguous data.
Automatic error detection facilitates target node to respond to errors without interrupting the intermediate CPUs.

Message is a chain of packets relayed from one processor to another processor. The first packet contains the header information or the address that the routing layer notes to decide which port or ports to use in creating the next link or links in the path. Even the routing layer makes note of the address to decide if the CPU needs to be notified upon arrival of the message. It is possible to establish complex tree-structured and multi-node paths by broadcasting and forwarding techniques.

The channels that the address packet or the header packet traverses on moving from processor to processor are reserved for data packets that succeed the header packet. The network communication unit can buffer two packets on every incoming channel. As soon as space is available, the network communication unit requests another packet until the end of transmission (EOT) packet is received.

The transmission errors that are detected by the intermediate processors are encoded in the packets during their traversal and they are detected by the destination processor.

CHAPTER 4. HIGH-LEVEL SYNTHESIS FOR HARDWARE DESIGN IN VLSI

In applications where very complicated electronic designs need to be implemented, gate level description becomes extremely tedious and verbose. In order to circumvent this difficulty, hardware description languages are used in place of the gate level schematic. Logic synthesis tools do the gate level design.

So far, two Hardware Description Languages have been developed: Verilog and VHDL. The Verilog hardware description language is very similar to C programming and is easy to implement. The language realizes the design process in two stages. At first, it provides the behavioral level of abstraction. In the second stage, it provides the final design from the structural level of abstraction. In complex descriptions, the language makes use of the concept of hierarchy.

- Synthesis

Synthesis is the translation of behavioral description into structural description. Every component in the Structural description is defined by a unique behavioral description. The component structure is a low level synthesis. Synthesis has an additional level of description which gives the required information for the next level of synthesis.

The various kinds of synthesis are enumerated [10]: They are Gate, Register-Transfer and Logic.

The Behavioral description takes into account a known set of inputs and outputs and outputs and a set of functions that explain the behavior of each of the outputs with respect to the inputs. It provides an interface description and the constraints that are imposed on the design. The input/output ports and timing relationships or protocols of signals at the ports are given in the interface description.

The behavioral modeling finds extensive application in the initial stage of the design. The main objective is to derive an accurate simulation that meets the intended behavior requirements. The performance of the system is measured even before the final implementation.

After the behavioral modeling has been made, structural models with accurate detail of the final implementation are designed and simulated for proving the functional correctness. Thus the behavioral modeling is the first step in the process of synthesizing structural design of that behavior. There can be more than one structural design that realize the same behavior.

At the high-level design or behavioral synthesis, an algorithm is described to synthesize large systems while at the lower level design or structural synthesis, register-transfer level behavior of a circuit specifying clock edges and preconditions for each of the system's register transfers is described.

4.1 Design Modeling

Design description is a tradeoff between behavioral and structural descriptions. Design can be visualized as a single process or as being broken down into a number of smaller processes. Each of the small components is described by a process. The difference between behavioral and structural descriptions is that the operations in behavioral description are written in the order of their occurrence in time, whereas, in structural description, they are grouped together according to the place of execution.

4.2 Design Quality Measure

It is necessary to have estimates during the early synthesis stage when the final quality is still not known. The estimates are required to meet the accuracy, fidelity and simplicity specified. Error is the difference of the estimated value and the actual real quality value. Fidelity is the deviation from the average error over all design points. When the error is of same magnitude at all the design points, the fidelity is said to be high. Estimates having high fidelity and low accuracy efficiently predict the superiority of one design over the other.

The high-level estimates must predict silicon area, wire delays and performance based on the transistor and wire layout.

4.3 High-Level Synthesis Algorithms

Synthesis is a transformation of a behavioral description into a set of connected storage and functional units. The algorithms most commonly used in high-level synthesis are partitioning, scheduling and allocation. Partitioning is used to divide a behavioral description or design structure into sub-descriptions or sub-structures to simplify the problem and meet the constraints of size of the chip, number of pins on the package. power dissipation or maximal wire length. Scheduling and allocation are a part of the partitioning problem. Scheduling algorithms partition the variable assignments and operations into time intervals and allocation partitions them into storage and functional units (hardware resources).

Scheduling and allocation are related. If scheduling is made to precede allocation, extra constraints are imposed on scheduling operation. As an example, if two operations cannot be performed by the same functional unit in the same time interval. Once the operations have been assigned to different time intervals and functional units, algorithms for optimizing control, storage and functional units are required. Control unit synthesis is sequential synthesis and that of functional units is called logic synthesis. Storage units are synthesized from sequential and combinational synthesis methods.

4.3.1 Databases and Environments

There are two types of databases for high-level synthesis. One is a component database where Register transfer components required in the high-level synthesis are stored. It provides an interface to the tools and gives an account of the component instances on three abstraction levels: Register transfer, logic and layout.

At the Register transfer level, estimates on component types and delay area of each component are given to the high-level synthesis tools to implement partitioning, scheduling and allocation.

At the logic-level, behavioral and structural descriptions of each component is given on the layout level design. The height and width of the layout and the input/ output port locations at the boundary area are given for floor planning. The Aspect ratio for floor planning and area delay ratio for scheduling are additional information to be furnished. Specific inquires are handled during scheduling and datapath allocation [6].

The step that follows Structural description is behavioral description for each component for simulation and structural description in terms of standard cells for custom and semi-custom design.

System database at high-level retains all information with regard to hierarchy of communicating processes or finite state machines and testing, manufacturing, documentation and management of the whole design.

The Central database keeps track of all the specifications and requirements of design and provides different design views when necessary. the information is initially derived from the original behavioral description and changes are made to it by a graphical user interface.

The best performance interactive synthesis is required that facilitates:

a) interactive mapping of behavior to structure

b) generation of quality measures or estimates and

c) verification of changes in behavioral and structural descriptions.

4.3.2 Design Description Languages

The design is initially done with flowcharts, state graphs, timing diagrams and block structures. In Capture and Simulate methodology, design information is added till a complete register-transfer(RT) design information is derived. The original specification is not always maintained. As there is not formalized description, simulation is difficult as the complete design is not maintained explicitly but is in designer's mind. The basic order that is adopted in the design process is illustrated in the form a flow chart in Figure 4.1.

Hardware description languages are used for varied applications (digital signal

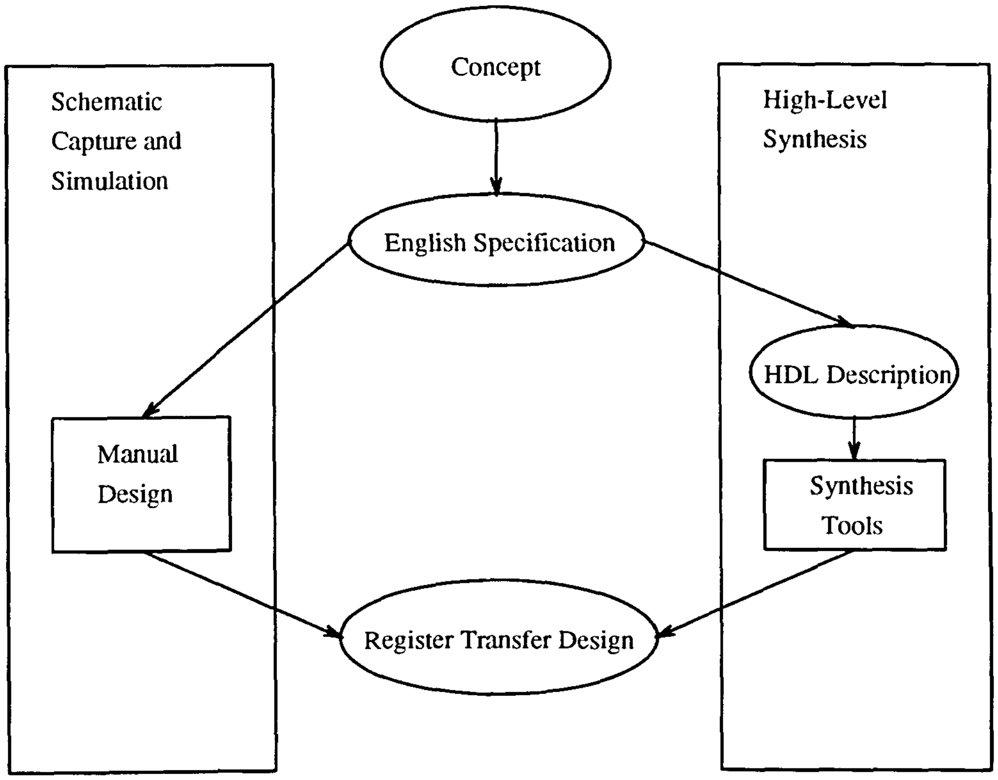

Figure 4.1: DESIGN METHODOLOGY

processing or real time control), different design aspects (simulate and synthesis) and target architecture (FSM and systems). Digital signal processing application is visualized as signal flow graphs, bus interface as timing diagrams. Description for simulation uses the concept of simulation time and the description for synthesis requires hardware description language constructs. Finite-state machine-based design requires the concept of states to be built in.

4.4 Hardware Description Language Formats

Different formats facilitate design description at various levels. At lower levels such as RT, logic and layout, graphical and other modes of design description and capture are used. Behavioral HDLs use textual form. A mixture of different modes is used in the design specification. Protocol and interface information is obtained from timing waveforms. Small state machines are described by graphical or tabular state diagrams and pure behavior is described in textual language format.

4.4.1 Textual HDL

Finds best application in capturing behavior containing assignment statements with complex data transformations. Examples of textual HDL are VHDL and Hardware C.

4.4.2 Graphical HDL

This format works most effectively in applications of execution, ordering, parallelism and control flow. Behavioral flow chart is a good example of a graphical capture mechanism, control flow is expressed graphically but operational behavior is best elucidated by textual assignments.

4.4.3 Tabular HDL

These are best suited for state based design descriptions where state based part of the design is in a state table and the datapath operations in textual form assigned to appropriate parts in the state table.

4.4.4 Waveform Based HDL

This format expresses interface behavior protocols and associated timing constraints. Timing diagrams represent changes on signals and give the sequence of events and the timing relationships between one another.

4.4.5 Simulation based HDL

This activates an event simulator that simulates a physical hardware behavior over the simulation time. VHDL derives hardware behavior from logic gate level to complex architectures in the design. At the highest level, it is a conceptualization of an interconnected network of entities. Each entity has an interface description called entity declaration and an architectural body. At the bottom of the hierarchical decomposition, the architectural body is a process block, data flow block or structural block. These are the three basic description facilities in VHDL.

- *Behavioral Domain*

System architecture is a set of concurrently executing processes and data flow blocks. Statements inside a process block are executed sequentially. State levels inside a data flow block are executed concurrently.

- *Structural Domain*

VHDL structural block has components, component instances, interconnection nets and design inter-connectivity as a mapping of component instance ports to interconnection nets, the design is made hierarchically.

- *Temporal Domain*

Describes VHDL time as discrete simulation time. the behavioral VHDL executes the behavior instantaneously at simulation time but schedules the outputs as transactions in future simulation time in accordance with the delays defined in the behavior. The concurrent processes in VHDL may cause a signal to receive many values at a particular instance of time, in such an event, a VHDL resolution function assigns a unique value to the signal for that instant of time.

4.5 HDL Compilation

VHDL is the input and the CDFG (control data flow graph) representation is the output. In the CDFG, HDL control flow constructs such as loop and conditionals are mapped to control flow nodes and blocks of assignment statements between control flow constructs are represented as data flow graphs.

The HDL parser generates parse trees by performing a one to one mapping of individual language statements to the parse trees. The parse trees describe the behavior of sequential assignment statements in VHDL, the statement list shows the execution ordering of parse trees. It is expressed in terms of the semantics os the input HDL description style. If the HDL semantics require that statements be concurrently executed, the parse trees are studied to make sure that all expressions on the right hand side of the assignment statements are concurrently executed before assigning values to the left hand side variables. The parse trees are intact except for the accesses to common right hand side source variables that are merged.

The dataflow analysis is completed by combining together all the parse trees into a single dataflow graph maintaining any data dependencies that may exist between different statements.

4.6 Olympus, High-Level Synthesis System

Olympus has certain unique features as a high-level synthesis tool.

- Vertically Integrated System

It is common to build a tool that is application specific to high-level synthesis, logic synthesis or circuit synthesis that synthesizes a particular level in the behavioral domain to a corresponding horizontal level in the structural domain of the conventional Y-chart. Olympus provides vertical synthesis. In vertical synthesis, it is possible to traverse the Y-chart vertically from high-level synthesis to logic-level synthesis and does the technology mapping and hence Olympus is a complete synthesis tool.

- Modularity

Enhancements can be added to the system and advancements made at a particular synthesis level. This makes Olympus very versatile and adaptive.

- Open Systems

Olympus can be easily interfaced with other design tools. The tools in Olympus communicate through machine and human readable formats.

- Applications Range

It is suited for application specific (ASIC) design and is not restricted to digital signal processing applications.

- Feedback and Simulation

An exhaustive feedback at both HLS and logic-level simulation is provided by Olympus. The feature supports design verification and modification.

4.6.1 Main Feature of Olympus

- Input/Output Specification

A synthesis-oriented hardware description language called Hardware C is used in specifying hardware for synthesis and output in the form of gate netlist is obtained.

Hardware C $\Rightarrow$ Olympus $\Rightarrow$ Gate netlist

- Constraint-driven Structural Synthesis

Timing and resource constraints defined at the high-level description are utilized to obtain the desired implementation. A synthesis of partially bound hardware descriptions is followed by the synthesis of the remaining operations.

- Not Library Dependent

Logic synthesis combines the datapath and control synthesis. high-level synthesis does not require the knowledge of library of well-defined resources. Each resources is representative of the modes synthesized and characterized by logic synthesis.

- ASIC Design

The timings at the logic-level are made use of at the high-level synthesis where architectural trade-offs and scheduling are done, this is an useful feature for ASIC designs. Logic synthesis ensures the correctness of high level synthesis.

- Technology Mapping

Technology mapping algorithm targets different design goals from performance to area.

- Automatic and User Driven Synthesis

The Hardware C can be easily transformed to the logic-level and supports driven synthesis. The user can intervene to enforce decisions based on design trade-offs.

4.6.2 Overall Description of Olympus Block Diagram

Olympus performs three major tasks:

- High-Level Synthesis

Translates the hardware behavioral description to register transfer level.

- Logic-Level Synthesis

Optimizes the area and performance of the logic representation.

- Technology Mapping

Maps the logic representation onto a predefined library of functional units.

The tasks are split between various blocks. The input to the system is Hardware C represented by the first block of the diagram. The function of each block is explained briefly. High-level synthesis in Olympus is performed by Hercules and Hebe. Hercules does the front end parsing on the Hardware C and generates an implementation independent description in a graph based representation called Sequencing

Intermediate Format or SIF. A simulator called Aridane simulates the SIF to provide the feed back on functional correctness. Hebe takes the SIF as the input and maps it into the logic-level implementation described by the structural logical intermediate format. Mercury, a logic synthesis tool supports transformation from multilevel logic optimization and logic-level simulation. Ceres performs technology mapping in which the logic description is changed into a purely structural representation defined in terms of cell instances of a predefined library for semi-custom implementation.

4.6.3 Hercules: Behavioral Synthesis

The objective of behavioral synthesis is to identify the maximum parallelism that exists in the input description. The result is an indication of the latest design that the system can produce assuming that a dedicated hardware component implements each operation in the design implementation. In Olympus, this is performed by Hercules.

First, Hercules parses the Hardware C description input and translates it into an abstract syntax-tree representation. This provides the underlying model for semantic analysis and behavioral transformations as shown in Figure 4.2.

• In selective line expansion of model calls, a call to a model is replaced by the functionality of the called model.

• In the selective operator-to-library mapping, operators such as $+$ or $\geq$ in the input description are mapped into calls to specific library template models.

• In automatic transformations the following optimizations are done

- Unrolling for Loops

Fixed iteration loops are unrolled to provide more opportunities for optimization.

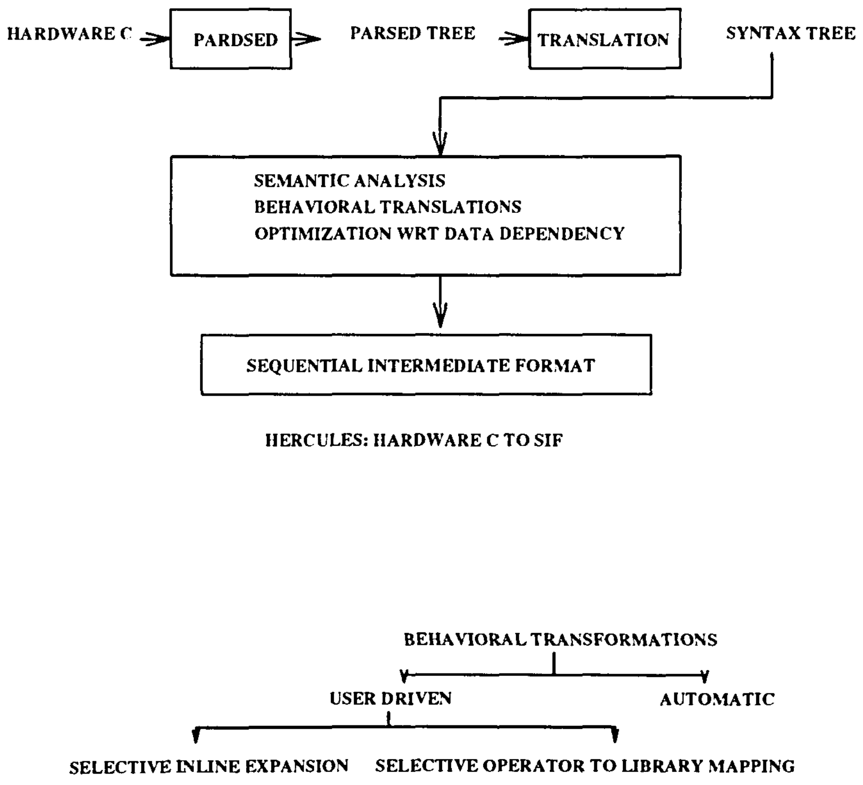
HARDWARE C
PARDSED
PARSED TREE
TRANSLATION
SYNTAX TREE
SEMANTIC ANALYSIS
BEHAVIORAL TRANSLATIONS
OPTIMIZATION WRT DATA DEPENDENCY
SEQUENTIAL INTERMEDIATE FORMAT
HERCULES: HARDWARE C TO SIF
BEHAVIORAL TRANSFORMATIONS
USER DRIVEN
AUTOMATIC
SELECTIVE INLINE EXPANSION
SELECTIVE OPERATOR TO LIBRARY MAPPING

Figure 4.2: HERCULES BEHAVIORAL SYNTHESIS

- Propagating Constants and Variables

The reference to a variable is replaced by its last assigned value.

- Resolving Reference Stacks

Multiple and conditional assignments to variables are resolved by creating multiplexed values that can be referenced and assigned.

- Eliminating Common Subexpressions

Redundant operations producing the same result are removed.

- Eliminating Dead Code

Operations whose effects are not visible outside the model are removed.

- Collapsing Conditionals

Conditionals with branches having only combinational logic are collapsed to provide more opportunity to apply logic synthesis.

- Analyzing Dataflow

Data and Control dependencies among operations are identified.

Thus Hercules outputs the SIF derived from the input Hardware C description. This SIF acts as the input to the next structural synthesis tool - Hebe.

CHAPTER 5. IMPLEMENTATION METHODOLOGY

The purpose of the thesis is to achieve a speed up in the application of message passing. Here the message is a sequence of IEEE 64 bit floating point data. The Huffman coding technique has been tested using both Hardware and Software approaches. The results of both the approaches have been provided in the form of a table. It can be seen from the table that the hardware approach gives better performance than the software approach.

The input sequence for which the Huffman coding is applied, is the IEEE floating point data that comprises of numbers *0, 1, 2, 3, 4, 5, 6, 7, 8, 9* and characters . and *E*. The . is a decimal point that separates the whole number from the fractional part and *E* is the exponent character.

The time required for creating the binary tree can be perceived as the set up time. The actual coding and transmission takes much less time than the set up time. Hence, the longer the message is, the better is the performance that can be achieved. This is because, the set up time becomes a small fraction of the total time when transmitting very long sequences of data.

A very high compression has been realized using Huffman coding. In the worst case, a reduction of 3 bits has been achieved. This is because the longest Huffman code is 5 bits in length. In the normal sequence of execution, each of the input character sequence is represented as an 8 bit ASCII value. A best case of 6 bits reduction has been achieved in the data. Thus, on an average a reduction of approximately 50 % has been achieved. This results in an improvement in the overall speedup by a factor of two.

The input character set and the equivalent Huffman code sequence are as shown in the table that follows:

Table 5.1: HUFFMAN CODE EQUIVALENTS OF THE ASCII CHARACTERS

ASCII character	Huffman code
E	10
.	11
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	00000
7	00001
8	00010
9	00011

The IEEE floating point numbers are converted into their ASCII equivalents and the coding is performed on the ASCII characters. There is a slight loss in data during the conversion to ASCII characters and hence lossy techniques could also be considered.

It consists of Three parts: Encoding, Transmission and Decoding operations.

5.1 Encoding

- Algorithm

(1) Provide the input sequence that has to be communicated. This is the IEEE 64 bit floating point data.

(2) Calculate the probability of occurrence of each of the twelve input characters that are to be coded.

(3) The highest probable character is assigned the shortest length Huffman code. As the probability of occurrence of a character diminishes, the length of the Huffman code it is assigned to increases. Eventually, the lowest probable sequence is assigned to the longest Huffman code.

(4) The process is repeated until all the characters have been encoded with their respective Huffman codes.

(5) The encoding process is stopped.

5.2 Transmission

The encoded message is then transmitted from one processor to another as the case may be. In this context, the message was communicated from one processor node to another in the nCUBE processor.

5.3 Decoding

- Algorithm

(1) The Huffman code is visualized as a binary tree. The left branch of the tree originating from any node is a binary *zero* and the right branch of the tree is a binary *one*. The binary tree for the Huffman code adopted is as shown in Figure 5.1.

(2) Based on the encoded sequence that arrives at the destination (Huffman code sequence), the binary tree is traversed. The right branch is traversed with the occurrence of a binary *one* and the left branch with the occurrence of a binary *zero*. The Algorithm for Binary Search is explained .

(3) Upon arriving at the leaf node of the binary tree, the value stored in the value field of the binary tree is output. It is the IEEE ASCII character.

(4) The decoding is done till the all the bit sequences in the encoded sequence have been substituted with their respective ASCII characters.

(5) The decoding is stopped and the sequence of original data is recovered.

5.3.1 Structure of the Binary Tree

Every node of the tree has a left and a right branch except the *leaf nodes*. Each node has four fields associated with it, the *Value* field, the *Label* field, the *Left pointer* and the *Right pointer*.

The node at the top-most level in the tree is called the *Root node*, the nodes that are internal to the tree are called the *Internal or Intermediate Nodes*. The nodes

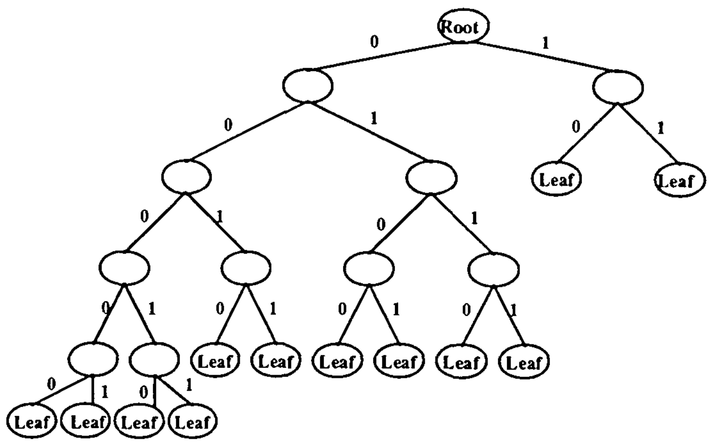

Figure 5.1: HUFFMAN TREE STRUCTURE

that have no nodes to their left or right are called the *Leaf nodes*.

The Tree has five levels. At the first and the third levels there are no leaf nodes, at the second there are two, they are represented as binary sequences 10 and 11. The fourth level has six leaf nodes, that are arrived at traversing the tree by binary sequences 0111, 0110, 0101, 0100, 0011 and 0010. The fifth level has four leaf nodes that are got by binary sequences 00011, 00010, 00001 and 00000.

5.3.2 Create Binary Tree

- Algorithm

(1) All the twelve Huffman codes that are used to represent each of the twelve characters that comprise the IEEE floating point number is provided as an input sequence.

(2) Since the size of each of the codes is different and varies with bit lengths of 2, 4 and 5, the size information of each of the codes is provided in a different string.

(3) The Huffman sequence is read character by character. At first, a zero is read. The node that is created is labeled the *Root node.*

(4) The value field of the node contains a *null.*

(5) The left and right pointers of the root node are initially set to *null.*

(6) The node is labeled with a number that identifies it uniquely at all times.

(7) Every time a character is read, a counter is incremented. The number of characters that have been read is compared with the size of the code.

(8) On the occurrence of a zero, the left pointer of the root node is incremented by one and the current pointer is set to the new node location.

(9) The left pointer is incremented and the current pointer is made to point to the address of the updated left pointer.

(10) The new node is created and a unique label is assigned to it.

(11) Upon the occurrence of a one, the right pointer is incremented and the current pointer is made to point to the address contained by the updated right pointer.

(12) The new node is created and a unique label is assigned to it.

(13) The counter value is compared with the size of the code and if they are equal, the value field of the node is assigned the original ASCII character it represents is copied into its value field.

(14) The current pointer is again positioned to point the root node.

(15) If the counter value is less than the size of the code, the node is an intermediate code and its value field is made to contain a *null.*

(16) Steps 7 onwards are repeated till the end of the Huffman code sequence is reached.

5.3.3 Binary Tree Search

- Algorithm

(1) The encoded sequence is read one bit at a time.

(2) If the bit is a binary *zero*, the left pointer is incremented by one and the current pointer is made to point to the updated address of the left pointer.

(3) If the node it is pointing to is a leaf node, the value in its value field is output.

(4) If the bit is a binary *one*, the right pointer is incremented by one and the current pointer is made to point to the updated address of the right pointer.

(5) If the node it is pointing to is a leaf node, the value in its value field is output.

(6) The Steps 1 onwards are repeated till the end of the encoded sequence.

In the Software approach, the code is written in C and is tested on the nCUBE processor. The graphs of time as a function of the length of the message is provided.

In the Hardware approach, the code is written in Verilog Hardware Description Language. The graph of time as a function of the length of the message is provided.

CHAPTER 6. RESULTS AND CONCLUSION

It can be seen from the tables that the software approach has a higher set up time. The process of creating a binary tree and searching it, can be perceived as the set up time.

The actual message transfer time is in the order of milliseconds. The percentage increase in time with respect to the time required to transfer 500 K bytes is calculated. This percent increase in time with increase in the message length that is transferred is very small.

There is a trade off between the two approaches, the software yields better performance at the expense of a higher setup time whereas, the hardware performs better except that there is a higher percent increase in time with increase in message length.

The reason for the long latency times in case of the hardware can be attributed to the fact that the the search performed at the decoding end does not employ the binary search tree method. On the other hand a comparison is done on the basis of the length of the code in bits. As the Huffman code is made up of different sizes, 2 bits, 4 bits and 5 bits in this case, a search is done considering 2 bits first and comparing it with the existing codes. If the code is not found in the Huffman coded sequence then, 4 bits are considered at a time and a similar search done. If it is still not found code 5 bits are considered and a final search done.

In the worst case scenario this would entail eleven searches before the actual code is found and in the best case the very first search would provide the corresponding code.

Table 6.1: RESULTS OF COMPRESSION USING SOFTWARE SIMULATIONS ON THE NCUBE

#	Message Length in K bytes	Time in μ seconds	% Increase in Time
1	500	867237	0
2	1000	868728	.17
3	1500	872324	.58
4	2000	874079	.78
5	2457	877149	1.13
6	2500	877587	1.18
7	3000	880657	1.53
8	3500	884078	1.92
9	4000	887148	2.27

Graphs of Time verses Message Length for both software and hardware are provided in Figure 6.1 and Figure 6.2. The cross-over points for both the cases are shown. It is the indication of the message length starting at which the compression technique yields better performance as against normal circumstance.

Table 6.2: RESULTS OF COMPRESSION BY HARDWARE SIMULATIONS USING VERILOG HDL

#	Message Length in K bytes	Time in μ seconds	% Increase in Time
1	500	452028	0
2	1000	453235	.26
3	1300	464285	2.64
4	1500	467256	3.28
5	2000	472131	4.33
6	2500	473803	4.69
7	3000	475799	5.12
8	3500	477749	5.54

Table 6.3: RESULTS OF SIMULATIONS WITHOUT COMPRESSION

#	Message Length in K bytes	Time in μ seconds
1	500	178500
2	1000	357000
3	1300	464100
3	1500	535500
4	2000	714000
5	2457	877149
5	2500	892500
6	3000	1071000
7	3500	1249500
8	4000	1428000

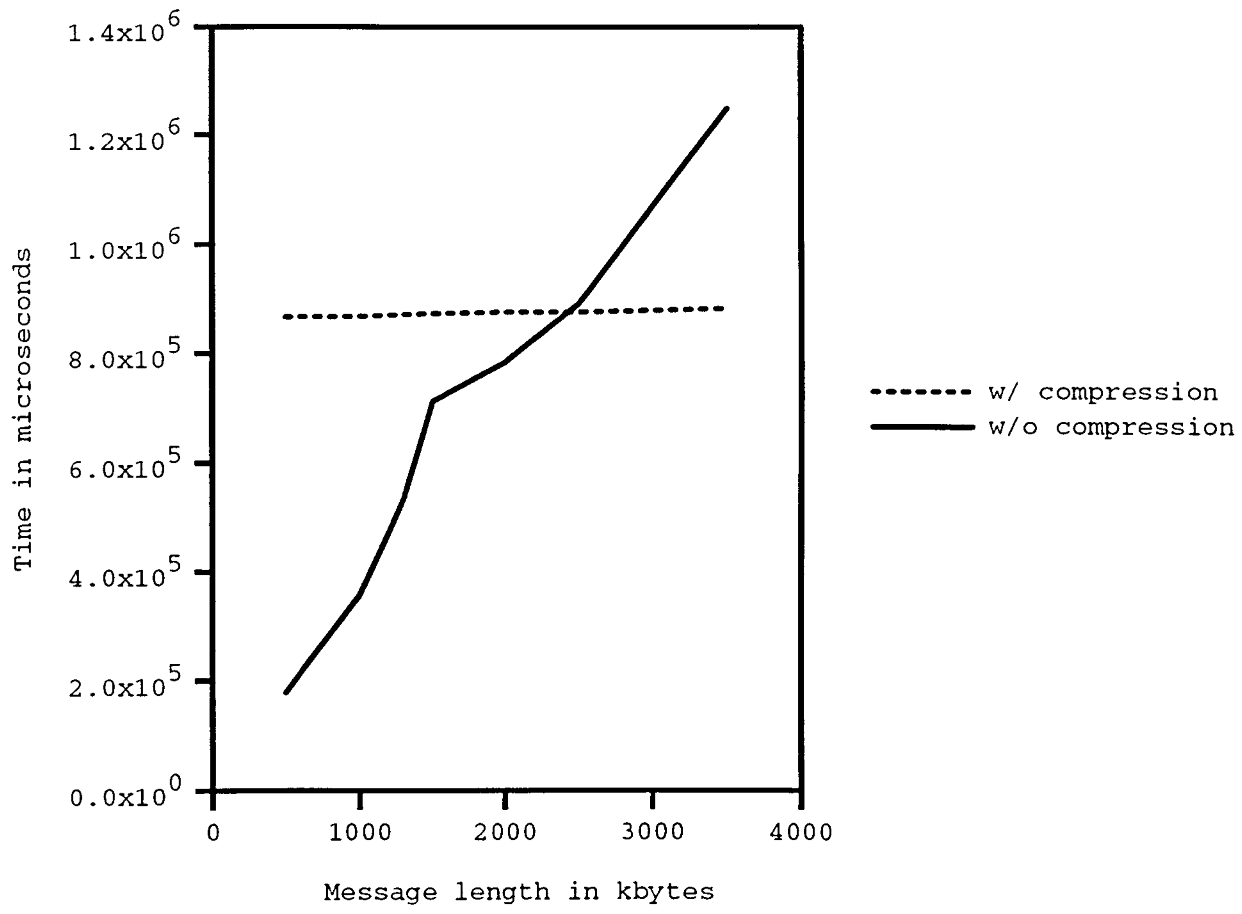

Figure 6.1: FEASIBILITY CROSS-OVER POINT FOR SOFTWARE SIMULATION

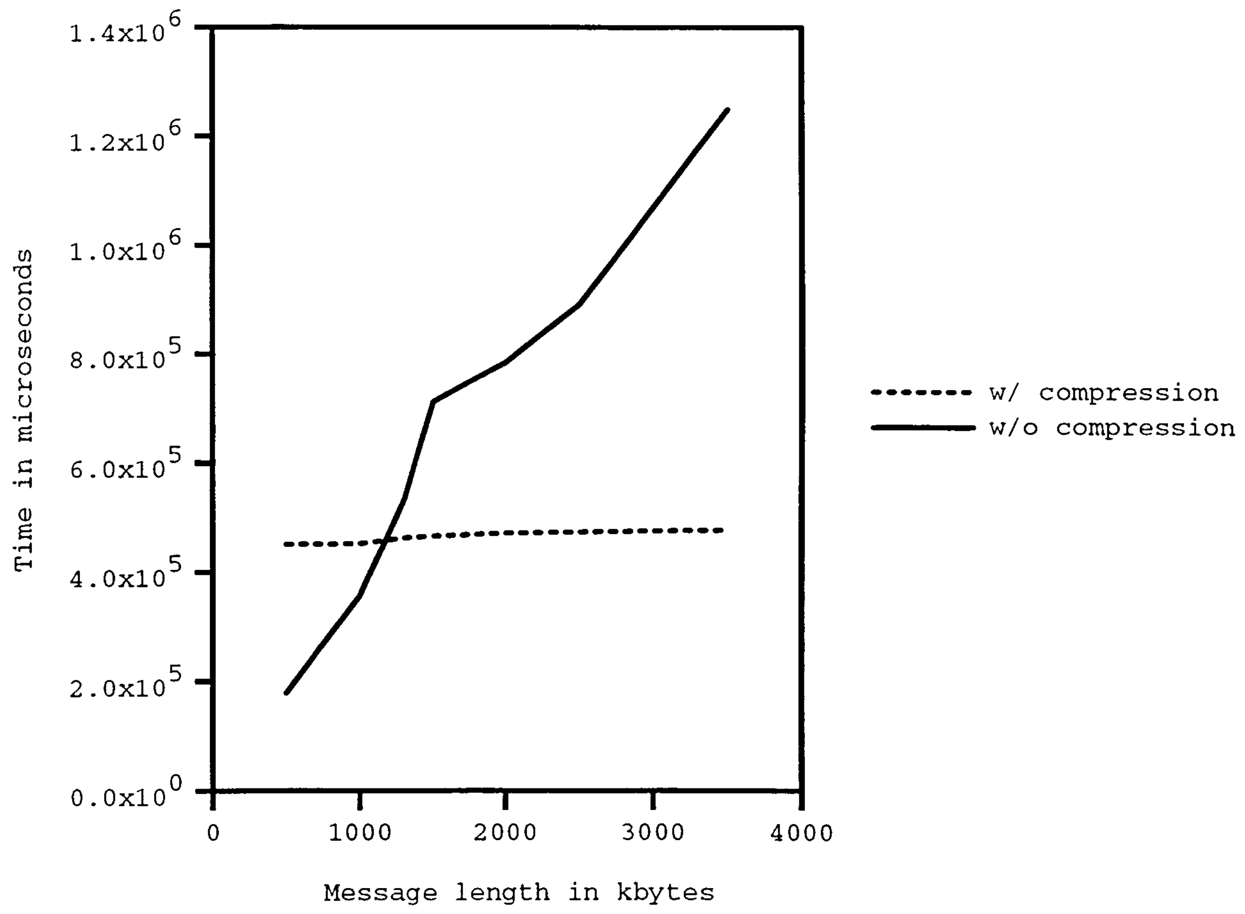

Figure 6.2: FEASIBILITY CROSS-OVER POINT FOR HARDWARE SIMULATION

BIBLIOGRAPHY

[1] T. H. Cormen, C. E. Leiserson and R. L. Rivest, *Introduction to Algorithms.* McGraw-Hill, New York City, NY, 1990.

[2] R. D. Cameron, "Source Encoding Using Syntactic Information Source Models", *IEEE Transactions on Information Theory*, pp 843–850, July 1988.

[3] J. G. Cleary and I. H. Witten, "Data Compression using adaptive coding and partial string matching", *IEEE Transactions on Information Theory,* pp396–402, April 1984.

[4] G. Held, *Data Compression: Techniques and Applications, Hardware and Software Considerations.* John Wiley & Sons, New York City, NY, 1987.

[5] J. A. Storer, *Data Compression: methods and theory.* Computer Science Press, Baltimore, MD, 1988.

[6] C. S. Ying and J. Wong, "An Analytical Approach to Floorplanning for Hierarchical Building Blocks Layout," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp.403–412, April 1989.

[7] B. S. Haroun and M. I. Elmasry, "Architectural Synthesis for DSP Silicon Compilers", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp.431–447, April 1989.

[8] P. Agrawal and W. J. Dally, "A Hardware Logic Simulation Systems", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* pp.19–29, January 1990.

[9] S. Devadas and A. R. Newton, "Algorithms for Hardware Allocation in Data Path Synthesis", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp.768–781, July 1989.

[10] D. E. Thomas and P. Moorby, *The Verilog Hardware Description Language*. Kluwer Academic Publishers, Amsterdam, Netherlands, 1991.

[11] R.Pai, "Development of a low-latency scalar communication routine on message-passing architectures", *Master's Thesis submitted to Iowa State University*, 1993.

[12] NCUBE Corp, *nCUBE 2 Processor Manual*, NCUBE Corp, Foster City, CA, Feb 1992.