FPGA Hardware Acceleration of Phylogenetic Tree Reconstruction with Maximum Parsimony Algorithms

(FPGA を用いた最節約法による進化系統樹構築アルゴリズムの高速化)

２０１８年　３月

Henry Jose Block Saldana

FPGA Hardware Acceleration of Phylogenetic Tree Reconstruction with Maximum Parsimony Algorithms

(FPGA を用いた最節約法による進化系統樹構築アルゴリズムの高速化)

Henry Jose Block Saldana

システム情報工学研究科

筑波大学

２０１８年　３月

Abstract

In this research, we investigate, propose, evaluate and implement an FPGA-hardware approach for molecular phylogenetic tree reconstruction under maximum parsimony. Phylogenetic reconstruction has been investigated for many decades in different fields such as biology and medicine. Several software algorithms and hardware solutions have been proposed for phylogenetics. However, there is still the need to reduce the time required to reach a reliable solution. This research aims to contribute in this regard by proposing a general approach, faster than current software or hardware approaches, that can work for large phylogenetic problems.

We study different software algorithms and methods for accelerating the tree reconstruction and the heuristic search for the best tree. First, we start from a basic approach and gradually improve it. In total, we design and implement five FPGA hardware approaches based on the software algorithms studied. Each approach uses a new idea from a software algorithm or acceleration method, which helps to improve the performance over the other approaches.

We evaluate the implementation of each approach for several real-world biological datasets by comparing the results obtained with those from previous approaches, and with those from the phylogenetic software TNT (Tree analysis using New Technology), known as the fastest available parsimony program. We compare for each dataset the total execution time, the execution time required for evaluating a single tree, and the best score obtained. The datasets used are of medium to large size and each one of them consists of hundreds of sequences, each of them with thousands of DNA characters.

Our fifth and current approach, the fastest of all five, achieves acceleration rates between 2.66 and 31.94 against TNT for the evaluation of a single tree. These acceleration rates achieved are thanks to a combination of using the Indirect Calculation of Tree Lengths method, the Incremental Tree Optimization method, and the parallel and pipeline processing used. In our proposed hardware approach, all the DNA characters in a sequence are processed in parallel.

The main contribution of this work is to present an FPGA hardware approach for phylogenetic tree reconstruction under maximum parsimony that effectively addresses the evaluation of a single tree rearrangement and the stochastic local search. For the first time in the literature, an approach that covers both the complete and incremental first- and second-pass optimization, as well as the tree rearrangement evaluation has been proposed.

Acknowledgements

My greatest gratitude is for my supervisor Professor Tsutomu Maruyama for his valuable guidance and unlimited support through all these 6 years. He gave me the opportunity to be part of his laboratory, where I found the ideal place to pursue the research I desired, and kindly gave me advice whenever I needed. This thesis could not have been possible without him.

I would also like to thank all the members in the Reconfigurable Computing Systems Laboratory for their friendship and incredible support. They always helped me without hesitation whenever I encountered any difficulties.

Finally, I would like to thank all my friends in Tsukuba and around for the great moments we spent together. Thanks to them I felt home since the first day, and motivated to work in my research.

Contents

List of Figures

List of Tables

This work is dedicated to my beloved family, who has constantly encouraged me to pursue my dreams, and who in spite of the long distance separating us has given me their unconditional support since I came to Japan. Thanks for always loving me.

Chapter 1

Introduction

Thanks to rapid advancements in technology during the past years, DNA sequencing has become a more easy, quick and inexpensive process. Today, molecular sequence data obtained from DNA sequencing is used as the most reliable material to study living organisms. As a result, molecular phylogenetics, which studies how living organisms have evolved through the years, is now a growing field with many applications in biology, medicine and bioinformatics.

Many software algorithms and tools have already been developed for phylogenetics. However, molecular phylogenetics involves computational-intensive operations that can difficult obtaining solutions in reasonable amounts of time. For this reason, new software algorithms and hardware approaches are being proposed to ease and accelerate molecular phylogenetic analysis.

In this regard, hardware platforms such as FPGAs offer great potential for accelerating the different computational-intensive operations involved. Therefore, it is necessary to propose efficient hardware approaches that can be implemented on FPGAs. This will contribute to the growing field of phylogenetics and its applications.

1.1 Phylogenetics

Phylogenetics is the field that studies and attempts to reconstruct the evolutionary relationships among biological entities. These entities, which are commonly referred as taxa (singular: taxon), can be species, genomes, genes, regions of a gene, protein sequences, molecule sequences, etc. [1] [2].

1.1.1 Phylogenetic Trees

The evolutionary relationships among the taxa are graphically represented on a binary tree as ancestor-descendant relationships in a branching pattern. The tree starts from a common ancestor, known as the root of the tree, and branches into distinct lineages that descend to the taxa that are being analyzed. An example of a phylogenetic tree for the True Flies (Diptera) is shown in figure 1.1.

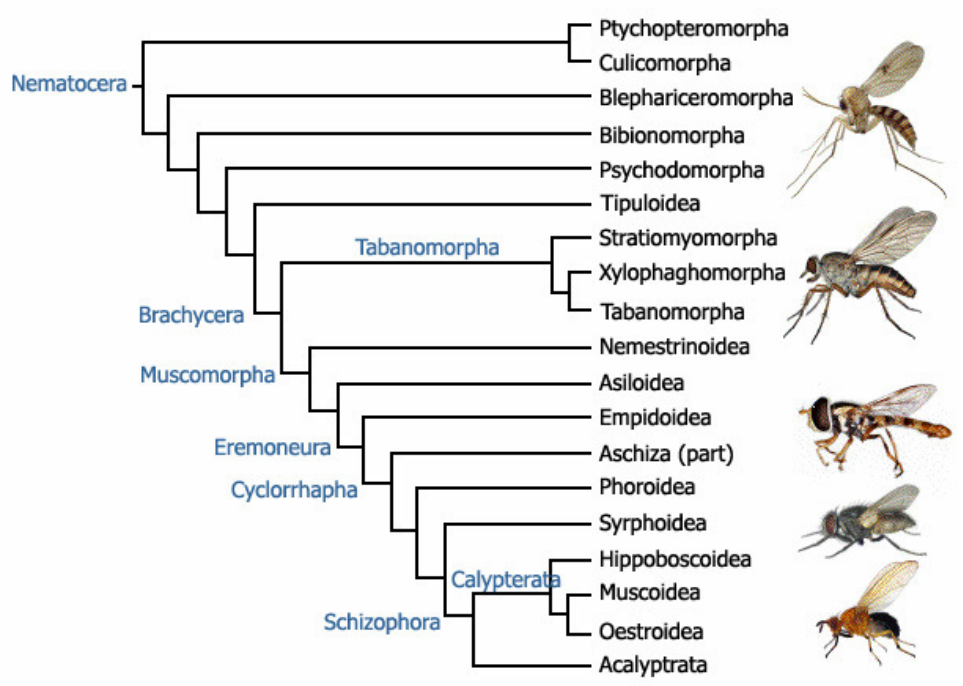


FIGURE 1.1: Phylogeny of True Flies (Diptera) [3]

As can be seen from figure 1.1, the current species of true flies descend from a common ancestor called *Nematocera*. This common ancestor evolved into distinct species of true flies, some of which are intermediate ancestors, others of which are the current species.

1.1.2 Phylogenetic Tree Inference Methods

There are different methods used to infer phylogenetic trees, but they can be classified into three main categories: Parsimony, Distance, and Likelihood methods [4]:

- Parsimony methods are based on the principle of Occam's Razor. This principle states that without any other information, the simplest explanation should be chosen. For phylogenetics, this means that the most likely tree to represent the true history of the taxa is the one that requires the fewest number of evolutionary changes. This is known as the maximum parsimony criterion, and the optimal tree as the most parsimonious tree.

- Distance methods are based on the assumption that it should be possible to infer a phylogenetic tree from the patterns of similarity among organisms. In other words, organisms that share a recent common ancestor should be more similar to each other than organisms whose last common ancestor was more ancient. The optimal tree is generated by using a distance matrix that contains the estimated evolutionary distances between all pairs of sequences.

- Likelihood methods are based on statistics. They resemble parsimony methods in that different trees are compared and given a score. However, the score is not based on the number of evolutionary changes, but on how likely the taxa evolved on the tree given a model of amino acid substitution probabilities. The optimal tree is the one that has the highest probability among all trees.

There is no strict consent on which method is better or preferred, since each has its advantages and disadvantages. However, there are several criteria used to evaluate the methods themselves and the results they produce. For example, two criteria that are always taken into consideration are the efficiency of the method (i.e. how fast it performs) and its consistency (i.e. how reliable it is to generate the correct tree) are [4].

In this research, we are concerned with phylogenetic tree reconstruction of molecular DNA sequence data using the maximum parsimony criterion. We chose this criterion not only because of its widely acceptance and use, but also because of its simplicity when calculating the score of a tree, which makes it more suitable for a hardware implementation on FPGA.

1.1.3 The Maximum Parsimony Criterion

As mentioned in section 1.1.2, the maximum parsimony criterion is based on the assumption that the most likely tree is the one that requires the fewest number of evolutionary changes to explain the given data [4]. Theoretically, this means that all possible trees have to be examined and evaluated [5]. However, this is not possible in practice, because they are too many trees, as shown in figure 1.2.

Taxa	Rooted Trees[a]	Unrooted Trees[b]
3	3	1
4	15	3
5	105	15
6	945	105
7	10,395	945
8	135,135	10,395
9	2,027,025	135,135
10	34,459,425	2,027,025

[a] $N_r = (2n-3) \times (2n-5) \times (2n-7) \times \cdots \times 3 \times 1 = (2n-3)!/[2^{n-2} \times (n-2)!]$.
[b] $N_u = (2n-5) \times (2n-7) \times \cdots \times 3 \times 1 = (2n-5)!/[2^{n-3} \times (n-3)!]$.

FIGURE 1.2: Number of possible trees versus number of taxa [4]

As can be seen from figure 1.2, as the number of taxa increases, the number of possible trees (either rooted or unrooted trees) grows exponentially. For this reason, phylogenetic tree reconstruction using maximum parsimony has been proven to be an NP-complete problem [6] [7].

For the above reason, heuristic methods are used in practice [8]. The aim of these approximate methods is to find a sub-optimal solution without having to evaluate all trees in the search space. Some examples include local search, genetic and memetic algorithms [9]. In this research, we focus on algorithms using local search.

1.1.4 Relevance of Molecular Sequence Data

Even though there are different methods to infer phylogenetic trees, today almost all of them use molecular sequence data [10]. This is mainly because of the following three reasons [11]:

- DNA is the inherited material;
- Genetic material can now be sequenced easily, quickly, inexpensively and reliably;
- Sequences are highly specific and are often rich in information.

1.1.5 Applications of Phylogenetics

The importance of Phylogenetics relies not only in the fact that it allows us to understand how species (molecular sequences in particular) evolved, but also in that it allows us to learn the general principles that enable us to predict how they will change in the future. This is very useful for many applications in areas such as biology, medicine and others. For example, some of the applications listed by The European Bioinformatics Institute (EMBL-EBI) can be summarized as following [12]:

- Classification: Phylogenetics using molecular sequences provides accurate descriptions of patterns of relatedness. Thanks to this, new species can be classified.
- Forensics: Phylogenetics is used to assess DNA evidence presented in court cases. This can clarify situations where someone has committed a crime, when food is contaminated, or where the father of a child is unknown.
- Identifying pathogens: Phylogenetics can be used to learn more about a new pathogen outbreak by finding out about which species the pathogen is related to and the likely source of transmission.

- Conservation: Phylogenetics can help conservation biologists in deciding which species they should try to prevent from becoming extinct.
- Bioinformatics: Many of the algorithms developed for phylogenetics have been used to develop software in other fields.

These applications are illustrated in figure 1.3.

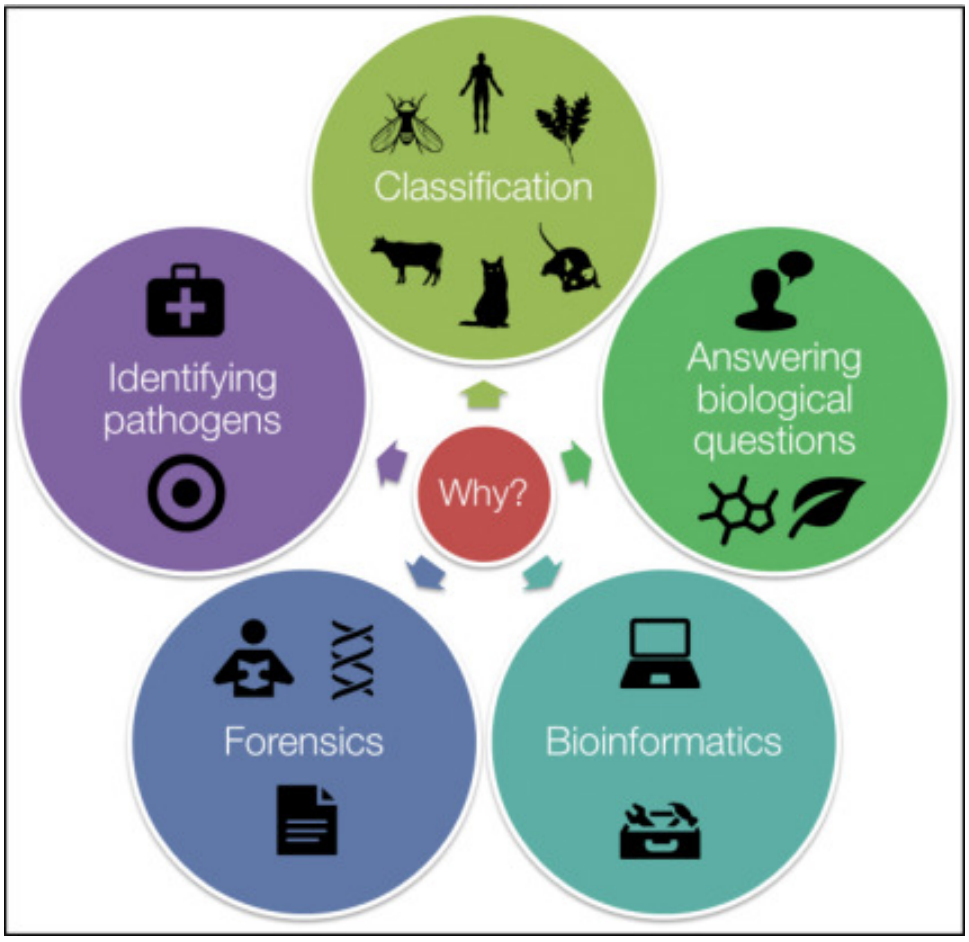


FIGURE 1.3: A summary of some applications of phylogenetics [12]

As can be seen from figure 1.3, the applications of phylogenetics are diverse and of great utility to our society. This makes phylogenetics an important field of study and application that certainly will continue to grow in the near future.

1.2 Background

1.2.1 FPGAs

Field Programmable Gate Arrays (FPGAs) are semiconductor devices that are based around a matrix of configurable logic blocks connected via programmable interconnects. Unlike Application Specific Integrated Circuits (ASICs), FPGAs can be reprogrammed after manufacturing [13]. To better understand the concept of an FPGA, we will use the simplified structure shown in figure 1.4.

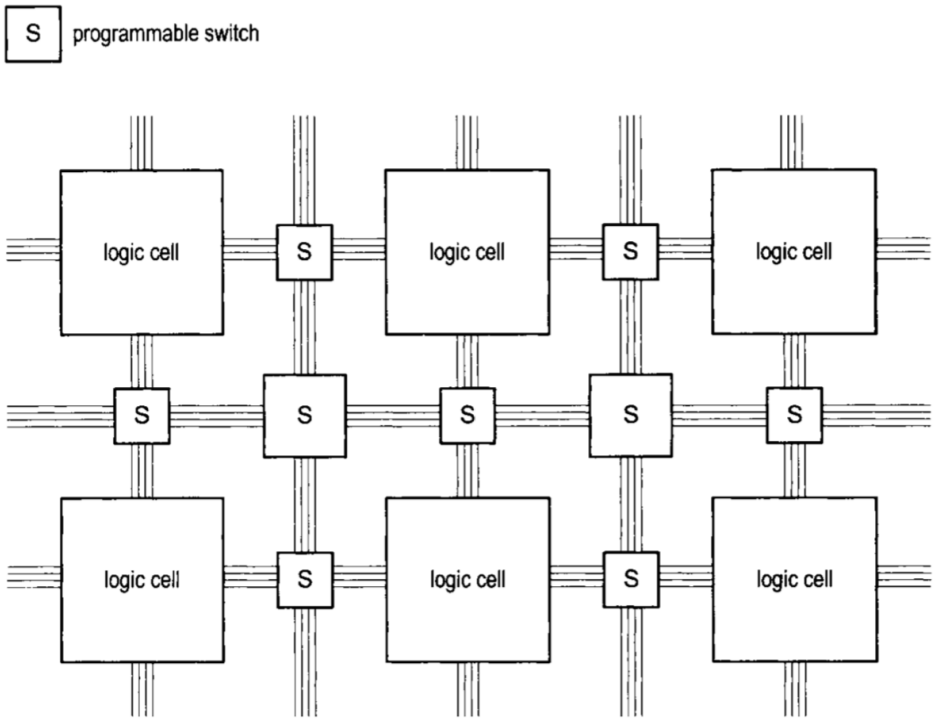


FIGURE 1.4: Conceptual structure of an FPGA [14]

As can be seen in figure 1.4, an FPGA is a two-dimensional array of logic cells and programmable switches. Logic cells can be configured to perform a specific function. They contain a combinational circuit such as a look-up table (LUT) and a sequential circuit such as a D-type flip-flop (D-FF). An n-input LUT can implement any n-input combinational function, and a D-FF can implement a one-bit sequential circuit synchronous to a clock signal [14]. An example of a logic cell is shown in figure 1.5.

In addition, FPGAs usually contain other cells known as macro blocks. These cells provide other complementary functions such as memory blocks, multipliers, clock management circuits, I/O interface circuits and even processor cores. A specific design of a digital circuit or system can then be implemented by specifying the function of each logic cell and macro cell, and connecting them through the programmable switches [14].

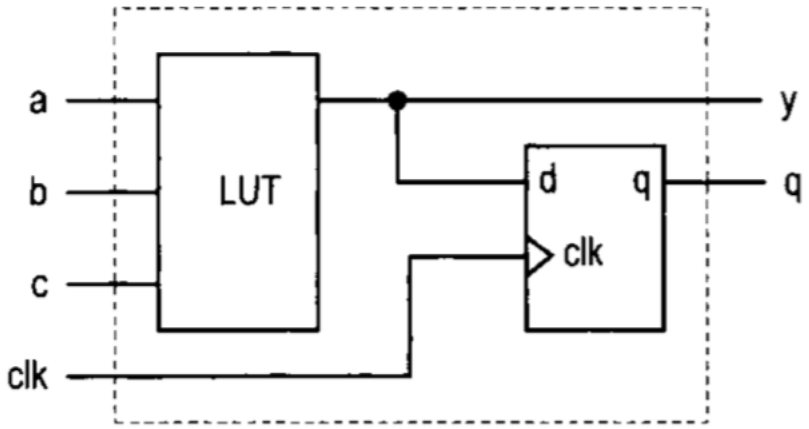


FIGURE 1.5: Conceptual diagram of a logic cell [14]

Because FPGAs are very versatile, they are used in numerous applications. Some examples include the following areas [13]:

- ASIC Prototyping
- High Performance Computing and Data Storage
- Industrial and Consumer Electronics
- Video and Image Processing
- Wired and Wireless Communications

1.2.2 Hardware Acceleration

Besides FPGAs, there are other hardware platforms used for the acceleration of software algorithms and applications. One accelerator that is frequently used like FPGAs are GPUs. Both FPGAs and GPUs can achieve better performance than CPUs on certain tasks. On one hand, FPGAs are highly customizable, while, on the other hand, GPUs provide massive parallel execution resources and high memory bandwidth [15].

In this research, we propose an approach targeted at an FPGA rather than at a GPU or other many-core accelerators, because the algorithm under consideration, as we will see in section X, involves detailed low-level hardware control operations and a lot of memory accesses that are not suit for high-level languages. In addition, GPUs or other many-core accelerators impose a fixed programming model, whereas FPGAs do not; thus, they allow a higher level of customization [15]. This means that we can aim to obtain a higher performance by designing a specific circuit that best suits the phylogenetic tree reconstruction algorithm under consideration.

1.2.3 Software Solutions for Phylogenetics

There are many programs available for phylogenetics, some of which are free to use, others of which are not. Of the extensive list available [16], some frequently-used free programs that include phylogenetic tree reconstruction under maximum parsimony are PHYLIP (Phylogeny Inference Package) [17], PAUP (Phylogenetic Analysis Using Parsimony) [18], MEGA (Molecular Evolution Genetic Analysis) [19] and TNT (Tree analysis using New Technology) [20].

From the above mentioned programs, TNT is known to be the fastest available parsimony program [21] [22]. Since we are interested in assessing how much acceleration our approach can provide with respect to a software implementation, we compare our results with the ones obtained from TNT.

1.2.4 Hardware Solutions for Phylogenetics

There have been several implementions of phylogenetic algorithms in different parallel hardware architectures that include not only FPGAs, but also GPUs, multi-core processors and supercomputer systems [23]. Most of these implementations are for phylogenetic tree reconstruction under maximum likelihood, while only a few have been for the maximum parsimony problem [24]. In this research, we are concerned with FPGA-hardware acceleration of phylogenetic tree reconstruction under maximum parsimony.

From the few FPGA implementations proposed for the maximum parsimony problem, particularly worth mentioning are the ones in [25], [26] and [27].

First, in [25], an approach for whole-genome phylogenetic reconstruction under maximum parsimony was proposed. It covered both the generation of the tree by exploring the search space, and the tree scoring by performing median computations over the internal vertices of the tree. Results reported high acceleration rates against a software implementation. However, given that the method used is not based on heuristics and is highly computational-intensive, the execution time would grow considerably with the number of taxa.

Second, in [26], an approach for phylogenetic analysis under maximum parsimony was proposed. In this case, the approach consisted of a linear systolic array composed of 20 processing elements each of which scores a different tree topology in parallel. Again, results reported high acceleration rates against a software implementation. However, given that the proposed array computes the scores of all theoretically possible trees, the approach is limited to a small number of taxa.

Finally, in [26], an approach for computing the parsimony function on evolutionary trees for any number of taxa was proposed. The approach was compared against a software implementation by the same authors, and results reported acceleration rates not so high.

1.2.5 Problem Statement

Even if heuristics or other approximate methods are used for phylogenetic tree reconstruction under maximum parsimony, a software approach can still require a considerable amount of time for larger phylogenetic problems.

As we saw in section 1.2.4, some FPGA hardware approaches have been proposed to deal with this problem, but they are still limited or not efficient enough to compete with the best software solutions available. In [25] and [26], the proposed approaches are not based on a heuristic method, but on the direct evaluation of all possible tress. As a result, they are limited to a small number of taxa. In [27], the approach is not restricted by the number of taxa, but it only addresses the computation of the parsimony function, and does not involve the search algorithm. In addition, it was only evaluated and compared against a software solution by the same authors.

In general, none of the current hardware approaches provides a solution for phylogenetic tree reconstruction under maximum parsimony that covers any number of taxa and implements the whole search algorithm. In addition, the approaches were not evaluated against the fastest available parsimony program, TNT (Tree analysis using New Technology) [20].

1.3 Purpose of this Research

The main purpose of this research is to propose an approach for the hardware acceleration of a phylogenetic tree reconstruction with maximum parsimony algorithm using an FPGA. The approach has to be faster than current hardware and software approaches. Furthermore, it has to be a general approach that can work for large phylogenetic problems consisting of hundreds of taxa, each of them having a sequence of hundreds or thousands of DNA nucleotides.

To achieve this purpose, we consider the following design requirements. First, the hardware architecture has to be modular. This is desirable not only to ease the design process, but to allow the different modules (units) to work in parallel. Second, the hardware architecture has to maximize the level of parallelism while minimizing the use of logical hardware resources. This means that it is necessary to take into account that tasks, which cannot be executed at the same time (i.e. in parallel), have to share as much as possible the same logical resources. Third, the hardware architecture has to be portable. This is important so it can be easily implemented on any FPGA without having to modify the source code. Last, but not least, all tasks that involve computational-intensive calculations have to work using pipeline processing to maximize the throughput of the hardware architecture. Our design goal is to ultimately obtain a throughput of one node per clock cycle.

Taking into consideration all of the before mentioned requirements, we propose, design and implement five hardware architecture approaches based on the different software algorithms that we explore. We evaluate our implementations by comparing them to the phylogenetic software TNT [20] for several real-world biological datasets, each of them consisting of hundreds of taxa and sequences.

1.4 Thesis Outline

This thesis is organized as follows. In chapter 2, we introduce the algorithms and methods that we researched for phylogenetic tree reconstruction under maximum parsimony. These algorithms are the base for all our hardware approaches that are detailed starting from chapter 3 through the end of chapter 6. In chapter 3, we present our first approach. In chapter 4, we present our second approach. In chapter 5, we present our third approach. In chapter 6, we present our fourth and fith approach. Finally, in chapter 7 we summarize all the results achieved and discuss about them. Following this, in chapter 8, we mention the overall conclusions and talk about the future work.

Parts of this thesis have already been published. Most of the first approach from chapter 3 has been published as a conference paper in [28]. Most of the second approach from chapter 4 has been published as a conference paper in [29] and as a journal paper in [30]. Most of the third approach from chapter 5 has been published as a conference paper in [31]. Finally, most of the fifth approach from section 6.2 of chapter 6 has been published as a conference paper in [32].

Chapter 2

Algorithms for Phylogenetic Tree Reconstruction

2.1 Phylogenetic Tree Reconstruction

2.1.1 Stochastic Local Search

In section 1.1.3, we saw the need for using heuristic methods for phylogenetic tree reconstruction. The idea of these approximate methods is to find a near-optimal solution without having to evaluate all possible trees. The most well-known heuristic methods are based on stochastic local search. Stochastic local search algorithms are among the most successful methods for solving computationally difficult problems in many areas of computer science [33].

Stochastic local search uses iterative improvements to find better solutions. In general, every local search algorithm includes the following four parts [9]:

- A search space composed of the candidate solutions.
- An evaluation function of a solution, also known as the score function.
- A neighborhood function used to obtain a new solution by slightly modifying the current solution.
- A transition strategy to accept or reject a neighboring solution.

In this research, we examine four algorithms that are a variant of a basic stochastic local search algorithm. In simple terms, the algorithm is based on the iterative descent, which is guided by the length of the tree as the score function [34]. The length of the tree is equivalent to the number of evolutionary changes in the tree. It starts from a randomly generated tree in the search space, and tries to improve it on each iteration. A neighbor tree is obtained by rearranging the current tree by cutting a part of it and reinserting it elsewhere on the tree. This is known as Subtree Pruning and Regrafting (SPR). Then, the neighbor tree replaces the current one if it has a lower score (i.e. if the score is better). The algorithm stops when there is no better neighbor tree or when the maximum number of iterations of the algorithm has been reached.

2.1.2 Subtree Pruning and Regrafting (SPR)

Subtree Pruning and Regrafting (SPR) is a rearrangement technique used to obtain a new tree by slightly modifying the current one. It consists of cutting a branch from the tree with a subtree attached to it, and reinserting it into the remaining tree in all possible places. Each reinsertion of the subtree creates a node into a branch of the remaining tree [35]. An example of SPR-rearrangement for an unrooted tree with 11 taxa is shown in figure 2.1.

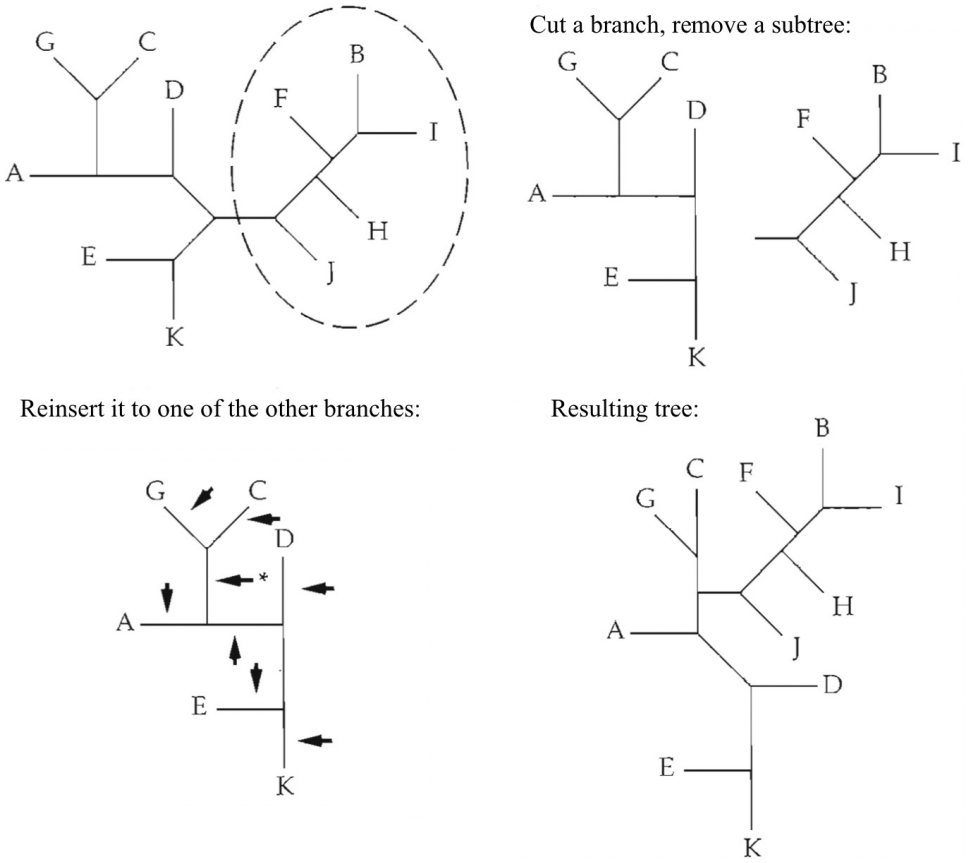


FIGURE 2.1: Example of Subtree Pruning and Regrafting (SPR) [35]

As can be seen from figure 2.1, a 5-taxa subtree is pruned from the original tree, and it is inserted into the remaining tree of 6 taxa, in one of the 9 possible places. If a tree of $n1 + n2$ species has a subtree of $n2$ species pruned from it, there will be $2n1 - 3$ possible places to reinsert it. One of these is the original location [35].

The advantage of SPR is that it can produce drastic changes in the tree topology. The resulting tree can have a greater variation from the starting tree than in other simpler techniques such as Nearest-Neighbor Interchange (NNI) [36], which consists in exchanging two adjacent branches of the tree. This helps in finding the globally optimal or near-optimal solution in the space of all trees [37].

2.1.3 Tree Optimization

The tree optimization process consists of two phases. The first phase is called the first-pass optimization. The second phase is called the second-pass optimization. Both phases described here refer to a complete optimization of all the nodes in the tree.

The first-pass optimization is used to obtain:

- the preliminary character states (DNA sequences) of the nodes,
- the lengths of the nodes, and the total length (score) of the tree.

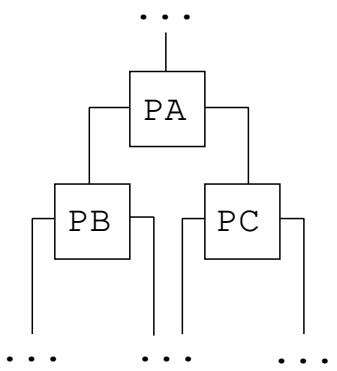
The second-pass optimization is used to obtain:

- the final character states (DNA sequences) of the nodes.

2.1.3.1 First-pass Optimization

The first-pass optimization is the phase where the preliminary character states (the preliminary DNA sequences) of the nodes, along with their lengths (scores) are found. There are two widely used methods that depend on the algorithm applied: Fitch's algorithm [38] and Sankoff's algorithm [39]. Here, we use Fitch's, because it is less complex to implement in hardware.

The first-pass optimization proceeds from the tips of the tree (taxa) by formulating a character state (DNA sequence) for each node in the tree, working backwards, until the character state for the most distant node (root) has been formulated. In addition, each time an evolutionary change (a change in the DNA sequence) takes place in a node, its length is increased by one. A node character state, along with its length, is inferred by using the algorithm in figure 2.2 for each of the characters in the node. This algorithm is based on Fitch's algorithm [38].




```
0. D = 0
1. PA = PB ∩ PC
2. If PA != 0 go to 5
3. PA = PB ∪ PC
4. D = D + 1
5. Store PA
6. LA = LB + LC + D
7. Store LA
```

FIGURE 2.2: First-pass optimization algorithm [40]

PA refers to the preliminary character state of the current node, and *PB* and *PC* to that of its descendant nodes. *D* refers to the score difference between *PB* and *PC*. *LA* refers to the length of node *A*, *LB* to the length of node *B*, and *LC* to the length of node *C*. As can be noted, the length of a node is obtained by summing up the individual lengths of the descendant nodes and the score difference between those nodes. Finally, the length of the tree is equivalent to the length of the root node. An example of the first-pass optimization is shown in figure 2.3 for a tree with 5 taxa. *TL* refers to the length of the tree.

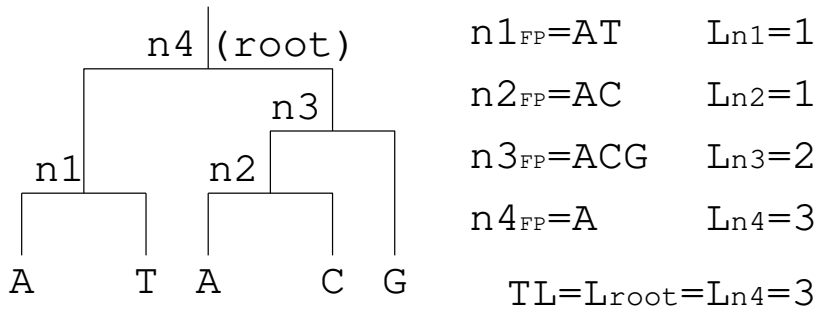


FIGURE 2.3: Example of the first-pass optimization

2.1.3.2 Second-pass Optimization

The second-pass optimization is the phase where the final character states (final DNA sequences) of the nodes are found. This pass proceeds in the reversed order, from the root to the tips of the tree. The final character state of a node is obtained by applying the algorithm in figure 2.4 for each of the characters in the node. This algorithm is based on Fitch's algorithm [38].

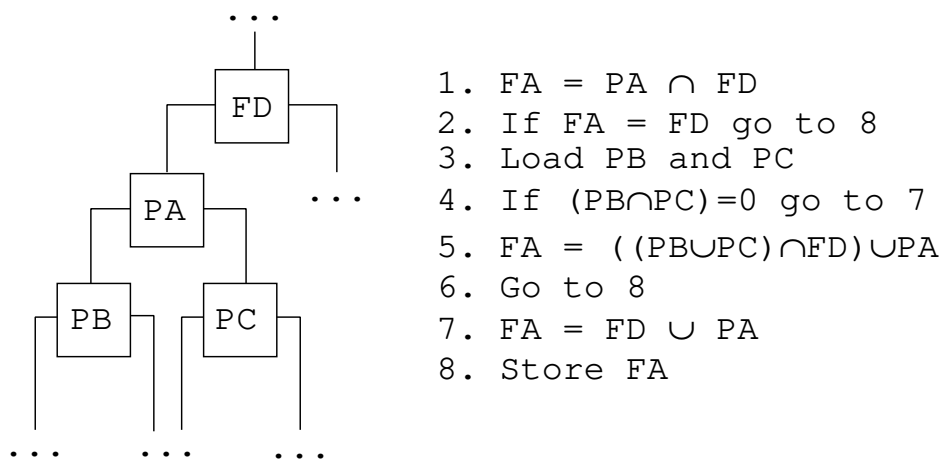


FIGURE 2.4: Second-pass optimization algorithm [40]

FA refers to the final character state of the current node *PA*, *FD* to that of its parent node, and *PB* and *PC* to the preliminary character

states of its descendants. For the root of the tree, *FA* is the same as *PA*, because the root has no parent node. An example of the second-pass optimization is shown in figure 2.5.

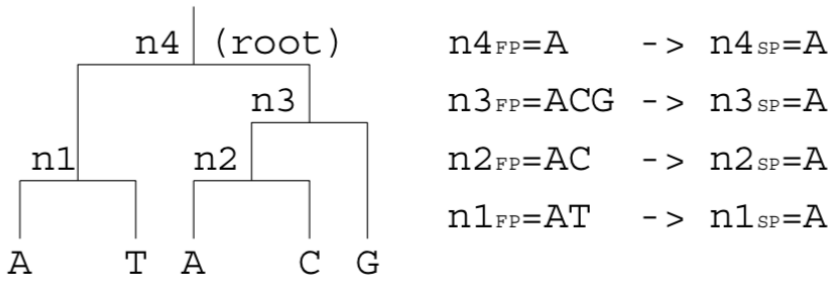


FIGURE 2.5: Example of the second-pass optimization

As can be observed from figure 2.5, the final character states of the nodes are not necessarily the same as the preliminary character states.

2.2 Software Algorithms

In this research, we examine four algorithms for phylogenetic tree reconstruction under maximum parsimony that are used to improve the consistency and the efficiency of the local search. In the following sections, we describe each one of them.

2.2.1 Progressive Tree Neighborhood

The Progressive Tree Neighborhood aims to combine the properties of large and small neighborhoods by changing its size as the search progresses. It starts with a large neighborhood and ends with a small neighborhood. Starting the search with a large neighborhood allows examining more neighbors, and ensures a more global search. Then, as the search progresses the neighborhood gets reduced, so the changes applied to the tree topology are more restricted [41].

A simple progressive tree neighborhood uses Subtree Pruning and Regrafting (SPR) as the large neighborhood and Nearest-Neighbor Interchange (NNI) as the small neighborhood. To change the size of the neighborhood during the search, a distance parameter is used to constrain the distance between the pruned branch and the branch where it is reinserted [41]. The progressive tree neighborhood that uses SPR and NNI can be described by Equations 2.1 and 2.2 [41].

$$\begin{matrix} N_{d_{init}}^{SPR} \equiv N^{SPR} \\ N_{d_{final}}^{SPR} \equiv N^{NNI} \end{matrix} \rightarrow \begin{pmatrix} d_{init} \\ d_{final} \end{pmatrix} = \begin{pmatrix} max\delta(v_i, v_j) \\ 1 \end{pmatrix} \tag{2.1}$$

where v_i is the node at which the branch is pruned, v_j is the node at which the pruned branch is reinserted, and $\delta(v_i, v_j)$ is the distance between these two nodes. It is given by

$$d = d_{init}(1 - \frac{i}{M}), \qquad i < M \tag{2.2}$$

where i is the i_{th} local search iteration and M is the maximum number of local search iterations. The distance parameter starts at d_{init} and ends at a value close to 1.

An example of how the distance parameter of the progressive tree neighborhood is used when a branch of the tree is pruned is given in figure 2.6. The distance parameter is equal to 3.

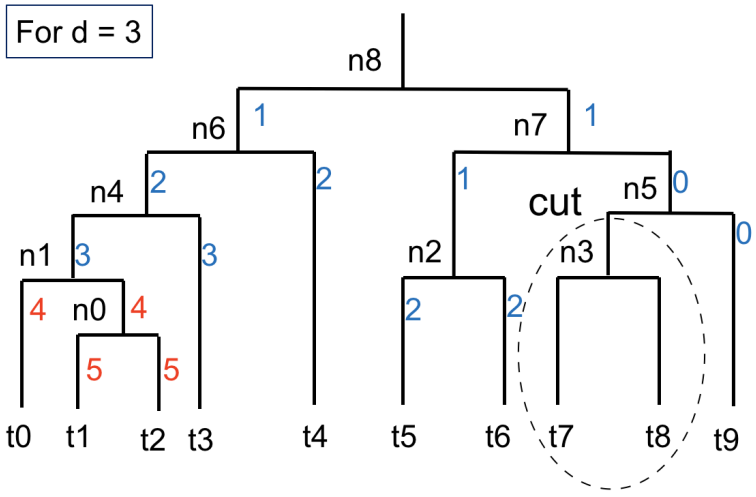


FIGURE 2.6: Example of the progressive tree neighborhood

Figure 2.6 shows a tree with 10 taxa and 9 internal nodes. In this example, the left branch of node $n5$, i.e. $n3$, is pruned from the tree, as indicated by the dashed circle in figure 2.6. Then, all branches in the remaining tree are given a value that corresponds to the distance between the respective branch and the pruned branch. In other words, branches that are further away from the pruned branch have a higher value than those which are closer. Finally, only those branches that have a value equal or lower than the current value of the distance parameter are chosen as candidates for reinserting the subtree in the remaining tree. In this example, the distance parameter has a current value of 3, so branches with a value of 4 or higher are considered to be out of the neighborhood, and; thus, not chosen.

2.2.2 Indirect Calculation of Tree Lengths

The Indirect Calculation of Tree Lengths (ICTL) is a method used to evaluate all possible neighbor tree rearrangements after a branch has been pruned from the tree during the Subtree Pruning and Regrafting step [42]. The ICTL method allows to evaluate all rearrangements without having to actually reinsert the subtree in the remaining tree. Thus, there is no need to do a first-pass optimization on every single rearrangement to know the score of it. According to this method, the score of a neighbor tree rearrangement can easily be calculated by following Equation 2.3.

$$S_{nt} = S_{mt} + S_{st} + D.Score \tag{2.3}$$

where S_{nt} is the score of the neighbor tree rearrangement being evaluated, S_{mt} is that of the main tree (i.e. the remaining tree), S_{st} is that of the subtree, and $D.Score$ is that of the difference between the subtree's root and the insertion branch in the main tree.

Equation 2.3 means that for each rearrangement the only value that has to be calculated independently is $D.Score$, since S_{mt} and S_{st} are common to all rearrangements.

Then, the difference score $D.Score$ can be calculated as shown in figure 2.7, where *NZ* is the final character state of the root of the subtree, *NX* is the final character state of a node in the main tree, and *NY* is that of its descendant. In other words, *NX* and *NY* form the branch where *NZ* could be reinserted. This process is done for each of the characters in the DNA sequence, and then the individual differences are sum up together to get the overall difference score.

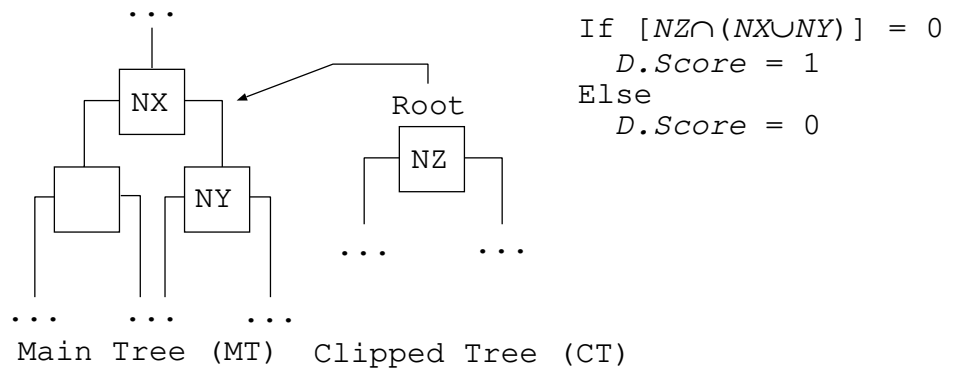


FIGURE 2.7: Difference Score Calculation [42]

To better understand the idea of how the Indirect Calculation of Tree Lengths method is used, an example for evaluating all neighbor tree rearrangements for a subtree with 3 taxa and a main tree with 7 taxa is shown in figure 2.8.

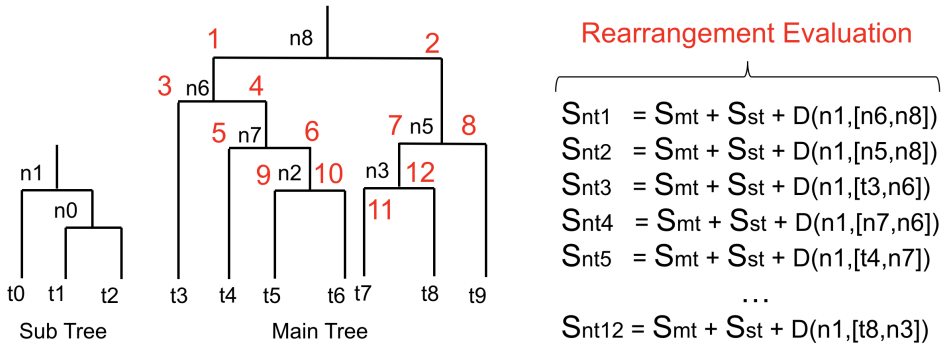


FIGURE 2.8: Example of the rearrangement evaluation process

Figure 2.8 shows a main tree and a subtree resulting from pruning a branch from the original tree with the subtree attached to it. The main tree has 12 possible branches where the subtree can be reinserted. For each of these branches, the score of the neighbor tree rearrangement for that branch is calculated by using Equation 2.3, and the $D.Score$ is calculated as indicated in figure 2.8. Finally, once the score of all possible neighbor tree rearrangements has been calculated, the branch that produces the rearrangement with the lowest score can be chosen as the candidate branch for reinserting the subtree into the main tree.

The ICTL method provides a fast technique to evaluate all tree rearrangements. In general, a first-pass optimization for a tree with T taxa requires visiting each one of the $T - 1$ internal nodes. Thus, the time required increases with T. On the other hand, the ICTL method allows to evaluate all rearrangements that can be constructed after pruning the tree without having to visit each one of them. The time required is approximately $1/T$, which does not increase with T. In addition, this method is exact, i.e. it will always produce the right score. However, it requires a second-pass optimization on top of the first-pass optimization every time the tree is pruned, because the final character states of all nodes are required [42].

2.2.3 Alternative Second-pass Optimization

In section 2.2.2 we mentioned that the Indirect Calculation of Tree Lengths (ICTL) required a second-pass optimization on top of the first-pass optimization, because the final character states of all the nodes are required to calculate the difference score $D.Score$ for a neighbor tree rearrangement. Furthermore, as we saw in section 2.1.3.2, doing a second-pass optimization involves an algorithm that is more complex than the algorithm for the first-pass optimization. If the final character states are not needed during the search, then performing a second-pass optimization might not be desirable due to

the computational complexity. In this regard, the alternative second-pass optimization is a method that allows to use the ICTL without having to do a second-pass optimization as described in section 2.1.3.2 [43].

Fitch's algorithm [38] examines the evolution of only one DNA character on the tree at a time. This means that each character evolves independently; thus, the character states are reversible. As a consequence of reversibility of character states, the tree may be rooted at any point with no change in the tree length. An example of how a tree can be rerooted with no change in its total length is shown in figure 2.9.

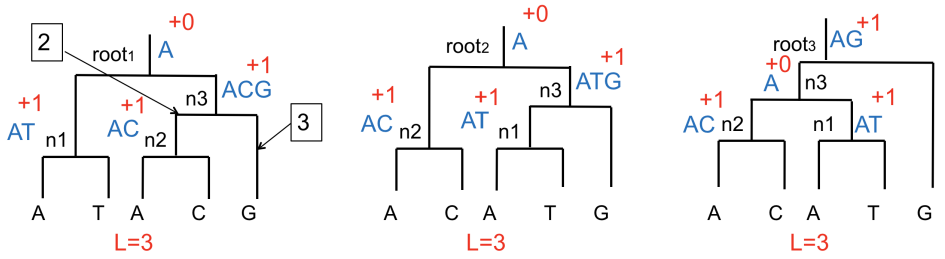


FIGURE 2.9: Example of rerooting the tree

Figure 2.9 shows three rooted trees with 5 taxa and 4 internal nodes each. The original tree is the one on the left. The tree in the middle has been obtained by rerooting the original tree in the branch formed by $n2$ and $n3$, as indicated by the arrow with a number 2 inside a box. Similarly, the tree on the right has been obtained by rerooting the original tree in the branch formed by G and $n3$, as indicated by the arrow with a number 3 inside a box. Both trees have the same length as the original one. In general, no matter where the tree is rooted, it will have the same length.

As we saw in section 2.2.2, the ICTL method uses the final character states of two nodes: *NX* and *NY* (see figure 2.7). Instead, we can reroot the tree at *N(X,Y)* as illustrated in figure 2.10.

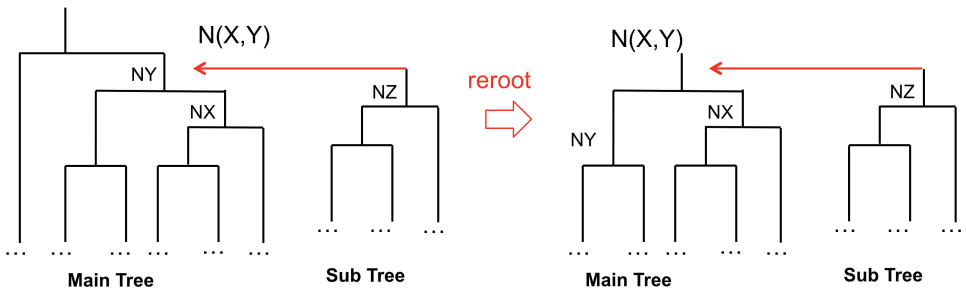


FIGURE 2.10: Rerooting the tree for the ICTL

Then we can do a first-pass optimization, and use the preliminary character states of *NX* and *NY*, and *NZ* to calculate the score

difference $D.Score$, because the final character states are the same as the preliminary character states for the root of the tree, as we saw in section 2.1.3.2.

Rerooting the tree means changing the order in which the original tree is traversed. In other words, there is no need to actually change the root of the tree and the tree topology every time the ICTL method is used. We only need to traverse the tree in all three possible ways, as shown in figure 2.11.

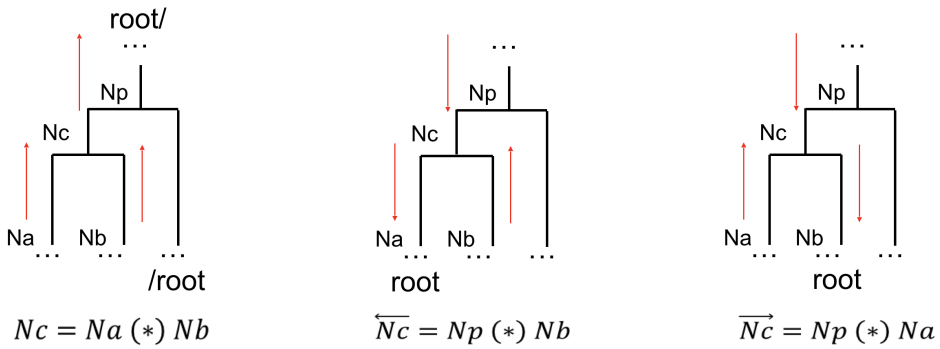


FIGURE 2.11: Traversing the tree in all three different ways

The (*) symbol in figure 2.11 represents the operation used to calculate the character state of a node during the first-pass optimization. For each node, there are three node character states. The "normal-path" character state, assuming that the root is above the node; the "left-path" character state, assuming that the root is below on the left side; and the "right-path" character state, assuming that the root is below on the right side. The "normal-path" character state is calculated during the first-pass optimization, and the "left-path" and "right-path" character states are calculated during this alternative optimization. In this way, the complexity of performing a complete second-pass optimization is reduced to that of performing two first-pass optimization for all nodes in the tree as indicated in figure 2.11.

2.2.4 Incremental Tree Optimization

The Incremental Tree Optimization consists on optimizing only those nodes that were affected by the pruning or reinsertion of a branch during the Subtree Pruning and Regrafting (SPR). In other words, only those nodes that might have changed character state (DNA sequence) and length (score) have to be recalculated. As a result, the number of nodes that has to be updated during an incremental tree optimization is much lower than during a complete tree optimization.

In a similar way to the complete tree optimization (explained in section 2.1.3), the incremental tree optimization consists of two

phases: the incremental first-pass optimization and the incremental second-pass optimization. Here we describe both of them as we have applied them in our algorithm in section Y.Y.Y. For a more detail and complete version of the Incremental Tree Optimization method, please refer to the work in [42].

2.2.4.1 Incremental First-pass Optimization

The incremental first-pass optimization consists on recalculating preliminary character states and node lengths only for those nodes that were affected by the pruning or reinsertion process [42]. The character state of a node is still calculated with the same algorithm used during the complete first-pass optimization (refer to figure 2.2). To better understand how the incremental first-pass optimization works, we show in figure 2.12 a simple example of how it is performed after the pruning process for a tree with 7 nodes.

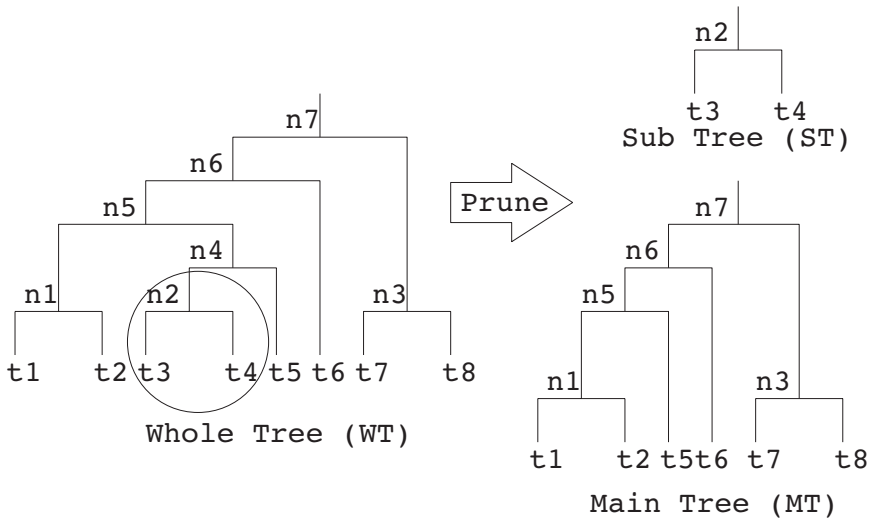


FIGURE 2.12: Example of the incremental first-pass optimization

As can be seen from Figure 2.12, pruning a branch from the whole tree only affects the nodes above the node. All the other nodes will have the same character states and lengths, because they have the same descendants as in the whole tree. In this example, $n5$, $n6$ and $n7$ can change, while $n1$ and $n3$ remain the same. Now, starting at the parent node of the node that had one of its branches pruned, i.e. $n5$, the new character states and the new length are calculated. Then, we proceed to $n6$, the parent node of $n5$, and do the same. Similarly, we proceed to $n7$ and do the same. Since $n7$ is the root of the tree, we finish here. But, if during the incremental first-pass optimization we find that the new character states of a node are the same as the previous ones, we do not have to continue calculating the character states for the other nodes, because they remain the same. Only the node lengths will have to be recalculated.

This is an important fact, because the lowest node with unmodified character states is the starting node in the incremental second-pass optimization. The incremental first-pass optimization is done in a similar way after the reinsertion process.

2.2.4.2 Incremental Second-pass Optimization

The incremental second-pass optimization consists on recalculating final character states only for those nodes affected by the pruning or reinsertion process [42]. The starting node is the lowest node with unmodified preliminary character states that was found during the incremental first-pass optimization. All nodes below this node, from both left and right side lineages, might change character states, so the incremental second-pass optimization has to cover them all. For this reason, the incremental second-pass optimization can be seen as a complete second-pass optimization that takes the lowest node with unmodified preliminary character states as the root node. Then, for each node we use the second-pass optimization algorithm shown before in figure 2.4 to obtain the new final character states.

Chapter 3

Approach for the Progressive Tree Neighborhood

3.1 Algorithm Overview

The algorithm is based on the stochastic local search algorithm described in section 2.1.1, and uses the Progressive Tree Neighborhood described in section 2.2.1. It is guided by the score function, i.e. the length of the tree. First, a tree in the search space is randomly generated. For this initial tree, a complete first-pass optimization is done to calculate the initial score of the tree. Then, at each iteration of the algorithm a neighbor tree rearrangement replaces the current one if it has a lower score, i.e. better score. For this, in each iteration, the following steps are performed:

1 A branch to prune from the whole tree (WT) is chosen randomly.

2 The main tree (MT) and the subtree (ST) derived from the previous pruning are created.

3 3.1 All possible branches from the MT where the ST can be reinserted are listed, according to the distance parameter.

3.2 A branch is randomly chosen from the previous listing.

4 The ST is inserted in the MT to create a new WT.

5 A first-pass optimization is done on the WT.

If the score of the WT is greater (worse) than the current score, go to step 6. Otherwise, go to step 1.

6 6.1 The previous tree topology is reconstructed.

6.2 Go to step 1.

The algorithm stops when there is no better neighbor tree or when the maximum number of iterations has been reached.

3.2 Phylogenetic Data Structure

For a given phylogenetic tree reconstruction problem consisting of N taxa, each of which has a sequence of L nucleobases, the sequence matrix is an N rows $\times$ L columns matrix. The characters in the sequences might include not only the DNA nucleobases Adenine (A), Cytosine (C), Guanine (G) and Thymine (T), but also the '-' character, which represents a gap, and the '?' character, which represents an undefined character. In total there are six characters, so a 3-bit binary representation can be used instead of the 8-bit ASCII representation to reduce the size of the memory. This is shown in table 3.1.

TABLE 3.1: First approach: 3-bit representation for DNA characters

DNA character	3-bit representation
'A'	000
'C'	001
'G'	010
'T'	011
'-'	100
'?'	101

Hence, a memory of $N \times L \times 3$ bits is required to store the sequence alignment matrix. On the other hand, the tree topology shows the connections between the internal nodes of the tree and the taxa. A tree with N taxa has $N-1$ nodes, including the root node. Since the tree is a binary tree, each node has a left branch and a right branch, and it has a parent node. Thus, the size of the memory required to store the tree topology is $(N-1) \times 3[log_2(N)]$ bits. For example, the tree topology memory and the sequence data memory for a tree with 6 taxa can be represented as shown in figure 3.1.

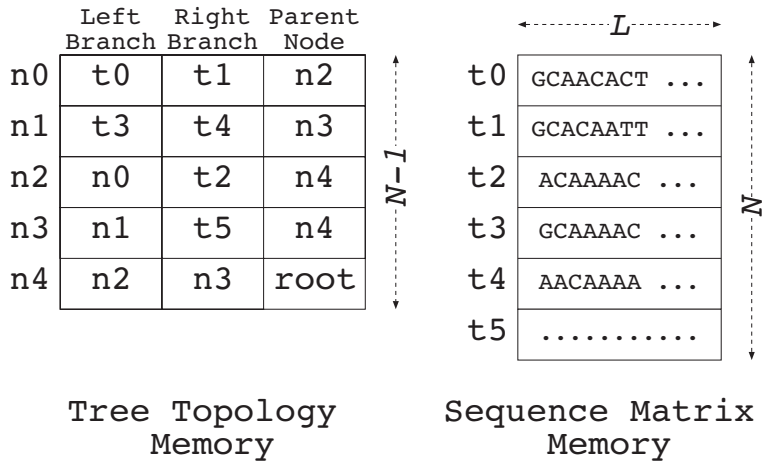


FIGURE 3.1: First approach: memory data structure

Taxa are labeled according to their memory position on the sequence matrix memory. Likewise, nodes are labeled according to their memory position on the tree topology memory. Since both nodes and taxa appear on the same memory, we use an additional bit to distinguish between the two of them: 0 for a node and 1 for a taxon (singular form of taxa). The root node doesn't have a parent. Instead, a full sequence of 1s is used to identify it.

3.3 Proposed Hardware Architecture

In section 3.1, we mentioned the steps involved in each iteration of the algorithm. To design the hardware architecture, we considered that steps 1 and 3 can be performed by the same hardware unit, since they are about choosing a branch from the tree. Similarly, steps 2 and 4 can be performed by the same hardware unit, since they are about modifying the tree topology. On the other hand, step 5 can be divided into two steps: listing the node order in which the score is calculated, and calculating the actual score of the tree by following that order. Each of these steps is carried out by a different hardware unit.

As a result, we designed the following hardware units to implement the algorithm described in section 3.1:

1 Prune and Reinsert Selection (PRS) unit

2 Tree Topology Update (TTU) unit

3 Node Order Listing (NOL) unit

4 Tree Score Calculation (TSC) unit

5 Global Control (GC) unit

In addition to this, we considered that the hardware units can work in parallel. Thus, more than one tree topology can be explored at the same time. For this, we use three Tree Topology memories instead of one. A general block diagram of the hardware architecture proposed is shown in figure 3.2.

It consists of the following elements: the Tree Topology Memory ($\times 3$), the Sequence Alignment Matrix Memory, the Node Order Memory ($\times 2$), the PRS unit, the TTU unit, the NOL unit, the TSC unit, and the GC unit. Black bars on figure 3.2 make reference to multiplexers. In the following sections, we explain how each of this hardware units works.

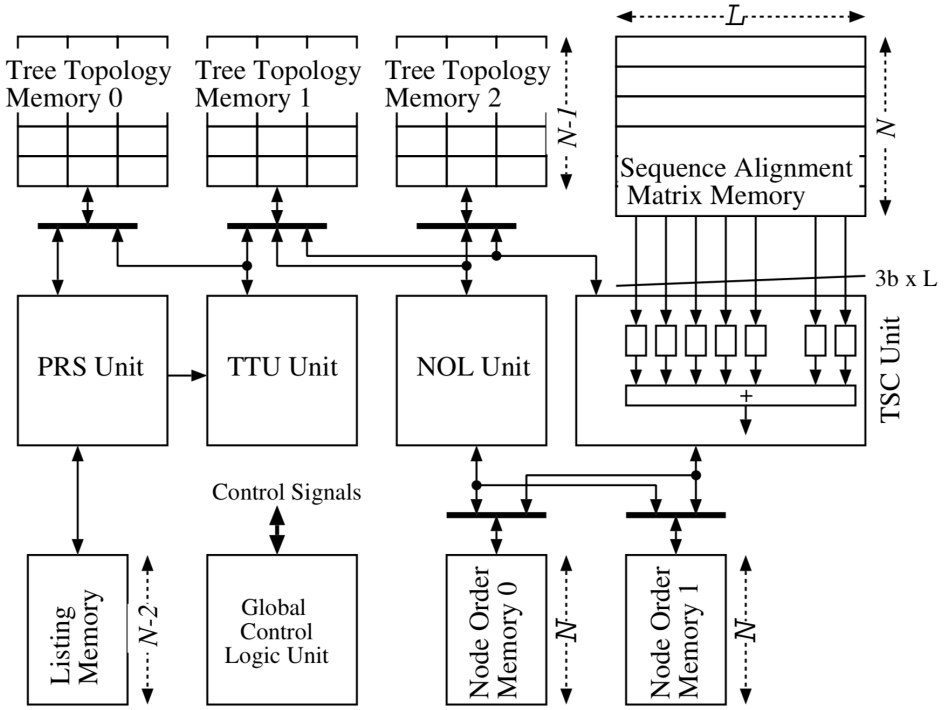


FIGURE 3.2: First approach: general block diagram of the proposed hardware architecture

3.3.1 Prune and Reinsert Selection (PRS) unit

The Pruning and Reinsert Selection (PRS) unit is in charge of deciding which branch of the whole tree (WT) is pruned, and where the pruned branch with the subtree (ST) attached to it is reinserted in the main tree (MT). Its general block diagram is shown in figure 3.3.

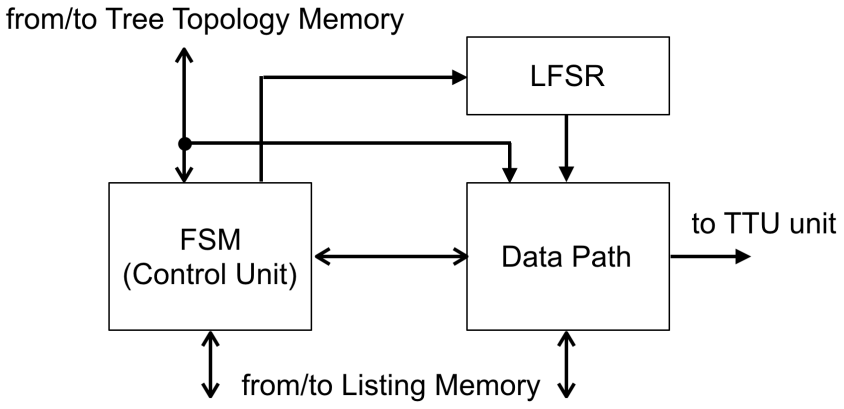


FIGURE 3.3: First approach: general block diagram of the PRS unit

As can be seen from figure 3.3, the PRS unit is implemented as a Finite State Machine (FSM) with a Data Path. In addition to this, it

has a linear-feedback shift register (LFSR), which works as a pseudo-random number generator.

The PRS unit works as follows. First, using the LFSR, a branch is randomly chosen to be pruned from the tree. Then, all possible branches where the pruned branch can be reinserted are listed in a memory (*Listing Memory* in figure 3.2). Since the distance between the pruned branch and the branch where it is reinserted is restricted by the distance parameter (refer to section 2.2.1), the branches are listed as valid or invalid. Finally, a valid branch from the *Listing Memory* is randomly selected for the pruned branch with the ST attached to it to be reinserted in the MT. For this, the LFSR is used again. An example of how the valid and invalid branches are stored in the *Listing Memory* is shown in table 3.2 for the tree of figure 2.6 that has 9 nodes and a distance parameter with a current value of 3.

TABLE 3.2: First approach: example of the memory listing
(1'b0 = invalid, 1'b1 = valid)

node	LB	RB
n0	1'b0	1'b0
n1	1'b0	1'b0
n2	1'b1	1'b1
n3	1'b1	1'b1
n4	1'b1	1'b1
n5	1'b1	1'b1
n6	1'b1	1'b1
n7	1'b1	1'b1
n8	1'b1	1'b1

As can be seen from table 3.2, only one bit is necessary to determine whether the left or right branch of a node is valid or invalid as candidate branch for reinserting the ST into the MT.

The PRS unit requires an execution time that depends on the number of nodes that has to be listed. However, since one clock cycle is required to list one node, the execution time never exceeds $N - 2$ clock cycles for the worst-case scenario where all nodes except one have to be listed.

3.3.2 Tree Topology Update (TTU) unit

The Tree Topology Update (TTU) unit is in charge of modifying the Tree Topology Memory to reflect the changes produced by the pruning and regrafting process as determined by the PRS unit. Its general block diagram is shown in figure 3.4.

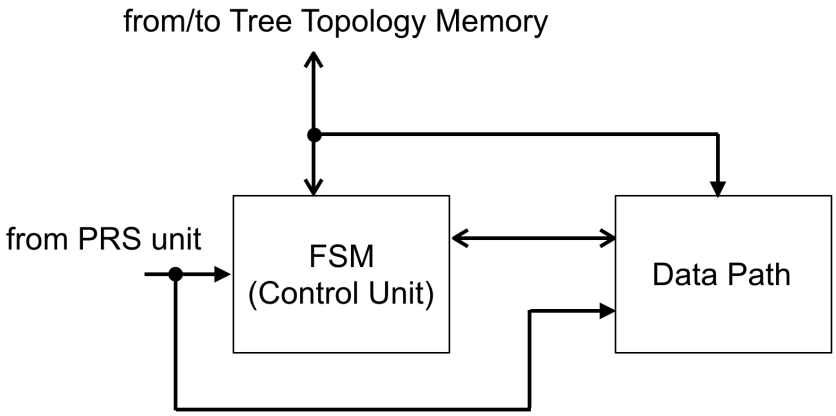
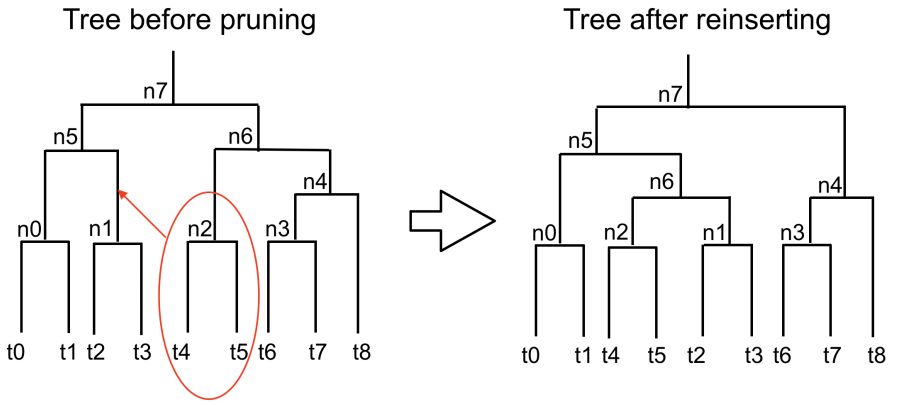


FIGURE 3.4: First approach: general block diagram of the TTU unit

As can be seen from figure 3.4, the TTU unit is implemented as a Finite State Machine (FSM) with a Data Path. The FSM works basically as a memory controller that is used to modify the content of a Tree Topology Memory.

Regardless of the number of taxa, when a branch with a subtree (ST) attached to it is pruned from the tree and reinserted elsewhere, at most 5 nodes are modified. An example is shown in figure 3.5 to illustrate this.



Node	LB	RB	PN
n0	t0	t1	n5
n1	t2	t3	n5
n2	t4	t5	n6
n3	t6	t7	n4
n4	n3	t8	n6
n5	n0	n1	n7
n6	n2	n4	n7
n7	n5	n6	root

Node	LB	RB	PN
n0	t0	t1	n5
n1	t2	t3	n6
n2	t4	t5	n6
n3	t6	t7	n4
n4	n3	t8	n7
n5	n0	n6	n7
n6	n2	n1	n5
n7	n5	n4	root

FIGURE 3.5: First approach: example of the nodes modified by the pruning and regrafting process

Figure 3.5 shows a tree with 9 taxa. On the left is the tree before pruning, and on the right is the tree after reinserting. The pruned branch is the left branch (LB) of *n6*, and the reinsertion branch is the right branch (RB) of *n5*. This pruning and reinsertion process causes changes in the tree structure that affect a total of 5 nodes.

Thus, 5 memory positions are modified in the Tree Topology Memory accordingly. These positions correspond to the following nodes:

1 The node where the branch is pruned from: *n6*

2 The parent node of the previous mentioned node: *n7*

3 The node on the opposite branch to the pruned branch: *n4*

4 The node up where the pruned branch is reinserted: *n5*

5 The node down where the pruned branch is reinserted: *n1*

The process of pruning and regrafting a branch leads to a maximum number of 5 changes in the tree topology. This means that it is not necessary to rewrite the whole Tree Topology Memory, but only to update those 5 memory positions. Since it takes 2 clock cycles to update a memory position, the TTU unit requires a maximum total of 10 clock cycles.

3.3.3 Node Order Listing (NOL) unit

The Node Order Listing (NOL) unit has the task of listing all the nodes in the tree in a post-order. Then, this listing can be used by the Tree Score Calculation (TSC) unit to do a tree traversal for the first-pass optimization. The general block diagram of the NOL unit is shown in figure 3.6.

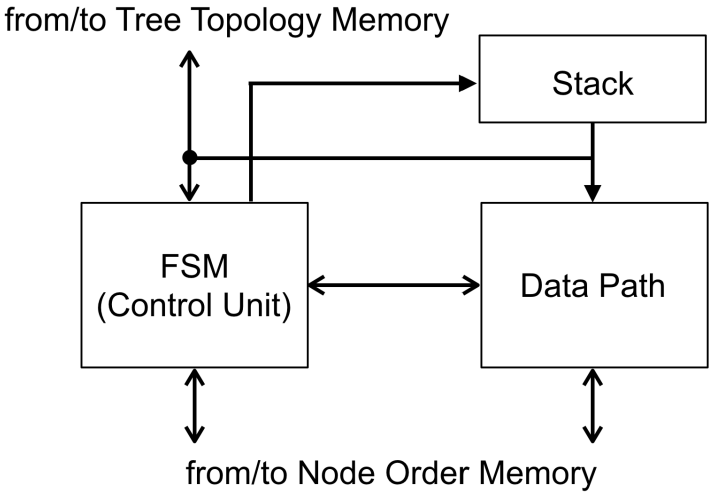


FIGURE 3.6: First approach: general block diagram of the NOL unit

As can be seen from figure 3.6, the NOL unit is implemented as a Finite State Machine (FSM) with a Data Path and a Stack, from which the data is read or written in a last-in-first-out (LIFO) order. The NOL unit is connected to a Tree Topology Memory and a Node Order Memory, which is also a Stack. The following algorithm is used to list all the nodes in the tree:

1 Read the memory position of the root node from the selected Tree Topology Memory (TTM)

2 Repeat until all nodes are listed.
Push each node visited into the Node Order Memory.

Case (Left Branch (LB), Right Branch (RB))

2.1 (Node, Node): Push RB into the Stack, read LB from TTM

2.2 (Node, Leaf): Read LB from TTM

2.3 (Leaf, Node): Read RB from TTM

2.4 (Leaf, Leaf): Pop a node and read it from TTM

As a result of the above mentioned algorithm, the last node visited becomes the first node to be read from the Node Order Memory. Thus, the desired post-order (reverse order of the visited nodes) is obtained. To illustrate this, an example is shown in figure 3.7.

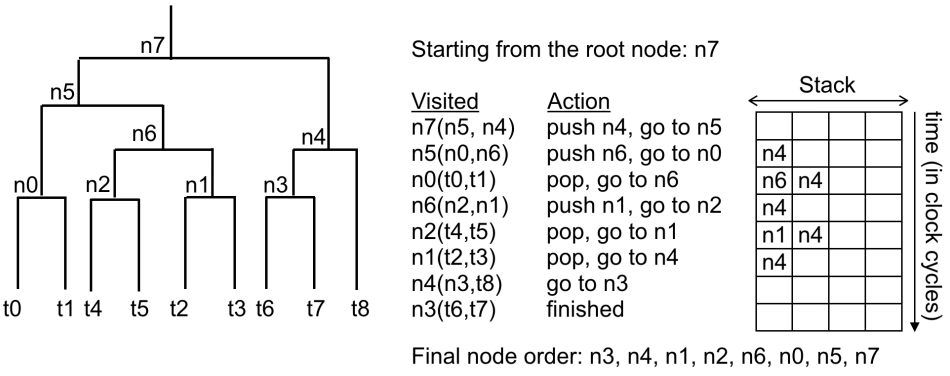


FIGURE 3.7: First approach: example of the NOL unit listing process

Figure 3.7 shows a tree with 9 taxa and 8 nodes. The visited nodes, the actions taken and the contents of the stack for every clock cycle are detailed on the right of the tree. At the end, the Node Order Memory has the reverse order of the visited nodes. This is the final node order.

The NOL unit takes one clock cycle to list each node, so it requires a total of $N-1$ clock cycles to list all the nodes, where N is the number of taxa.

3.3.4 Tree Score Calculation (TSC) unit

The Tree Score Calculation (TSC) unit computes the total length of the tree (i.e. the score) by following the node order stored in the selected Node Order memory. Its general block diagram is shown in figure 3.8.

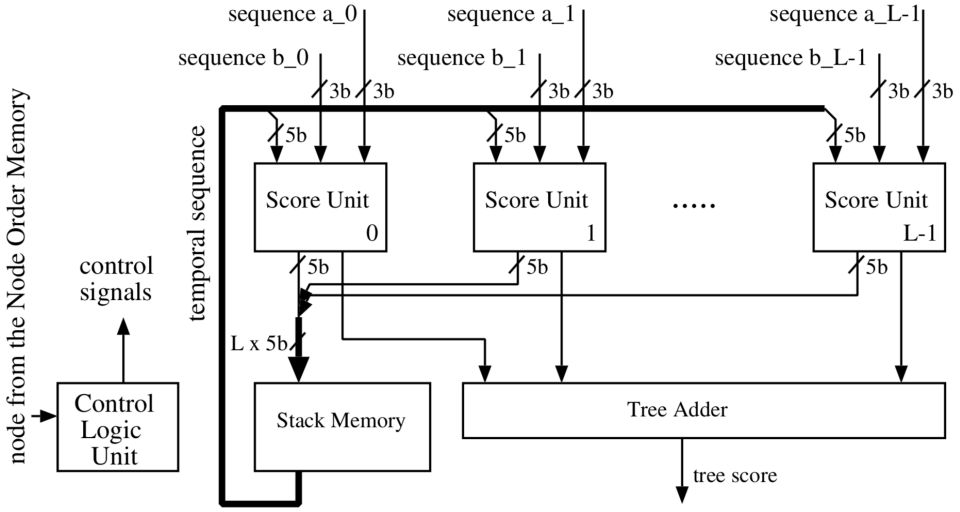


FIGURE 3.8: First approach: general block diagram of the TSC unit

As seen in figure 3.8, the TSC unit is composed of L Score Units, where L is equivalent to the number of DNA characters in a sequence (refer to figure 3.1), a Tree Adder, a Stack Memory, and a Control Logic unit. It receives two DNA sequences from the Sequence Alignment Matrix Memory: *sequence_a* and *sequence_b*, and the node order from the Node Order Memory. This allows to process one node at a time. The Tree Adder is used at the end to add all individual scores from each Score Unit in order to obtain the final score of the tree.

The TSC uses L Score Units to implement the first-pass optimization algorithm (see section 2.1.3.1), which consists in obtaining the character state and the length of the node. However, for this approach neither the character states nor the lengths of the nodes are needed, so only the score is accumulated. The general block diagram of a Score Unit is shown in figure 3.9.

As can be seen from figure 3.9, the Score Unit is composed of two 3-bit to 5-bit decoders, one AND and one OR logic gates, a comparator, an accumulator that includes a D-type flip-flop, and a D-type flip-flop-based register of 5 bits.

The decoders convert the 3-bit DNA characters to a 5-bit representation, as shown in table 3.3. Thanks to this final representation, the union can be performed by the binary *OR* operation, and the intersection by the binary *AND* operation [44].

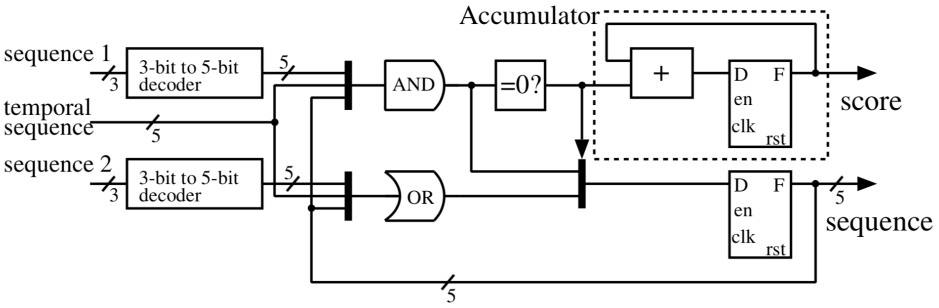


FIGURE 3.9: First approach: general block diagram of the Score Unit

TABLE 3.3: First approach: conversion of the 3-bit to 5-bit representation for DNA characters

DNA character	3-bit DNA	5-bit DNA
'A'	000	00010
'C'	001	00100
'G'	010	01000
'T'	011	10000
'-'	100	00001
'?'	101	11111

The TSC unit works by using the algorithm listed below. This algorithm uses the Stack Memory shown in figure 3.8, and the sequence output from all the Score Units. The (*) symbol represents the operation performed by all the Score Units.

Repeat the following steps until all nodes are processed:

1 Pop a node from the Node Order Memory.
Read it from the Tree Topology Memory.

2 If both branches are leaves (taxa), push the sequence (seq) into the Stack Memory (except for the first time).

3 Case (Left Branch (LB), Right Branch (RB))

3.1 (Node, Node): Pop a node from the Stack.
seq = seq (*) temporal sequence

3.2 (Node, Leaf): Read the sequence for RB (seq_RB).
seq = seq (*) seq_RB

3.3 (Leaf, Node): Read the sequence of LB (seq_LB).
seq = seq $(*)$ seq_LB

3.4 (Leaf, Leaf): Read both sequences from LB and RB.
seq = seq_LB $(*)$ seq_RB

To illustrate the above mentioned algorithm, an example is given in figure 3.10. This example is for the tree with 9 taxa shown in figure 3.7 of section 3.3.3.

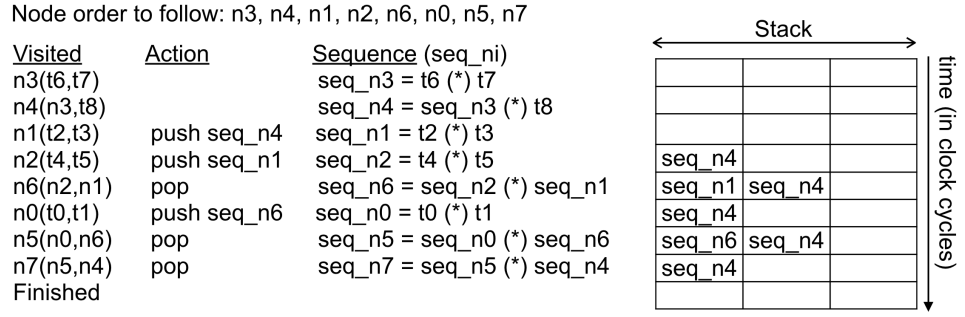


FIGURE 3.10: First approach: example of TSC unit processing

As can be seen from figure 3.10, processing a single node requires one clock cycle thanks to the pipeline stages used in the TSC unit. As a result, processing all nodes requires an approximate time of $N - 1$ clock cycles.

3.3.5 Global Control (GC) unit

The Global Control (GC) logic unit is a Finite State Machine (FSM) that commands the other four units: PRS, TTU, NOL and TSC. The GC unit controls these units so they work in pipelined stages as illustrated in figure 3.11, where TTMi refers to the Tree Topology Memory *i*, and NOMi refers to the Node Order Memory *i*. At the beginning, TTM0, TTM1 and TTM2 start with the same tree topology.

The pipelined flow in figure 3.11 was designed by taking into account that most of the time the new tree topology has a worse score than that of the previous tree topology. For this reason, it is better to explore more than one tree topology at a time. Thus, one tree topology is stored in TTM1 and another one in TTM2.

If the score for a particular TTM is better, then the contents of it is copied into the other TTMs. Otherwise, the TTM is rebuild to its previous state, as shown in figure 3.11.

Updating or rebuilding a TTM takes less than 10 clock cycles, so these added stages do not have a significant impact on the overall performance. On the contrary, thanks to the pipelined flow, the process of calculating the score of a tree is reduced by an approximate of one third, in the worst-case scenario.

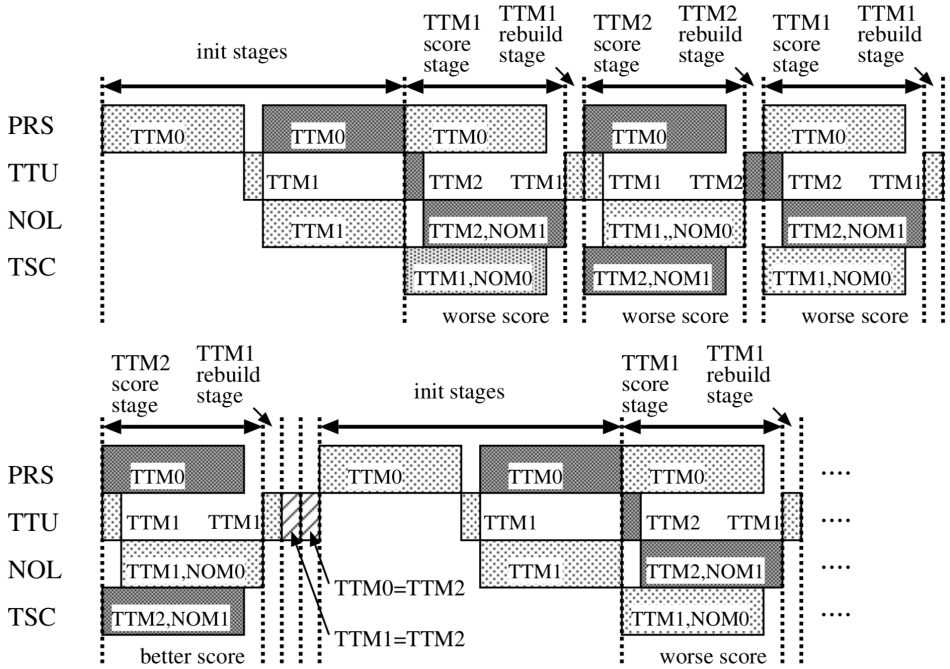


FIGURE 3.11: First approach: execution flow of the proposed hardware architecture

3.4 Implementation Results

In this section we show implementation results for four real-world biological datasets. The datasets were obtained from the repository of phylogenetic information TreeBASE [45], see table 3.4.

TABLE 3.4: First approach: datasets used [45]

ID	M972	M2355	M3452	M3875
#taxa	155	150	116	228
#characters	355	829	1,157	1,435

3.4.1 Hardware Utilization and Performance Results

The hardware utilization and performance results are shown in table 3.5. The targeted FPGA is a Kintex-7 XC7K325T-FF2-900.

The implementation covers any of the four datasets in table 3.4. In other words, problems up to $N = 1,024$ and $L = 1,435$ can be processed with this amount of hardware logical resources.

TABLE 3.5: First approach: implementation results on a Kintex-7 FPGA

Logic Utilization	Used	Available	Utilization
Number of Slices	16,821	50,950	34%
Number of Slice Registers	33,468	407,600	8%
Number of Slice LUTs	67,282	203,800	33%
Number of BRAMs (36 Kb)	224	445	50%
Maximum Frequency	157.031MHz		

3.5 Comparison and Performance Evaluation

3.5.1 Execution Time for the Score Calculation

First, we compare the execution time required for calculating the score of a single tree. In other words, the time required to do a first-pass optimization. A theoretical software approach would require an approximate time of that shown in Equation 3.1.

$$t \approx (N-1)(L)(F^{-1})(OP) \tag{3.1}$$

where N is the number of taxa, L is the number of characters, F is the operational frequency of the CPU, and OP is the number of CPU operations required per node calculation.

On the other hand, our hardware implementation takes the time required by the NOL and TSC units together. This is shown in equation 3.2. Note that this is not the maximum throughput of the circuit.

$$t \approx 2(N-1)(F^{-1}) \tag{3.2}$$

where N is the number of taxa, and F is the operational frequency of the FPGA. From this equation, it should be noted that the execution time of the hardware implementation is independent of the number of DNA characters.

For example, if we consider that a software implementation runs at a frequency of 2.8 GHz, and requires around 30 CPU operations per node in the best-case scenario, the FPGA implementation, considered that it runs only at a frequency of 157 MHz, will be in the order of hundreds or thousands, and will increase with the number of characters.

3.5.2 Local Search Results Comparison

Now, we compare the execution time required for the whole local search. We compare our hardware approach to our C++ implementation, both of which implement the same local search algorithm with

the progressive tree neighborhood. In both cases the score of each tree is evaluated by a complete post-order traversal.

Furthermore, we also make a comparison with TNT (Tree analysis using New Technology) [20]., the fastest available parsimony program. To make the comparison as fair as possible, we use the traditional search of TNT based on Subtree Pruning and Regrafting (SPR). Moreover, since the total number of examined trees is not the same, we show the execution time required for a single iteration.

The CPU used is an Intel Core i7 860@2.80GHz with 4 GB RAM. The targeted FPGA runs at 153.8 MHz. The results are summarized in table 3.6.

TABLE 3.6: First approach: results for the local search

Dataset		SW C++	FPGA	Accel.	TNT
M972	Total time (s)	133.15	0.0622	2141 (1.23 clk /taxa)	0.05
	Time/tree (μs)	2661.27	1.2432		0.1841
	Visited trees	50,031	←		271,584
	Score (Best: 1,529)	1,548	←		1,551
M2355	Total time (s)	152.68	0.0555	2751 (1.14 clk /taxa)	0.06
	Time/tree (μs)	3051.71	1.1093		0.2750
	Visited trees	50,032	←		218,148
	Score (Best: 2,748)	2,749	←		2,757
M3452	Total time (s)	125.65	0.0499	2518 (1.32 clk /taxa)	0.06
	Time/tree (μs)	2510.69	0.9971		0.2809
	Visited trees	50,046	←		213,634
	Score (Best: 3,608)	3,633	←		3,638
M3875	Total time (s)	240.40	0.0828	2903 (1.11 clk /taxa)	0.05
	Time/tree (μs)	4791.54	1.6503		0.0991
	Visited trees	50,172	←		504,305
	Score (Best: 561)	605	←		565

In comparison to our C++ implementation, the FPGA acceleration (Accel. in table 3.6) is in the order of thousands, and it increases with the number of characters. The values between parentheses in the Accel. column show the average number of clock cycles required to calculate the score per taxa. The total number of clock cycles required to calculate the score of the tree can be obtained by multiplying the number of taxa by these values. As can be seen, the values are very close to 1.00 (the minimum number of clock cycles), which shows that the pipeline works well. When the circuit continues to fail to generate a tree with a lower score, the value becomes closer to 1.00. On the contrary, when a better tree is found, the pipeline is stalled; thus, the value becomes higher.

On the other hand, in comparison to TNT, there is no acceleration. In fact, TNT is faster than our FPGA implementation. For example, only 99 *ns* are required to evaluate the score of one tree in

problem M3865. This is extremely fast, even if we consider that the operational frequency is 2.8 GHz, and that multi-core processing (4 cores in this evaluation) with SIMD instructions is used. Here, the following should be remarked. First, the search performed by TNT is not based on the Progressive Tree Neighborhood, but on Random Addition Sequences (RAS) with the Subtree Pruning and Regrafting (SPR) neighborhood. Second, TNT's search strategy does not require a de-novo computation of the tree score in each iteration. TNT implements optimization methods to reduce the evaluation time of tree rearrangements. Hence, its execution time is not in accordance with Equation 3.1.

Our implementation does not involve any optimization methods. A post-order tree traversal is performed for each tree being evaluated. Nevertheless, the scores obtained by our approach are comparable to those obtained by TNT (the values between parentheses are the lowest scores reported so far), as well as the total execution time.

3.6 Discussion

We first compared the execution time for the score calculation of a single tree between a software and a hardware approach. In this regard, the acceleration rate of our hardware approach can be in the order of hundreds or thousands. This high performance is achieved by parallel processing of all the characters in the Sequence Alignment Matrix Memory by using L Score Units (see section 3.3.4), where L is the number of DNA characters in the sequence. As a result, the execution time for the tree score calculation is independent of the number of characters in our approach.

Then, we compared the execution time required to perform the whole local search for four real-world biological datasets, which consist of hundreds of taxa and DNA characters. When compared to our C++ implementation, acceleration rates in the order of thousands were achieved. However, when compared to a highly optimized parsimony program like TNT, there is no acceleration. In fact, TNT performs faster.

From this approach we learned that to achieve faster execution times than TNT with a hardware implementation, it is necessary to consider that the score of the tree does not have to be recalculated from the start in each iteration, since only a small portion of the tree changes by the Subtree Pruning and Regrafting process.

Chapter 4

Approach for the Indirect Calculation of Tree Lengths

4.1 Algorithm Overview

The algorithm is based on the stochastic local search algorithm described in section 2.1.1. It uses both the Progressive Tree Neighborhood, described in section 2.2.1, and the Indirect Calculation of Tree Lengths, described in section 2.2.2.

The algorithm starts from a randomly generated tree in the search space, and tries to improve it on each iteration. For the initial tree, a list for all the branches is created. This list will denote which rearrangements have to be tried. Then, a first-pass optimization is done to calculate the initial score of the tree. At each iteration of the algorithm a neighbor tree rearrangement replaces the current one if it has a lower score, i.e. better score. For this, in each iteration, the following steps are performed:

1 1.1 A branch to prune from the whole tree (WT) is randomly chosen from the list.

1.2 The main tree (MT) and the subtree (ST) derived from the previous pruning are created.

2 All possible branches from the MT where the ST can be reinserted are listed, according to the distance parameter from the Progressive Tree Neighborhood.

3 A first-pass optimization is done on the MT and the ST.

If the sum of the scores of the MT and the ST is greater (worse) than the current score, go to step 7.

4 A second-pass optimization is done on the MT.

5 All rearrangements within the neighborhood are evaluated by the ICTL.

If the sum of the scores of the MT, the ST and the difference (*D.Score*) is greater (worse) than the current score, go to step 7.

6 6.1 The ST is inserted in the MT to create a new WT.
  6.2 All branches are added to the list again.
  6.3 Go to step 1.

7 7.1 The previous tree topology is reconstructed.
  7.2 The chosen branch is removed from the list. If there are still branches in the list, go to step 1.

4.2 Phylogenetic Data Structure

For a given phylogenetic tree reconstruction problem consisting of N taxa, each of which has a sequence of L nucleobases, the sequence matrix is an N rows $\times$ L columns matrix. The characters in the sequences might include not only the DNA nucleobases Adenine (A), Cytosine (C), Guanine (G) and Thymine (T), but also the '-' character, which represents a gap, and the '?' character, which represents an undefined character. These are the six basic characters, but a combination of them is also possible thanks to the five-bit binary representation shown in table 4.1.

TABLE 4.1: Second approach: 5-bit representation for DNA characters [44]

DNA character	5-bit representation
'-'	00001
'A'	00010
'C'	00100
'G'	01000
'T'	10000
'?'	11111

As can be seen from table 4.1, each character is represented by a power of 2, from $2^0 = 1$ ($5'b00001$ for '-') to $2^4 = 16$ ($5'b10000$ for 'T'), except for '?', which is coded by the value 31 ($5'b11111$), since it can represent any character. Thanks to this five-bit representation, the union can be performed by the binary *OR* operation, and the intersection by the binary *AND* operation [44]. This eases the hardware implementation of the first-pass optimization algorithm, described in section 2.1.3.1 and shown in figure 2.2.

Hence, a memory of $N \times L \times 5$ bits is required to store the sequence alignment matrix. On the other hand, the tree topology shows the connections between the internal nodes of the tree and the taxa. A tree with N taxa has $N-1$ nodes, including the root node. Since the tree is a binary tree, each node has a left branch and a right branch, and it has a parent node. Thus, the size of the memory required to store the tree topology is $(N-1) \times 3[log_2(N)]$ bits.

Finally, the size of the memory required to store the node character states is of $(N-1) \times L \times 5$ bits. For example, the tree topology, the sequence data matrix memory and the node data matrix memory for a tree with 6 taxa are represented in figure 4.1.

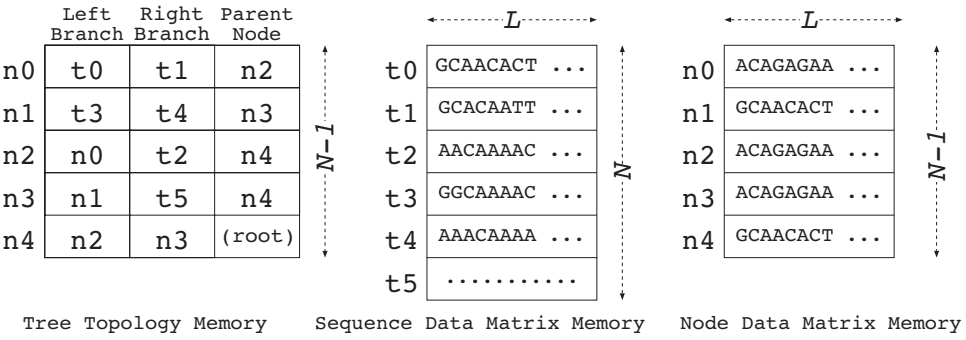


FIGURE 4.1: Second approach: memory data structure

Taxa are labeled according to their memory position on the sequence matrix memory. Likewise, nodes are labeled according to their memory position on the tree topology memory. Since both nodes and taxa appear on the same memory, we use an additional bit to distinguish between the two of them: 0 for a node and 1 for a taxon (singular form of taxa). The root node doesn't have a parent. Instead, a full sequence of 1s is used to identify it.

4.3 Proposed Hardware Architecture

In section 4.1, we mentioned the steps involved in each iteration of the algorithm. To design the hardware architecture, we considered that steps 1, 6 and 7 can be performed by the same hardware unit, since they are about modifying the tree topology. Similarly, steps 3, 4 and 5 can be performed by the same hardware unit, since they are about doing some operations on the nodes of the tree. However, steps 3 and 4 are divided into two additional steps each: listing the node order in which the first- and second-pass optimization is calculated, and performing the actual first- and second-pass optimization, respectively. Listing the node order is performed by a different hardware unit. Finally, step 2 is performed by a single hardware unit.

This leads to the following hardware units that we designed to implement the algorithm described in section 4.1:

1 Tree Topology Update (TTU) unit

2 Progressive Neighborhood Listing (PNL) unit

3 Node Order Listing (NOL) unit

4 First-, Second-pass and Rearrangement evaluation (FSR) unit

5 Global Control (GC) unit

The TTU unit is in charge of updating the tree topology memory to reflect the changes produced by the Subtree Pruning and Reinserting (SPR) process. The PNL unit is in charge of listing all possible nodes in the main tree where the pruned branch with a subtree (ST) attached to it can be reinserted in the main tree (MT). The NOL unit has the task of listing the nodes of the tree for a post-order tree traversal. The FSR unit has the most important tasks, which are doing a first- and second-pass optimization, and evaluating all possible rearrangements according to the Indirect Calculation of Tree Lengths (ICTL) method. A general block diagram of the hardware architecture proposed is shown in figure 4.2.

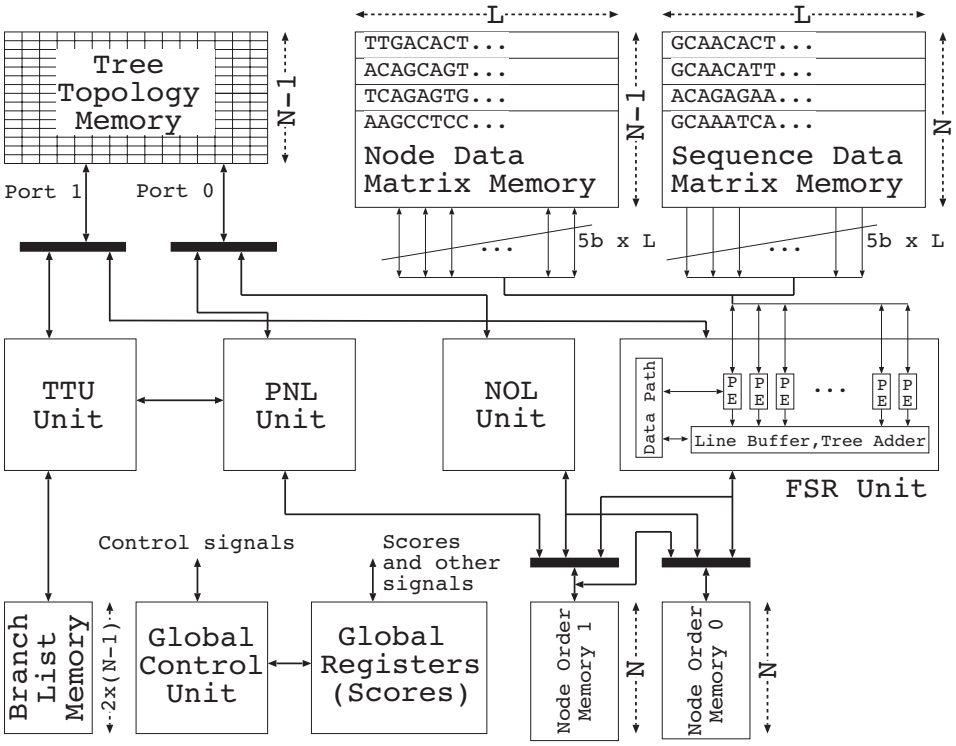


FIGURE 4.2: Second approach: general block diagram of the proposed hardware architecture

It consists of the following elements: the dual-port Tree Topology Memory (TTM), the dual-port Sequence Data Matrix Memory (SDM), the dual-port Node Data Matrix Memory (NDM), two Node Order Memories (NOM0 and NOM1), the Branch List Memory (BLM), the TTU unit, the PNL unit, the NOL unit, the FSR unit, and a Global Control unit with some registers. Black bars on the diagram make reference to multiplexers. In the following sections, we explain how each of this hardware units works.

4.3.1 Tree Topology Update (TTU) unit

The Tree Topology Update (TTU) unit is in charge of modifying the Tree Topology Memory to reflect the changes produced by the Subtree Pruning and Reinserting (SPR) process.

It has three main tasks:

1 Pruning a branch from the whole tree (WT) to create the main tree (MT) and the sub tree (ST).

2 Inserting the ST in the MT to create a new WT.

3 Rebuilding the previous WT when the score does not improve.

And two sub tasks:

1 Storing the value of the pruned branch and reinsertion branch.

2 Storing the values of the WT, MT and ST roots.

The general block diagram of the TTU unit is shown in figure 4.3.

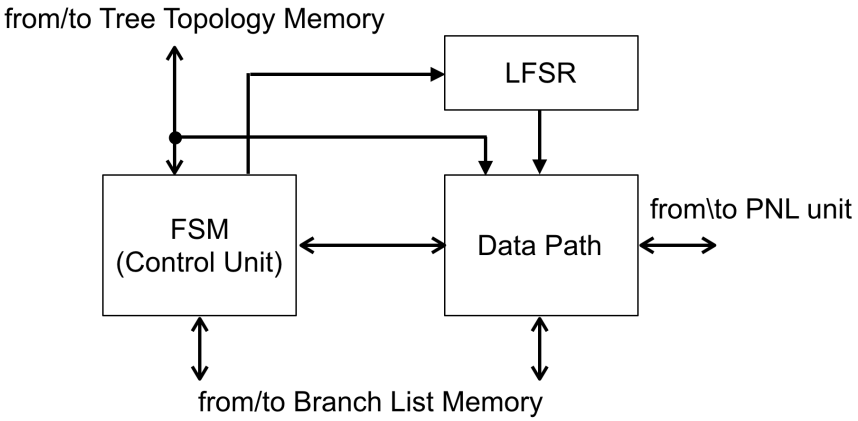
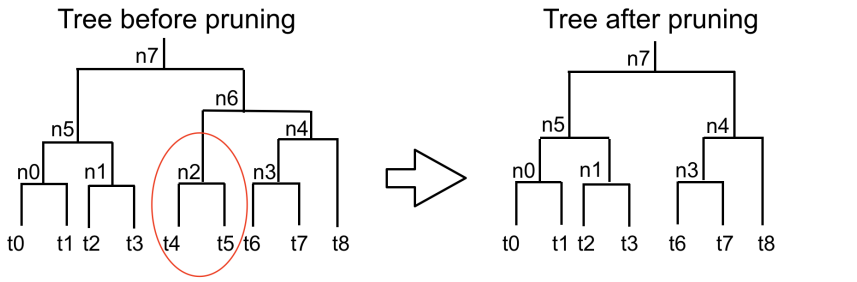


FIGURE 4.3: Second approach: general block diagram of the TTU unit

As can be seen from figure 4.3, the TTU unit is implemented as a Finite State Machine (FSM) with a Data Path. The FSM works basically as a memory controller that is used to modify the content of the Tree Topology Memory. In addition to this, it has a linear-feedback shift register (LFSR), which works as a pseudo-random number generator.

Regardless of the number of taxa, when a branch with a subtree (ST) attached to it is pruned from the tree, at most 2 nodes are modified. These nodes correspond to the parent node of the node where the branch is pruned from, and the node on the opposite branch to the pruned branch. An example of this is shown in figure 4.4.



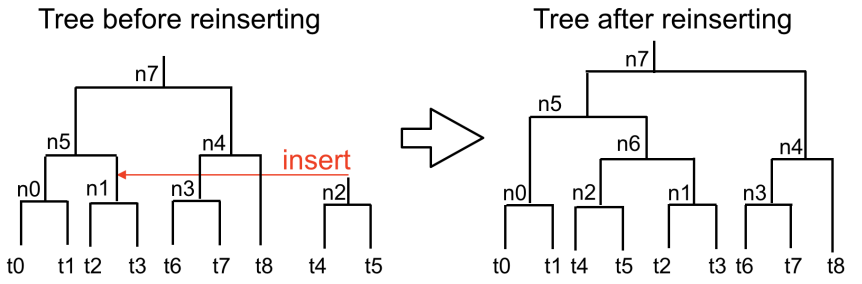
Node	LB	RB	PN
n0	t0	t1	n5
n1	t2	t3	n5
n2	t4	t5	n6
n3	t6	t7	n4
n4	n3	t8	n6
n5	n0	n1	n7
n6	n2	n4	n7
n7	n5	n6	root

Node	LB	RB	PN
n0	t0	t1	n5
n1	t2	t3	n5
n2	t4	t5	n6
n3	t6	t7	n4
n4	n3	t8	n7
n5	n0	n1	n7
n6	n2	n4	n7
n7	n5	n4	root

FIGURE 4.4: Second approach: example of the nodes modified by the pruning process

Figure 4.4 shows a tree with 9 taxa. On the left is the tree before pruning the left branch (LB) of *n6*, and on the right is the tree after pruning it. This pruning process causes changes in the tree structure that affect a total of 2 nodes. Thus, 2 memory positions are modified in the Tree Topology Memory accordingly.

Similarly, when a branch with a ST attached to it is inserted in the MT, at most 3 nodes are modified. These nodes correspond to the node where the pruned branch comes from, and the nodes up and down where the pruned branch is reinserted. An example of this is shown in figure 4.5.



Node	LB	RB	PN
n0	t0	t1	n5
n1	t2	t3	n5
n2	t4	t5	n6
n3	t6	t7	n4
n4	n3	t8	n7
n5	n0	n1	n7
n6	n2	n4	n7
n7	n5	n4	root

Node	LB	RB	PN
n0	t0	t1	n5
n1	t2	t3	n6
n2	t4	t5	n6
n3	t6	t7	n4
n4	n3	t8	n7
n5	n0	n6	n7
n6	n2	n1	n5
n7	n5	n4	root

FIGURE 4.5: Second approach: example of the nodes modified by the reinsertion process

Figure 4.5 shows the same tree with 9 taxa after reinserting the left branch (LB) of *n6* on the right branch (RB) of *n5*. This reinsertion process causes changes in the tree structure that affect a total of 3 nodes. Thus, 3 memory positions are modified in the Tree Topology Memory accordingly.

The pruning process involves modifying at most 2 nodes. Since it takes 2 clock cycles to modify a node from the Tree Topology Memory, the execution time is of 4 clock cycles at most. Similarly, inserting the sub tree involves modifying at most 3 nodes. Thus, it takes 6 clock cycles at most. Rebuilding the tree is equivalent to reverting the pruning process; hence, this takes also 4 clock cycles at most.

4.3.2 Progressive Neighborhood Listing (PNL) unit

The Progressive Neighborhood Listing (PNL) unit is in charge of listing all possible nodes in the main tree (MT) where the pruned branch with the subtree (ST) attached to it can be reinserted. For this, it takes into account the distance parameter from the Progressive Tree Neighborhood (see section 2.2.1). Its general block diagram is shown in figure 4.6.

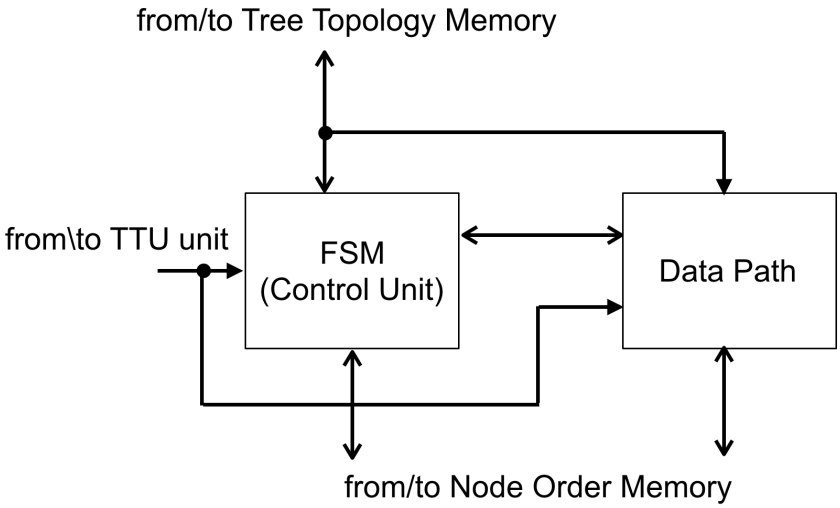


FIGURE 4.6: Second approach: general block diagram of the PNL unit

As can be seen from figure 4.6, the PNL unit is implemented as a Finite State Machine (FSM) with a Data Path. It is connected to the TTU unit, the Tree Topology Memory and one of the Node Order memories (see the general block diagram in figure 4.2).

It works as follows. Starting from the parent node of the node where the branch is pruned from, all nodes in the main tree (MT) are visited in order. Each time a node located at a relative distance less than or equal to the current value of the distance parameter is visited, the node is pushed into the Node Order Memory. Other nodes,

i.e. those nodes that have a relative distance that exceeds the current value of the distance parameter, are not visited. As a result, only those nodes from the MT that comply with the distance parameter from the Progressive Tree Neighborhood are listed in the Node Order Memory. An example of this is shown in figure 4.7.

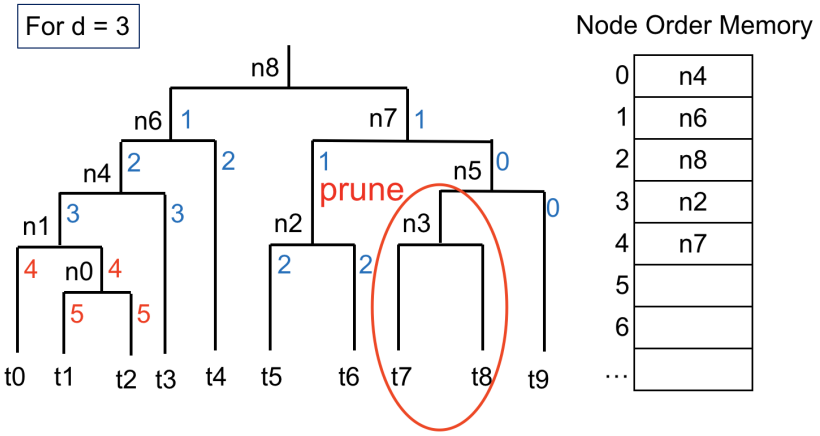


FIGURE 4.7: Second approach: example of the PNL listing

Figure 4.7 shows a tree with 10 taxa and 9 nodes with the relative distance values for all their branches. For this example, the current value of the distance parameter is equal to 3, so all branches that exceed this value are not visited by the PNL unit. At the end, the reversed order of the visited nodes is stored in the Node Order Memory, as shown on the right of figure 4.7. This listing is then used by the FSR unit to evaluate all the rearrangement trees within the neighborhood.

The PNL unit requires an execution time that depends on the number of nodes that have to be listed. However, since it requires two clock cycle to list each node at maximum, its execution time never exceeds $2 \times (N - 2)$ clock cycles for the worst- case scenario where all nodes except one have to be listed. Moreover, this unit works in parallel with the FSR unit, so its execution time does not add any delay.

4.3.3 Node Order Listing (NOL) unit

The Node Order Listing (NOL) unit has the task of listing all the nodes in the tree in a post-order. This tree can be the whole tree (WT), the main tree (MT) or the subtree (ST), depending on the root node chosen. The general block diagram of the NOL unit is shown in figure 4.8.

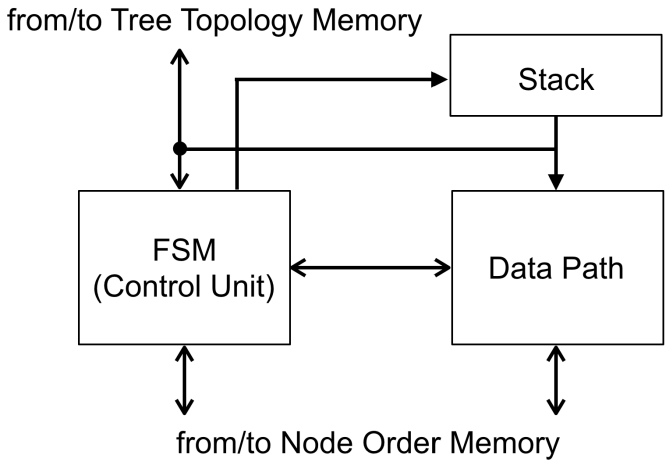


FIGURE 4.8: Second approach: general block diagram of the NOL unit

As can be seen from figure 4.8, the NOL unit is implemented as a Finite State Machine (FSM) with a Data Path and a Stack, from which the data is read or written in a last-in-first-out (LIFO) order. The NOL unit is connected to a Tree Topology Memory and a Node Order Memory, which is also a Stack. The following algorithm is used to list all the nodes in the tree:

1 Read the memory position of the chosen root node from the Tree Topology Memory (TTM)

2 Repeat until all nodes are listed.
Push each node visited into the Node Order Memory.

Case (Left Branch (LB), Right Branch (RB))

2.1 (Node, Node): Push RB into the Stack, read LB from TTM

2.2 (Node, Leaf): Read LB from TTM

2.3 (Leaf, Node): Read RB from TTM

2.4 (Leaf, Leaf): Pop a node and read it from TTM

As a result of the above mentioned algorithm, the last node visited becomes the first node to be read from the Node Order Memory. Thus, the desired post-order (reverse order of the visited nodes) is obtained. To illustrate this, an example is shown in figure 4.9.

Figure 4.9 shows a tree with 9 taxa and 8 nodes. The visited nodes, the actions taken and the contents of the stack for every clock cycle are detailed on the right of the tree. At the end, the Node Order Memory has the reverse order of the visited nodes. This is the final node order.

The NOL unit takes one clock cycle to list each node, so it requires a total of n clock cycles to list all the nodes, where n is the number of nodes.

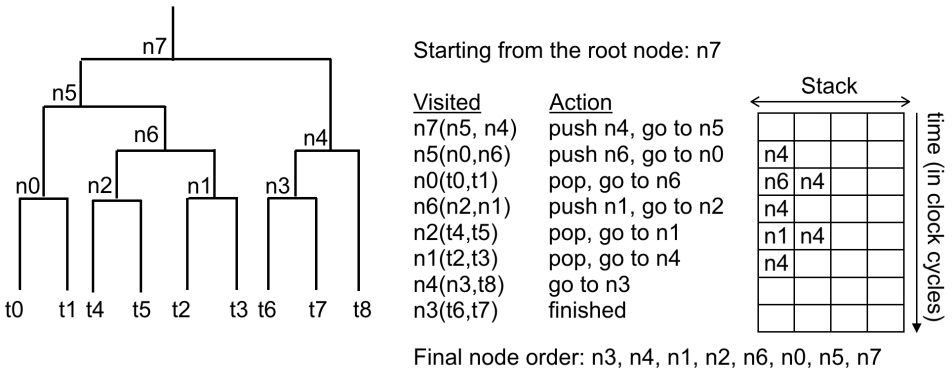


FIGURE 4.9: Second approach: example of the NOL unit listing process

4.3.4 First-, Second-pass and Rearrangement Evaluation (FSR) unit

This unit is the most important unit. It has three main tasks:

1 Doing a first-pass optimization following the order stored in the Node Order Memory (post-order)

2 Doing a second-pass optimization following the order stored in the Node Order Memory (reversed-order)

3 Evaluating all possible tree rearrangements following the order stored in the Node Order Memory (PNL-order)

Its general block diagram is shown in figure 4.10.

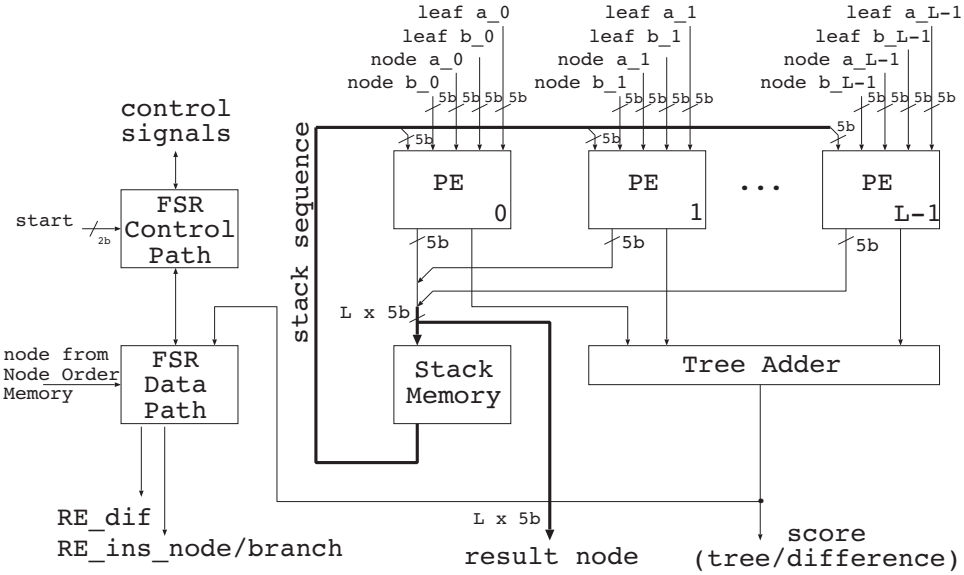


FIGURE 4.10: Second approach: general block diagram of the FSR unit

As seen in figure 4.10, the FSR unit is composed of L processing elements (PE), where L is equivalent to the number of DNA characters in a sequence (refer to figure 4.1), a Tree Adder, a Stack Memory, and a Control Logic unit with a Data Path. The inputs of the FSR unit are two leaves (taxa) from the Sequence Data Matrix Memory: *leaf a* and *leaf b*, two nodes from the Node Data Matrix Memory: *node a* and *node b*, and the node order from one of the Node Order memories. The Tree Adder is used at the end of the first-pass optimization to add all individual scores from each PE in order to obtain the final score of the tree. Furthermore, the FSR unit outputs the result node of all the PEs, so it can be stored in the Node Data Matrix Memory, and the best candidate node and branch for reinserting the subtree (ST) in the main tree (MT), along with the respective difference score: *RE_dif*, *RE_ins_node/branch*.

The FSR unit uses L PEs to implement the first-pass optimization algorithm (see section 2.1.3.1), the second-pass optimization algorithm (see section 2.1.3.2), and the neighbor trees rearrangement evaluation by using the Indirect Calculation of Tree Lengths (see section 2.2.2). These L PEs allow to process all the characters of two nodes or leaves (taxa) in parallel. The general block diagram of a PE is shown in figure 4.11.

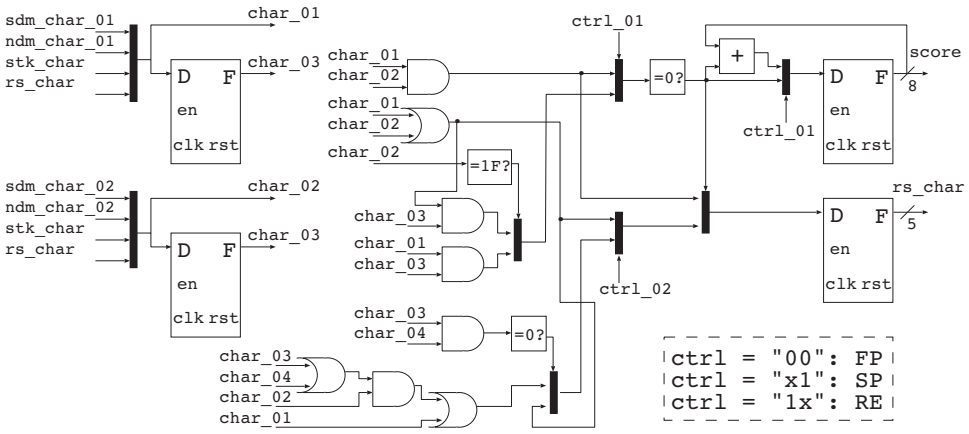


FIGURE 4.11: Second approach: general block diagram of the Processing Element (PE)

As can be seen from figure 4.11, the PE is composed of 4 D-type flip-flop-based registers of 5 bits each, and a collection of logic gates. These logic gates implement the three main tasks of the FSR unit described at the beginning of this section, but for a single DNA character. Depending on a control signal (ctrl in figure 4.11) the PE changes its functionality. The idea behind this is to share the same resources for the first, second-pass optimization, and for the rearrangement evaluation, since they need not to work at the same time. Each of these tasks works using pipeline. In the following sections we describe each one of them.

4.3.4.1 First-pass Optimization (FSR-FP)

The FSR-FP works by following the 4-stage pipelined algorithm listed below. This algorithm uses the Stack Memory shown in figure 4.10 and the sequence output from all the PEs.

Stage 01 Pop a node from the NOM and read it from the TTM.

Stage 02 Read LB and RB from the SDM1 and SDM2.

Stage 03 Do the following two tasks:

1. Case{Node(LB, RB)}

   (Leaf, Leaf): Push the node from Stage 04 into the Stack.

   (Node, Node): Pop a node from the Stack.
2. Do a first-pass optimization on the node.

Stage 04 Store the resulting node into NDM1.

To illustrate the above mentioned algorithm and how the 4-stage pipeline works, an example is given in figure 4.12 for a tree with 9 taxa and 8 nodes. The order in which the nodes are processed is already stored in the Node Order Memory (NOM).

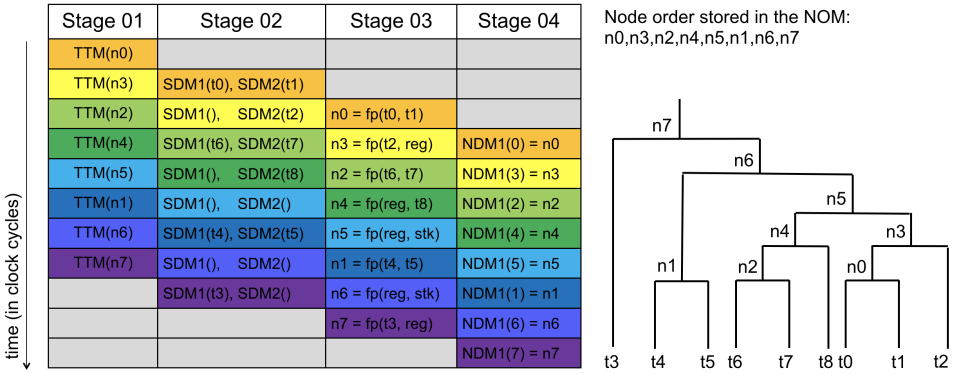


FIGURE 4.12: Second approach: example of the pipeline processing during the FSR-FP

In figure 4.12, TTM refers to the Tree Topology Memory, SDM to the Sequence Data Matrix Memory and NDM to the Node Data Matrix Memory. Both the SDM and the NDM are dual-port memories. In *Stage 03*, *fp(LB,RB)* refers to the first-pass optimization that is a function of the left and right branches of the node (see section 2.1.3.1), *reg* refers to the output sequence from all the PEs, and *stk* to the output sequence from the Stack Memory.

As can be seen from figure 4.12, each stage processes a node at a time. The FSR-FP finishes when all nodes in the Node Order Memory have been processed. Then, the final score of the tree is obtained after summing the individual results from all the PEs using the Tree Adder. The total execution time approximates $n + T$ clock cycles, where n is the number of nodes in the main tree (MT) or subtree (ST), and T the latency of the Tree Adder.

4.3.4.2 Second-pass Optimization (FSR-SP)

The FSR-SP works by following the 5-stage pipelined algorithm listed below. This algorithm uses the Stack Memory shown in figure 4.10 and the sequence output from all the PEs.

Stage 01 Pop a node from the NOM and read it from the TTM.

Stage 02 Read LB and RB from the SDM1/SDM2 or NDM1/NDM2.

Stage 03 Do the following two tasks:

1. Select the other operands of the node.
2. Read the node from NDM1.

Stage 04 Do a second-pass optimization on the node.

Stage 05 Store the resulting node into NDM2.

This pipeline works in two phases. In phase one, stages 1, 3 and 5 work in parallel. In the other phase, stages 2 and 4 work in parallel. It works in two phases, because not all operands can be read at the same time from the memories.

To illustrate the above mentioned algorithm and how the 5-stage pipeline works, an example is given in figure 4.13 for a tree with 9 taxa and 8 nodes. The order in which the nodes are processed is the reversed order from the first-pass optimization.

In figure 4.13, TTM refers to the Tree Topology Memory, SDM to the Sequence Data Matrix Memory and NDM to the Node Data Matrix Memory. Both the SDM and the NDM are dual-port memories. In *Stage 04*, *sp(PN, N, LB, RB)* refers to the second-pass optimization that is a function of the parent node (PN), the node (N) and the left (LB) and right (RB) branches (see section 2.1.3.2). In the same stage, *reg* refers to the output sequence from all the PEs, and *stk* to the output sequence from the Stack Memory.

The execution time approximates 2 clock cycles per node as the number of nodes increases. Since the Tree Adder is not used for the second-pass optimization, the total execution time approximates $2n$, where n is the number of nodes in the main tree (MT) or subtree (ST).

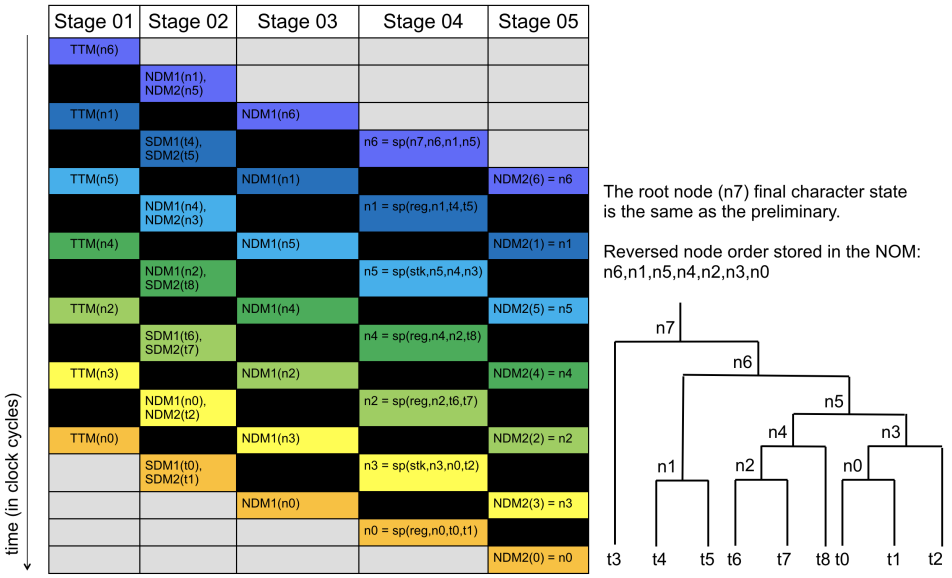


FIGURE 4.13: Second approach: example of the pipeline processing during the FSR-SP

4.3.4.3 Rearrangement Evaluation (FSR-RE)

The FSR-RE works by following the 4-stage pipelined algorithm listed below. This algorithm uses the Stack Memory shown in figure 4.10 and the sequence output from all the PEs. Moreover, this unit works in two phases. In phase one, stages 1 and 3 work. In the other, stages 2 and 4 work.

Initialization Read the root of the subtree (ST) from NDM1 or SDM1.

Stage 01 Pop a node from the NOM and read it from the TTM.

Stage 02 Read the node from NDM1 and LB from SDM2 or NDM2.

Stage 03 Do the following two tasks:

1. Read the node from NDM2 and RB from SDM2 or NDM2.
2. Evaluate the first tree rearrangement.

Stage 04 Evaluate the second tree rearrangement.

To illustrate the above mentioned algorithm and how the 4-stage pipeline works, an example is given in figure 4.14 for a tree with 9 taxa and 8 nodes. The order in which the nodes are processed is the order determined by the Progressive Neighborhood Listing (PNL) unit.

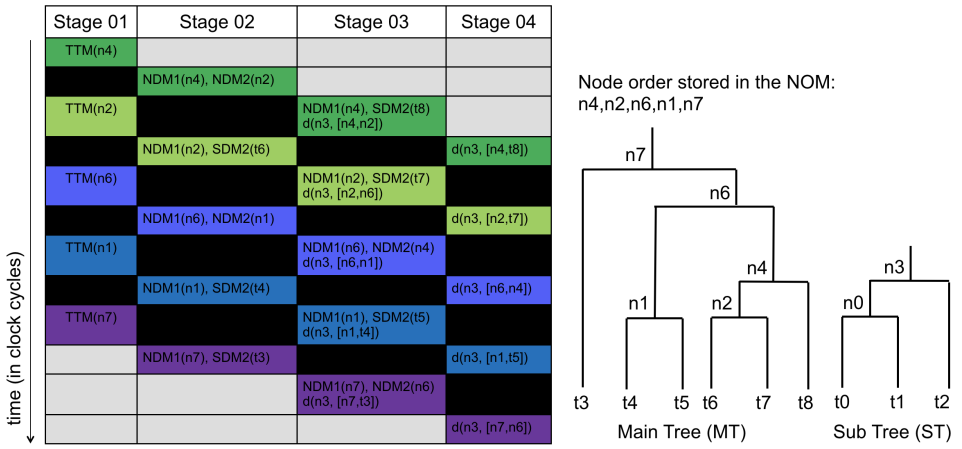


FIGURE 4.14: Second approach: example of the pipeline processing during the FSR-RE

In figure 4.14, TTM refers to the Tree Topology Memory, SDM to the Sequence Data Matrix Memory and NDM to the Node Data Matrix Memory. Both the SDM and the NDM are dual-port memories. In *Stage 03* and *Stage 04*, *d(SR, [LB, RB])* refers to the difference score that is a function of the root node of the subtree (SR) and the left (LB) and right (RB) branches of the MT (see section 2.2.2). In the same stage, *reg* refers to the output sequence from all the PEs, and *stk* to the output sequence from the Stack Memory. The difference score values are added by the Tree Adder from figure 4.10 to obtain the total difference score for the tree rearrangement. Finally, the rearrangement with the lowest score is kept as candidate to reinsert the subtree (ST) in the main tree (MT). For this purpose, inside the Data path of the FSR unit, a line buffer and some comparison registers are used.

Since two rearrangements are evaluated consecutively, the execution time will approximate 1 clock cycle per tree rearrangement as the number of nodes increases. The total execution approximates $2n + T$, where n is the number of nodes in the main tree (MT), and T is the latency of the Tree Adder.

4.3.5 Global Control (GC) unit

The Global Control (GC) logic unit is a Finite State Machine (FSM) that commands the other four units: TTU, PNL, NOL and FSR. The GC unit controls these units so they work in parallel as illustrated in figure 4.15, where TTMi refers to the port i of the Tree Topology Memory, and NOMi refers to the Node Order Memory i. FP refers to the first-pass optimization, SP to the second-pass optimization, and RE to the rearrangement evaluation.

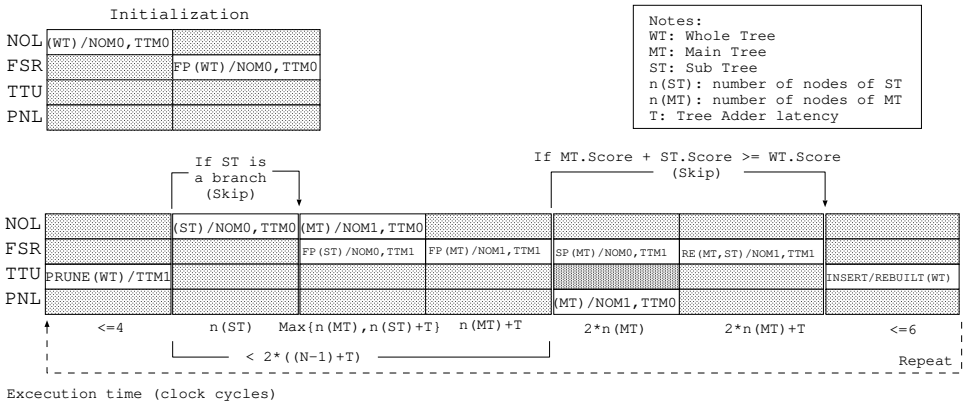


FIGURE 4.15: Second approach: execution flow of the proposed hardware architecture

As can be seen from figure 4.15, the NOL unit and the FSR-FP, the PNL unit and the FSR-SP work in parallel. This explains the use of a dual-port Tree Topology Memory and two Node Order Memories as shown in figure 4.2.

4.4 Implementation Results

In this section we show implementation results for four real-world biological datasets and simulation results for two other real-world biological datasets. The datasets were obtained from the repository of phylogenetic information TreeBASE [45], see table 4.2.

TABLE 4.2: Second approach: datasets used [45]

ID	M972	M2355	M3452	M3875	M17200	M2616
#taxa	155	150	116	228	326	330
#characters	355	829	1,157	1,435	1,434	1,711

4.4.1 Hardware Utilization and Performance Results

The hardware utilization and performance results are shown in table 4.3. The targeted FPGA is a Kintex-7 XC7K325T-FF2-900.

The implementation covers any of the first four datasets in table 4.2. In other words, problems up to $N = 1,024$ and $L = 1,435$ can be processed with this amount of hardware logical resources. The number of LUTs is almost proportional to$L$, the number of BRAMS to $L \times N$. For problems M17200 and M2616 a larger FPGA, e.g. XC7K410T, would be required.

TABLE 4.3: Second approach: implementation results on a Kintex-7 FPGA

Logic Utilization	Used	Available	Utilization
Number of Slices	23,610	50,950	46%
Number of Slice Registers	55,174	407,600	13%
Number of Slice LUTs	94,442	203,800	46%
Number of BRAMs (36 Kb)	402	445	90%
Maximum Frequency	163.826MHz		

4.5 Comparison and Performance Evaluation

Here we compare our hardware approach (second approach) with the first approach (see chapter 3) and with TNT (Tree analysis using New Technology) [20]. The results are summarized in table 4.4.

TABLE 4.4: Second approach: results for the local search

Dataset		First	Second	TNT
M972	Total time (ms)	62.2	11.27	890
	Time/tree (μs)	1.243	0.031	0.065
	Visited trees	50,031	364,177	13,637,086
	Best score	1548	1533	1543
M2355	Total time (ms)	55.5	9.9	500
	Time/tree (μs)	1.109	0.025	0.059
	Visited trees	50,032	400,368	8,497,522
	Best score	2749	2724	2771
M3452	Total time (ms)	49.9	9.64	180
	Time/tree (μs)	0.997	0.029	0.103
	Visited trees	50,046	329,025	1,749,117
	Best score	3633	3632	3624
M3875	Total time (ms)	82.8	27.76	510
	Time/tree (μs)	1.65	0.037	0.021
	Visited trees	50,172	760,180	23,818,061
	Best score	605	567	564
M17200*	Total time (ms)	No Data	53.56	2260
	Time/tree (μs)	No Data	0.019	0.046
	Visited trees	No Data	2798944	49662276
	Best score	No Data	4344	4340
M2616*	Total time (ms)	No Data	42.25	4700
	Time/tree (μs)	No Data	0.027	0.09
	Visited trees	No Data	1564515	53330179
	Best score	No Data	10003	10004

*Notes: Results for the last two datasets are simulation results.

To make the comparison as fair as possible, we use the traditional search of TNT based on SPR, and start from a random tree. This is the closest setting of TNT that resembles our algorithm, and the first one (see section 3.1). Moreover, since the total number of examined trees is not the same, we show the average execution time required per tree. The targeted PC and FPGAs were the following:

- PC: Intel Core-i7 860, 4GB RAM @ 2.80 Ghz (TNT)
- FPGA: Kintex-7 XC7K325T-FF2-900 @ 153.80 Mhz (first approach)
- FPGA: Kintex-7 XC7K325T-FF2-900 @ 156.25 Mhz (second approach)

Now, using the results from table 4.4, we show the acceleration rates obtained for the whole local search and for the evaluation of a single tree in figures 4.16 and 4.17, respectively.

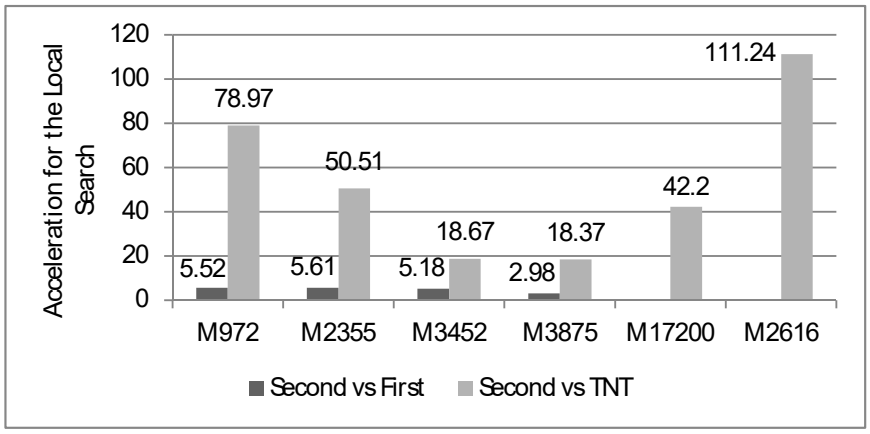


FIGURE 4.16: Second approach: acceleration rate for the local search

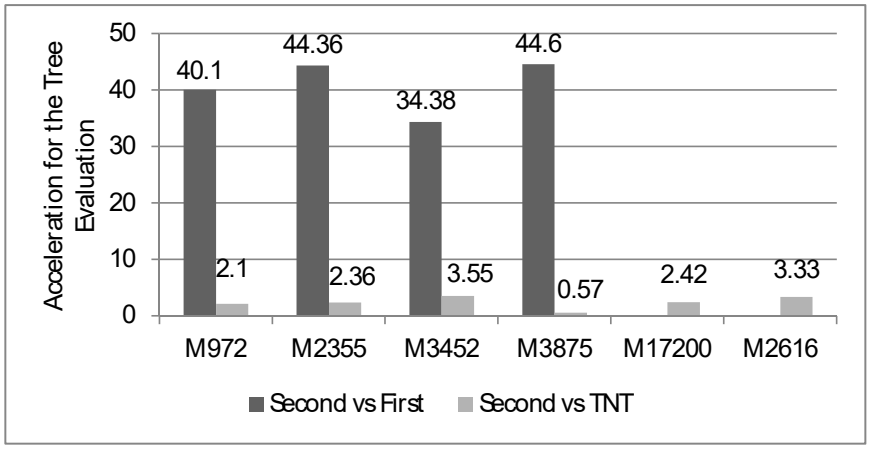


FIGURE 4.17: Second approach: acceleration rate for the tree evaluation

Compared to the first approach, our second approach provides an acceleration rate between 2 and 6 for the local search, and between 34 and 45 for the evaluation of a single tree. Here, we can make the following remarks. Although our second approach evaluates more trees than the first approach, there is still an acceleration rate for the local search. Moreover, the acceleration rate for the evaluation of a single tree is considerable. This is thanks to having applied the Indirect Calculation of Tree Lengths (ICTL) method.

On the other hand, in comparison to TNT, our approach yields an acceleration rate between 18 and 112 for the local search, and between 2 and 4 for the evaluation of a single tree, except for problem M3875, for which there is no acceleration. We think that the reason for this might lie in the particularities of problem M3875 itself, which make it easier for TNT. Naturally, there is a high acceleration rate for the local search, because our second approach evaluates less trees than TNT. However, it should still be noted that our second approach reaches a similar score (see table 4.4) faster than TNT. For problems M972, M2355 and M2616, the score is better, and for problems M3452, M3875 and M17200 it is worse. The acceleration rate for the evaluation of a single tree grows with the number of characters, except for problems M3452 and M3875. Problem M3452 has the highest acceleration rate (3.55), although it is not the problem with more characters. Problem M3875 has the lowest acceleration rate (0.57), although it is not the problem with fewer taxa.

4.6 Discussion

We compared execution times against our first approach (see chapter 3) and TNT. Compared to our first approach, our second approach is faster for all the problems. Compared to TNT, our second approach is faster for the local search and the evaluation of a single tree, except for problem M3875. The implementation of the Indirect Calculation of Tree Lengths (ICTL) method served to exceed by far the first approach. On the other hand, our second approach only slightly surpasses TNT. Although the algorithm implemented by TNT is not open, it is known that it implements its own version of the Indirect Calculation of Tree Lengths (ICTL) method. Furthermore, TNT uses multi-core processing (4 cores in this evaluation) with SIMD instructions.

The high performance we obtained is achieved by parallel processing of all the characters in the Sequence Alignment Matrix Memory or the Node Data Matrix Memory by using L Processing Elements (PEs) (see section 4.3.4), where L is the number of DNA characters in the sequence. As a result, the execution time for the tree

optimization, whether a first- or second-pass optimization, is independent of the number of characters in our approach.

From this approach, like in the first approach, we learned that to achieve faster execution times than TNT with a hardware implementation, it is necessary to consider that the score of the tree does not have to be recalculated from the start in each iteration, since only a small portion of the tree changes by the Subtree Pruning and Regrafting (SPR) process.

Chapter 5

Approach for the Alternative Second-pass

5.1 Algorithm Overview

The algorithm is based on the stochastic local search algorithm described in section 2.1.1. It uses the Progressive Tree Neighborhood, described in section 2.2.1, the Indirect Calculation of Tree Lengths, described in section 2.2.2, and the Alternative Second-pass Optimization, described in section 2.2.3.

The algorithm starts from a randomly generated tree in the search space, and tries to improve it on each iteration. For the initial tree, a list for all the branches is created. This list denotes which tree rearrangements have to be tried. Then, a first-pass optimization is done to calculate the initial score of the tree and the preliminary character states of the nodes. At each iteration of the algorithm a neighbor tree rearrangement replaces the current one if it has a lower score, i.e. better score. For this, in each iteration, the following steps are performed:

1 1.1 A branch to prune from the whole tree (WT) is randomly chosen from the list.

1.2 The main tree (MT) and the subtree (ST) derived from the previous pruning are created.

2 All possible branches from the MT where the ST can be reinserted are listed, according to the distance parameter from the Progressive Tree Neighborhood.

3 A first-pass optimization is done on the MT and the ST.

If the sum of the scores of the MT and the ST is greater (worse) than the current score, go to step 7.

4 An alternative second-pass optimization is done on the MT.

5 All rearrangements within the neighborhood are evaluated by the ICTL.

If the sum of the scores of the MT, the ST and the difference (*D.Score*) is greater (worse) than the current score, go to step 7.

6 6.1 The ST is inserted in the MT to create a new WT.
6.2 All branches are added to the list again.
6.3 Go to step 1.

7 7.1 The previous tree topology is reconstructed.
7.2 The chosen branch is removed from the list. If there are still branches in the list, go to step 1.

5.2 Phylogenetic Data Structure

For a given phylogenetic tree reconstruction problem consisting of N taxa, each of which has a sequence of L nucleobases, the sequence matrix is an N rows $\times$ L columns matrix. The characters in the sequences might include not only the DNA nucleobases Adenine (A), Cytosine (C), Guanine (G) and Thymine (T), but also the '-' character, which represents a gap, and the '?' character, which represents an undefined character. These are the six basic characters, but a combination of them is also possible thanks to the five-bit binary representation shown in table 5.1.

TABLE 5.1: Third approach: 5-bit representation for DNA characters [44]

DNA character	5-bit representation
'-'	00001
'A'	00010
'C'	00100
'G'	01000
'T'	10000
'?'	11111

As can be seen from table 5.1, each character is represented by a power of 2, from $2^0 = 1$ ($5'b00001$ for '-') to $2^4 = 16$ ($5'b10000$ for 'T'), except for '?', which is coded by the value 31 ($5'b11111$), since it can represent any character. Thanks to this five-bit representation, the union can be performed by the binary *OR* operation, and the intersection by the binary *AND* operation [44]. This eases the hardware implementation of the first-pass optimization algorithm, described in section 2.1.3.1 and shown in figure 2.2.

Hence, a memory of $N \times L \times 5$ bits is required to store the sequence alignment matrix. On the other hand, the tree topology shows the connections between the internal nodes of the tree and the taxa. A tree with N taxa has $N-1$ nodes, including the root node. Since the

tree is a binary tree, each node has a left branch and a right branch, and it has a parent node. Thus, the size of the memory required to store the tree topology is $(N-1) \times 3[log_2(N)]$ bits.

Finally, the size of the memory required to store the node character states is of $(N-1) \times L \times 5$ bits. Since we have to store the node character states for the normal, left and right path, we need three of these memories. For example, the tree topology, the sequence data matrix memory and the node data matrix memories for a tree with 6 taxa are represented in figure 5.1.

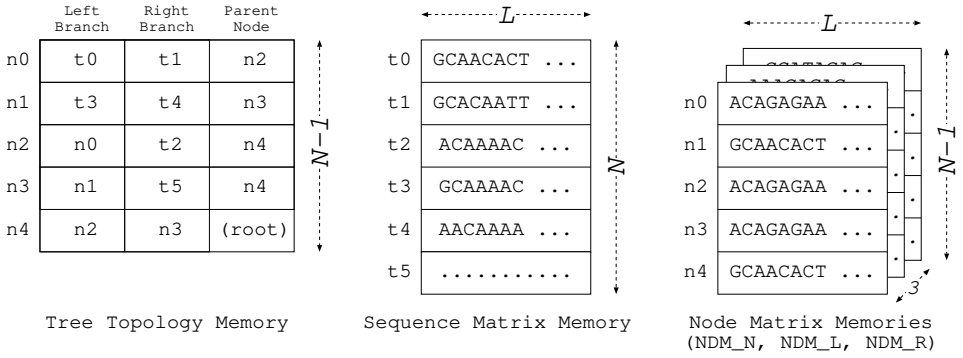


FIGURE 5.1: Third approach: memory data structure

NDM_N, NDM_R and NDM_L refer to the node data matrix memories used to store the normal, right and left path character states, respectively. Taxa are labeled according to their memory position on the sequence matrix memory. Likewise, nodes are labeled according to their memory position on the tree topology memory. Since both nodes and taxa appear on the same memory, we use an additional bit to distinguish between the two of them: 0 for a node and 1 for a taxon (singular form of taxa). The root node doesn't have a parent. Instead, a full sequence of 1s is used to identify it.

5.3 Proposed Hardware Architecture

In section 5.1, we mentioned the steps involved in each iteration of the algorithm. To design the hardware architecture, we considered that steps 1, 6 and 7 can be performed by the same hardware unit, since they are about modifying the tree topology. Similarly, steps 3, 4 and 5 can be performed by the same hardware unit, since they are about doing some operations on the nodes of the tree. However, steps 3 and 4 are divided into two additional steps each: listing the node order in which the first- and the alternative second-pass optimization is calculated, and performing the optimizations, respectively. Listing the node order is performed by a different hardware unit. Finally, step 2 is performed by a single hardware unit.

This leads to the following hardware units that we designed to implement the algorithm described in section 4.1:

1 Tree Topology Update (TTU) unit

2 Progressive Neighborhood Listing (PNL) unit

3 Node Order Listing (NOL) unit

4 First-, alternative Second-pass optimization and Rearrangement evaluation (FSR) unit

5 Global Control (GC) unit

The TTU unit is in charge of modifying the tree topology memory to reflect the changes produced by the Subtree Pruning and Reinserting (SPR) process. The PNL unit is in charge of listing all possible nodes in the main tree where the pruned branch with a subtree (ST) attached to it can be reinserted in the main tree (MT). The NOL unit has the task of listing the nodes of the tree for a post-order tree traversal. The FSR unit has the most important tasks, which are doing a first- and an alternative second-pass optimization, and evaluating all possible rearrangements according to the Indirect Calculation of Tree Lengths (ICTL) method. A general block diagram of the hardware architecture proposed is shown in figure 5.2.

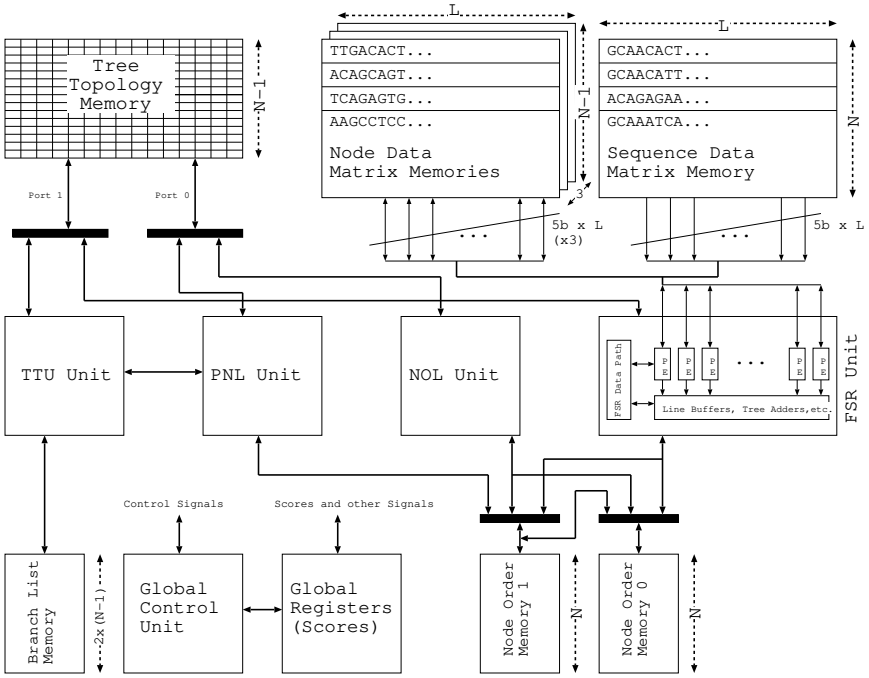


FIGURE 5.2: Third approach: general block diagram of the proposed hardware architecture

It consists of the following elements: the dual-port Tree Topology Memory (TTM), the dual-port Sequence Data Matrix Memory (SDM), the dual-port Node Data Matrix Memories (NDM_N, NDM_L and NDM_R), two Node Order Memories (NOM0 and NOM1), the Branch List Memory (BLM), the TTU unit, the PNL unit, the NOL unit, the FSR unit, and a Global Control unit with some registers. Black bars on the diagram make reference to multiplexers. In the following sections, we explain how each of this hardware units works.

5.3.1 Tree Topology Update (TTU) unit

The Tree Topology Update (TTU) unit is in charge of modifying the Tree Topology Memory to reflect the changes produced by the Subtree Pruning and Reinserting (SPR) process.

It has three main tasks:

1 Pruning a branch from the whole tree (WT) to create the main tree (MT) and the sub tree (ST).

2 Inserting the ST in the MT to create a new WT.

3 Rebuilding the previous WT when the score does not improve.

And two sub tasks:

1 Storing the value of the pruned branch and reinsertion branch.

2 Storing the values of the WT, MT and ST roots.

The general block diagram of the TTU unit is shown in figure 5.3.

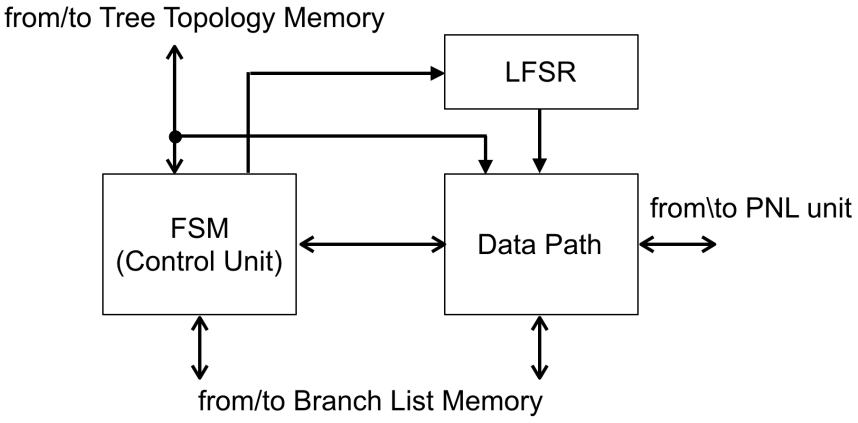
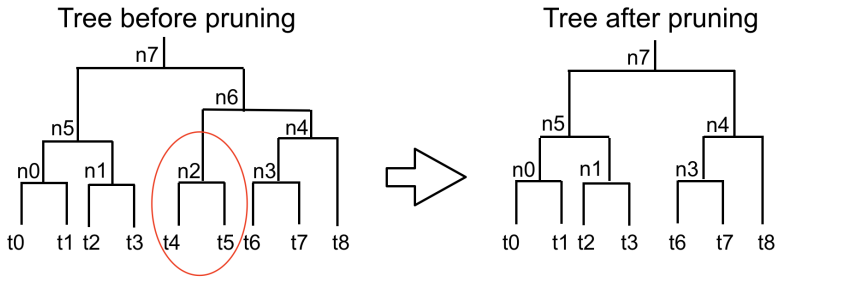


FIGURE 5.3: Third approach: general block diagram of the TTU unit

As can be seen from figure 5.3, the TTU unit is implemented as a Finite State Machine (FSM) with a Data Path. The FSM works basically as a memory controller that is used to modify the content of the Tree Topology Memory. In addition to this, it has a linear-feedback shift register (LFSR), which works as a pseudo-random number generator.

Regardless of the number of taxa, when a branch with a subtree (ST) attached to it is pruned from the tree, at most 2 nodes are modified. These nodes correspond to the parent node of the node where the branch is pruned from, and the node on the opposite branch to the pruned branch. An example of this is shown in figure 5.4.



Node	LB	RB	PN
n0	t0	t1	n5
n1	t2	t3	n5
n2	t4	t5	n6
n3	t6	t7	n4
n4	n3	t8	n6
n5	n0	n1	n7
n6	n2	n4	n7
n7	n5	n6	root

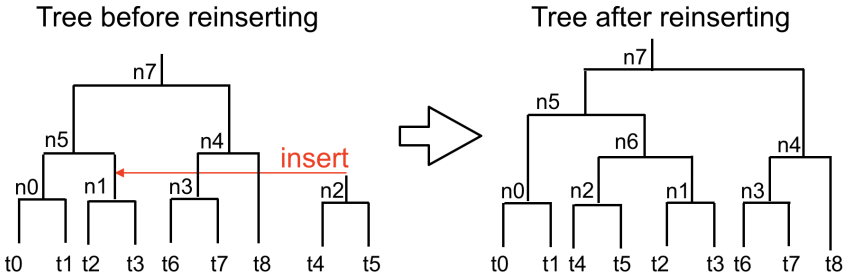
Node	LB	RB	PN
n0	t0	t1	n5
n1	t2	t3	n5
n2	t4	t5	n6
n3	t6	t7	n4
n4	n3	t8	n7
n5	n0	n1	n7
n6	n2	n4	n7
n7	n5	n4	root

FIGURE 5.4: Third approach: example of the nodes modified by the pruning process

Figure 5.4 shows a tree with 9 taxa. On the left is the tree before pruning the left branch (LB) of *n6*, and on the right is the tree after pruning it. This pruning process causes changes in the tree structure that affect a total of 2 nodes. Thus, 2 memory positions are modified in the Tree Topology Memory accordingly.

Similarly, when a branch with a ST attached to it is inserted in the MT, at most 3 nodes are modified. These nodes correspond to the node where the pruned branch comes from, and the nodes up and down where the pruned branch is reinserted. An example of this is shown in figure 5.5.

Figure 5.5 shows the same tree with 9 taxa after reinserting the left branch (LB) of *n6* on the right branch (RB) of *n5*. This reinsertion process causes changes in the tree structure that affect a total of 3 nodes. Thus, 3 memory positions are modified in the Tree Topology Memory accordingly.



Node	LB	RB	PN
n0	t0	t1	n5
n1	t2	t3	n5
n2	t4	t5	n6
n3	t6	t7	n4
n4	n3	t8	n7
n5	n0	n1	n7
n6	n2	n4	n7
n7	n5	n4	root

Node	LB	RB	PN
n0	t0	t1	n5
n1	t2	t3	n6
n2	t4	t5	n6
n3	t6	t7	n4
n4	n3	t8	n7
n5	n0	n6	n7
n6	n2	n1	n5
n7	n5	n4	root

FIGURE 5.5: Third approach: example of the nodes modified by the reinsertion process

The pruning process involves modifying at most 2 nodes. Since in 2 clock cycles a node gets modified, it takes 4 clock cycles at most. Similarly, inserting the sub tree involves modifying at most 3 nodes. Thus, it takes 6 clock cycles at most. Reconstructing the tree is equivalent to reverting the pruning process; hence, it takes also 4 clock cycles at most.

5.3.2 Progressive Neighborhood Listing (PNL) unit

The Progressive Neighborhood Listing (PNL) unit is in charge of listing all possible nodes in the main tree (MT) where the pruned branch with the subtree (ST) attached to it can be reinserted. It takes into account the distance parameter from the Progressive Tree Neighborhood (section 2.2.1). Its general block diagram is shown in figure 5.6.

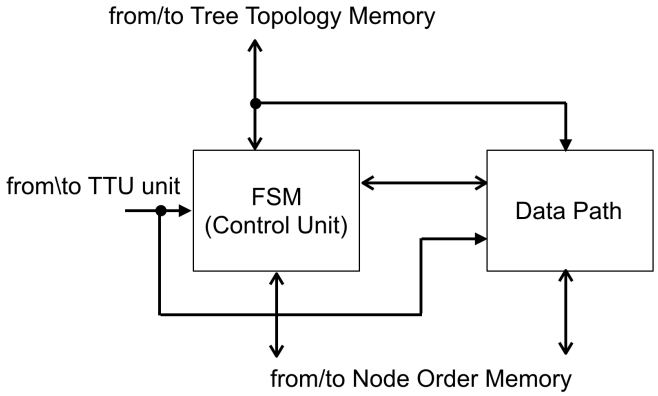


FIGURE 5.6: Third approach: general block diagram of the PNL unit

As can be seen from figure 5.6, the PNL unit is implemented as a Finite State Machine (FSM) with a Data Path. It is connected to the TTU unit, the Tree Topology Memory and one of the Node Order memories (see the general block diagram in figure 5.2).

It works as follows. Starting from the parent node of the node where the branch is pruned from, all nodes in the main tree (MT) are visited in order. Each time a node located at a relative distance less than or equal to the current value of the distance parameter is visited, the node is pushed into the Node Order Memory. Other nodes, i.e. those nodes that have a relative distance that exceeds the current value of the distance parameter, are not visited. As a result, only those nodes from the MT that comply with the distance parameter from the Progressive Tree Neighborhood are listed in the Node Order Memory. An example of this is shown in figure 5.7.

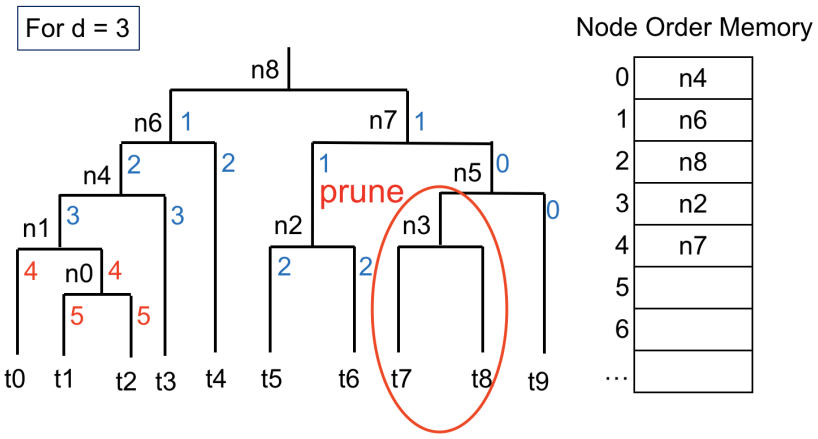


FIGURE 5.7: Third approach: example of the PNL listing

Figure 5.7 shows a tree with 10 taxa and 9 nodes with the relative distance values for all their branches. For this example, the current value of the distance parameter is equal to 3, so all branches that exceed this value are not visited by the PNL unit. At the end, the reversed order of the visited nodes is stored in the Node Order Memory, as shown on the right of figure 5.7. This listing is then used by the FSR unit to evaluate all the rearrangement trees within the neighborhood.

The PNL unit requires an execution time that depends on the number of nodes that have to be listed. However, since it requires two clock cycle to list each node at maximum, its execution time never exceeds $2 \times (N - 2)$ clock cycles for the worst- case scenario where all nodes except one have to be listed. Moreover, this unit works in parallel with the FSR unit, so its execution time does not add any delay.

5.3.3 Node Order Listing (NOL) unit

The Node Order Listing (NOL) unit has the task of listing all the nodes in the tree in a post-order. This tree can be the whole tree (WT), the main tree (MT) or the subtree (ST), depending on the root node chosen. The listing is then used by the First-, Second-pass and Rearrangement evaluation (FSR) unit. The general block diagram of the NOL unit is shown in figure 5.8.

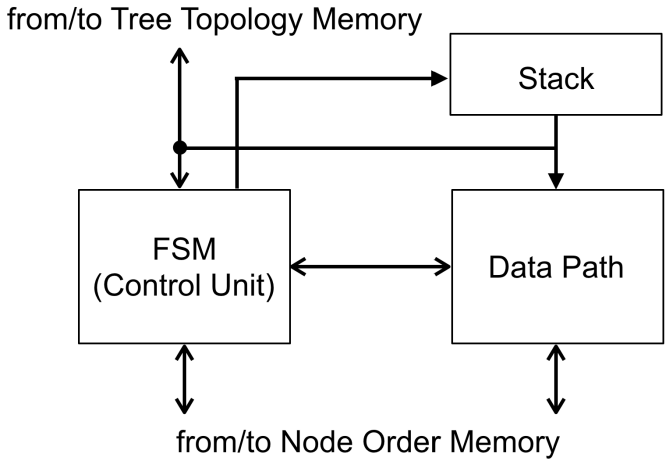


FIGURE 5.8: Third approach: general block diagram of the NOL unit

As can be seen from figure 5.8, the NOL unit is implemented as a Finite State Machine (FSM) with a Data Path and a Stack, from which the data is read or written in a last-in-first-out (LIFO) order. The NOL unit is connected to a Tree Topology Memory and a Node Order Memory, which is also a Stack. The following algorithm is used to list all the nodes in the tree:

1 Read the memory position of the chosen root node from the Tree Topology Memory (TTM)

2 Repeat until all nodes are listed.
Push each node visited into the Node Order Memory.

Case (Left Branch (LB), Right Branch (RB))

2.1 (Node, Node): Push RB into the Stack, read LB from TTM

2.2 (Node, Leaf): Read LB from TTM

2.3 (Leaf, Node): Read RB from TTM

2.4 (Leaf, Leaf): Pop a node and read it from TTM

As a result of the above mentioned algorithm, the last node visited becomes the first node to be read from the Node Order Memory. Thus, the desired post-order (reverse order of the visited nodes) is obtained. To illustrate this, an example is shown in figure 5.9.

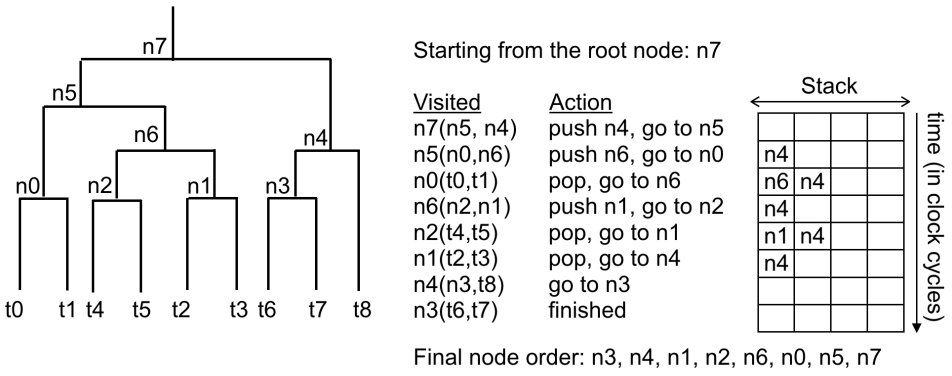


FIGURE 5.9: Third approach: example of the NOL unit listing process

Figure 5.9 shows a tree with 9 taxa and 8 nodes. The visited nodes, the actions taken and the contents of the stack for every clock cycle are detailed on the right of the tree. At the end, the Node Order Memory has the reverse order of the visited nodes. This is the final node order.

The NOL unit takes one clock cycle to list each node, so it requires a total of n clock cycles to list all the nodes, where n is the number of nodes.

5.3.4 First-, alternative Second-pass and Rearrangement evaluation (FSR) unit

This unit is the most important unit. It has three main tasks:

1 Doing a first-pass optimization following the order stored in the Node Order Memory (post-order)

2 Doing an alternative second-pass optimization following the order stored in the Node Order Memory (reversed-order)

3 Evaluating all possible tree rearrangements following the order stored in the Node Order Memory (PNL-order)

Its general block diagram is shown in figure 5.10.

As seen in figure 5.10, the FSR unit is composed of L processing elements (PE), where L is equivalent to the number of DNA characters in a sequence (refer to figure 5.1), two tree adders: Tree Adder 1 and Tree Adder 2, a Stack Memory, and a Control Logic unit with a Data Path. The inputs of the FSR unit are two leaves (taxa) from the Sequence Data Matrix Memory: *leaf a* and *leaf b*, two nodes from any of the Node Data Matrix Memories: *node a* and *node b*, and the node order from one of the Node Order memories.

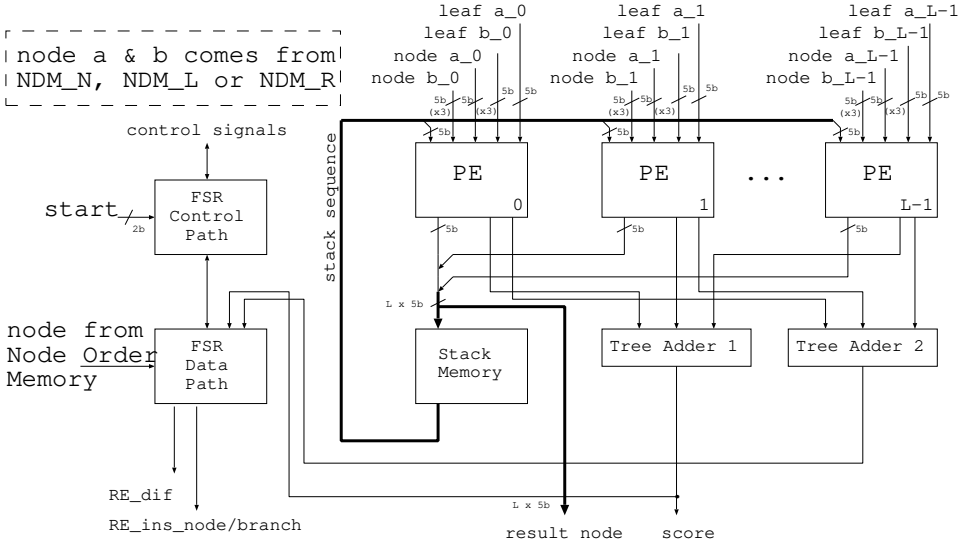


FIGURE 5.10: Third approach: general block diagram of the FSR unit

The Tree Adder 1 is used at the end of the first-pass optimization to add all individual scores from each PE in order to obtain the final score of the tree. In addition, both Tree Adder 1 and Tree Adder 2 are used during the tree rearrangement evaluation to add all individual difference scores from each PE in order to obtain the total difference score. Two tree adders are used, since there are two branches in a node: the left branch (LB) and the right branch (RB). Furthermore, the FSR unit outputs the result node of all the PEs, so it can be stored in the Node Data Matrix Memory, and the best candidate node and branch for reinserting the subtree (ST) in the main tree (MT), along with the respective difference score: *RE_dif*, *RE_ins_node/branch*.

The FSR unit uses L PEs to implement the first-pass optimization algorithm (see section 2.1.3.1), the alternative second-pass optimization algorithm (see section 2.2.3), and the neighbor tree rearrangement evaluation by using the Indirect Calculation of Tree Lengths (see section 2.2.2). These L PEs allow to process all the characters of two nodes or leaves (taxa) in parallel. The general block diagram of a PE is shown in figure 5.11.

As can be seen from figure 5.11, the PE is composed of 5 D-type flip-flop-based registers of 5 bits each, and a collection of logic gates. These logic gates implement the three main tasks of the FSR unit described at the beginning of this section, but for a single DNA character. Depending on a control signal (ctrl in figure 5.11) the PE changes its functionality. As should be noted, the circuit used for the alternative second-pass optimization is basically the same as the one for the first-pass optimization. This is due to the fact that the alternative second-pass optimization is equivalent to performing two first-pass optimizations.

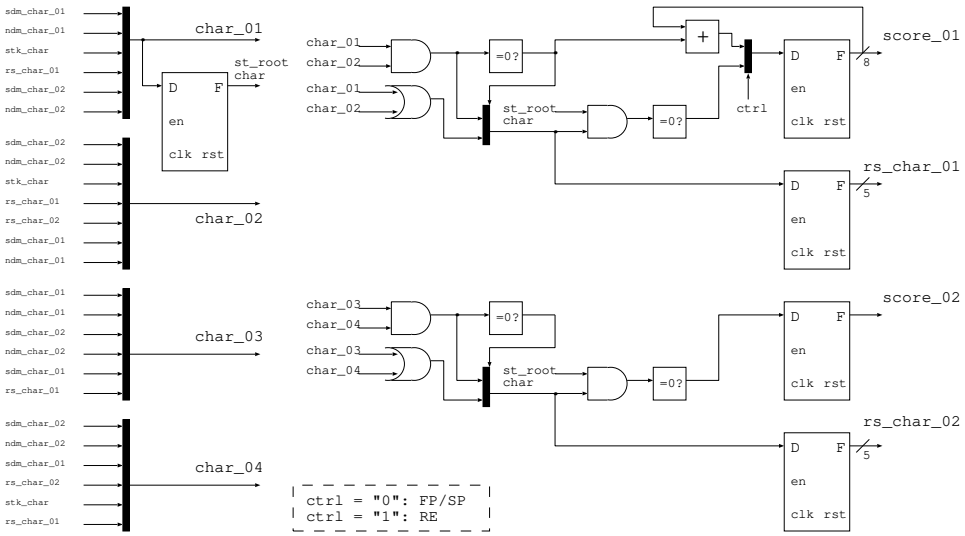


FIGURE 5.11: Third approach: general block diagram of the Processing Element (PE)

The idea behind having these three main tasks in one unit is to share the same resources for the first, alternative second-pass optimization, and for the rearrangement evaluation, since they need not to work at the same time. Each of these tasks works using pipeline. In the following sections we describe each one of them.

5.3.4.1 First-pass Optimization (FSR-FP)

The FSR-FP works by following the 4-stage pipelined algorithm listed below. This algorithm uses the Stack Memory shown in figure 5.10 and the sequence output from all the PEs.

Stage 01 Pop a node from the NOM and read it from the TTM.

Stage 02 Read LB and RB from the SDM1 and SDM2.

Stage 03 Do the following two tasks:

1. Case{Node(LB, RB)}

   (Leaf, Leaf): Push the node from Stage 04 into the Stack.

   (Node, Node): Pop a node from the Stack.

2. Do a first-pass optimization on the node.

Stage 04 Store the resulting node into NDM1.

To illustrate the above mentioned algorithm and how the 4-stage pipeline works, an example is given in figure 5.12 for a tree with 9 taxa and 8 nodes. The order in which the nodes are processed is already stored in the Node Order Memory (NOM).

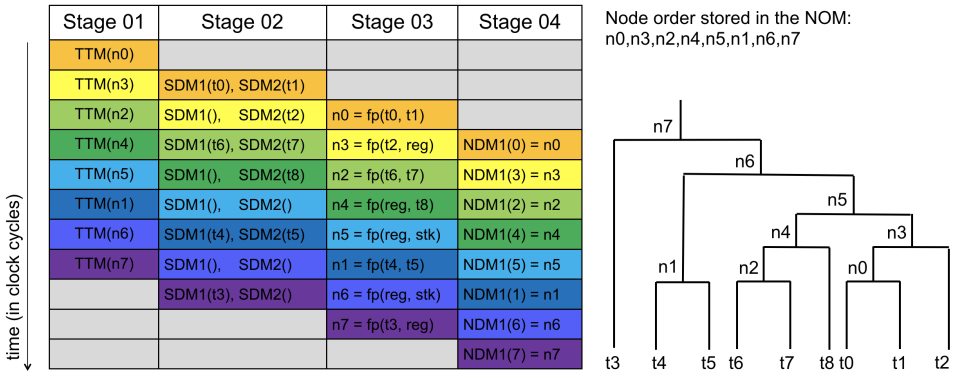


FIGURE 5.12: Third approach: example of the pipeline processing during the FSR-FP

In figure 5.12, TTM refers to the Tree Topology Memory, SDM to the Sequence Data Matrix Memory and NDM to the Node Data Matrix Memory. Both the SDM and the NDM are dual-port memories. In *Stage 03*, *fp(LB,RB)* refers to the first-pass optimization that is a function of the left and right branches of the node (see section 2.1.3.1), *reg* refers to the output sequence from all the PEs, and *stk* to the output sequence from the Stack Memory.

As can be seen from figure 5.12, each stage processes a node at a time. The FSR-FP finishes when all nodes in the Node Order Memory have been processed. Then, the final score of the tree is obtained after summing the individual results from all the PEs using the Tree Adder. The total execution time approximates $n + T$ clock cycles, where n is the number of nodes in the main tree (MT) or subtree (ST), and T the latency of the Tree Adder.

5.3.4.2 Alternative Second-pass Optimization (FSR-ASP)

The FSR-ASP works by following the 4-stage pipelined algorithm listed below. This algorithm uses the sequence output from all the PEs shown in figure 5.10.

Stage 01 Pop a node from the NOM and read it from the TTM.

Stage 02 Do the following two tasks:

1. Read LB and RB from SDM1/SDM2 or NDM_N1/NDM_N2
2. Read PN from NDM_L1 and NDM_R1.

Stage 03 Do an alternative second-pass optimization on the node.

Stage 04 Store the resulting nodes into NDM_L2 NDM _R2.

To illustrate the above mentioned algorithm and how the 4-stage pipeline works, an example is given in figure 5.13 for a tree with 9 taxa and 8 nodes. The order in which the nodes are processed is the reversed order from the first-pass optimization.

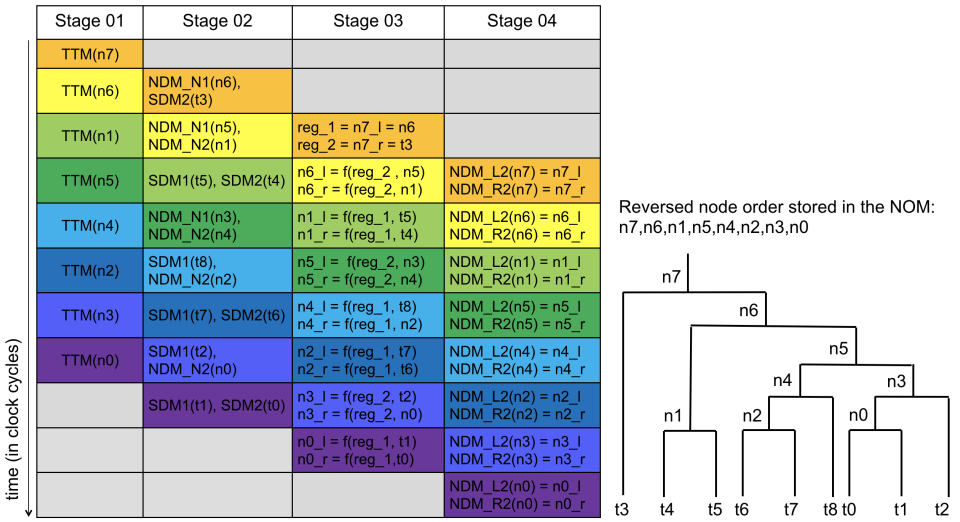


FIGURE 5.13: Third approach: example of the pipeline processing during the FSR-ASP

In figure 5.13, TTM refers to the Tree Topology Memory, SDM to the Sequence Data Matrix Memory and NDM to the Node Data Matrix Memory. Both the SDM and the NDM are dual-port memories. In *Stage 03*, *f(PN, LB/RB)* refers to the alternative second-pass optimization that is a function of the parent node (PN) and the left (LB) and right (RB) branches (see section 2.1.3.2). In the same stage, *reg_1* refers to the first output sequence from all the PEs, and *reg_2* to the second.

The execution time approximates n clock cycles, where n is the number of nodes in the main tree (MT). This is half the time required by the FSR-SP of the second approach. In this third approach, it has been reduced by half thanks to processing two nodes (for the left and the right paths) at the same time.

5.3.4.3 Rearrangement Evaluation (FSR-RE)

The FSR-RE works by following the 3-stage pipelined algorithm listed below. This algorithm uses the sequence output from all the PEs shown in figure 5.10.

Initialization Read the subtree (ST) root from NDM_N1 or SDM1.

Stage 01 Pop a node from the NOM and read it from the TTM.

Stage 02 Do the following two tasks:

1. Read the node from NDM_L1 and NDM _R1.
2. Read LB and RB from SDM1/SDM2 or NDM_N1/NDM_N2.

Stage 03 Evaluate the first and second tree rearrangement.

To illustrate the above mentioned algorithm and how the 3-stage pipeline works, an example is given in figure 5.14 for a tree with 9 taxa and 8 nodes. The order in which the nodes are processed is the order determined by the Progressive Neighborhood Listing (PNL) unit.

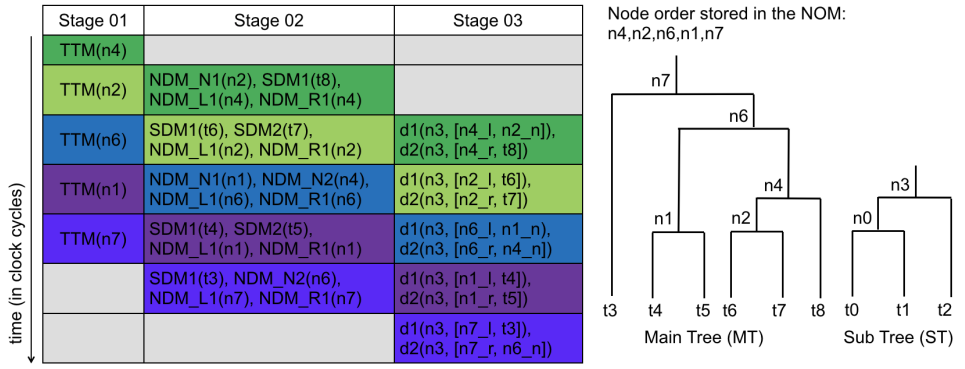


FIGURE 5.14: Third approach: example of the pipeline processing during the FSR-RE

In figure 5.14, TTM refers to the Tree Topology Memory, SDM to the Sequence Data Matrix Memory and NDM to the Node Data Matrix Memory. NDM_N, NDM_R and NDM_L refer to the NDMs used to store the normal, right and left path character states, respectively. Both the SDM and the NDM are dual-port memories. In *Stage 03*, *d(SR, [LB, RB])* refers to the difference score that is a function of the root node of the subtree (SR) and the left (LB) and right (RB) branches of the MT (see section 2.2.2).

The difference scores from the first and second tree rearrangements are summed using the two tree adders: Tree Adder 1 and Tree Adder 2 (see figure 5.10). Finally, the rearrangement with the lowest score is kept as candidate to reinsert the subtree (ST) in the main tree (MT). For this purpose, inside the Data path of the FSR unit, a line buffer and some comparison registers are used.

The execution time approximates $n+T$ clock cycles as the number of nodes increases, where n is the number of nodes in the main tree (MT), and T is the latency of the Tree Adder. This is almost half the time required by the FSR-RE of the second approach. In this third approach, it has been reduced by half thanks to processing two tree rearrangements at the same time.

5.3.5 Global Control (GC) unit

The Global Control (GC) logic unit is a Finite State Machine (FSM) that commands the other four units: TTU, PNL, NOL and FSR. The GC unit controls these units so they work in parallel as illustrated in figure 5.15, where TTMi refers to the port *i* of the Tree Topology Memory, and NOMi refers to the Node Order Memory *i*. FP refers to the first-pass optimization, ASP to the alternative second-pass optimization, and RE to the rearrangement evaluation.

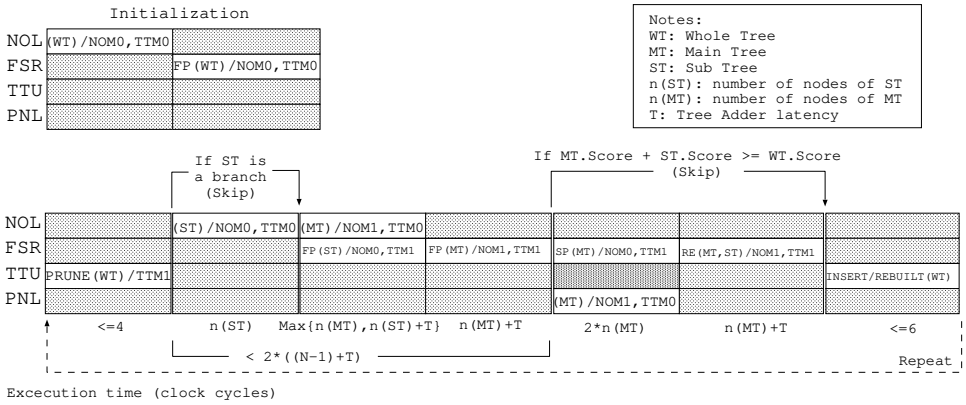


FIGURE 5.15: Third approach: execution flow of the proposed hardware architecture

As can be seen from figure 5.15, the NOL unit and the FSR-FP, the PNL unit and the FSR-ASP work in parallel. This explains the use of a dual-port Tree Topology Memory and two Node Order Memories as shown in figure 5.2.

5.4 Implementation Results

In this section we show implementation results for six real-world biological datasets. The datasets were obtained from the repository of phylogenetic information TreeBASE [45], see table 5.2.

TABLE 5.2: Third approach: datasets used [45]

ID	M972	M2355	M3452	M3875	M17200	M2616
#taxa	155	150	116	228	326	330
#characters	355	829	1,157	1,435	1,434	1,711

5.4.1 Hardware Utilization and Performance Results

The hardware utilization and performance results are shown in table 5.3. The targeted FPGA is a Virtex-7 XC7VX690T-FFG1157-2 FPGA.

TABLE 5.3: Third approach: implementation results on a Virtex-7 FPGA

Logic Utilization	Used	Available	Utilization
Number of Slices	39,627	108,300	37%
Number of Slice Registers	62,828	866,400	7%
Number of Slice LUTs	158,508	433,200	36%
Number of BRAMs (36 Kb)	1,074	1,470	73%
Maximum Frequency	167.792MHz		

The implementation covers any of the six datasets in table 5.2. In other words, problems up to $N = 1,024$ and $L = 1,711$ can be processed with this amount of hardware logical resources. The number of LUTs is almost proportional to$L$, the number of BRAMS to $L \times N$.

As can be seen from table 5.3, most of the resources used correspond to memory resources. BRAMs are used to store the Tree Topology Memory, the Sequence Data Matrix Memory, and the three Node Data Matrix memories (NDM_N, NDM_L and NDM_R).

5.5 Comparison and Performance Evaluation

Here we compare our hardware approach (third approach) with the second approach (see chapter 4) and with TNT (Tree analysis using New Technology) [20]. To make the comparison as fair as possible, we use the traditional search of TNT based on Subtree Pruning and Regrafting (SPR), and start from a random tree. This is the closest setting of TNT that resembles our algorithm, and the second one (see section 4.1). Moreover, since the total number of examined trees is not the same, we show the average execution time required for each tree. The results are summarized in table 5.4.

The targeted PC and FPGAs were the following:

- PC: Intel Core-i7 860, 4GB RAM @ 2.80 Ghz (TNT)
- FPGA: Kintex-7 XC7K325T-FF2-900 @ 156.25 Mhz (second approach)
- FPGA: Virtex-7 XC7VX690T-FFG1157 @ 156.25 Mhz (third approach)

TABLE 5.4: Third approach: results for the local search

Dataset		Second	Third	TNT
M972	Total time (ms)	11.27	9.06	890
	Time/tree (μs)	0.031	0.026	0.065
	Visited trees	364,177	349,012	13,637,086
	Best score	1,533	← same	1,543
M2355	Total time (ms)	9.9	6.18	500
	Time/tree (μs)	0.025	0.021	0.059
	Visited trees	400,368	294,360	8,497,522
	Best score	2,724	← same	2,771
M3452	Total time (ms)	9.64	8.09	180
	Time/tree (μs)	0.029	0.024	0.103
	Visited trees	329,025	337,014	1,749,117
	Best score	3,632	← same	3,624
M3875	Total time (ms)	27.76	18.02	510
	Time/tree (μs)	0.037	0.032	0.021
	Visited trees	760,180	562,514	23,818,061
	Best score	567	← same	564
M17200	Total time (ms)	53.56	44.82	2260
	Time/tree (μs)	0.019	0.016	0.046
	Visited trees	2,798,944	2,801,301	49,662,276
	Best score	4,344	← same	4,340
M2616	Total time (ms)	42.25	35.67	4700
	Time/tree (μs)	0.027	0.022	0.09
	Visited trees	1,564,515	1,621,566	53,330,179
	Best score	10,003	← same	10,004

Now, using these results, we show the acceleration rates obtained for the whole local search and for the evaluation of a single tree in figures 5.16 and 5.17, respectively.

Compared to the second approach, our third approach provides an acceleration rate between 1.18 and 1.6 for the local search, and between 1.16 and 1.23 for the evaluation of a single tree rearrangement. This acceleration rate obtained is thanks to the improvements done in the second-pass optimization by using an alternative optimization and in the rearrangement evaluation units. Although we reduced by half the individual execution times for both the FSR-ASP and FSR-RE units, the overall execution time is not reduced as much.

On the other hand, in comparison to TNT, our approach yields an acceleration rate between 22.25 and 131.76 for the whole local search, and between 2.5 and 4.29 for the evaluation of a single tree, except for problem M3875, for which there was no acceleration. We think that the reason for this might lie in the particularities of problem M3875 itself, which make it easier to evaluate for TNT.

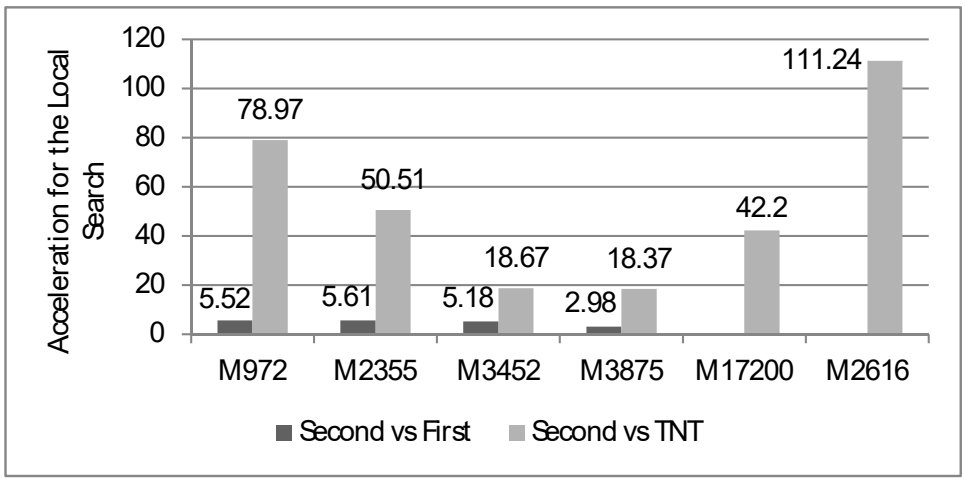


FIGURE 5.16: Third approach: acceleration rate for the local search

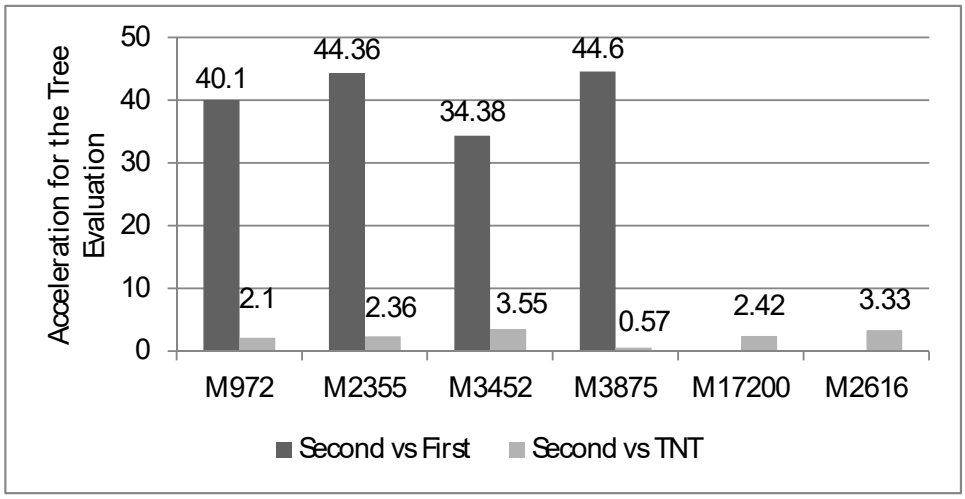


FIGURE 5.17: Third approach: acceleration rate for the tree evaluation

5.6 Discussion

We compared execution times against our second approach (refer to chapter 4) and TNT. Compared to our second approach, our third approach is faster for all problems. Compared to TNT, our third approach is faster for all problems except one: M3875. The implementation of the Alternative Second-Pass Optimization method served to improve the performance of our third approach over the previous one. However, as a disadvantage, this third approach requires three Node Data Matrix memories, while the second approach requires only one.

The acceleration rate should increase with the number of characters, because we process all the characters in parallel in our approach. On the contrary, a software approach like TNT requires to process characters serially; thus, taking more time as its number increases.

From this approach we learned that, although using the Alternative Second-Pass optimization method provides a slightly improvement in the acceleration rate, it requires a considerable amount of memory that might not be available depending on the targeted FPGA. For this reason, it is necessary to consider other optimization methods. In particular, we have to consider that not all node states of the tree have to be recalculated, since only a small portion of the tree changes by the Subtree Pruning and Regrafting (SPR) process.

Chapter 6

Approach for the Incremental Tree Optimization

6.1 Approach One

This approach is the fourth approach in total, but the first for the Incremental Tree Optimization.

6.1.1 Algorithm Overview

The algorithm is based on the stochastic local search algorithm described in section 2.1.1. It employs the Indirect Calculation of Tree Lengths, described in section 2.2.2, and the Incremental Tree Optimization, described in section 2.2.4.

The algorithm starts from a randomly generated tree in the search space, and tries to improve it on each iteration. For the initial tree, a list of all the branches in the tree is created. This list will denote which tree rearrangements have to be tried. Then, a complete first-pass optimization is done to calculate the initial score of the tree and the preliminary node character states. Following this, a complete second-pass optimization is done to obtain the final node character states. At each iteration of the algorithm a neighbor tree rearrangement replaces the current one if it has a lower score, i.e. better score. For this, in each iteration, the following steps are performed:

1 1.1 A branch to prune from the whole tree (WT) is randomly chosen from the list.

1.2 The main tree (MT) and the subtree (ST) derived from the previous pruning are created.

2 An incremental first-pass optimization is done on the MT.

If the sum of the scores of the MT and the ST is greater (worse) than the current score, go to step 6.

3 An incremental second-pass optimization is done on the MT.

4 All rearrangements within the neighborhood are evaluated by the ICTL.

If the sum of the scores of the MT, the ST and the difference (*D.Score*) is greater (worse) than the current score, go to step 6.

5 5.1 The ST is inserted in the MT to create a new WT.

5.2 An incremental first- and second-pass optimizations are done on the WT.

5.3 All branches are added to the list again. Go to step 1.

6 6.1 The previous tree topology is reconstructed.

6.2 The chosen branch is removed from the list. If there are still branches in the list, go to step 1.

This algorithm will always converge to a local optimum after all branches in the list have been tried. In other words, after it has been found that no rearrangement is better than the current tree.

6.1.2 Phylogenetic Data Structure

For a given phylogenetic tree reconstruction problem consisting of N taxa, each of which has a sequence of L nucleobases, the sequence matrix is an N rows $\times$ L columns matrix. The characters in the sequences might include not only the DNA nucleobases Adenine (A), Cytosine (C), Guanine (G) and Thymine (T), but also the '-' character, which represents a gap, and the '?' character, which represents an undefined character. These are the six basic characters, but a combination of them is also possible thanks to the five-bit binary representation shown in table 6.1.

TABLE 6.1: Fourth approach: 5-bit representation for DNA characters [44]

DNA character	5-bit representation
'-'	00001
'A'	00010
'C'	00100
'G'	01000
'T'	10000
'?'	11111

As can be seen from table 6.1, each character is represented by a power of 2, from $2^0 = 1$ ($5'b00001$ for '-') to $2^4 = 16$ ($5'b10000$ for 'T'), except for '?', which is coded by the value 31 ($5'b11111$), since it can represent any character. Thanks to this five-bit representation, the union can be performed by the binary *OR* operation, and the intersection by the binary *AND* operation [44]. This eases the hardware implementation of the first-pass optimization algorithm, described in section 2.1.3.1 and shown in figure 2.2.

Hence, a memory of $N \times L \times 5$ bits is required to store the sequence alignment matrix. On the other hand, the tree topology shows the connections between the internal nodes of the tree and the taxa. A tree with N taxa has $N-1$ nodes, including the root node. Since the tree is a binary tree, each node has a left branch and a right branch, and it has a parent node. Thus, the size of the memory required to store the tree topology is $(N-1) \times 3[log_2(N)]$ bits.

Furthermore, this approach requires the use of several memories for the implementation of the Incremental Tree Optimization method (section 2.2.4). First, it requires a memory to store the preliminary node character states and a memory to store the final node character states. The size of any of these memories is $(N-1) \times L \times 5$ bits. Next, it requires a memory to store the lengths of the nodes for all the nodes in the tree. The size of this memory is $(N-1) \times w$ bits, where w is the maximum number of bits required to store the length of the tree in binary codification. In this approach, 18 bits is enough to do so. Finally, it requires three buffer memories of equal size to their counterparts, one for each of the previously described memories. These buffer memories are used in case the pruning process is not successful and the tree topology has to be reconstructed. For example, the tree topology, the sequence data matrix memory, the node data matrix memories, and the node length memory for a tree with 6 taxa are represented in figure 6.1.

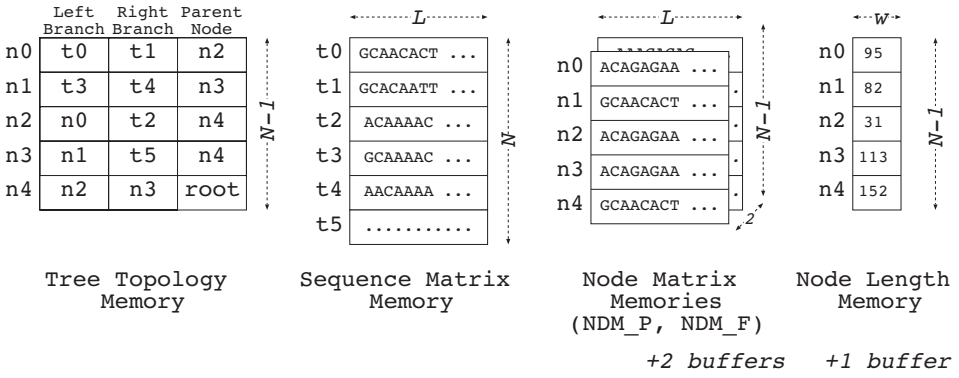


FIGURE 6.1: Fourth approach: memory data structure

NDM_P refers to the node data matrix memory for the preliminary character states; while NDM_F, for the final character states. Taxa are labeled according to their memory position on the sequence matrix memory. Likewise, nodes are labeled according to their memory position on the tree topology memory. Since both nodes and taxa appear on the same memory, we use an additional bit to distinguish between the two of them: 0 for a node and 1 for a taxon (singular form of taxa). The root node doesn't have a parent. Instead, a full sequence of 1s is used to identify it.

6.1.3 Proposed Hardware Architecture

In section 6.1.1, we mentioned the steps involved in each iteration of the algorithm. To design the hardware architecture, we considered that steps 1, 5.1 and 6.1 can be performed by the same hardware unit, since they are about modifying the tree topology. Similarly, steps 2, 3, 4 and 5.2 can be performed by the same hardware unit, since they are about doing some operations on the nodes of the tree. On the other hand, listing the node order for the complete optimization is performed by a single hardware unit.

This leads to the following hardware units that we designed to implement the algorithm described in section 6.1.1:

1 Tree Topology Update (TTU) unit

3 Node Order Listing (NOL) unit

4 First-, Second-pass and Rearrangement evaluation (FSR) unit

5 Global Control (GC) unit

The TTU unit is in charge of modifying the tree topology memory to reflect the changes produced by the Subtree Pruning and Reinserting (SPR) process. The NOL unit has the task of listing the nodes of the tree for a post-order tree traversal. The FSR unit has the most important tasks, which are doing a first- and second-pass optimization, whether complete or incremental, and evaluating all possible rearrangements according to the Indirect Calculation of Tree Lengths (ICTL) method. A general block diagram of the hardware architecture proposed is shown in figure 6.2.

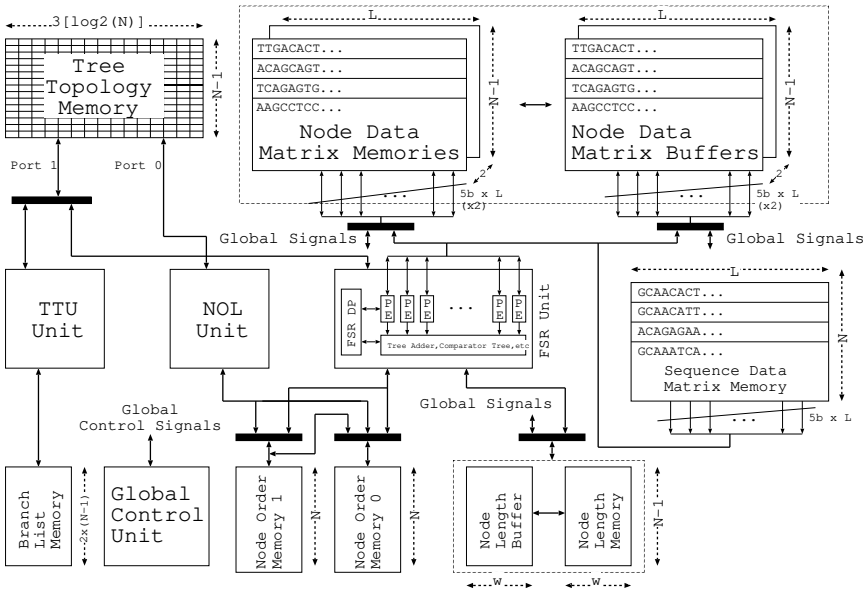


FIGURE 6.2: Fourth approach: general block diagram of the proposed hardware architecture

It consists of the following elements: the dual-port Tree Topology Memory, the dual-port Sequence Data Matrix Memory, the dual-port Node Data Matrix Memories, two Node Order Memories, the Node Length Memory, the Branch List Memory, the buffer memories, the TTU unit, the NOL unit, the FSR unit, and a Global Control unit. Black bars on the diagram make reference to multiplexers. In the following sections, we explain how each of these hardware units work.

6.1.3.1 Tree Topology Update (TTU) unit

The Tree Topology Update (TTU) unit is in charge of modifying the Tree Topology Memory to reflect the changes produced by the Subtree Pruning and Reinserting (SPR) process.

It has three main tasks:

1 Pruning a branch from the whole tree (WT) to create the main tree (MT) and the sub tree (ST).

2 Inserting the ST in the MT to create a new WT.

3 Rebuilding the previous WT when the score does not improve.

And two sub tasks:

1 Storing the value of the pruned branch and reinsertion branch.

2 Storing the values of the WT, MT and ST roots.

The general block diagram of the TTU unit is shown in figure 6.3.

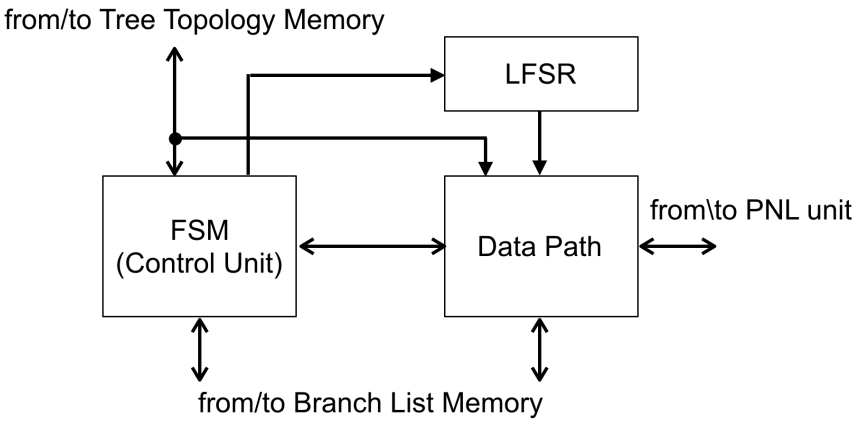
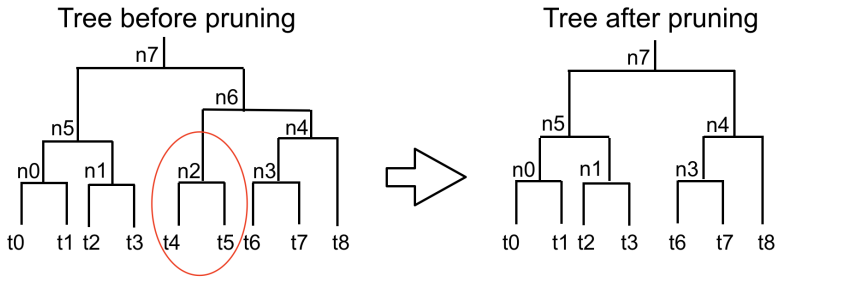


FIGURE 6.3: Fourth approach: general block diagram of the TTU unit

As can be seen from figure 6.3, the TTU unit is implemented as a Finite State Machine (FSM) with a Data Path. The FSM works basically as a memory controller that is used to modify the content of the

Tree Topology Memory. In addition to this, it has a linear-feedback shift register (LFSR), which works as a pseudo-random number generator.

Regardless of the number of taxa, when a branch with a subtree (ST) attached to it is pruned from the tree, at most 2 nodes are modified. These nodes correspond to the parent node of the node where the branch is pruned from, and the node on the opposite branch to the pruned branch. An example of this is shown in figure 6.4.



Node	LB	RB	PN
n0	t0	t1	n5
n1	t2	t3	n5
n2	t4	t5	n6
n3	t6	t7	n4
n4	n3	t8	n6
n5	n0	n1	n7
n6	n2	n4	n7
n7	n5	n6	root

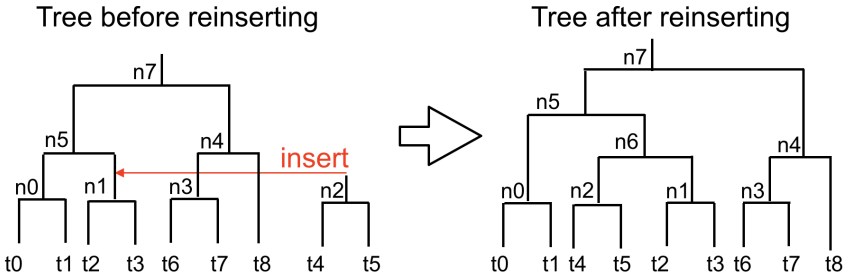
Node	LB	RB	PN
n0	t0	t1	n5
n1	t2	t3	n5
n2	t4	t5	n6
n3	t6	t7	n4
n4	n3	t8	n7
n5	n0	n1	n7
n6	n2	n4	n7
n7	n5	n4	root

FIGURE 6.4: Fourth approach: example of the nodes modified by the pruning process

Figure 6.4 shows a tree with 9 taxa. On the left is the tree before pruning the left branch (LB) of *n6*, and on the right is the tree after pruning it. This pruning process causes changes in the tree structure that affect a total of 2 nodes. Thus, 2 memory positions are modified in the Tree Topology Memory accordingly.

Similarly, when a branch with a ST attached to it is inserted in the MT, at most 3 nodes are modified. These nodes correspond to the node where the pruned branch comes from, and the nodes up and down where the pruned branch is reinserted. An example of this is shown in figure 6.5.

Figure 6.5 shows the same tree with 9 taxa after reinserting the left branch (LB) of *n6* on the right branch (RB) of *n5*. This reinsertion process causes changes in the tree structure that affect a total of 3 nodes. Thus, 3 memory positions are modified in the Tree Topology Memory accordingly.



Node	LB	RB	PN
n0	t0	t1	n5
n1	t2	t3	n5
n2	t4	t5	n6
n3	t6	t7	n4
n4	n3	t8	n7
n5	n0	n1	n7
n6	n2	n4	n7
n7	n5	n4	root

Node	LB	RB	PN
n0	t0	t1	n5
n1	t2	t3	n6
n2	t4	t5	n6
n3	t6	t7	n4
n4	n3	t8	n7
n5	n0	n6	n7
n6	n2	n1	n5
n7	n5	n4	root

FIGURE 6.5: Fourth approach: example of the nodes modified by the reinsertion process

The pruning process involves modifying at most 2 nodes. Since it takes 2 clock cycles to modify a node from the Tree Topology Memory, the execution time is of 4 clock cycles at most. Similarly, inserting the sub tree involves modifying at most 3 nodes. Thus, it takes 6 clock cycles at most. Rebuilding the tree is equivalent to reverting the pruning process; hence, this takes also 4 clock cycles at most.

6.1.3.2 Node Order Listing (NOL) unit

The Node Order Listing (NOL) unit has the task of listing all the nodes in the tree in a post-order. This tree can be the whole tree (WT), the main tree (MT) or the subtree (ST), depending on the root node chosen. The listing is then used by the First-, Second-pass and Rearrangement evaluation (FSR) unit. The general block diagram of the NOL unit is shown in figure 6.6.

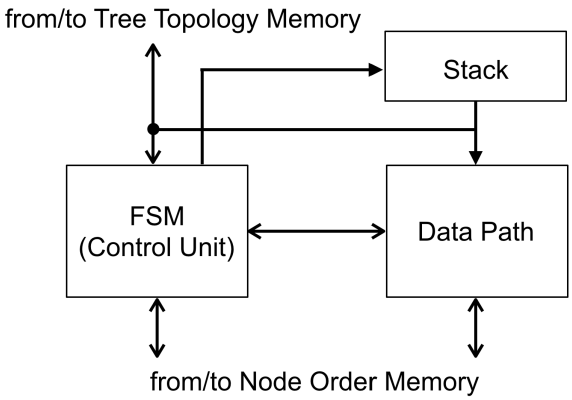


FIGURE 6.6: Fourth approach: general block diagram of the NOL unit

As can be seen from figure 6.6, the NOL unit is implemented as a Finite State Machine (FSM) with a Data Path and a Stack, from which the data is read or written in a last-in-first-out (LIFO) order. The NOL unit is connected to a Tree Topology Memory and a Node Order Memory, which is also a Stack. The following algorithm is used to list all the nodes in the tree:

1 Read the memory position of the chosen root node from the Tree Topology Memory (TTM)

2 Repeat until all nodes are listed.
Push each node visited into the Node Order Memory.

Case (Left Branch (LB), Right Branch (RB))

2.1 (Node, Node): Push RB into the Stack, read LB from TTM

2.2 (Node, Leaf): Read LB from TTM

2.3 (Leaf, Node): Read RB from TTM

2.4 (Leaf, Leaf): Pop a node and read it from TTM

As a result of the above mentioned algorithm, the last node visited becomes the first node to be read from the Node Order Memory. Thus, the desired post-order (reverse order of the visited nodes) is obtained. To illustrate this, an example is shown in figure 6.7.

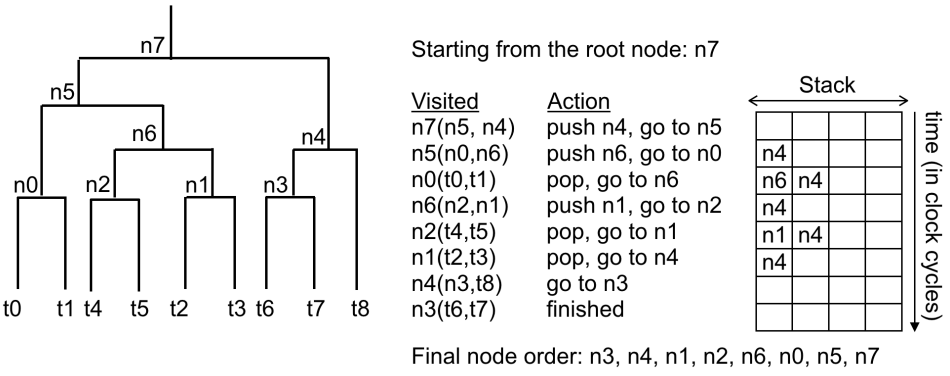


FIGURE 6.7: Fourth approach: example of the NOL unit listing process

Figure 6.7 shows a tree with 9 taxa and 8 nodes. The visited nodes, the actions taken and the contents of the stack for every clock cycle are detailed on the right of the tree. At the end, the Node Order Memory has the reverse order of the visited nodes. This is the final node order.

The NOL unit takes one clock cycle to list each node, so it requires a total of n clock cycles to list all the nodes, where n is the number of nodes.

6.1.3.3 First-, Second-pass and Rearrangement Evaluation (FSR) unit

This unit is the most important unit. It has five main tasks:

1 Doing a complete first-pass optimization following the order stored in the Node Order Memory 1 (post-order)

2 Doing a complete second-pass optimization following the order stored in the Node Order Memory 2 (reversed-order)

3 Doing an incremental first-pass optimization.

4 Doing an incremental second-pass optimization.

5 Evaluating all possible tree rearrangements.

Its general block diagram is shown in figure 6.8.

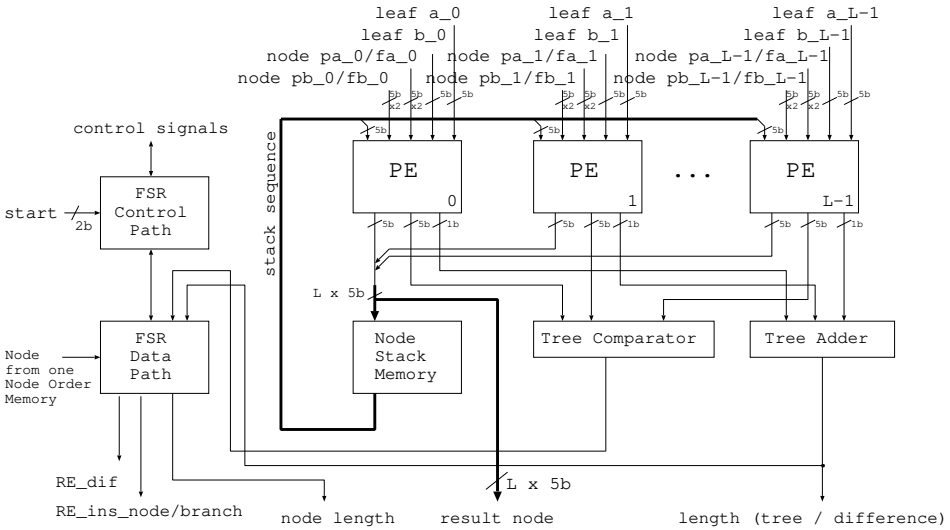


FIGURE 6.8: Fourth approach: general block diagram of the FSR unit

As seen in figure 6.8, the FSR unit is composed of L processing elements (PE), where L is equivalent to the number of DNA characters in a sequence (refer to figure 6.1), a Tree Adder, a Node Stack Memory, a Tree Comparator, and a Control Logic unit with a Data Path. The inputs of the FSR unit are two leaves (taxa) from the Sequence Data Matrix Memory: *leaf a* and *leaf b*, two nodes from the Preliminary Node Data Matrix Memory: *node pa* and *node pb*, two nodes from the Final Node Data Matrix Memory: *node fa* and *node fb*, and the node order from one of the Node Order memories.

The Tree Adder is used during the first-pass optimization to add all individual scores from the PEs in order to obtain the length of the node. In addition, it is used during the tree rearrangement evaluation to add all individual difference scores from the PEs in order to

obtain the total difference score. Furthermore, the FSR unit outputs the result node from the PEs, so it can be stored in the Preliminary or Final Node Data Matrix Memory. It outputs the best candidate node and branch for reinserting the subtree (ST) in the main tree (MT), and the respective difference score: *RE_dif*, *RE_ins_node/branch*. The Tree Comparator is used to compare the new node character state against the previous one during the incremental first-pass optimization to obtain the lowest node with unmodified character state. This node is used during the incremental second-pass optimization as the root node.

The FSR unit uses L PEs to implement the first-pass optimization algorithm (see section 2.1.3.1), the second-pass optimization algorithm (see section 2.2.3), and the neighbor tree rearrangement evaluation by using the Indirect Calculation of Tree Lengths (see section 2.2.2). These L PEs allow to process all the characters of two nodes or leaves (taxa) in parallel. The general block diagram of a PE is shown in figure 6.9.

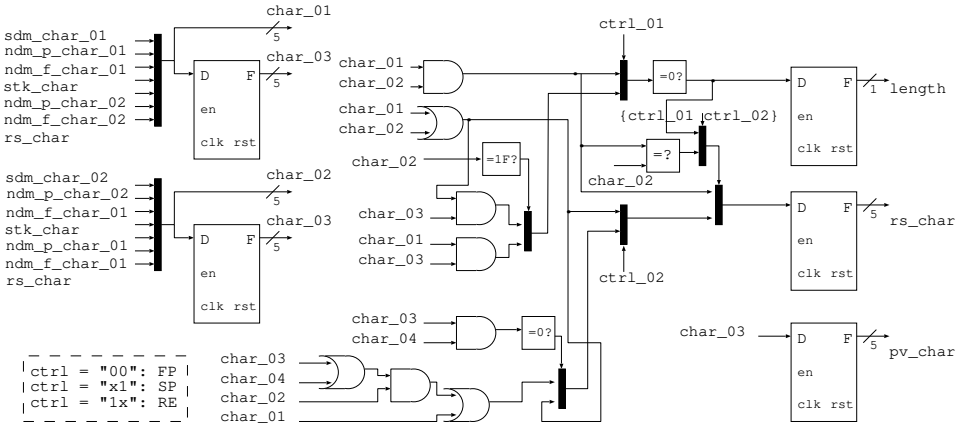


FIGURE 6.9: Fourth approach: general block diagram of the Processing Element (PE)

As can be seen from figure 6.9, the PE is composed of four D-type flip-flop-based registers of 5 bits each, one D-type flip-flop-based register of 1 bit, and a collection of logic gates. These logic gates implement the five main tasks of the FSR unit described at the beginning of this section, but for a single DNA character. Depending on a control signal (ctrl in figure 6.9) the PE changes its functionality.

The idea behind having these five main tasks in one unit is to share the same resources for the complete and incremental first- and second-pass optimization, and for the rearrangement evaluation, since they need not to work at the same time. Each of these tasks works using pipeline. In the following sections we describe each one of them.

6.1.3.3.1 Complete First-Pass Optimization (FSR-CFP) The FSR-CFP performs a first-pass optimization on all the nodes of the whole tree (WT). It uses the Node Stack Memory and the sequence output from all the PEs (see figure 6.8). The FSR-CFP works by following the extended 6-stage pipelined algorithm listed below.

Stage 01 Pop a node from the NOM and read it from the TTM.

Stage 02 Read LB and RB from the SDM1 and SDM2.

Stage 03 Do the following three tasks:

1. Case{Node(LB, RB)}
   (Leaf, Leaf): Push the node from Stage 04 into the Stack.
   (Node, Node): Pop a node from the Stack.
2. Do a first-pass optimization on the node.
3. Add the lengths of both branches.

Stage 04 Store the resulting node into NDM_P1.

... Line Buffer / Tree Adder ...

Stage 05 Calculate the node length.

Stage 06 Store the resulting node length into the NLM1.

TTM refers to the Tree Topology Memory, SDM to the Sequence Data Matrix Memory, NDM_P to the Preliminary Node Data Matrix Memory. Both the SDM and the NDM are dual-port memories. This is represented by the index attached at the end of the name of the memory, which refers to either the port 1 or 2, respectively. To illustrate the above mentioned algorithm and how the extended 6-stage pipeline works, an example is given in figures 6.10 and 6.11 for a tree with 9 taxa and 8 nodes. The order in which the nodes are processed is already stored in the Node Order Memory (NOM).

In figure 6.10, in *Stage 03*, *fp(LB,RB)* refers to the first-pass optimization that is a function of the left and right branches of the node (see section 2.1.3.1), *reg* refers to the output sequence from all the PEs, and *stk* to the output sequence from the Node Stack Memory. Furthermore, *LnbLB,RB* refers to the sum of the lengths of the left and right branches of the node. In *Stage 04*, *d[LB,RB]* refers to the difference score between the left and right branch for a single DNA.

In figure 6.11, in *Stage 05*, the total length of the node is calculated after the Tree Adder has summed the individual results from the PEs, and in *Stage 06* the node length is stored into the Node Length Memory (NLM).

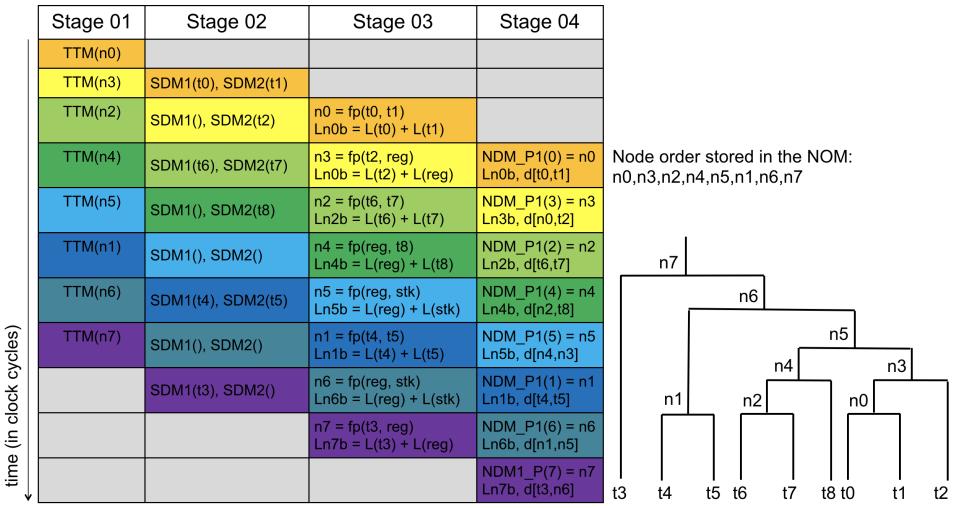


FIGURE 6.10: Fourth approach: example of the pipeline processing during the FSR-CFP (nodes)

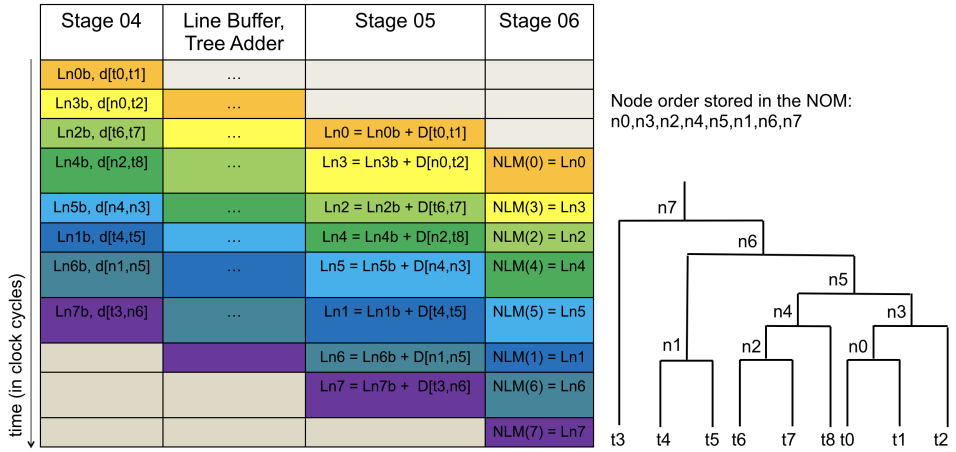


FIGURE 6.11: Fourth approach: example of the pipeline processing during the FSR-CFP (lengths)

As can be seen from figures 6.10 and 6.11, each stage processes a node at a time. The FSR-CFP finishes when all nodes in the Node Order Memory have been processed. The total execution time approximates $N + T$ clock cycles, where N is the number of nodes in the whole tree (WT), and T the latency of the Tree Adder.

6.1.3.3.2 Incremental First-Pass Optimization (FSR-IFP) The FSR-IFP performs a first-pass optimization only on those nodes that were affected by the Subtree Pruning and Reinserting (SPR). It works by following the extended 6-stage pipeline algorithm listed below.

Stage 01 Do the following two tasks:

1. Read the node from the TTM and the NDM_P1.
2. Read the length of the node from the NLM1.

Stage 02 Do the following three tasks:

1. Store the node and the length in the buffer memories.
2. Read either the LB or the RB from SDM1/NDM_P1.
3. Read either the length of the node in the LB or the RB from NLM1.

Stage 03 Do the following two tasks:

1. Do a first-pass optimization on the node.
2. Add the lengths of both branches.

Stage 04 Do the following two tasks:

1. Store the resulting node into NDM_P2.
2. Compare the new value of the node with its previous value.

... Line Buffer / Tree Adder ...

Stage 05 Calculate the node length.

Stage 06 Store the resulting node length into the NLM2.

TTM refers to the Tree Topology Memory, SDM to the Sequence Data Matrix Memory, NDM_P to the Preliminary Node Data Matrix Memory, and NLM to the Node Length Memory. All memories are dual-port memories. This is represented by the index attached at the end of its name, which refers to either the port 1 or 2, respectively.

The pipeline works in two phases. In phase one, stages 1 and 3 work in parallel. In the other phase, stages 2 and 4 work in parallel. In addition, stages 5 and 6 always work in different phases. The reason why the pipeline has to work in two phases is because not all operands can be read at the same time from the memories. To illustrate the above mentioned algorithm and how the extended 6-stage pipeline works, an example is given in figures 6.12 and 6.13 for a main tree (MT) with 6 taxa.

In figure 6.12, in stage *Stage 02*, NBM_P refers to the Node Buffer Memory, and LBM to the Length Buffer Memory. In *Stage 03*, *fp(LB,RB)* refers to the first-pass optimization that is a function of the left and right branches of the node (see section 2.1.3.1), and *reg* refers to the output sequence from all the PEs. Furthermore, *LnbLB,RB* refers to the sum of the lengths of the left and right branches of the node. In *Stage 04*, *d[LB,RB]* refers to the difference score between the left and right branch for a single PE.

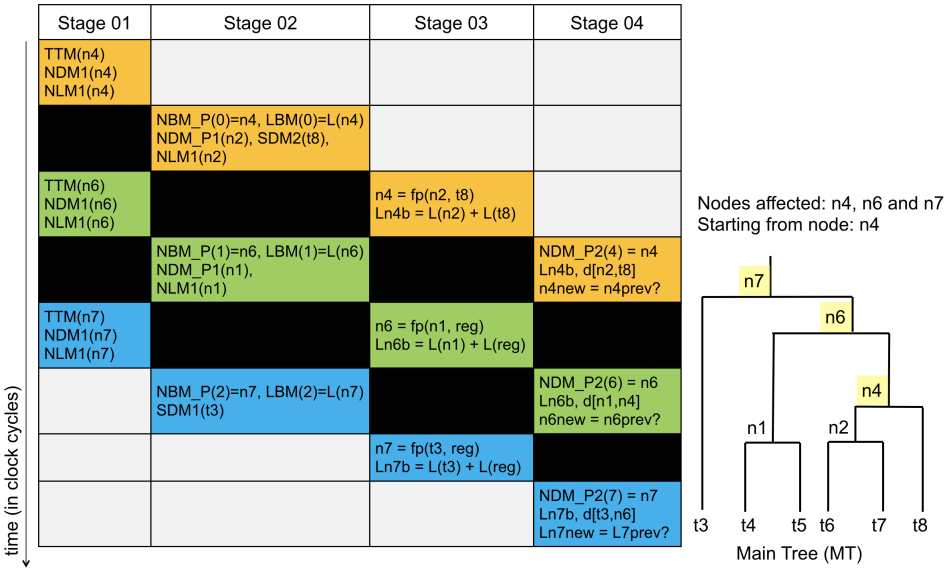


FIGURE 6.12: Fourth approach: example of the pipeline processing during the FSR-IFP (nodes)

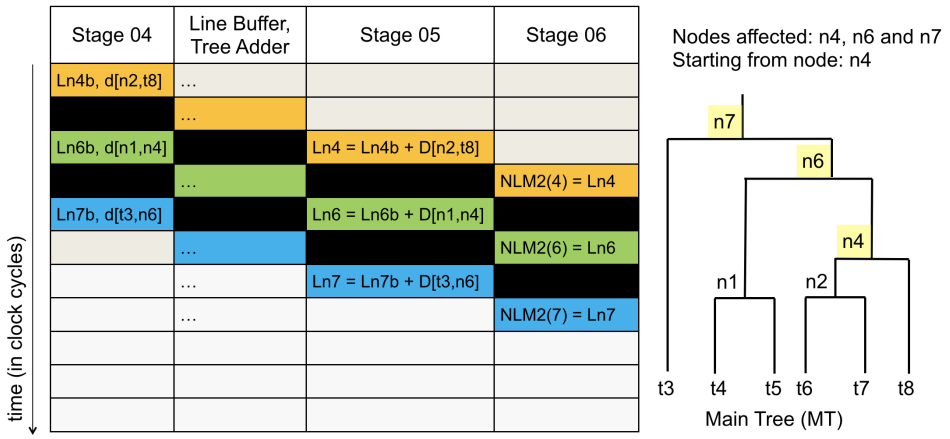


FIGURE 6.13: Fourth approach: example of the pipeline processing during the FSR-IFP (lengths)

In figure 6.13, in *Stage 05*, the total length of the node is finally calculated after the Tree Adder has summed the individual results from all the PEs, and in *Stage 06* the node length is stored into the Node Length Memory (NLM).

As can be seen from figures 6.12 and 6.13, each stage processes a node every two clock cycles. The FSR-IFP finishes when all nodes affected by the Subtree Pruning and Reinserting (SPR) have been processed. The total execution time approximates $2n + T$ clock cycles, where n is the number of nodes in the main tree (MT) or subtree (ST), and T the latency of the Tree Adder.

6.1.3.3.3 Complete Second-Pass Optimization (FSR-CSP) The FSR-CSP performs a second-pass optimization on all the nodes of the whole tree (WT). It uses the Node Stack Memory and the sequence output from all the PEs (see figure 6.8). The FSR-CSP works by following the 5-stage pipelined algorithm listed below.

Stage 01 Pop a node from the NOM and read it from the TTM.

Stage 02 Read LB and RB from SDM1/SDM2 or NDM_P1/NDM_P2.

Stage 03 Do the following two tasks:

1. Select the other operands of the node.
2. Read the node from NDM_P1.

Stage 04 Do a second-pass optimization on the node.

Stage 05 Store the resulting node into NDM_F2.

TTM refers to the Tree Topology Memory, SDM to the Sequence Data Matrix Memory, NDM_P to the Preliminary Node Data Matrix Memory, and NDM_F to the Final Node Data Matrix Memory. Both the SDM and the NDM are dual-port memories. This is represented by the index attached at the end of its name, which refers to either the port 1 or 2, respectively.

The pipeline works in two phases. In phase one, stages 1, 3 and 5 work in parallel. In the other phase, stages 2 and 4 work in parallel. It works in two phases, because not all operands can be read at the same time from the memories. To illustrate the above mentioned algorithm and how the 5-stage pipeline works, an example is given in figure 6.14 for a tree with 9 taxa and 8 nodes. The order in which the nodes are processed is the reversed order from the first-pass optimization.

In figure 6.14, in *Stage 04*, *sp(PN, N, LB, RB)* refers to the second-pass optimization that is a function of the parent node (PN), the node (N) and the left (LB) and right (RB) branches (see section 2.1.3.2). In the same stage, *reg* refers to the output sequence from all the PEs, and *stk* to the output sequence from the Stack Memory.

The execution time approximates 2 clock cycles per node as the number of nodes increases. Since the Tree Adder is not used for the second-pass optimization, the total execution time approximates $2N$, where N is the number of nodes in the whole tree (WT).

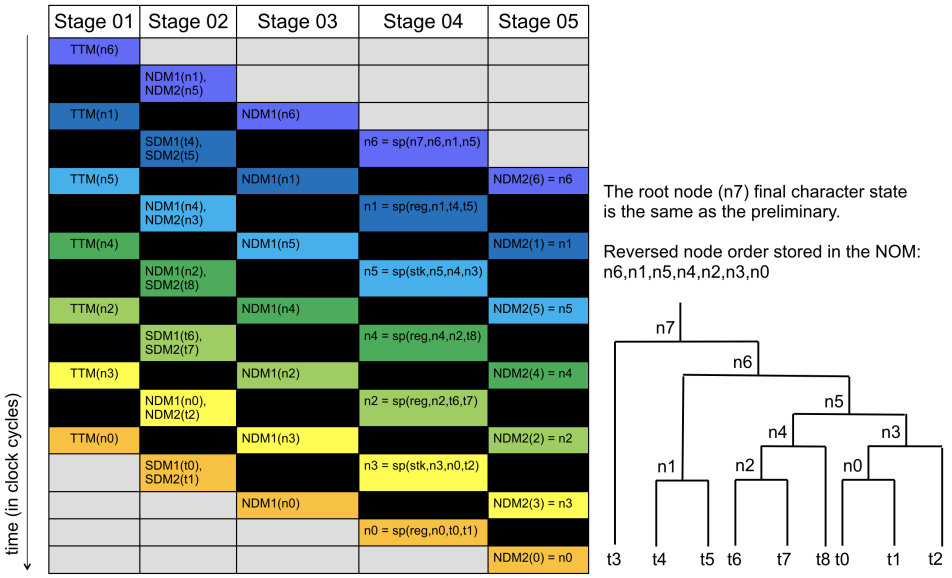


FIGURE 6.14: Fourth approach: example of the pipeline processing during the FSR-CSP

6.1.3.3.4 Incremental Second-Pass Optimization (FSR-ISP) The incremental second-pass optimization is equivalent to a complete second-pass optimization that starts from the lowest node with unmodified character states that was obtained during the incremental first-pass optimization. In other words, this node with unmodified character states is considered the root of the tree for the incremental second-pass optimization.

6.1.3.3.5 Rearrangement Evaluation (FSR-RE) The FSR-RE evaluates all the tree rearrangements within the neighborhood using the Indirect Calculation of Tree Lengths (ICTL). It uses the Stack Memory and the sequence output from all the PEs (see figure 6.8). The FSR-RE works by following the 4-stage pipelined algorithm listed below.

Initialization Read the root of the subtree from NDM_P1 or SDM1.

Stage 01 Pop a node from the NOM and read it from the TTM.

Stage 02 Do the following two tasks:

1. Read the node from NDM_F1.
2. Read the LB from SDM2 or NDM_F2.

Stage 03 Do the following three tasks:

1. Read the node from NDM_F1.
2. Read the RB from SDM2 or NDM_F2.
3. Evaluate the first tree rearrangement.

Stage 04 Evaluate the second tree rearrangement.

TTM refers to the Tree Topology Memory, SDM to the Sequence Data Matrix Memory, NDM_P to the Preliminary Node Data Matrix Memory, and NDM_F to the Final Node Data Matrix Memory. Both the SDM and the NDM are dual-port memories. This is represented by the index attached at the end of its name, which refers to either the port 1 or 2, respectively.

The pipeline works in two phases. In phase one, stages 1 and 3 work. In the other, stages 2 and 4 work. To illustrate the above mentioned algorithm and how the 4-stage pipeline works, an example is given in figure 6.15 for a tree with 9 taxa and 8 nodes. The order in which the nodes are processed is the same as for the complete second-pass optimization (i.e. the reversed order).

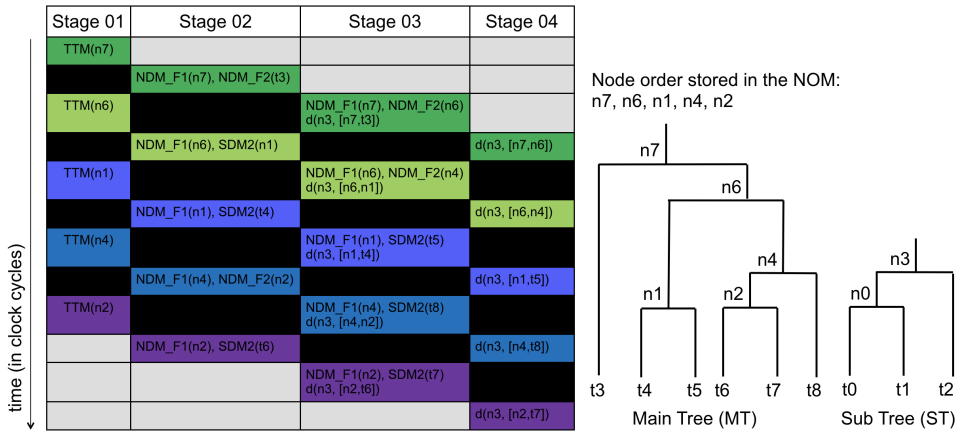


FIGURE 6.15: Fourth approach: example of the pipeline processing during the FSR-RE

In figure 6.15, in *Stage 03* and *Stage 04*, *d(SR, [CN, LB/RB])* refers to the difference score that is a function of the root node of the subtree (SR), the current node (CN), and the left branch (LB) or the right branch (RB) of the main tree (MT) (see section 2.2.2).

The difference score values are added by the Tree Adder from figure 6.8 to obtain the total difference score for every tree rearrangement. Finally, the rearrangement with the lowest score is kept as candidate to reinsert the subtree (ST) in the main tree (MT). For this purpose, inside the Data path of the FSR unit, a line buffer and some comparison registers are used.

Since two rearrangements are evaluated consecutively, the execution time will approximate 1 clock cycle per tree rearrangement as the number of nodes increases. The total execution approximates $2n + T$, where n is the number of nodes in the main tree (MT), and T is the latency of the Tree Adder.

6.1.3.4 Global Control (GC) unit

The Global Control (GC) logic unit is a Finite State Machine (FSM) that commands the other three units: TTU, NOL and FSR. The GC unit controls these units and supervises all memory accesses. In addition, the GC unit is in charge of restoring the Node Data Matrix and the Node Length memories by using their respective buffer memories, whenever the tree has to be reconstructed. The overall processing of our hardware approach executed by the GC unit is shown in figure 6.16. FULL-FP refers to the complete first-pass optimization; INC-FP refers to the incremental first-pass optimization; FULL-SP to the complete second-pass optimization; INC-SP to the incremental second-pass optimization; and, RE to the tree rearrangement evaluation.

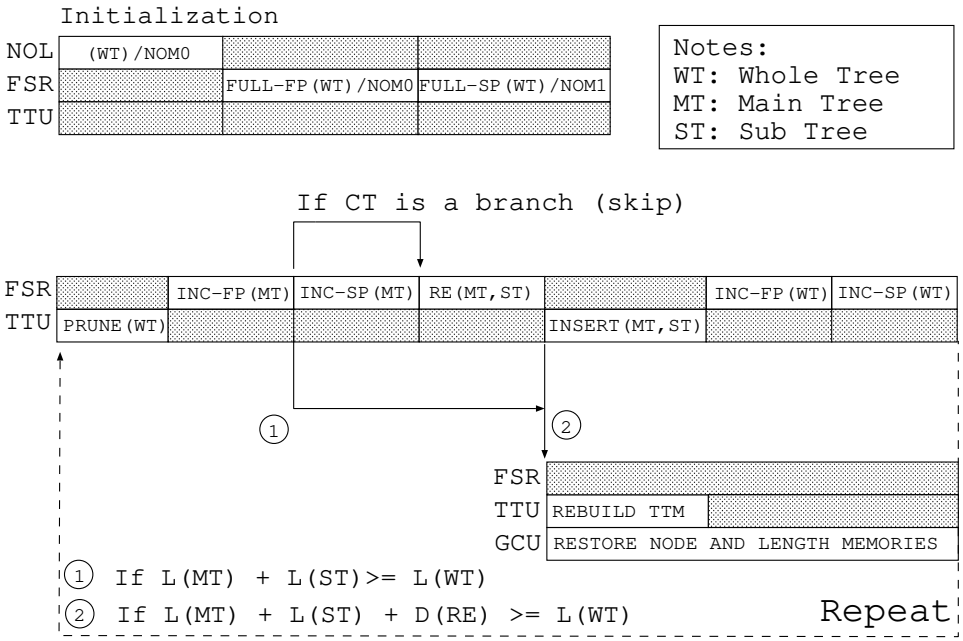


FIGURE 6.16: Fourth approach: execution flow of the proposed hardware architecture

As can be seen from figure 6.16, when the algorithm succeeds to find a better neighbor tree, the subtree (ST) is inserted in the main tree (MT) to create a new whole tree (WT). If, on the other hand, it fails to find a better neighbor tree, the previous tree topology is rebuilt. Along with this, the Node Data Matrix and Node Length memories are restored to their previous values.

6.1.4 Implementation Results

In this section we show simulation and implementation results for six real-world biological datasets. The datasets were obtained from the repository of phylogenetic information TreeBASE [45], see table 6.2.

TABLE 6.2: Fourth approach: datasets used [45]

ID	M972	M2355	M3452	M3875	M17200	M2616
#taxa	155	150	116	228	326	330
#characters	355	829	1,157	1,435	1,434	1,711

We implemented our system on a Virtex-7 XC7VX690T-FFG1157-2 FPGA. Results for the first two datasets were obtained directly from the implementation on the FPGA, while results for the last four datasets were obtained from post-synthesis simulations.

6.1.4.1 Hardware Utilization and Performance Results

The hardware utilization and performance results are shown in table 6.3. The targeted FPGA is a Virtex-7 XC7VX690T-FFG1157-2 FPGA.

TABLE 6.3: Fourth approach: implementation results on a Virtex-7 FPGA

Logic Utilization	Used	Available	Utilization
Number of Slices	23,968	108,300	22%
Number of Slice Registers	21,664	866,400	2%
Number of Slice LUTs	58,398	433,200	13%
Number of BRAMs (36 Kb)	577	1,470	39%
Maximum Frequency	158.767MHz		

The implementation covers the first two datasets in table 6.2. In other words, problems up to $N = 1,024$ and $L = 829$ can be processed with this amount of hardware logical resources. The number of LUTs is almost proportional to$L$, the number of BRAMS to $L \times N$.

As can be seen from table 6.3, most of the resources used correspond to memory resources. BRAMs are used to store the Tree Topology Memory, the Sequence Data Matrix Memory, the two Node Data Matrix memories (NDM_P, and NDM_P), the Node Length Memory, and the respective buffer memories.

6.1.5 Comparison and Performance Evaluation

Here we compare our hardware approach (fourth approach) against the second approach (see chapter 4), the third approach (see chapter 5), and TNT (Tree analysis using New Technology) [20]. To make the comparison as fair as possible, we use the traditional search of TNT based on Subtree Pruning and Regrafting (SPR), and start from a random tree. This is the closest setting of TNT that resembles our algorithms (see sections 4.1, 5.1, and 6.1.1). Moreover, since the total

number of examined trees is not the same, we show the average execution time required per tree. The results are shown in table 6.4.

TABLE 6.4: Fourth approach: results for the local search

Dataset		Second	Third	Fourth	TNT
M972	Total time (ms)	11.27	9.06	5.78	890
	Time/tree (μs)	0.031	0.026	0.014	0.065
	Visited trees	364,177	349,012	419,128	13,637,086
	Best score	1,533	$\leftarrow$ same	1,537	1,543
M2355	Total time (ms)	9.9	6.18	5.8	500
	Time/tree (μs)	0.025	0.021	0.013	0.059
	Visited trees	400,368	294,360	462,136	8,497,522
	Best score	2,724	$\leftarrow$ same	2,728	2,771
M3452	Total time (ms)	9.64	8.09	3.82	180
	Time/tree (μs)	0.029	0.024	0.014	0.103
	Visited trees	329,025	337,014	275,360	1,749,117
	Best score	3,632	$\leftarrow$ same	3,605	3,624
M3875	Total time (ms)	27.76	18.02	8.35	510
	Time/tree (μs)	0.037	0.032	0.01	0.021
	Visited trees	760,180	562,514	683,634	23,818,061
	Best score	567	$\leftarrow$ same	562	564
M17200	Total time (ms)	53.56	44.82	18.29	2260
	Time/tree (μs)	0.019	0.016	0.012	0.046
	Visited trees	2,798,944	2,801,301	1,595,264	49,662,276
	Best score	4,344	$\leftarrow$ same	4,342	4,340
M2616	Total time (ms)	42.25	35.67	26.25	4700
	Time/tree (μs)	0.027	0.022	0.010	0.09
	Visited trees	1,564,515	1,621,566	2,569,577	53,330,179
	Best score	10,003	$\leftarrow$ same	10,004	10,004

The targeted PC and FPGAs were the following:

- PC: Intel Core-i7 860, 4GB RAM @ 2.80 Ghz (TNT)
- FPGA: Kintex-7 XC7K325T-FF2-900 @ 156.25 Mhz (Second)
- FPGA: Virtex-7 XC7VX690T-FFG1157 @ 156.25 Mhz (Third)
- FPGA: Virtex-7 XC7VX690T-FFG1157 @ 150.00 Mhz (Fourth)

In table 6.4, we can see the total execution time required for the algorithm to converge, the execution time for the evaluation of a tree rearrangement, the number of trees visited during the search, and the best score obtained for each of the datasets. With regard to the best score, we achieve values similar to those of the second and third approach, and TNT. Now, using these results, we show in figure 6.17 the acceleration rate per tree rearrangement obtained.

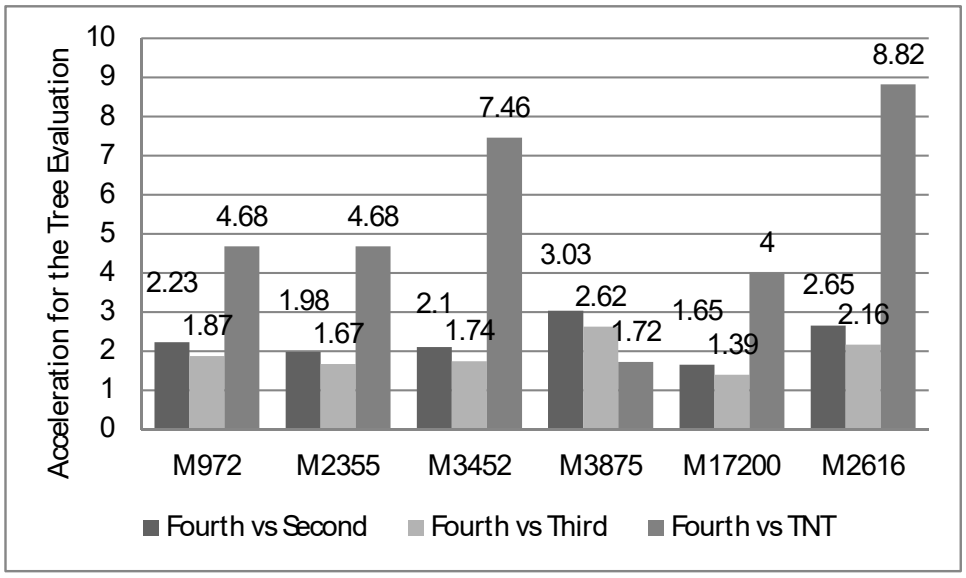


FIGURE 6.17: Fourth approach: acceleration rate for the tree evaluation

As can be seen from figure 6.17, we obtain acceleration rates per tree rearrangement between 1.65 and 3.03 against our second approach; between 1.39 and 2.62 against our third approach; and, between 1.72 and 8.82 against TNT (see figure 6.17).

6.1.6 Discussion

We compared execution times against our second approach (refer to chapter 4), the approach without the Incremental Tree Optimization method; against our third approach (refer to chapter 5), the approach that uses the Alternative Second-pass optimization; and TNT. Compared to both approaches and TNT, our fourth approach is faster. The acceleration rates achieved are thanks to a combination of using the Indirect Calculation of Tree Lengths (ICTL), the Incremental Tree Optimizaton, and the parallel and pipeline processing used in every unit in our approach. First, the Indirect Calculation of Tree Lengths method allows to evaluate all tree rearrangements within the neighborhood, and obtain the best one. Second, the Incremental Tree Optimization reduces the number of nodes to be updated during the first- and second-pass optimization. Thus, reducing significantly the time spent in optimizing the nodes of the tree. Last but not least, the parallel processing of all the characters in a sequence using L PEs makes it possibly to evaluate a node of the tree in one or at most two clock cycles. The acceleration rate against a software approach should increase theoretically with the number of characters, because in our approach we process all characters in parallel, while a software approach requires to process characters sequentially; thus, taking more time as its number increases.

With regard to hardware resources, our hardware approach can be applied for even larger problems as long as there are enough resources in the targeted FPGA; in particular, BRAM resources. Furthermore, this approach requires two node matrix memories (one for the preliminary character states and one for the final) and two buffer memories of equal size. Compared to this, the other previous approaches required less memory resources.

From this approach we learned that, although using the Incremental Tree Optimization method provides a significant improvement in the acceleration rates obtained, the use of additional buffer memories imposes a constraint on the size of the problems that can be implemented. This approach requires a considerable amount of memory that might not be available depending on the targeted FPGA. For this reason, it is necessary to consider an algorithm that does not use these additional buffer memories.

6.2 Approach Two

This approach is the fifth approach in total, but the second for the Incremental Tree Optimization.

6.2.1 Algorithm Overview

As was the case with the fourth approach, the algorithm proposed here is based on the stochastic local search (section 2.1.1), and uses the Indirect Calculation of Tree Lengths (section 2.2.2) and the Incremental Tree Optimization (section 2.2.4). However, this time, reconstructing the previous tree topology is done by reinserting the sub tree (ST) back into the main tree (MT) on the same branch from which it was pruned. We chose this approach, rather than reconstructing the tree topology in the same way as in the fourth approach, to avoid increasing the amount of memory needed for the Incremental Tree optimization.

The algorithm starts from a randomly generated tree in the search space, and tries to improve it on each iteration. For the initial tree, a list of all the branches in the tree is created. This list will denote which tree rearrangements have to be tried. Then, a complete first-pass optimization is done to calculate the initial score of the tree and the preliminary node character states. Following this, a complete second-pass optimization is done to obtain the final node character states. At each iteration of the algorithm a neighbor tree rearrangement replaces the current one if it has a lower score, i.e. better score. For this, in each iteration, the following steps are performed:

1 1.1 A branch to prune from the whole tree (WT) is randomly chosen from the list.

1.2 The main tree (MT) and the subtree (ST) derived from the previous pruning are created.

2 An incremental first-pass optimization is done on the MT.

If the sum of the scores of the MT and the ST is greater (worse) than the current score, go to step 6.

3 An incremental second-pass optimization is done on the MT.

4 All rearrangements within the neighborhood are evaluated.

If the sum of the scores of the MT, the ST and the difference (*D.Score*) is greater (worse) than the current score, go to step 6.

5 5.1 The ST is inserted in the MT in the branch that gives the best tree rearrangement to create a new WT.

5.2 An incremental first-pass optimization is done on the WT,

followed by an incremental second-pass optimization.

5.3 All branches are added to the list again. Go to step 1.

6 6.1 The previous tree topology is reconstructed by reinserting the ST in the MT in the same branch from which it was pruned.

6.2 An incremental first-pass optimization is done on the WT, followed by an incremental second-pass optimization.

6.3 The chosen branch is removed from the list. If there are still branches in the list, go to step 1. Otherwise, finish.

This algorithm will always converge to a local optimum after all branches in the list have been tried. In other words, after it has been found that no tree rearrangement is better than the current one.

6.2.2 Phylogenetic Data Structure

The data structure used to store the phylogenetic information is the same as for the fourth approach. However, since the previous tree topology is now reconstructed by reinserting the sub tree (ST) back into the main tree (MT) on the same branch from which it was pruned, no additional buffer memories are needed.

For a given phylogenetic problem consisting of N taxa, each of which has a sequence of L nucleobases, the sequence alignment matrix is an N rows $\times$ L columns matrix. The characters in the sequences include not only the DNA nucleobases Adenine (A), Cytosine (C), Guanine (G) and Thymine (T), but also the '-' character, which represents a gap, and the '?' character, which represents an undefined character. These are the six basic characters, but a combination of them is also possible thanks to the five-bit binary representation shown in table 6.1 of section 6.1.2.

Therefore, a memory of $N \times L \times 5$ bits is required to store the sequence alignment matrix. On the other hand, the tree topology shows the connections between the internal nodes of the tree and the taxa. A tree with N taxa has $N - 1$ nodes, including the root node. Since the tree is a binary tree, each node has a left and a right branch, and a parent node. Thus, the size of the memory required to store the tree topology is $(N - 1) \times 3[log_2(N)]$ bits.

Furthermore, this approach requires the use of three memories. First, it requires a memory to store the preliminary node character states, and, second, a memory to store the final node character states. The size of any of these memories is $(N - 1) \times L \times 5$ bits. Third, it requires a memory to store the node lengths for all the nodes in the tree. The size of this memory is $(N - 1) \times 18$ bits, where 18 bits is the number of bits required to store the length of the tree in binary

codification. For example, the tree topology, the sequence data matrix memory, the node data matrix memories, and the node length memory for a tree with 6 taxa are represented in figure 6.18.

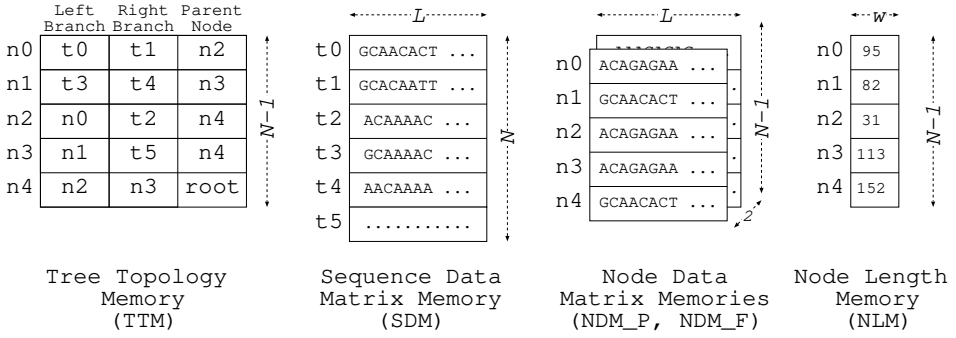


FIGURE 6.18: Fifth approach: memory data structure

NDM_P refers to the Node Data Matrix Memory used to store the preliminary character states; while NDM_F, to the Node Data Matrix Memory used to store the final character states. Taxa are labeled according to their memory positions on the Sequence Data Matrix Memory. Likewise, nodes are labeled according to their memory positions on the Tree Topology Memory. Since both taxa and nodes appear on the same memory, we use an additional bit to distinguish between the two of them: 0 for a node and 1 for a taxon (singular form of taxa). The root node doesn't have a parent. Instead, a full sequence of 1s is used to identify it.

6.2.3 Proposed Hardware Architecture

In section 6.2.1, we mentioned the steps involved in each iteration of the algorithm. To design the hardware architecture, we considered that steps 1, 5.1 and 6.1 can be performed by the same hardware unit, since they are about modifying the tree topology. Similarly, steps 2, 3, 4, 5.2 and 6.2 can be performed by the same hardware unit, since they are about processing the nodes of the tree in a certain manner.

This leads to the following hardware units that we designed to implement the algorithm described in section 6.2.1:

1 Tree Topology Update (TTU) unit

2 First-, Second-pass and Rearrangement evaluation (FSR) unit

3 Global Control (GC) unit

The TTU unit is in charge of modifying the tree topology memory to reflect the changes produced by the Subtree Pruning and Reinserting (SPR) process. The FSR unit has the most important tasks, which are doing a first- and second-pass optimization, whether complete or incremental, and evaluating all possible rearrangements according to

the Indirect Calculation of Tree Lengths (ICTL). A general block diagram of the hardware architecture proposed is shown in figure 6.19.

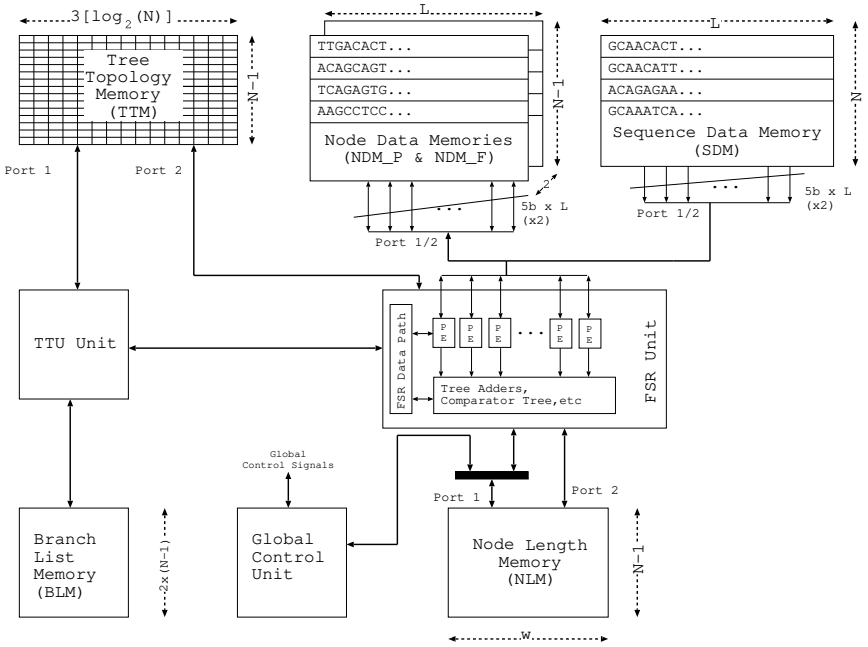


FIGURE 6.19: Fifth approach: general block diagram of the proposed hardware architecture

It consists of the following elements: the dual-port Tree Topology Memory (TTM), the dual-port Sequence Data Memory (SDM), the dual-port Node Data Memories (NDM_P and NDM_F), the Node Length Memory (NLM), the Branch List Memory (BLM), the TTU unit, the FSR unit, and the Global Control (GC) unit. Black bars on the diagram make reference to multiplexers.

As can be seen from figure 6.19, the proposed hardware architecture has been drastically simplified compared to the fourth approach. This simplification has contributed to ease the design of the remaining units and improve the overall performance. First, the TTU unit was redesigned to include reconstructing the previous tree topology by reinserting the sub tree (ST) back into the main tree (MT) on the same branch from which it was pruned. Second, the FSR unit was redesigned to reduce the execution times in the five main tasks it has. Finally, the GC unit was redesigned to execute all the steps in the algorithm presented in section 6.2.1. In the following sections, we explain how each of these three hardware units work.

6.2.3.1 Tree Topology Update (TTU) unit

The Tree Topology Update (TTU) unit is in charge of modifying the Tree Topology Memory to reflect the changes produced by the Subtree Pruning and Reinserting (SPR) process, whether it involves the creation of a new tree or the reconstruction of the previous one.

It has two main tasks:

1 Pruning a branch from the whole tree (WT) to create the main tree (MT) and the sub tree (ST).

2 Inserting the ST in the MT to create a new WT or to reconstruct the previous WT.

And two sub tasks:

1 Storing the value of the pruned branch and reinsertion branch.

2 Storing the values of the WT, MT and ST roots.

The general block diagram of the TTU unit is shown in figure 6.20.

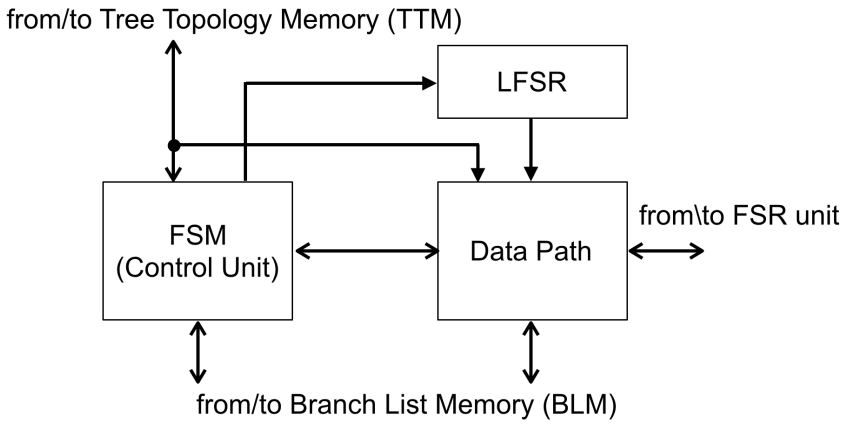


FIGURE 6.20: Fifth approach: general block diagram of the TTU unit

As can be seen from figure 6.20, the TTU unit is implemented as a Finite State Machine (FSM) with a Data Path. The FSM works basically as a memory controller that is used to modify the content of the Tree Topology Memory (TTM). In addition to this, it has a linear-feedback shift register (LFSR), which works as a pseudo-random number generator.

As was shown in section 6.1.3.1, regardless of the number of taxa, when a branch with a subtree (ST) attached to it is pruned from the tree, at most 2 nodes are modified. Similarly, when a branch with a ST attached to it is inserted in the MT, at most 3 nodes are modified. This is the same either when the ST is inserted in the MT to create a new WT, or when the ST is inserted in the MT to reconstruct the previous tree.

The pruning process involves modifying at most 2 nodes. Since it takes 2 clock cycles to modify a node from the Tree Topology Memory, the execution time is of 4 clock cycles at most. Similarly, inserting the sub tree involves modifying at most 3 nodes. Thus, it takes 6 clock cycles at most. Rebuilding the tree is equivalent to reverting the pruning process; hence, this takes also 4 clock cycles at most.

6.2.3.2 First-, Second-pass and Rearrangement Evaluation (FSR) unit

This unit is the most important unit. It has five main tasks:

1 Doing a complete first-pass optimization (post-order).

2 Doing a complete second-pass optimization (reversed-order).

3 Doing an incremental first-pass optimization.

4 Doing an incremental second-pass optimization.

5 Evaluating all possible tree rearrangements.

Its general block diagram is shown in figure 6.21.

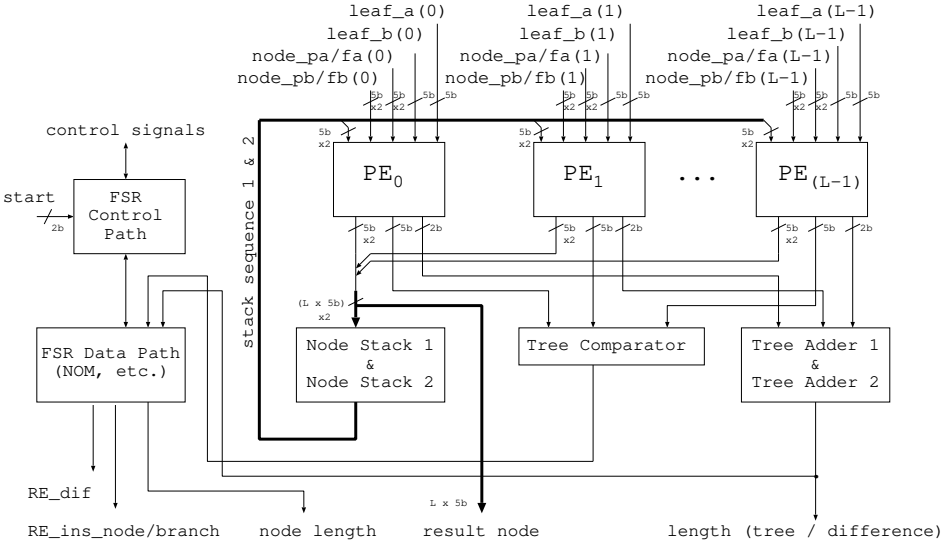


FIGURE 6.21: Fifth approach: general block diagram of the FSR unit

As seen in figure 6.21, the FSR unit is composed of L processing elements (PE), where L is equivalent to the number of DNA characters in a sequence (see figure 6.18), two tree adders: Tree Adder 1 and Tree Adder 2, two Node Stack memories: Node Stack 1 and Node Stack 2, a Tree Comparator, and a Control Logic unit with a Data Path. Inside the Data Path is a Node Order Memory. This memory is used for the complete first-pass optimization to list the nodes in the post-order, and for the second-pass optimization to list the nodes in the reversed-order.

The inputs of the FSR unit are two leaves (taxa) from the Sequence Data Matrix Memory: *leaf a* and *leaf b*, two nodes from the Preliminary Node Data Matrix Memory: *node pa* and *node pb*, and two nodes from the Final Node Data Matrix Memory: *node fa* and *node fb*.

The Tree Adder 1 is used during the complete and incremental first-pass optimization to add all individual scores from the PEs to obtain the length of the node. In addition, the Tree Adder 1 and the Tree Adder 2 are used during the tree rearrangement evaluation to add all individual difference scores from the PEs in order to obtain the total difference scores. The Tree Comparator is used during the incremental first-pass optimization to compare the new node character state against the previous one in order to obtain the lowest node with unmodified character state. This node is used during the incremental second-pass optimization as the root node.

The FSR unit outputs the result node from the PEs, so it can be stored in the Preliminary or Final Node Data Matrix Memory. It also outputs the best candidate node and branch for reinserting the subtree (ST) in the main tree (MT), and the respective difference score: *RE_dif*, *RE_ins_node/branch*.

The FSR unit uses L PEs to implement the complete and incremental first-pass optimization algorithm (see section 2.1.3.1 and section 2.2.4.1), the complete and incremental second-pass optimization algorithm (see section 2.2.3 and section 2.2.4.2), and the tree rearrangement evaluation by using the Indirect Calculation of Tree Lengths (see section 2.2.2). These L PEs allow to process all the characters of two nodes or leaves (taxa) in parallel. The general block diagram of a PE is shown in figure 6.22.

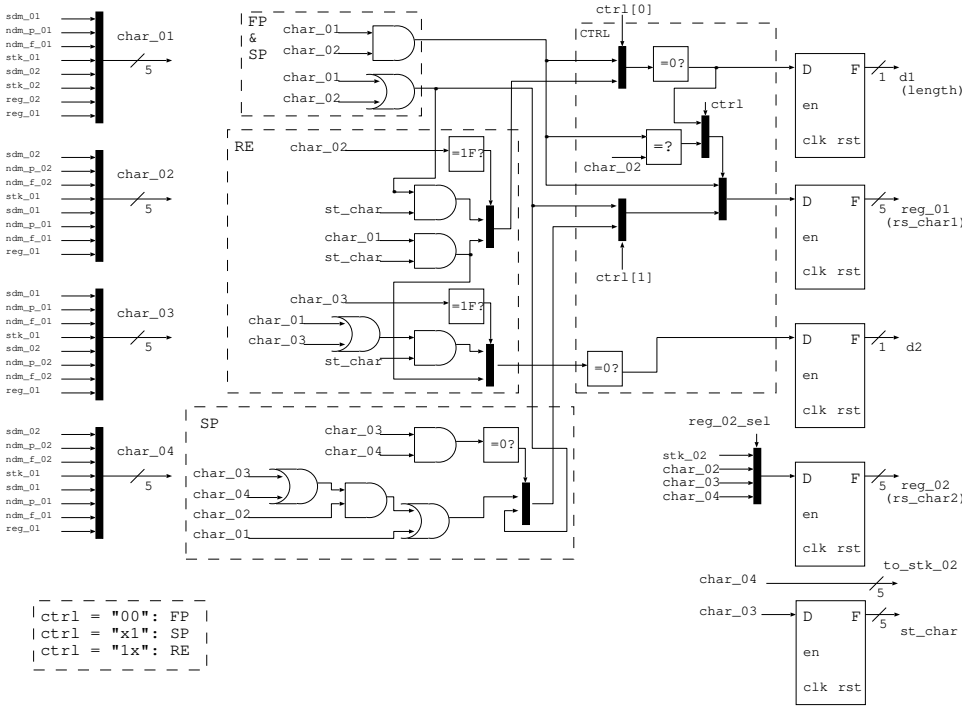


FIGURE 6.22: Fifth approach: general block diagram of the Processing Element (PE)

As can be seen from figure 6.22, the PE is composed of three D-type flip-flop-based registers of 5 bits each, two D-type flip-flop-based register of 1 bit, and a collection of logic gates. These logic gates implement the five main tasks of the FSR unit described at the beginning of this section, but for a single DNA character. Depending on a control signal (ctrl in figure 6.22) the PE changes its functionality. It should be noted that the amount of logic resources inside the PE has been almost doubled compared to the PE from the fourth approach (see section 6.1.3.3. The amount of logic resources was increased to reduce the execution times for the tree optimization and the tree rearrangement evaluation, as we explain in the next subsections.

The idea behind having these five main tasks in one unit is to share the same resources for the complete and incremental first- and second-pass optimization, and for the rearrangement evaluation, since they need not to work at the same time. Each of these tasks works using a pipelined algorithm. In the following sections we describe each one of them.

6.2.3.2.1 Complete First-Pass Optimization (FSR-CFP) The FSR-CFP performs a first-pass optimization on all the nodes of the whole tree (WT). It works in almost the same way as in the fourth approach, but now the node order is stored in the Node Order Memory from the Data Path of the FSR unit. In addition, it uses the Node Stack 1 and the sequence output from all the PEs, as shown in figure 6.21. It follows the extended 6-stage pipelined algorithm listed below.

Stage 01 Pop a node from the NOM and read it from the TTM.

Stage 02 Read the LB and the RB from the SDM1 and the SDM2.

Stage 03 Do the following three tasks:

1. Case{Node(LB, RB)}
   (Leaf, Leaf): Push the node from Stage 04 into the Stack.
   (Node, Node): Pop a node from the Stack.
2. Do a first-pass optimization on the node.
3. Add the lengths of both branches.

Stage 04 Store the resulting node into the NDM_P1.

... Line Buffer / Tree Adder ...

Stage 05 Calculate the node length.

Stage 06 Store the resulting node length into the NLM1.

As in the fourth approach, each stage processes a node at a time. The FSR-CFP finishes when all nodes in the Node Order Memory have been processed. The total execution time approximates $N + T$ clock cycles, where N is the number of nodes in the whole tree (WT), and T the latency of the Tree Adder.

6.2.3.2.2 Incremental First-Pass Optimization (FSR-IFP) The FSR-IFP performs a first-pass optimization on those nodes that were affected by the Subtree Pruning and Reinserting (SPR) process. Unlike the fourth approach, now the FSR-IFP processes one node per clock cycle thanks to the extended 6-stage pipeline algorithm listed below. This algorithm is now similar to the one used for the complete first-pass optimization.

Stage 01 Read the node from the TTM.

Stage 02 Read either the LB or the RB from the SDM1 or the NDM_P1.

Stage 03 Do a first-pass optimization on the node.

Stage 04 Do the following two tasks:

1. Store the resulting node into NDM_P2.
2. Compare the new value of the node with its previous value.

... Line Buffer / Tree Adder ...

Stage 4.5 Read either the LB or the RB from the NLM_1.

Stage 05 Calculate the node length.

Stage 06 Store the resulting node length into the NLM2.

TTM refers to the Tree Topology Memory, SDM to the Sequence Data Matrix Memory, NDM_P to the Preliminary Node Data Matrix Memory and NLM to the Node Length Memory. All memories are dual-port memories.

To illustrate the above mentioned algorithm and how the extended 6-stage pipeline works, an example is given in figures 6.23 and 6.24 for a main tree (MT) with 6 taxa.

In figure 6.23, in *Stage 03*, *fp(LB,RB)* refers to the first-pass optimization that is a function of the left and right branches of the node (see section 2.1.3.1), and *reg1* refers to the output sequence 1 from the PEs. In *Stage 04*, *d[LB,RB]* refers to the difference score between the left and right branch for a single DNA character.

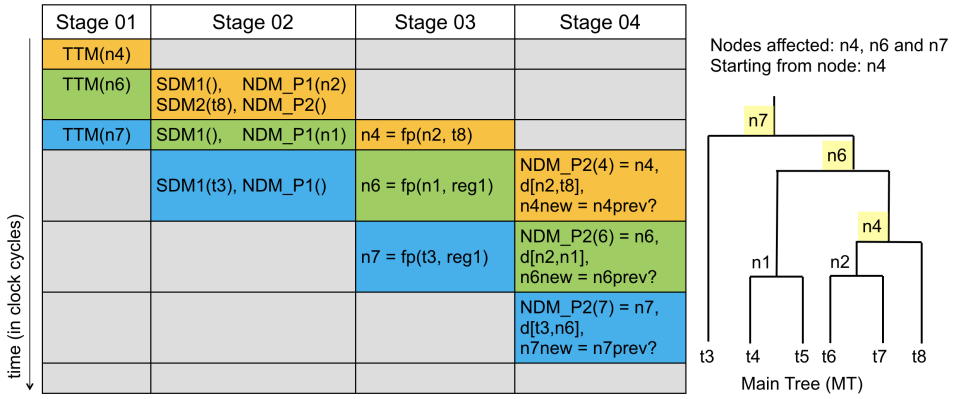


FIGURE 6.23: Fifth approach: example of the pipeline processing during the FSR-IFP (nodes)

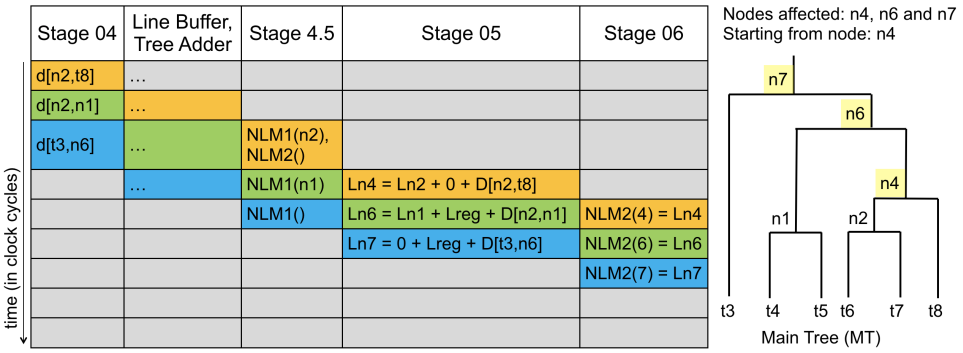


FIGURE 6.24: Fifth approach: example of the pipeline processing during the FSR-IFP (lengths)

In figure 6.24, in *Stage 05*, the total length of the node is finally calculated after the Tree Adder has summed the individual results from all the PEs, and in *Stage 06* the node length is stored into the Node Length Memory (NLM).

As can be seen from figures 6.23 and 6.24, each stage processes a node at a time. The FSR-IFP finishes when all nodes affected by the Subtree Pruning and Reinserting (SPR) have been processed. The total execution time approximates $n + T$ clock cycles, where n is the number of nodes in the main tree (MT) or subtree (ST), and T the latency of the Tree Adder. This is almost half the time required by the FSR-IFP of the fourth approach.

6.2.3.2.3 Second-Pass Optimization (FSR-SP) The FSR-SP performs a second-pass optimization on all the nodes of the given tree, whether they belong to the whole tree (WT) or the main tree (MT). Unlike the fourth approach, in this approach we decided to merge both optimizations in one, because they are basically the same.

The incremental second-pass optimization is equivalent to a complete second-pass optimization that starts on a different root node

(i.e. the lowest node with unmodified character states during the incremental first-pass optimization).

The FSR-SP uses both Node Stack memories shown in figure 6.21: Node Stack 1 and Node Stack 2, and both sequence outputs from all the PEs. By selecting the root node, it can perform either the complete second-pass optimization or the incremental second-pass optimization. Unlike the fourth approach, now the FSR-SP processes one node per clock cycle thanks to the 4-stage pipelined algorithm listed below.

Stage 01 Read the current node from the TTM.

Stage 02 Read LB and RB from SDM1/SDM2 or NDM_P1/NDM_P2.

Stage 03 Do the following two tasks:

1. Select the operands of the second-pass function.
2. Do a second-pass optimization on the node.

Stage 04 Store the resulting node into NDM_F2.

TTM refers to the Tree Topology Memory, SDM to the Sequence Data Matrix Memory, NDM_P to the Preliminary Node Data Matrix Memory, and NDM_F to the Final Node Data Matrix Memory. All memories are dual-port memories. This is represented by the index attached at the end of the name of the memory, which refers to either the port 1 or 2, respectively.

The current node, which determines the order in which the tree is traversed (i.e. the reversed order), is obtained during Stage 01 and Stage 02 by using the following algorithm:

Initialization Set the current node to the given root node.

Stage 01 Read the current node from the TTM.

Stage 02 Case(LB, RB) of the current node.

(Node, Node): Push the RB into the Node Order Memory (NOM).

(Node, Leaf): Set the current node to the LB.

(Leaf, Node): Set the current node to the RB.

(Leaf, Leaf): If the stack is not empty:

1. Pop a node.
2. Set the current node to the popped node.

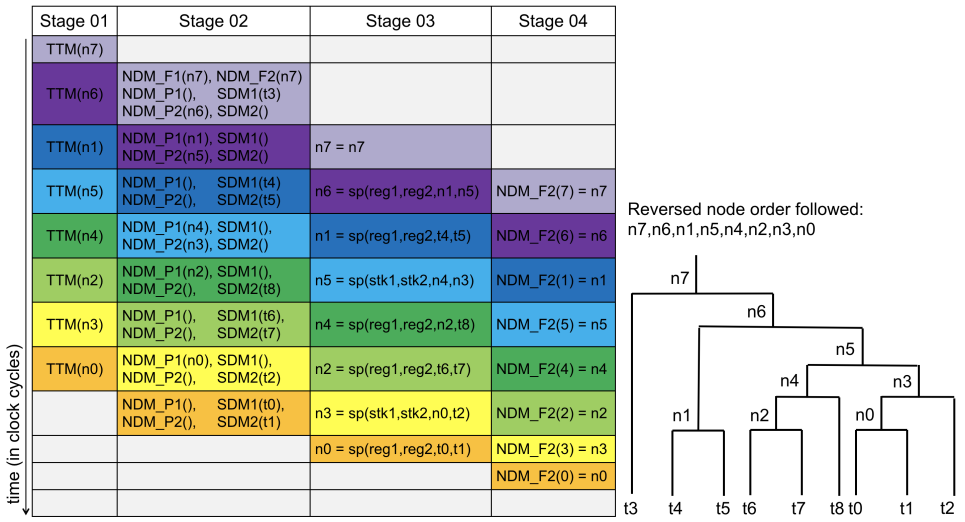


FIGURE 6.25: Fifth approach: example of the pipeline processing during the FSR-SP

To illustrate how the 4-stage pipeline of the FSR-SP works, an example is given in figure 6.25 for a tree with 9 taxa and 8 nodes. In this example, the starting node is the root of the tree; however, it could be any node from the tree.

In figure 6.25, in *Stage 03*, *sp(PN, CN, LB, RB)* refers to the second-pass optimization that is a function of the parent node (PN), the current node (CN) and the left (LB) and right (RB) branches (see section 2.1.3.2). In the same stage, *reg1* and *reg2* refer to the output sequences 1 and 2 from all the PEs, and *stk1* and *stk2* to the output sequences 1 and 2 from the Node Stack memories, respectively.

The execution time approximates 1 clock cycle per node as the number of nodes increases. The total execution time approximates $2n$, where n is the number of nodes optimized in the given tree. This is almost half the time required by the FSR-CSP or FSR-ISP of the fourth approach.

6.2.3.2.4 Rearrangement Evaluation (FSR-RE) The FSR-RE evaluates all the tree rearrangements within the neighborhood using the Indirect Calculation of Tree Lengths (ICTL). The algorithm uses both score outputs (*d1* and *d2* in figure 6.22) from all the PEs. Unlike the fourth approach, now one node is processed every clock cycle. The FSR-PE works by following the 3-stage pipelined algorithm listed below.

Initialization Do the following two tasks:

1. Read the root of the subtree (ST) from NDM_F1 or SDM1.
2. Read the root of the main tree (MT) from NDM_F2.

Stage 01 Read the curret node from TTM.

Stage 02 Read LB and RB from NDM_F1/NDM_F2 or SDM1/SDM2

Stage 03 Do the following two tasks:

1. Select the operands of the evaluation function.
2. Evaluate the left and right tree rearrangements.

TTM refers to the Tree Topology Memory, SDM to the Sequence Data Matrix Memory and NDM_F to the Final Node Data Matrix Memory. Both the SDM and the NDM are dual-port memories. This is represented by the index attached at the end of the name of the memory, which refers to either the port 1 or 2, respectively. The current node, which determines the order in which the tree is traversed (i.e. the reversed order), is obtained during Stage 01 and Stage 02 by using the same algorithm explained in section 6.2.3.2.3.

To illustrate the above mentioned algorithm and how the 3-stage pipeline works, an example is given in figure 6.26 for a tree with 9 taxa and 8 nodes. The order in which the nodes are processed is the the reversed order, starting from the root of the tree.

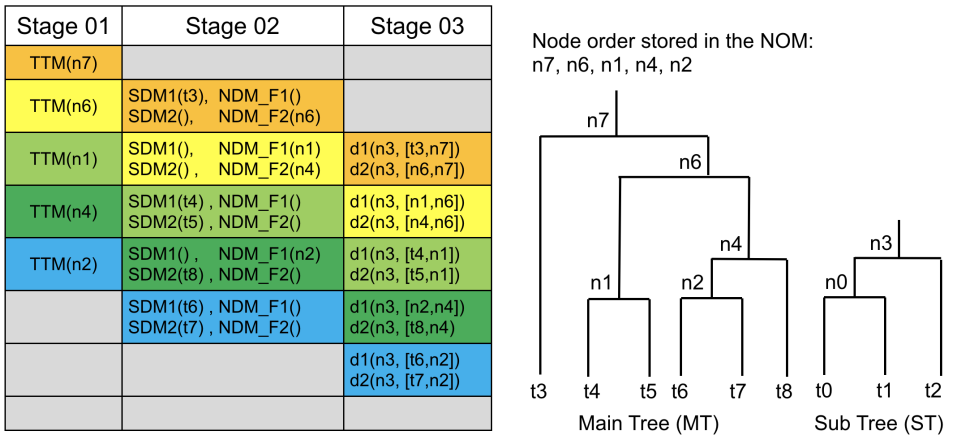


FIGURE 6.26: Fifth approach: example of the pipeline processing during the FSR-RE

In figure 6.26, in *Stage 03*, *di(SR, [LB/RB, CN])* refers to the difference score that is a function of the root node of the subtree (SR), the left branch (LB) or the right branch (RB) of the MT, and the current node (CN) (see section 2.2.2).

The difference score values *d1* and *d2* are added by the trees adders Tree Adder 1 and Tree Adder 2 from figure 6.21 to obtain the total difference score for those two tree rearrangements. The same is done for all pairs of tree rearrangements. Finally, the rearrangement with the lowest score is kept as candidate to reinsert the subtree (ST) in

the main tree (MT). For this purpose, inside the Data path of the FSR unit, a line buffer and some comparison registers are used.

Since two tree rearrangements are evaluated at the same time, the execution time will approximate 0.5 clock cycles per tree rearrangement as the number of nodes increases. The total execution approximates $n + T$, where n is the number of nodes in the main tree (MT), and T is the latency of the Tree Adder. This is almost half the time required by the FSR-RE of the fourth approach.

6.2.3.3 Global Control (GC) unit

The Global Control (GC) unit is a Finite State Machine (FSM) that is in charge of executing the algorithm described in section 6.2.1. It commands the other two units (TTU and FSR) and supervises all memory accesses. The overall processing of our hardware approach executed by the GC unit is shown in figure 6.27.

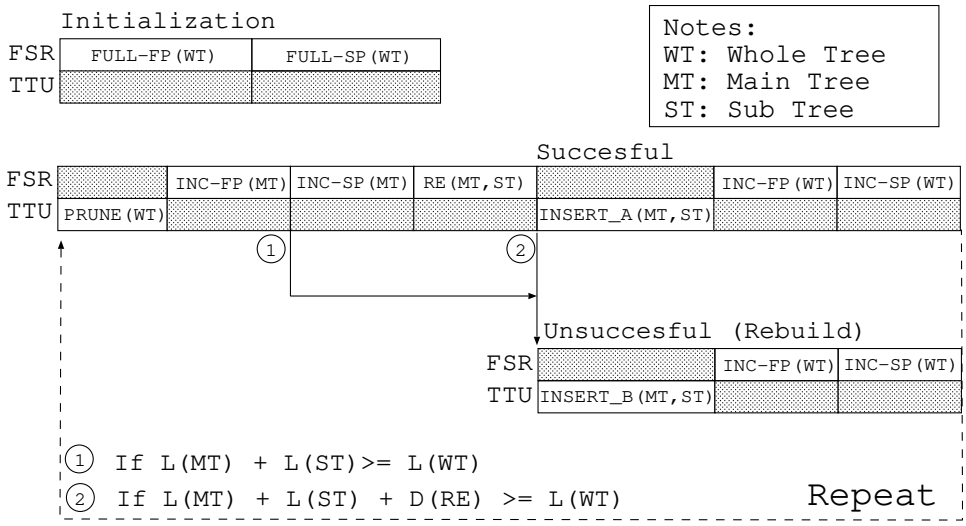


FIGURE 6.27: Fifth approach: execution flow of the proposed hardware architecture

In figure 6.27, FULL-FP refers to the complete first-pass optimization; INC-FP, to the incremental first-pass optimization; FULL-SP, to the complete second-pass optimization; INC-SP, to the incremental second-pass optimization; and RE, to the tree rearrangement evaluation. INSERT_A refers to the reinsertion process of the sub tree into the best branch found, while INSERT_B refers to the reinsertion process of the sub tree into the branch from which it was originally pruned. In other words, INSERT_B is equivalent to reconstructing the previous tree.

6.2.4 Implementation Results

In this section we show implementation results for eight real-world biological datasets. The datasets were obtained from the repository of phylogenetic information TreeBASE [45], see table 6.5.

TABLE 6.5: Fifth approach: datasets used [45]

ID	M972	M2355	M3452	M3875
#taxa	155	150	116	228
#characters	355	829	1,157	1,435
ID	M17200	M2616	M14883	M18088
#taxa	326	330	330	296
#characters	1,434	1,711	2,290	3,222

We implemented our system on a Virtex-7 XC7VX690T-FFG1157-2 FPGA. We limited our evaluation to only these eight datasets. However, we consider them enough and adequate to measure the performance of our approach. Each dataset was chosen carefully based on its number of taxa and characters. As can be seen from table 6.5, the number of characters increases from one dataset to the other, starting at 355 and ending at $3,222$. The number of taxa also has a tendency to increase. These datasets can be considered of medium to large size, which is an optimal size for evaluating the performance.

6.2.4.1 Hardware Utilization and Performance Results

The hardware utilization and performance results are shown in table 6.6. The targeted FPGA is a Virtex-7 XC7VX690T-FFG1157-2 FPGA.

TABLE 6.6: Fifth approach: implementation results on a Virtex-7 FPGA

Logic Utilization	Used	Available	Utilization
Number of Slice Registers	115,788	866,400	13.36%
Number of Slice LUTs	376,925	433,200	87.01%
Number of BRAMs (36 Kb)	1,346	1,470	91.56%
Maximum Frequency	146MHz		

The implementation covers all of the eight datasets shown in table 6.5. In other words, problems up to $N = 1,024$ and $L = 3,222$ can be processed with this amount of logical resources. BRAMs are used to store the Tree Topology Memory (TTM), the Sequence Matrix Data Memory (SDM), the two Node Data Matrix memories (NDM_P and NDM_F), and the Node Length Memory (NLM). The number of LUTs is almost proportional to$L$, the number of BRAMS to $L \times N$.

6.2.5 Comparison and Performance Evaluation

6.2.5.1 Logical Resources Utilization Comparison

In this section, we compare the amount of logical resources required by our current approach (the fifth approach) against the fourth approach. In table 6.3 of section 6.1.4.1, we showed the amount of logical resources needed by the fourth approach to implement problems up to $N = 1,024$ and $L = 829$ (i.e. the amount needed for the first two problems: M972 and M2355). We show these results again in table 6.7.

Now, to make a comparison in the same terms, we show the amount of logical resources required by our current approach (the fifth approach) to implement problems up to the same number of taxa and characters (i.e. $N = 1,024$ and $L = 829$). The results are summarized in table 6.8.

TABLE 6.7: Fourth approach: implementation results on a Virtex-7 FPGA for $N = 1,024$ and $L = 829$

Logic Utilization	Used	Available	Utilization
Number of Slices	23,968	108,300	22%
Number of Slice Registers	21,664	866,400	2%
Number of Slice LUTs	58,398	433,200	13%
Number of BRAMs (36 Kb)	577	1,470	39%
Maximum Frequency	158.767MHz		

TABLE 6.8: Fifth approach: implementation results on a Virtex-7 FPGA for $N = 1,024$ and $L = 829$

Logic Utilization	Used	Available	Utilization
Number of Slices	24,214	108,300	22.36%
Number of Slice Registers	29,127	866,400	3.36%
Number of Slice LUTs	96,853	433,200	22.36%
Number of BRAMs (36 Kb)	350	1,470	23.81%
Maximum Frequency	166MHz		

Now, using the results summarized in tables 6.7 and 6.8, we show in figure 6.28 the proportion of Registers, LUTs and BRAMs required by the fifth approach against the fourth.

As can be seen from figure 6.28, the fifth approach requires more registers and LUTs than the fourth approach does. This is to be expected, since more logical resources are used inside each PE to process one node per clock cycle during the complete or incremental first- and second-pass optimization, and during the tree rearrangement evaluation. On the other hand, the fifth approach only requires around 60% of the memory resources (BRAMs) that the fourth approach required.

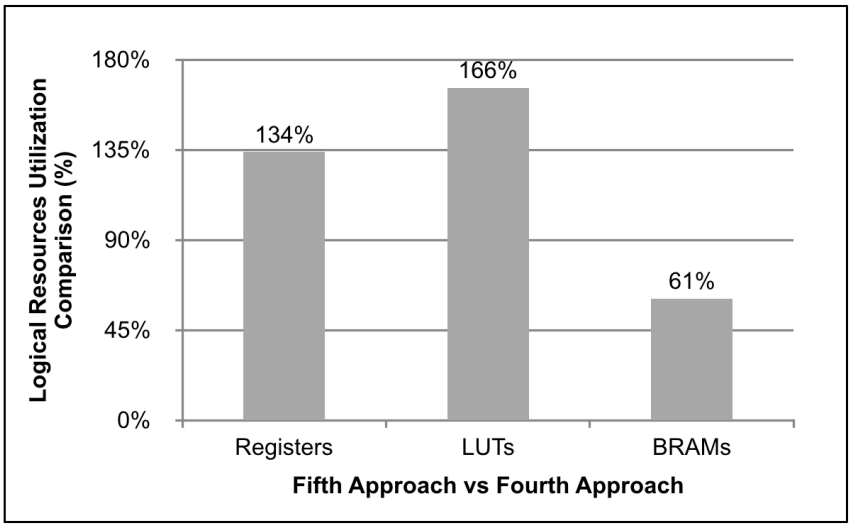


FIGURE 6.28: Fifth approach: comparison of logical resources utilization

This is a huge improvement, given that the BRAMs are the most required logical resource from the FPGA. Thanks to this reduction, the fifth approach is able to implement problems up to $N = 1,024$ and $L = 3,222$, while the fourth approach can only implement problems up to $N = 1,024$ and $L = 829$ on a Virtex-7 FPGA.

6.2.5.2 Local Search Results Comparison

Here we compare our current hardware approach (fifth approach) against the second approach (see chapter 4), the third approach (see chapter 5), and with TNT (Tree analysis using New Technology) [20]. We omit here the comparison against the fourth approach, because both approaches implement basically the same algorithm, which uses the Incremental Tree Optimization.

To make the comparison as fair as possible, we use the traditional search of TNT based on Subtree Pruning and Regrafting (SPR), and start from a random tree. This is the closest setting of TNT that resembles our algorithms (see sections 4.1, 5.1, and 6.2.1). Moreover, since the total number of examined trees is not the same, we show the average execution time required for each tree. The results are summarized in table 6.9.

The targeted PC and FPGAs were the following:

- PC: Intel Core-i7 860, 4GB RAM @ 2.80 Ghz (TNT)
- FPGA: Kintex-7 XC7K325T-FF2-900 @ 156.25 Mhz (Second)
- FPGA: Virtex-7 XC7VX690T-FFG1157 @ 156.25 Mhz (Third)
- FPGA: Virtex-7 XC7VX690T-FFG1157 @ 145.00 Mhz (Fifth)

TABLE 6.9: Fifth approach: results for the local search

Dataset		Second	Third	Fifth	TNT
M972	Total time (ms)	11.27	9.06	4.40	890
	Time/tree (μs)	0.031	0.026	0.0088	0.065
	Visited trees	364,177	349,012	505,181	13,637,086
	Best score	1,533	$\leftarrow$ same	1,531	1,543
M2355	Total time (ms)	9.9	6.18	4.20	500
	Time/tree (μs)	0.025	0.021	0.0084	0.059
	Visited trees	400,368	294,360	499,952	8,497,522
	Best score	2,724	$\leftarrow$ same	2,740	2,771
M3452	Total time (ms)	9.64	8.09	2.55	180
	Time/tree (μs)	0.029	0.024	0.0092	0.103
	Visited trees	329,025	337,014	281,126	1,749,117
	Best score	3,632	$\leftarrow$ same	3,610	3,624
M3875	Total time (ms)	27.76	18.02	5.85	510
	Time/tree (μs)	0.037	0.032	0.0079	0.021
	Visited trees	760,180	562,514	736,662	23,818,061
	Best score	567	$\leftarrow$ same	570	564
M17200	Total time (ms)	53.56	44.82	13.52	2,260
	Time/tree (μs)	0.019	0.016	0.0073	0.046
	Visited trees	2,798,944	2,801,301	1,841,018	49,662,276
	Best score	4,344	$\leftarrow$ same	4,345	4,340
M2616	Total time (ms)	42.25	35.67	22.36	4,700
	Time/tree (μs)	0.027	0.022	0.0063	0.09
	Visited trees	1,564,515	1,621,566	3,549,853	53,330,179
	Best score	10,003	$\leftarrow$ same	10,003	10,004
M14883	Total time (ms)	No Data	No Data	11.20	1630
	Time/tree (μs)	No Data	No Data	0.0076	0.038
	Visited trees	No Data	No Data	1,468,782	43,186,692
	Best score	No Data	No Data	1,132	1,139
M18088	Total time (ms)	No Data	No Data	22.96	9,000
	Time/tree (μs)	No Data	No Data	0.0073	0.23
	Visited trees	No Data	No Data	3,145.022	39,370,427
	Best score	No Data	No Data	29,729	29,856

The last two datasets were not evaluated on the second and fourth approach, so there is no data available for these two. In table 6.9, we can see the total execution time required for the algorithm to converge (*Total time*), the execution time for the evaluation of a tree rearrangement (*Time/tree*), the number of trees visited during the search (*Visited trees*), and the best score obtained for each of the datasets (*Best score*). With regard to the best score, we achieve values similar to those of the second and third approach, and TNT. Now, using these results, in figure 6.29 we show the acceleration rate per tree rearrangement obtained.

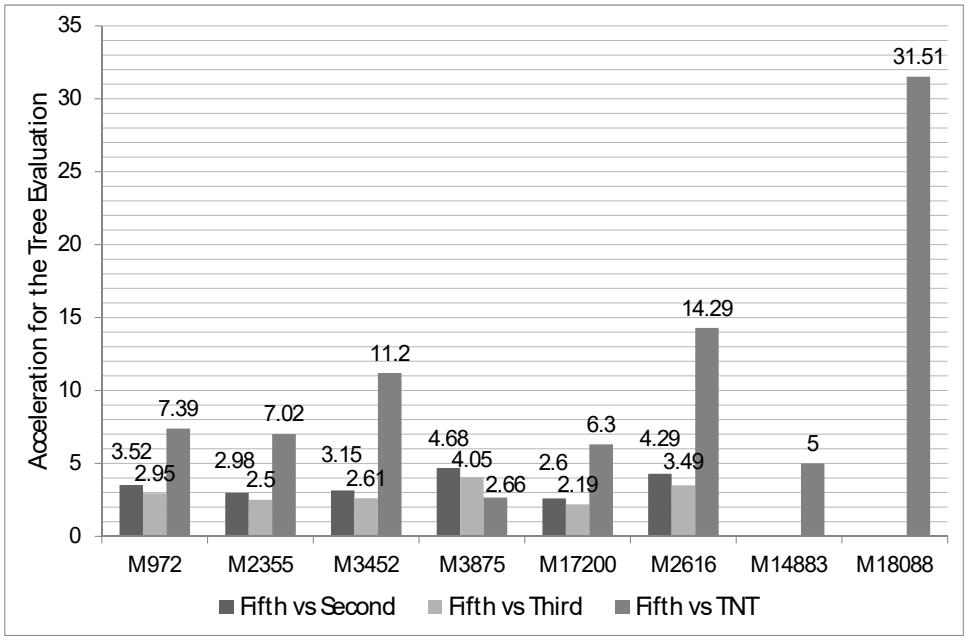


FIGURE 6.29: Fifth approach: acceleration rate for the tree evaluation

As can be seen from figure 6.29, we obtain acceleration rates per tree rearrangement between 2.60 and 4.68 against the second approach; between 2.19 and 4.05 against the third approach; and, between 2.66 and 31.94 against TNT.

6.2.6 Discussion

We compared execution times against our second approach (refer to chapter 4), the approach without the Incremental Tree Optimization method; against our third approach (refer to chapter 5), the approach that uses the Alternative Second-pass optimization; and TNT. Compared to both approaches and TNT, our current fifth approach is faster.

The acceleration rates achieved are thanks to a combination of using the Indirect Calculation of Tree Lengths method, the Incremental Tree Optimizaton method, and the redesigned parallel and pipeline processing used in our current fifth approach. First, the Indirect Calculation of Tree Lengths method allows to evaluate all tree rearrangements within the neighborhood, and obtain the best rearrangement. Second, the Incremental Tree Optimization reduces the number of nodes to be updated during the first- and second-pass optimization. Thus, reducing significantly the time spent in optimizing the nodes of the tree. Last but not least, the parallel processing of all the characters in a sequence using L PEs makes it possibly to evaluate a node of the tree in one clock cycle. The acceleration rate against a software approach should increase theoretically with the number

of characters, because in our approach we process all characters in parallel, while a software approach requires to process characters sequentially; thus, taking more time as its number increases.

Moreover, our approach achieves these acceleration rates without using additional memory resources for the incremental optimization compared to the fourth approach. We only need to store the current memory states; thus, eliminating the need of a memory to store the previous tree topology, the previous node character states (both preliminary and final), and the previous node lengths. As a result, the total amount of memory needed is decreased, the number of memory accesses is reduced, and the performance is increased.

With regard to hardware resources, our approach can be applied for larger problems as long as there are enough logical resources in the targeted FPGA. For this current implementation, we used almost all of the memory resources available on the FPGA. For larger datasets, FPGAs such as the Virtex UltraScale XCVU190 FPGA can be used.

Chapter 7

General Discussion

Our first approach (chapter 3) implemented the first algorithm examined in section 2.2.1 that includes the Progressive Tree Neighborhood. We showed implementation results on a Kintex-7 FPGA for four real-world biological datasets. In comparison to our C++ implementation of the same algorithm, the first approach provided an acceleration rate that was in the order of thousands. In comparison to TNT, there was no acceleration achieved. Nevertheless, the scores obtained by the first approach were comparable to those from TNT. From this approach we learned that to achieve higher acceleration rates we had to employ other optimization methods to reduce the evaluation time of tree rearrangements.

Our second approach (chapter 4) implemented the second algorithm examined in section 2.2.2 that includes the Indirect Calculation of Tree Lengths. We showed implementation results on a Kintex-7 FPGA and simulation results for four and two real-world biological datasets, respectively. In comparison to the first approach, the second approach provided an acceleration between 2 and 6 for the whole local search, and between 34 and 45 for the evaluation of a single tree rearrangement. In comparison to TNT, the second approach achieved an acceleration rate between 18 and 112 for the whole local search, and between 2 and 4 for the evaluation of a single tree rearrangement, except for one problem, for which there was no acceleration obtained. From this approach we learned that to achieve even faster execution times than TNT, it was necessary to consider other optimization methods.

Our third approach (chapter 5) implemented the third algorithm examined in section 2.2.3 that includes the Alternative Second-pass Optimization. We showed implementation results on a Virtex-7 FPGA for six real-world biological datasets. In comparison to the second approach, the third approach provided an acceleration between 1.18 and 1.6 for the whole local search, and between 1.16 and 1.23 for the evaluation of a single tree rearrangement. In comparison to TNT, the third approach achieved an acceleration between 22.25 and 131.76 for the whole local search, and between 2.5 and 4.29 for the evaluation of a single tree rearrangement, except for one problem again.

From this approach we learned that, although using the Alternative Second-pass Optimization provided a slightly improvement in the acceleration rate, it required a considerable amount of memory. For this reason, it was necessary to consider other optimization methods; in particular, that not all the node character states of the tree have to be recalculated when a branch from the tree is pruned, since only a small portion of the tree is affected.

Our fourth approach (section 6.1 of chapter 6) implemented the fourth algorithm examined in section 2.2.4 that includes the Incremental Tree Optimization. We showed implementation results on a Virtex-7 FPGA and simulation results for two and four real-world biological datasets. In comparison to the second approach, the fourth approach achieved an acceleration rate between 1.65 and 3.03 for the evaluation of a single tree rearrangement, and compared to the third approach; between 1.39 and 2.62. In comparison to TNT, the fourth approach achieved an acceleration rate between 1.72 and 8.82 against TNT for the evaluation of a single tree rearrangement. From this approach we learned that, although using the Incremental Tree Optimization provided a significant improvement in the acceleration rates obtained, the use of additional buffer memories imposed a constraint on the size of the problems that could be implemented. For this reason, it was necessary to consider an algorithm that did not use additional buffer memories.

Our fifth and last approach (section 6.2 of chapter 6) implemented also the fourth algorithm examined in section 2.2.4 that includes the Incremental Tree Optimization. We showed results on a Virtex-7 FPGA for eight real-world biological datasets. In comparison to the second approach, the fifth approach achieved an acceleration rate between 2.60 and 4.68 for the evaluation of a single tree rearrangement, and compared to the third approach; between 2.19 and 4.05. In comparison to TNT, the fifth approach achieved an acceleration rate between 2.66 and 31.94 for the evaluation of a single tree rearrangement. From this approach we learned that using the Incremental Tree Optimization effectively provides a significant improvement in the acceleration rates obtained without excessively increasing the amount of memory needed.

The acceleration rates achieved are thanks to a combination of using the Indirect Calculation of Tree Lengths, the Incremental Tree Optimization, and the parallel and pipeline processing used in our approach. First, the Indirect Calculation of Tree Lengths method allows evaluating all tree rearrangements within the neighborhood, and obtaining the best one. Second, the Incremental Tree Optimization reduces the number of nodes to be updated during the first and second-pass optimizations. Thus, reducing significantly the time

spent in optimizing the nodes of the tree. Last but not least, the parallel processing of all the characters in a sequence using L PEs makes it possibly to evaluate a node of the tree in one clock cycle.

We believe that our FPGA implementation is the best approach to achieve the highest performance. We think that the performance that could be obtained by implementing the algorithm on a GPU or other many-core accelerators such as CUDA-implementation would be poor for the following reasons.

First, the amount of shared memory (on-chip fast- access memory) available on a GPU or other many-core accelerators is much limited. For example, the GTX 980 Ti GPU has a total of 1,152kB of shared memory, which is much less than the 6,615kB of the Virtex-7 690T FPGA. Thus, using a GPU would either limit the size of phylogenetic datasets that can be processed or it would decrease the performance significantly by having to use the global memory.

Second, to calculate the score of a tree, we simply add up the individual results from each processing element (PE) by using a tree adder that has a latency equal to its depth. However, in the case of a GPU, the individual results from each thread block, which would be stored in the shared memory, would have to be written first into the global memory before they can all be added. This would signify a great delay that would reduce the overall performance.

Third, the first- and second-pass optimization algorithms are composed of some conditional clauses and low-level hardware operations. In the FPGA we implemented them as a group of logical gates and multiplexers inside the processing elements (PEs). Thus, it only takes one clock cycle to obtain the desired output. However, in a GPU this would not be possible. It would take more clock cycles to calculate the output, since the conditional clauses cannot be flattened and the low-level hardware operations cannot be directly implemented.

Our approach can be applied for larger problems as long as there are enough resources in the targeted FPGA. For even larger datasets than the ones we use, FPGAs such as the Virtex UltraScale XCVU190 FPGA can be used. On the other hand, the acceleration rate should increase theoretically with the number of DNA characters, because in our approach we process all characters in parallel. On the contrary, a software approach like TNT requires to process characters serially; thus, taking more time as its number increases.

Chapter 8

Conclusions and Future Directions

In this research we proposed and implemented five FPGA hardware approaches for molecular phylogenetic tree reconstruction using the maximum parsimony criterion.

Each approach was based on a stochastic local search algorithm (section 2.1.1) that combined one or more of the following algorithms: the Progressive Tree Neighborhood (section 2.2.1), the Indirect Calculation of Tree Lengths (section 2.2.2), the Alternative Second-pass Optimization (section 2.2.3), and the Incremental Tree Optimization (section 2.2.4).

We verified and evaluated the implementation of the proposed approaches for several real-world biological datasets of medium to large size. The datasets consisted of hundreds of sequences, each of them with thousands of DNA characters. We compared the implementation results from each approach with those from previous ones, and with those obtained from the phylogenetic software TNT (Tree analysis using New Technology), the fastest available parsimony program.

The implementation showed that our fifth and last approach was the fastest of all five. It achieved acceleration rates for the evaluation of a single tree rearrangement between 2.60 and 4.68 against the second approach (chapter 4); between 2.19 and 4.05 against the third approach (chapter 5); and, between 2.66 and 31.94 against TNT.

In conclusion, an efficient hardware approach for phylogenetic tree reconstruction has been successfully implemented. The fifth and last approach allowed us to obtain even higher acceleration rates in comparison to the first, second, third and fourth approaches and TNT, and required less memory resources than the fourth approach. These acceleration rates achieved are thanks to a combination of using the Indirect Calculation of Tree Lengths method, the Incremental Tree Optimization method, and the parallel and pipeline processing used in this approach.

8.1 Contributions of this Work

The main contribution of this work is to present an FPGA hardware approach for phylogenetic tree reconstruction under maximum parsimony that effectively addresses the evaluation of a single tree rearrangement and the stochastic local search.

To the best of our knowledge, we achieved the next contributions. Our first approach presented the first FPGA hardware implementation of the Progressive Tree Neighborhood algorithm. Our second approach presented the first FPGA hardware implementation of the Indirect Calculation of Tree Lengths algorithm. Our third approach presented the first FPGA hardware implementation of the Alternative Second-pass Optimization algorithm. Finally, our fourth and fifth approaches presented the first FPGA hardware implementation of the Incremental Tree Optimization algorithm.

In short, for the first time in the literature an FPGA hardware approach that covers both the complete and incremental first- and second-pass optimization, as well as the tree rearrangement evaluation has been proposed. Thanks to the parallel and pipeline processing, our approach achieves an ideal throughput of one node per clock cycle (as the number of nodes in the tree increases) for both the tree optimization and the tree rearrangement evaluation. In addition, our approach can be applied for larger problems as long as there are enough logical resources in the targeted FPGA. And given that it is modular and portable, it can easily be implemented on other FPGAs.

8.2 Future Directions

Our approach can be applied for larger phylogenetic problems as long as there are enough resources in the targeted FPGA; in particular, memory resources (BRAMs). For even larger problems, we suggest using an array of FPGAs.

The computation of the first- and second-pass optimizations, as well as the tree rearrangement evaluation is performed independently for each column of the Sequence Alignment Matrix. This means that the Sequence Alignement Matrix Memory, as well as other memories used to store the DNA character states, can be divided in smaller segments, so that each FPGA can store one of them. By doing this, each FPGA can perform calculations on the given segment of the memory, and the individual results regarding the score of the tree can be added together to obtain the total score of the tree.

Moreover, we suggest implementing our approach as part of a disk-covering method (DCM) such as the ones presented in works [46] [47] and [48]. This, we believe, would lead to achieve a higher phylogenetic tree reconstruction accuracy and reduction of memory consumption.

Bibliography

[1] A.W.F. Edwards and L.L. Cavalli-Sforza. "The reconstruction of evolution". In: *Annals of Human Genetics* 27 (1963), pp. 105–106.

[2] E.O. Wiley and B.S. Lieberman. *Phylogenetics: Theory and Practice of Phylogenetic Systematics*. 2nd ed. Wiley-Blackwell, 2011.

[3] D.K. Yeates, R. Meier, and B. Wiegmann. *Flytree*. 2017. URL: http://wwx.inhs.illinois.edu/research/flytree/flyphylogeny/ (visited on 12/07/2017).

[4] N.H. Barton, D.E.G. Briggs, J.A. Eisen, D.B. Goldstein, and N.H. Patel. *Evolution*. 1st ed. Cold Spring Harbor Laboratory Press, 2007.

[5] Barry G. Hall. *Phylogenetic Trees Made Easy: A How To Manual*. 4th ed. Sinauer Associates, Inc., 2011.

[6] L.R. Founds and R.L. Graham. "The Steiner problem in phylogeny is NP-complete". In: *Advances in Applied Mathematics* 3 (1982), pp. 43–49.

[7] W. Day, D. Johnson, and D. Sankoff. "The computational complexity of inferring rooted phylogenies by parsimony". In: *Mathematical biosciences* 81.33-42 (1986), p. 299.

[8] A.A. Andreatta and C.C. Ribeiro. "Heuristics for the Phylogeny Problem". In: *Journal of Heuristics* 8 (2002), pp. 429–447.

[9] A. Goëffon, J.-M. Richer, and J.-K. Hao. "Heuristic Methods for Phylogenetic Reconstruction with Maximum Parsimony". In: *Algorithms in Computational Molecular Biology: Techniques, Approaches and Applications* (2011).

[10] D.M. Hillis, C. Moritz, and B.K. Mable. *Molecular Systematics*. 2nd ed. Sinauer Associates, 1996.

[11] The European Bioinformatics Institute (EMBL-EBI). *Why use molecular data?* 2017. URL: https://www.ebi.ac.uk/training/online/course/introduction-phylogenetics/what-phylogenetics/why-use-molecular-data (visited on 12/07/2017).

[12] The European Bioinformatics Institute (EMBL-EBI). *Why is phylogenetics important?* 2017. URL: https://www.ebi.ac.uk/training/online/course/introduction-phylogenetics/why-phylogenetics-important (visited on 12/07/2017).

[13] Xilinx. *What is an FPGA?* 2017. URL: http://www.xilinx.com/training/fpga/fpga-field-programmable-gate-array.htm (visited on 12/12/2017).

[14] Pong P. Chu. *FPGA Prototyping By Verilog Examples: Xilinx Spartan-3 Version*. 1st ed. Wiley-Interscience, 2008.

[15] S. Che, J. Li, J.W. Sheaffer, K. Skadron, and J. Lach. "Accelerating compute-intensive applications with GPUs and FPGAs". In: *Application Specific Processors, SASP* (2008), pp. 101–107.

[16] University of Washington. *Phylogeny Programs*. 2017. URL: http://evolution.genetics.washington.edu/phylip/software.html (visited on 12/14/2017).

[17] University of Washington. *PHYLIP*. 2017. URL: http://evolution.genetics.washington.edu/phylip/ (visited on 12/15/2017).

[18] David Swofford. *PAUP*. 2017. URL: http://paup.scs.fsu.edu (visited on 12/15/2017).

[19] S. Kumar, G. Stecher, and K. Tamura. *MEGA*. 2017. URL: http://www.megasoftware.net/home (visited on 12/15/2017).

[20] Goloboff, Farris, and Nixon. *TNT*. 2017. URL: http://www.lillo.org.ar/phylogeny/tnt (visited on 12/14/2017).

[21] P. Goloboff, J. Farris, and K. Nixon. "TNT, a free program for phylogenetic analysis". In: *Cladistics, The International Journal of the Willi Hennig Society* 24 (2008), pp. 774–786.

[22] P. Goloboff and S. Catalano. "TNT version 1.5, including a full implementation of phylogenetic morphometrics". In: *Cladistics, The International Journal of the Willi Hennig Society* 32 (2016), pp. 221–238.

[23] J.M. Hancock and M.J. Zvelebil. *Concise Encyclopaedia of Bioinformatics and Computational Biology*. 2nd ed. Wiley-Blackwell, 2014.

[24] S.Sarkar, T. Majumder, A. Kalyanaraman, and P.P. Pande. "Hardware accelerators for biocomputing: A survey". In: *Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCAS)* (2010).

[25] J.D. Bakos and P.E. Elenis. "A Special-Purpose Architecture for Solving the Breakpoint Median Problem". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 16.12 (2008), pp. 1666–1676.

[26] S. Kasap and K. Benkrid. "High performance phylogenetic analysis with maximum parsimony on reconfigurable hardware". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 99 (2010), pp. 1–13.

[27] N. Alachiotis and A. Stamatakis. "FPGA acceleration of the phylogenetic parsimony kernel?" In: *Proc. 21st International Conference on Field Programable Logic (FPL)* (2011), pp. 417–422.

[28] H. Block and T. Maruyama. "A hardware acceleration of a phylogenetic tree reconstruction with maximum parsimony algorithm using FPGA". In: *International Conference on Field-Programmable Technology (FPT)* (2013), pp. 318–321.

[29] H. Block and T. Maruyama. "An FPGA hardware acceleration of the indirect calculation of tree lengths method for phylogenetic tree reconstruction". In: *24th International Conference on Field Programmable Logic and Applications (FPL)* (2014), pp. 1–4.

[30] H. Block and T. Maruyama. "FPGA Hardware Acceleration of a Phylogenetic Tree Reconstruction with Maximum Parsimony Algorithm". In: *IEICE Transactions on Information and Systems* E100.D.2 (2017), pp. 256–264.

[31] H. Block and T. Maruyama. "An FPGA implementation of a phylogenetic tree reconstruction algorithm using an alternative second-pass optimization". In: *25th International Conference on Field Programmable Logic and Applications (FPL)* (2015), pp. 1–4.

[32] H. Block and T. Maruyama. "An FPGA hardware implementation approach for a phylogenetic tree reconstruction algorithm with incremental tree optimization". In: *27th International Conference on Field Programmable Logic and Applications (FPL)* (2017), pp. 1–8.

[33] H.H. Hoos and T. Stuetzle. *Stochastic Local Search: Foundations and Applications*. 1st ed. Morgan Kaufmann, 2004.

[34] A. Goeffon, J. M. Richer, and J. K. Hao. "Local search for the maximum parsimony problem". In: *Lecture Notes in Computer Science* 3612 (2005), pp. 678–683.

[35] J. Felsenstein. *Inferring Phylogenies*. 2nd ed. Sinauer Associates is an imprint of Oxford University Press, 2003.

[36] M.S. Waterman and T.F. Smith. "On the similarity of dendograms". In: *Journal of Theoretical Biology* 73 (1978), pp. 789–800.

[37] M. Zvelebil and J. Baum. *Understanding Bioinformatics*. 1st ed. Garland Science, 2007.

[38] W. Fitch. "Towards defining course of evolution: minimum change for a specified tree topology". In: *Systematic Zoology* 20 (1971), pp. 406–416.

[39] D. Sankoff and P. Rousseau. "Locating the vertices of a Steiner tree in an arbitrary metric space". In: *Mathematical Programming* 9 (1975), pp. 240–246.

[40] Fredrik Ronquist. "Fast Fitch-Parsimony Algorithms for Large Data Sets". In: *Cladistics* 14 (1998), pp. 387–400.

[41] A. Goeffon, J. M. Richer, and J. K. Hao. "Progressive tree neighborhood applied to the maximum parsimony problem". In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 5 (2008).

[42] P. Goloboff. "Methods for faster parsimony analysis". In: *Cladistics* 12 (1996), pp. 199–220.

[43] M. Yan and D.A. Bader. "Fast character optimization in parsimony phylogeny reconstruction". In: *Technical Report* (2003).

[44] J.M. Richer, A. Goffon, and J.K. Hao. "A Memetic Algorithm for Phylogenetic Reconstruction with Maximum Parsimony". In: *Evolutionary Computation, Machine Learning and Data Mining in Bioinformatics* (2009).

[45] *TreeBASE: "A database of phylogenetic knowledge"*. 2017. URL: `http://www.treebase.org/` (visited on 12/28/2017).

[46] D. Huson, S. Nettles, L. Parida, T. Warnow, and S. Yooseph. "The disk-covering method for tree reconstruction". In: *Proc. Algorithms and Experiments* (1998), pp. 62–75.

[47] D. Huson, S. Nettles, and T. Warnow. "Disk-covering, a fast-converging method for phylogenetic tree reconstruction". In: *Journal of Computational Biology* 6.3-4 (1999), pp. 369–386.

[48] U. Roshan, T. Warnow, B. Moret, and T. Williams. "Rec-I-DCM3: a fast algorithmic technique for reconstructing phylogenetic trees". In: *Computational Systems Bioinformatics Conference, 2004. CSB 2004. Proceedings. 2004 IEEE*. IEEE. 2004, pp. 98–109.

Research Achievements

Journals (First author)

- H. Block and T. Maruyama. "FPGA Hardware Acceleration of a Phylogenetic Tree Reconstruction with Maximum Parsimony Algorithm". In: *IEICE Transactions on Information and Systems* E100.D.2 (2017), pp. 256-264

International Conference Papers (First author)

- H. Block and T. Maruyama. "A hardware acceleration of a phylogenetic tree reconstruction with maximum parsimony algorithm using FPGA". In: *International Conference on Field-Programmable Technology (FPT)* (2013), pp. 318-321
- H. Block and T. Maruyama. "An FPGA hardware acceleration of the indirect calculation of tree lengths method for phylogenetic tree reconstruction". In: *24th International Conference on Field Programmable Logic and Applications (FPL)* (2014), pp. 1-4
- H. Block and T. Maruyama. "An FPGA implementation of a phylogenetic tree reconstruction algorithm using an alternative second-pass optimization". In: *25th International Conference on Field Programmable Logic and Applications (FPL)* (2015), pp. 1-4
- H. Block and T. Maruyama. "An FPGA hardware implementation approach for a phylogenetic tree reconstruction algorithm with incremental tree optimization". In: *27th International Conference on Field Programmable Logic and Applications (FPL)* (2017), pp. 1-8