Technical University of Crete

Department of Electronic and Computer Engineering

Master's Thesis

Exploitation of Parallel Search Space Evaluation with FPGAs in Combinatorial Problems:The Eternity II Case

Pavlos Malakonakis

Chania 2011

Technical University of Crete

Department of Electronic and Computer Engineering

Master's Thesis

Exploitation of Parallel Search Space Evaluation with FPGAs in Combinatorial Problems:The Eternity II Case

Examination Committee:	Prof. Apostolos Dollas (Supervisor)
	Prof. Dionysios Pnevmatikatos
	Prof. Ioannis Papaefstathiou

Pavlos Malakonakis

Chania 2011

ACKNOWLEDGEMENTS

I would like to thank Prof. Apostolos Dollas for the supervision, guidance and helpful hints for completing this work. I would also like to thank Prof. Dionysios Pnevmatikatos and Prof. Ioannis Papaefstathiou for their helpful comments.

I wish to acknowledge Mr. Miltiadis Smerdis who wrote the optimized software which was used in this work for comparison purposes, as well as Ms. Katerina Papayannaki and Mr. Antonis Papayannakis who pointed out this problem to us and helped us with the very cumbersome tabulation of the puzzle pieces.

I would also like to thank Mr. Markos Kimionis and all undergraduate and graduate students of Microprocessor and Hardware Laboratory for their assistance whenever needed. Last the Microprocessor and Hardware Laboratory for the supplies I had at my disposal during my study for my Master.

CONTENTS

ABSTRACT

The Eternity II puzzle is a combinatorial search problem which qualifies as a computational grand challenge. As no known closed form solution exists, its solution is based on exhaustive search, making it an excellent candidate for FPGA-based architectures, in which complex data structures and non-trivial recursion are implemented in hardware. This paper presents such an architecture, which was designed and fully implemented on a Virtex5 FPGA (XUP ML505 board). Despite the serial nature of the recursion, as parallelism can be applied with the initiation of multiple searches, the system shows a measured speedup of 2,6 vs. a high-end multi-core compute server.

I. INTRODUCTION

The Eternity II is an edge-matching puzzle which involves placing 256 square puzzle pieces into a 16 by 16 grid, constrained by the requirement to match adjacent edges. Each puzzle piece has its edges marked with different color combinations, each of which must match with its neighboring side on each adjacent piece when the puzzle is complete. It was invented by Christopher Monckton, and is marketed and copyrighted by TOMY UK Ltd [1],[2]. A puzzle competition was released on 28 July 2007 and a $2 million prize is being offered to the first complete solution. As expected, none has been found to date. The puzzle comprises of a 16X16 board with a pre-paced initial tile, and an additional 255 square tiles. A completed 4x4 puzzle is presented on Figure 1. There are a total of 22 different colors, out of which five are exclusive to the corner and border pieces, and 17 for the inner board pieces. Taking into account the fixed piece in the center and the restrictions set on the pieces on the edge, the number of possible configurations are [2]:

$$1 \times 4! \times 56! \times 195! \times 4^{195} = 1.115 \times 10^{557} \qquad \text{[Formula 1]}$$

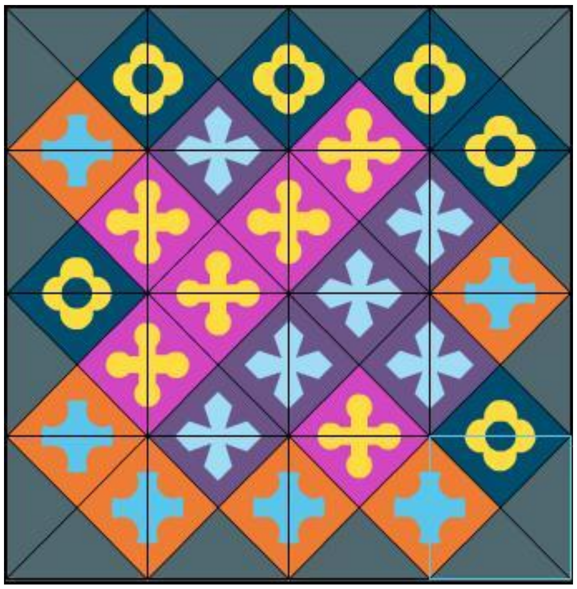

Figure 1. A completed 4X4 Eternity-II puzzle

Solving the puzzle is computationally hard. More precisely, it has been proven by Yasuhiko Takenaga et.al. [3], that deciding if there is a tiling of the Tetravex board is NP-complete and hence it has no simple closed form solution. The Tetravex game is the same puzzle game as the Eternity II. The extremity of the computation requirements for the Eternity II has attracted the interest of the research community. In this project we exploit the parallelism of an existing exhaustive search approach, in order to increase its performance.

We chose an exhaustive search approach for solving the puzzle using FPGAs. The main contribution of this work is that we tackle a problem which we know from previous research from other groups and ourselves [7-11] that it does not fare well vs. software implementations for a single instance of the algorithm. The existence of optimized software for this problem allowed for a good challenge on the hardware design process. This work (like our previous one) remains highly experimental, with full implementation on a Xilinx FPGA, the XC5V110T, on which the algorithm run for months at a time (with checkpointing for robustness).

The rest of the paper is organized as follows: Chapter II presents related work. Chapter III presents applications related to puzzle solving. Chapter IV presents the algorithm. In Chapter V our software implementations are analyzed. Chapter VI presents the proposed architecture. Chapter VII presents the space vs time tradeoffs of the various architectures and architecture improvements. Chapter VIII presents the software running on the host PC in order to establish communication with the FPGA. Chapter IX presents implementation results, the architecture’s performance evaluation from actual runs and comparisons to software solutions, followed by Conclusions in Chapter X.

II. RELATED WORK

A. *Eternity II Software Solvers*

The Eternity II caused the interest of the scientific community by being an NP-complete problem therefore multiple algorithms have been published since 2007 when the puzzle was introduced.

Pierre Schaus et. al.[4] presented an algorithm that first initializes the board with constraint programming by relaxing the problem and then the solution is improved with a very large neighborhood stochastic local search. The neighborhood is very large (i.e. exponential) but can be explored in polynomial time by solving an assignment problem.

Wim Vancroonenburg et. al.[5] introduced a two phase hyper-heuristic search method for solving the Eternity II puzzle. A hyper-heuristic manages a set of low level, problem specific, heuristics without knowledge of the problem domain, and tries to apply them to the problem in a meaningful way. In their approach they use a common hyper-heuristic framework. It consists of an iterative framework with two sub-mechanisms: a heuristic selection mechanism which uses some strategy to select a low level heuristic for generating a new (partial or complete) solution in the current iteration, and an acceptance mechanism to decide on the acceptability of the new solution.

Wei-Sin Wang et. al.[6] proposed a two-phase approach to solve the Eternity-II puzzles mainly based on the tabu search algorithm. The first phase solves the outside region of the puzzle, and then based on the result obtained in the first phase the entire puzzle is solved in the second phase. The neighborhood functions are based on swap and rotation. Random perturbation and simulated annealing are included to escape from the local optima.

Jorge Munoz et .al.[12], in their work evaluate a genetic algorithm and a multiobjective evolutionary algorithm in a constraint satisfaction problem. The problem that was chosen is the Eternity II puzzle. The objective is to analyze the results and the convergence of both algorithms in a problem that is not purely multiobjective but that can be split into multiple related objectives. For the genetic algorithm two different fitness functions were used, the first one as the score of the puzzle and the second one as a combination of the multiobjective algorithm objectives.

Marijn J.H. Heule [13], proposed a way of solving edge-matching problems with satisfiability solvers. The usefulness of these solvers does not only depend on their strength and the properties of a certain problem, but also on how the problem is translated into SAT. To show the impact of the translation on the performance, encodings of edge-matching problems was studied. There exists no straightforward translation into SAT for edge-matching problems such as Eternity II. A compact translation of edge-matching problems into CNF was used, which can be extended by using redundant clauses representing additional knowledge about the problem. The results show that these redundant clauses can guide the search – both for complete and incomplete SAT solvers – yielding significant performance gains.

Also, many unpublished exhaustive search software implementations can be found on the internet.

B. *Special-Purpose Hardware for Combinatorial Problems*

A special purpose hardware addressing the Eternity II problem was implemented by Vladimír Kašík [18]. In his work the implementation of a fast computation of the Backtracking Algorithm with FPGA logic is presented. The specific problem is encoded into the FPGA structure and solved in the hardware. Each partial candidate of the solution is then put / rejected (pushed / popped) in a single clock cycle only. This implementation can achieve a performance of about 8MNodes/sec. A comparison between his implementation and ours is done in Chapter VIII.

There also exist a variety of combinatorial problems that have been studied and the corresponding special purpose hardware was implemented with the scope of achieving SpeedUps vs the equivalent software implementations.

Marco Platzner [17] in his work states that reconfigurable accelerators can improve process time on combinatorial problems with fine-grained parallelism. Such problems contain a huge number of logical operations (NOT, AND, and OR) that can evaluate simultaneously, a characteristic that varies considerably from problem to problem. Because of this variability, such combinatorial problems are approached using instance-specific reconfiguration—hardware tailored to a specific algorithm and a specific set of input data. Boolean satisfiability is a common combinatorial problem that exhibits fine-grained parallelism that varies considerably based on the situation. Its solution is thus an ideal candidate for improvement with instance-specific reconfiguration. In fact, simulations of an instance-specific accelerator show potential speedups by a factor of up to 140,000 in execution time over the solution by a software solver.

The Golomb ruler derivation problem was thoroughly studied at the Microprocessor and Hardware Laboratory of the Technical University of Crete. The results were three hardware implementations of the Shift Algorithm. These architectures were GE1 [14] which is a single processor engine with 20 Xilinx XC5000 FPGAs. GE1 was followed by GE2 [15] which is an FPGA multiprocessor architecture developed jointly at TUC and Virginia Tech. In GE2, PC-run software produced stubs (Golomb Rulers with fewer marks than the actual ruler of the search) and many consumer GEs (hardware) calculated each stub, until all the search space is searched. The last architecture developed at TUC is GE3 [16], which is a single-chip client-server architecture. The new client architecture supports parallel evaluation of multiple hypotheses (up to 16), each implemented as a shift operation and one server can support many clients. This architecture presented a SpeedUp of about 570, with a very large FPGA in terms of slice resources, vs the optimized software.

Spyridon Ninos et.al. [11], presented a general, data-oriented approach to implement recursion on reconfigurable hardware. Recursion is a powerful technique used to solve problems with repeating patterns, and is a fundamental structure in software. They demonstrate that recursion can be efficiently implemented in a general way on FPGAs. This architecture was used for the implementation of two algorithms; the knight's tour and a binary tree search.

Valery Sklyarov [7] in his work suggests a novel method for implementing recursive algorithms in hardware. The required support for recursion has been provided through a modular and a hierarchical specification of a control unit that can be translated to an implementation of the respective hardware circuit on the basis of a recursive hierarchical finite state machine and through a mechanism that permits the contents of an execution unit to be stored/restored between hierarchical calls/returns.

Valery Sklyarov et.al. [10], at a later publication analyses and compares alternative iterative and recursive implementations of N-ary search algorithms in FPGAs. The improvements over their previous results have been achieved with the aid of the proposed novel methods for the fast implementation of hierarchical algorithms. The methods possess the following distinctive features: 1) providing sub-algorithms with multiple entry points; 2) fast stack unwinding for exits from recursive sub-algorithms; 3) hierarchical returns based on two alternative approaches; 4) rational use of embedded memory blocks for the design of a hierarchical finite state machine.

III. Applications

As stated in Chapter II, the Eternity II puzzle caused the interest of the scientific community about edge matching puzzle problems. The architecture presented here was designed in order to measure the capabilities of reconfigurable hardware on such problems. Also this architecture, with the appropriate modifications, can be used for solving a variety of problems.

From a Puzzle Type to Another

This architecture can address any edge matching puzzle with the appropriate adjustments, e.g. the initialization of the memories that contain the puzzle pieces, adjustments concerning the puzzle size. Also as shown by Erik D. Demaine and Martin L. Demaine in [19], jigsaw puzzles, edge-matching puzzles and polyomino packing puzzles (introduced by Solomon W. Golomb), are all NP-complete and furthermore, any puzzle of one type can be converted into an equivalent puzzle of any other type. Jigsaw puzzles have a guiding image and each side of a piece has only one “mate”, although a few harder variations have blank pieces and/or pieces with ambiguous mates. In an edge-matching puzzle (Eternity II), the goal is to arrange a given collection of several identically shaped but differently patterned tiles (typically squares) so that the patterns match up along the edges of adjacent tiles. In a harder abstract form of edge-matching puzzles, edges also have a sign (+ or −) and adjacent tiles must have opposite sign (like magnetism). Polyomino packing puzzles were introduced by Golomb around 1965 [20]. These puzzles were popularized by the Eternity puzzle. A polyomino arises from gluing unit squares together edge-to-edge. In general, a polyomino arises from edge-to-edge gluing of several copies of a simple shape, such as a square, an equilateral triangle, or an equilateral triangle cut in half (as in Eternity). In a polyomino packing puzzle, the goal is to pack a given collection of several polyforms into precisely a given shape—the target shape—such as a larger rectangle, rhombus, or dodecagon (as in Eternity).

They show that, computationally, these three types of puzzles are all effectively the same. A puzzle of one type can be converted into an equivalent puzzle of each of the other types, with a small blowup in puzzle size. Every solution in one puzzle corresponds to a solution in the other puzzle, by a simple, efficient, and invertible conversion. The transition from a puzzle type to another is shown on Figure 2.

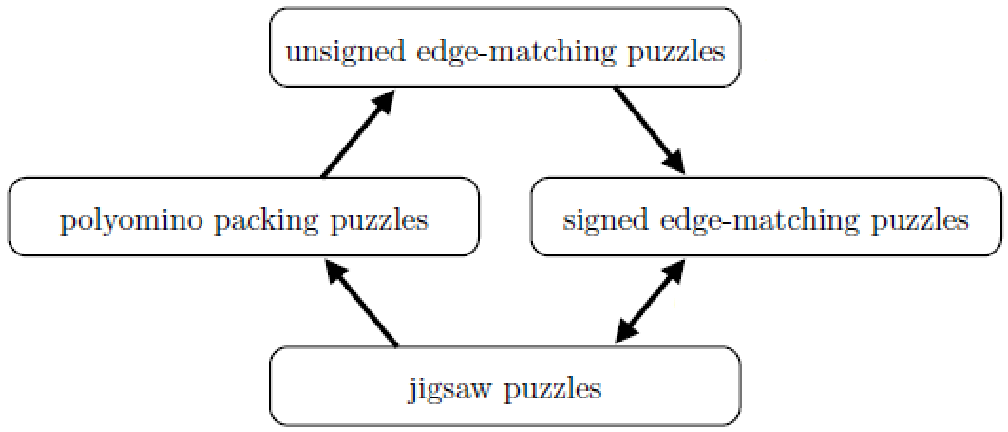
unsigned edge-matching puzzles
polyomino packing puzzles
signed edge-matching puzzles
jigsaw puzzles

Figure 2. Transitions between puzzle types

The Eternity II puzzle is an unsigned edge matching puzzle. By taking into account the methods described in [19] we can state that our system is able to solve any kind of these puzzles. The steps needed for a puzzle to become eligible to be solved by our architecture are:

- Make the appropriate changes on the puzzle pieces so the puzzle will become an edge matching puzzle.
- Create the piece data structures that need to be loaded on the architecture.
- Make any adjustments on the architecture if needed.
- Allow the hardware to solve the problem.
- Translate the result back to its original form.

Speech Descrambling

This allows our architecture to address a variety of problems that can be translated in any kind of the puzzles described above. Such a problem is presented in [21]. Y-X. Zhao et. al. presented the application of puzzle solving in speech descrambling. The security problem of speech communication has always been a demanding problem in military and business areas. A common approach to realizing end-to-end security is the use of a scrambler. Most of the scramblers are based on permutation of speech signals in the time domain and/or frequency domain. On the other hand, descramblers are used to eavesdrop information from scrambled speech signals.

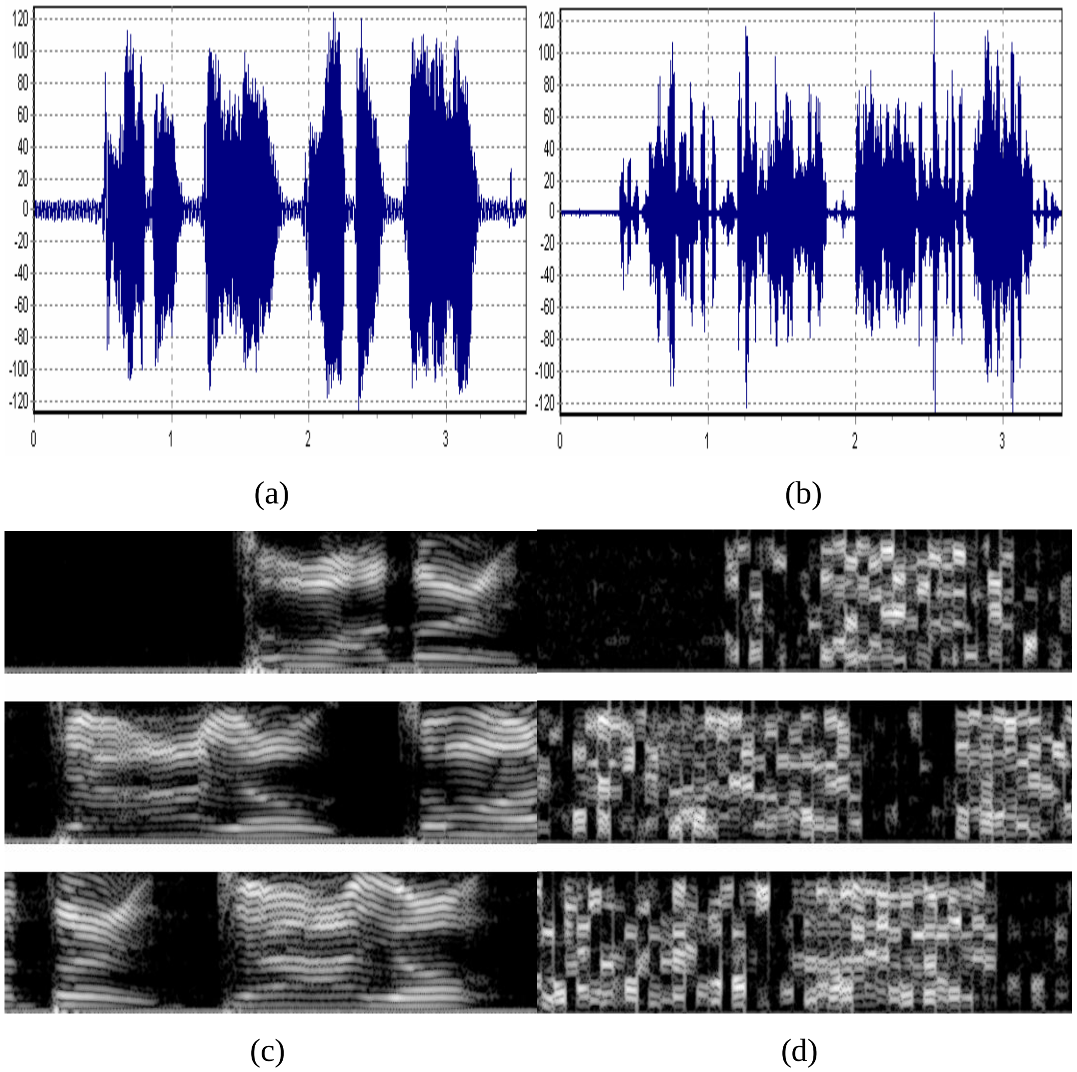

Figure 3. Scrambling a speech signal: (a) the original speech signal in time domain. (b) scrambled speech signal in time domain. (c) the original spectrogram. (d) the scrambled spectrogram.

They propose a new approach to implement a descrambler. They treat the descrambling problem as a puzzle solving problem. In this considered puzzle problem, each piece is a rectangular-shaped gray scaled image puzzle. They propose two different methods to assemble puzzles. The first method is based on human heuristics and the second method is based on the Ant Colony System (ACS) algorithm.

The scrambled signal is composed of a number of rectangular-shaped gray scaled pieces as shown on Figure 3. These pieces match the attributes of the edge matching puzzle pieces. So this puzzle can be considered as an edge matching puzzle. In order for it to be applicable for our architecture the only thing that needs to be done is the coding of each piece's edges.

VLSI Placement Problems

Another application that our architecture could address is the placement problems of VLSI modules and die-set parts. As stated by Kouki Kimoto et.al. in [22], developing a new algorithm for two-dimensional (2D) polyomino packing problems is expected to lead to another new algorithm for real applications, such as the placement problems of VLSI modules and die-set parts. Floor-plan design in VLSI circuit layout is a typical example of placement problems, where circuit modules, usually rectangularly shaped or at most L-shaped, are placed so as to minimize the total area occupied by them. Several approaches to solving the placement problem of VLSI modules have been presented. In these approaches, the typical procedure involves representing placement topology by a graph-theoretic order-pair expression or block-slicing-Polish expression, giving fluctuations to the topology by simulated annealing, and then improving the objective for optimal placement. They first explored two-dimensional (2D) polyomino packing problems (2DPPPs) of polyominoes of any shape topology. It is well known that a 2D-PPP of this type shares common features with general placement problems; however, it has, by definition, its own special features such that it has at least one solution of placement, that a feasible solution is always optimal, and that not many solutions are expected. This problem requires the transition from a polyomino packing puzzle to an edge matching. The method for doing so is described in [19].

Image Watermarking

Muhammad Jamil Anwar et.al. introduce in their paper [23] a new method of adaptive blind digital image watermarking in spatial domain. Watermarking is a way to hide secret information in a cover image or host medium. In the proposed method both cover and secret images are partitioned into equal size blocks. Then host for each secret image block is intelligently selected through Genetic Algorithm. At the watermark extraction phase only watermarked image is required from which using Jigsaw Puzzle Solver (JPS) the secret image is reconstructed (Figure 4). JPS works by combining blocks based on block edge and texture information in it. In JPS an image is reconstructed on the basis of image contents instead of image geometry. Initially a scattered image piece is taken as a complete image and randomly remaining pieces are taken and are matched with the completed piece to form the actual image. Our architecture could work on the reconstruction stage as the Jigsaw Puzzle Solver. Actually the pieces being rectangular and only the edges matter for the solving of the puzzle basically makes it an edge matching puzzle. As a result it can be solved by our architecture with the only need of the coding of each puzzle piece's edges.

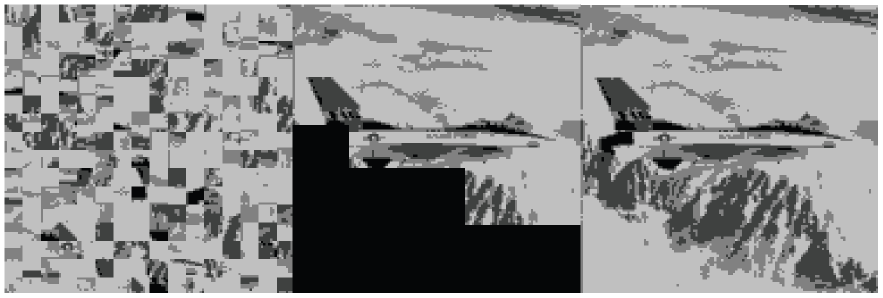

Figure 4. (a) Scattered secret image. (b) Secret image partially retrieved after application of JPS. (c) Retrieved complete secret image.

Archeology

Jigsaw puzzle solving, and thus our work, can be used also in archeology. Florian Kleber et. al. in [30] states that an automated assembling of shredded/torn documents (2D) or broken pottery (3D) will support philologists, archaeologists and forensic experts. They present an overview about current puzzle applications in Cultural Heritage. As stated by Benedict J. Brown et. al. in [24], at Akrotiri on the volcanic island of Thera (modern-day Santorini, Greece) as at other excavations, recent computer graphics research may significantly improve the quality of artifact documentation and reduce the human labor involved in matching the "jigsaw puzzle" of fragments (currently estimated at 75% of the total human effort at the site), freeing up time for other important tasks including conservation and restoration. Also Georgios Papaioannou et.al. in their work "Virtual Archaeologist: Assembling the past" [25] stated that the reconstruction of arbitrary objects from their fragments can be regarded as a 3D puzzle. They present a complete method, encapsulated in Virtual Archaeologist system, for the full reconstruction of archaeological finds from 3D scanned fragment (Figure 5). The archeologists create a database by scanning the fragments that are found at an excavation site. These fragments can be stored and then translated into edge matching pieces by mathematical equations. These pieces have different sizes and structures. Each fragment can be coded with more than one piece, which have exclusive ids at their adjacent edges making them inseparable at the final solution. Then the piece memory data structures of our architecture can be initialized by those pieces. A starting piece will be predefined and if the search cannot move any further the result will be sent to the output and the search will continue with a different starting fragment/piece to produce another partial solution. These partial solutions can correspond to different archeological finds. For example by starting from a certain fragment (piece) the result can be a part of a pottery and by starting from another the result can be a small statue or marble inscription. These results are then checked by archeologists in order to ensure the correctness of the matching.

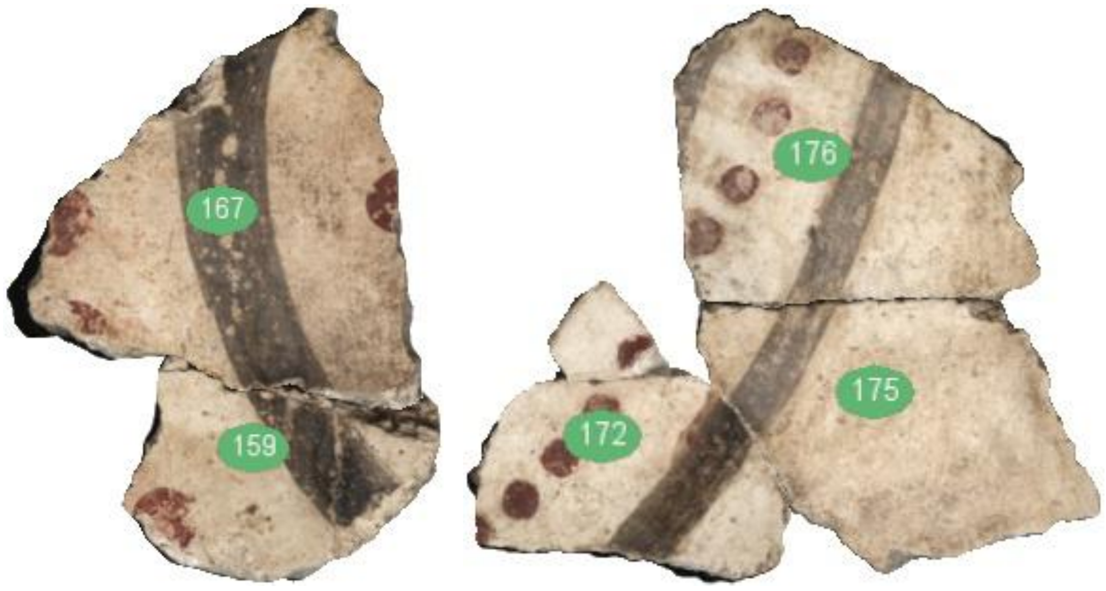

Figure 5. Reconstruction of archaeological finds from scanned fragments.

Ripped-up Document Reconstruction

Puzzle solving can be also used for the reassembly of torn documents. Florian Kleber et. al. in their work [26] state that reassembling of torn documents is related to the traditional puzzle games. The main difference to canonical jigsaw puzzle games is the irregular shape of the fragments and the content (mainly text in documents compared to images in jigsaw puzzles). Also Liangjia Zhu et. al. in [29] proposed a method for the recovery of ripped-up documents. Ripped-up document reconstruction is a problem that often arises in archival study and investigation science. Documents may be ripped up by hand or shredded by a machine. In both cases, the automatic or semiautomatic reconstruction of the original document would alleviate the manual effort, which is difficult and time-consuming. They proposed a method for the ripped-up document reconstruction by first finding candidate matches from document fragments using curve matching and then disambiguating these candidates through a relaxation process to reconstruct the original document. The same methodology as for the archeology fragments can be used for the reconstruction of fragmented documents. Through the same preprocessing the pieces initialize the appropriate memories and then the partial results, which will probably be different independent pages or parts of pages, can be checked by the investigators and be reconstructed. An example of ripped-up document reconstruction is shown on Figure 6.

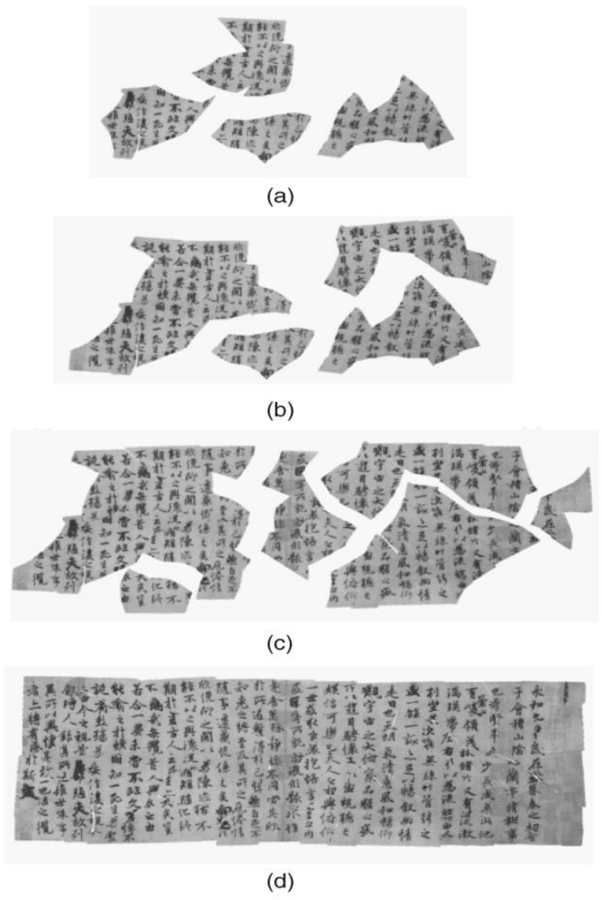

Figure 6. Ripped-up Document Reconstruction.

Cutting and Packing Problems

F. Hoshi et. al. presented in their work [27] that polyomino packing solving can be used in many applications involving cutting and packing, which are encountered in many industries. The cutting problems and the packing problems are essentially the same both on their objectives and theoretical implications. The objectives of the cutting problems are to cut a given set of product parts on given mother materials so as to maximize the usage of the mother materials, that is, to minimize the "trim loss". On the other hand the objectives of the latter problems are to place a set of pieces on given plates or bins so as to maximize the usage of their space. The wood-, glass- and paper industry are mainly concerned with the cutting of regular figures, whereas in the ship building, textile and leather industry, irregular, arbitrary shaped items are to be packed. Also, we can see examples of the packing problems in such as vehicle loading and pallet loading problems. They proposed a new deterministic algorithm for the problem. They devised a game-theoretic solution for two-dimensional (2D) cases, which were treated as rectilinear jigsaw puzzles, i.e., specialized 2D polyomino packing problems such that there is at least one placement solution. The new method is developed by extending the polyomino packing algorithms to solve the given problems. Our architecture could take the place of the polyomino packing algorithm by making the appropriate changes to the problem in order for it to become an edge matching puzzle. Each polyomino piece is translated into many edge matching pieces that have the size of the initial unit blocks of the polyomino. Each polyomino block's pieces have their adjacent edges ids being exclusive for each edge matching of that polyomino so at the solution of the puzzle these pieces are connected back together to form the starting polyomino. The outer edges have common ids in order to allow other polyominos to be connected to them. The solution of this puzzle has to be translated back to the initial polyominos by decoding the pieces that are exclusively connected together.

Protein- Folding

Another field of science that puzzle solving can be used is in molecular biology. P. Gorder in [28] presents how scientists took one small but intriguing step toward solving the protein- folding problem by synthesizing a protein called Top7. How protein molecules form into useful shapes— and what causes proteins to go wrong is unknown. It's a puzzle called the protein-folding problem, and it's key to developing treatments for diseases as diverse as Alzheimer's, Parkinson's, cataracts, cystic fibrosis, and diabetes' most common form. Building the protein is like creating a jigsaw puzzle. There are 20 amino acids commonly found in proteins, and each one can rotate to form some 10 different shapes. That means that 200 options exist for each puzzle piece's shape and orientation— and a protein can contain hundreds of pieces. By solving the jigsaw puzzle the scientists managed to synthesize the Top7 protein. By providing the pieces, appropriately coded/colored as edge matching pieces, to our architecture it can take the place of the protein-folding puzzle solver. The solver will start from a certain piece and try to connect as many pieces as possible until it returns the result. After that the starting piece will change in order for another result to be produced. These output results can be proteins that will have to be confirmed by biologists.

DNA self-Assembly

X. Ma and F. Lombardi in [31] present that DNA self-assembly has been advocated as a bottom up manufacturing technology (as applicable in the nano scales) and for algorithmic computation. They consider the synthesis of tile sets for DNA self-assembly and analyze it as a combinatorial optimization problem. This problem is referred to as PATS (Pattern Assembling Tile-set Synthesis). As CMOS is approaching its physical limits, emerging technologies have been investigated for building future computing systems. DNA-based self-assembly has emerged as a promising technology for manufacturing from “bottom-up” without conventional lithography. Using DNA complexes, or so-called tiles as building blocks, programmable self-assembly of nanoscale structures has been demonstrated. For manufacturing, DNA self-assembly relies on defining a set of tiles to target a specific pattern. The programmability of DNA in the form of tiles can be exploited to manufacture circuits based on single-molecule electronic devices, such as molecular transistors or quantumdot cellular automata (QCA) cells. The basic principle is to utilize the programmability of DNA tiles to self-assemble periodic and aperiodic lattice structures. Then, it is possible for other molecular electronic devices to selectively attach to the lattice. An obvious advantage of this technique is that a massive parallel assembly is possible, such that millions of copies of the desired structure can be built simultaneously. This technique has been advocated as a method for “bottomup” nanofabrication.

The abstract Tile Assembly Model (aTAM) provides the basis for analysis of algorithmic self-assembly in ideal cases [32]. A tile set consists of a finite set of unique square-shaped tiles that are used to self-assemble into a DNA aggregate. Each of the four sides of a tile has a bond type. Each bond type has an associated bond strength. Bond types can be null (strength of 0), single (strength of 1) or double (strength of 2). Two bonds of the same type can bond together, with a corresponding bond strength. It is assumed that the strength between different bond types is always 0. In aTAM, self-assembly always begins with a seed tile. A tile can be added to the existing aggregate when its total bond strength to the aggregate is greater than or equal to 2. The Sierpinski triangle tile set is a widely studied tile set [33]. The Sierpinski tile set is shown in Figure 7 (a). The four integers in a tile denote the bond types. The growth of an infinite Sierpinski triangle is shown in Figure 7(b), with smaller tiles representing the correct addition of tiles. The correct growth direction for this tile set is indicated by the arrows in Figure 7(b). The assembled pattern is shown in Figure 7(c). A simulation tool Xgrow [32] has been developed to emulate the DNA self-assembly. Xgrow is used to verify the tile set generated by the algorithms proposed in this paper.

This problem can also be addressed by our work. The DNA tiles resemble the edge matching puzzle pieces. The only difference is that a tile can be added to the existing aggregate when its total bond strength to the aggregate is greater than or equal to 2. Also the way the puzzle is filled is different, starts from a corner and expands towards the opposite direction. These changes can be implemented on our design in order for it to adapt to the needs of this problem. The bond strength that needs to be greater than or equal to 2 can be added as an extra constraint at the search of an appropriate tile and the filling order can easily be changed by initializing the appropriate memories with the filling sequence.

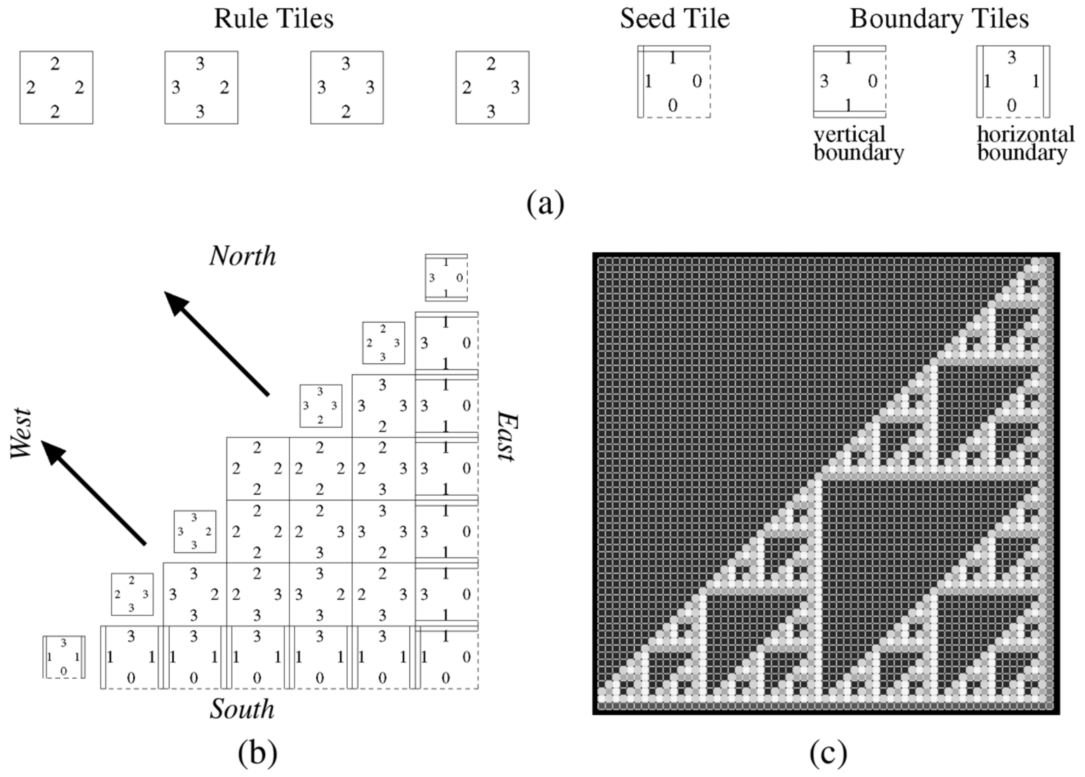

Figure 7. The Sierpinski Triangle Pattern. (a) Sierpinski tile set. (b) Growth of tile set. (c) Assembled pattern.

IV. Algorithm Selection

The selected algorithm is based on a straightforward exhaustive search with a depth first search (DFS) algorithm. The pieces are continuously placed on the board, taking into account all neighboring restrictions. If at any moment no piece can be placed without violation of those restrictions, the most recently inserted piece is removed, and an alternative piece will be selected for that position.

The Depth-First-Search (DFS) algorithm of choice requires complex data structures (e.g. in order to know which tiles are placed and which are available for a large number of tiles which are non-uniformly grouped in the puzzle), it has a recursive descend depth of 196 (even after we have excluded the borders) vs. 64 of the knight's tour on a chessboard problem which we had studied and reported before [11], and it is control intensive. Therefore it is as "bad" a problem to map on FPGA's as possible. This line of research aims at the long-term development of suitable hardware data structures for FPGA's, to cope with their low clock rate when compared to general-purpose processors when they tackle control-intensive serial problems.

The order, in which the empty positions of the board are filled, is graphically presented in the Figure 8 below, and it follows a center-counterclockwise spiral pattern.

35	34	33	32	31	30
16	15	14	13	12	29
17	4	3	2	11	28
18	5	0	1	10	27
19	6	7	8	9	26
20	21	22	23	24	25

Figure 8. Board Filling Order

The insertion of the pieces begins at the adjustment places of the fixed piece, and gradually surrounds the current square until they completely fill the inner square. This strategy was chosen to exploit the search space reduction that the fixed piece provides.

The architecture implemented to run on hardware concerns only the 196 inner pieces. The 60 border pieces are solved by software that receives the 196-piece solutions from the FPGA, which is a fairly trivial task for a general-purpose computer. This way the complexity for the FPGA is lowered by a factor of 4! × 56!. So the Formula 1 becomes for the hardware design:

$$195! \times 4^{195} = 6.55 \times 10^{480} \qquad \text{[Formula 2]}$$

It is useful to note here that each piece can be placed on the board with any of its four rotations. As a result each piece counts as four different pieces when it is checked in order to be placed on the board.

A pseudo code for the DFS algorithm is presented below:

```
Board.initialize(); // set up fixed piece
Pieces.initialize(); // a set with all pieces
Slot.initialize(); // Initial position of the board
Popid = 0; // Used for exclusion of previously used pieces

   for all Piece in Pieces
         if Piece.getUsed() == False and Piece.id() > Popid then
                for all Orientation of Piece
                      if Board.match(Slot, Piece, Orientation) then
                           Board.place(Slot, Piece, Orientation);
                           Piece.setUsed(True);
                           Popid = 0;
                           Slot.next();

         if Board.isEmpty(Slot) then
                Slot.prev();
                Popid = Board.piece(Slot).id();
                Board.piece(Slot).setUsed(False);
                Board.clear(Slot);

         If Board.isFull() then
                Celebrate(); // solution is found

until Board.isEmpty() == False
```

V. Software

After the selection of the algorithm the equivalent software was implemented. As described above the filling of the table follows a center-clockwise spiral pattern. For each slot on the table all the available pieces are checked in order to find a piece that has the requested values at its edges, which have to be the same as its neighboring already placed pieces. This software is not optimized, as it has to search between all the pieces for each slot. The performance it can achieve is about 0,6 MNodes/sec on an Intel Xeon at 2,66GHz.

As a project for a postgraduate course, Architecture of Parallel and Distributed Computers, a parallel implementation of this software was created in order to run on the Grid processors. This architecture is a producer (server) – consumer (client) system which allows a divide-and-conquer approach on the search space. The single producer provides a centralized job distribution service and is also responsible for the allotment of the search space between the consumers. On the other hand, each of the consumers constitutes a computation unit that will thoroughly search the designated chunk of the search space for solutions. The Division of the search space, by the producer, is implemented by applying a depth limit on the initial algorithm. Each of the partial solutions will be sent to the consumers to complete the search. The communication between the processors is done by using MPI. The scalability of the parrallized algorithm was proved to be linear as shown on Table I and Figure 9 below. Also we need to note even the single core implementation is much slower on the Grid processors than the Intel Xeon at 2,66GHz processor and in fact 3 times slower. This is due to the processor architecture and the clock frequency.

TABLE I. PERFORMANCE ON GRID

	Performance (MNodes/sec)	Speed up
Grid Single core	0,19	1
Grid 1 Consumers / 1 Producer	0,19	1,01
Grid 4 Consumers / 1 Producer	0,76	4,02
Grid 8 Consumers / 1 Producer	1,54	8,11
Grid 16 Consumers / 1 Producer	3,10	16,30
Grid 32 Consumers / 1 Producer	6,25	32,85
Grid 36 Consumers / 1 Producer	7,03	36,96

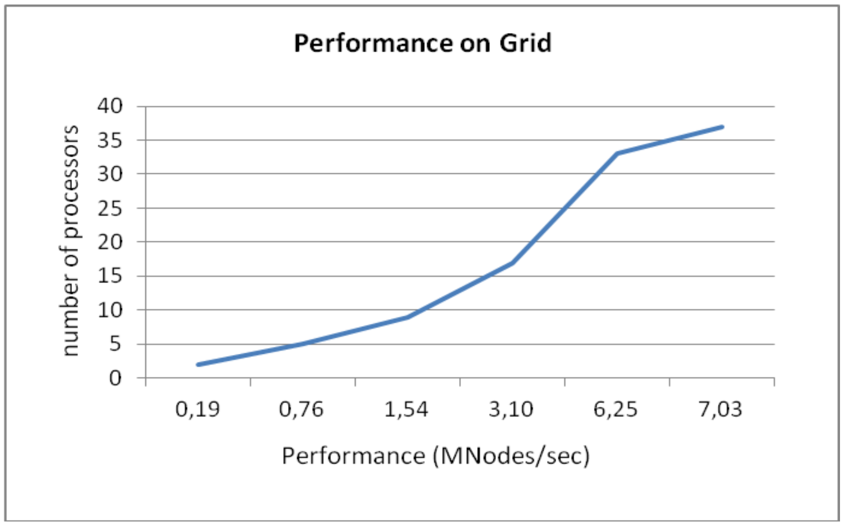

Figure 9. Performance on Grid

As the software presented above was not optimized a new version was implemented that can achieve a performance of 26MNodes/sec on an Intel Xeon at 2,66GHz. This software was developed by Mr. Miltiadis Smerdis at the Microprocessor and Hardware Laboratory of the Technical University of Crete. This software followed the hardware implementation and has the same algorithmic improvements. The most important improvement is the clustering of the pieces with regard to 1 edge and thus has 17 independent lists for single edged search (corner pieces) and 17X17 lists for the double edged search. So instead of searching between 196 pieces for a corner piece it searches between 44-49 and for double edged piece between 0 – 7 pieces. This effect provides a performance improvement of over 35. This improvement also decreases the complexity of the problem by a very significant amount. Actually the complexity is calculated as follows: 47^{27} for the corner board slots, where 47 is the average number of pieces that has each direction value and 27 represents the corner board slots, multiplied by 3^{169} which implies to the rest of the board slots (double edge search).So the actual complexity is:

$$47^{27} \times 3^{169} = 6{,}02 \times 10^{125} \qquad \text{[Formula 3]}$$

As a result the complexity goes from 6.55×10^{480} [Formula 2] to the exponentially less $6{,}02 \times 10^{125}$.

This optimized software implementation is analyzed further in Chapter VII. Also this software is considered optimal as unpublished implementations on the internet claim a performance of about 20MNodes/sec.

On the next Table we compare the performance of our first software with the optimized software.

TABLE II. COMAPARISON OF SOFTWARES

Architecture	Performance	SpeedUp
Smerdis Software on Intel Xeon 2,66GHz	26 MNodes/sec	1
First Software on Intel Xeon 2,66GHz	O,6 MNodes/sec	1/43
Grid Single core	0,19 MNodes/sec	1/137
Grid 36 Consumers / 1 Producer	7,03 MNodes/sec	1/3,5

As shown on the Table above the same software running on the Intel Xeon processor is about 3 times faster than on a grid processor. This happens due to the processor capabilities.

Also the optimized software is 43 time faster than the previous software and thus making it 3,5 times faster even if the first software runs on 36 grid processors. So, all the comparisons at Chapter VII are done with the hardware and the optimized software.

VI. An FPGA-Based Architecture

The FPGA-based architecture was based on two main blocks, one of which maintains the board and one which addresses placement of tiles, performs bookkeeping (i.e. keeps track of used/available tiles) and incorporates some search-space reduction techniques (e.g. in some cases in which a search can lead to no solution). The architecture will be described below.

A. *Top Level Architecture*

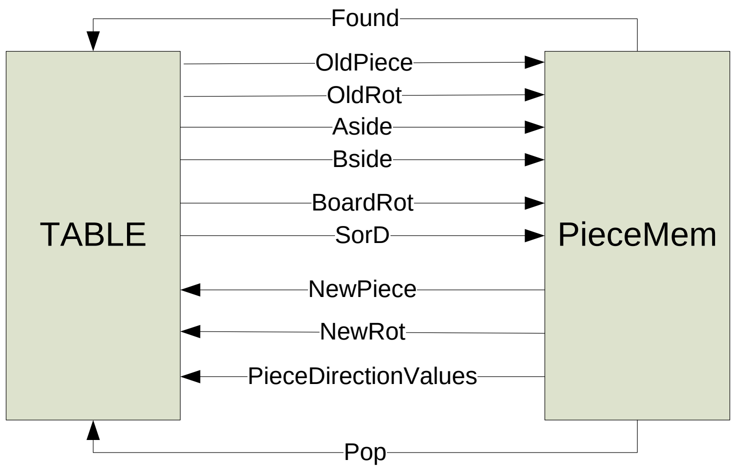

Figure 10. Top Level Architecture (EtSolver)

The Eternity II Solver (EtSolver) architecture has two distinct modules as shown on Figure 10. The TABLE module is the implementation of the board. It contains the appropriate memories for the board and makes requests at the PieceMem for the next piece that has to be placed on the board, by providing the necessary information. PieceMem is the memory that contains all the pieces and their side values. It receives a request from the TABLE and it responds with a piece matching the request or responds that there is no piece available with the attributes needed and thus the TABLE must pop the last placed piece. PieceMem module has also to keep track of the already used and the available pieces. The architecture of these two modules is presented thoroughly below.

B. *TABLE Architecture*

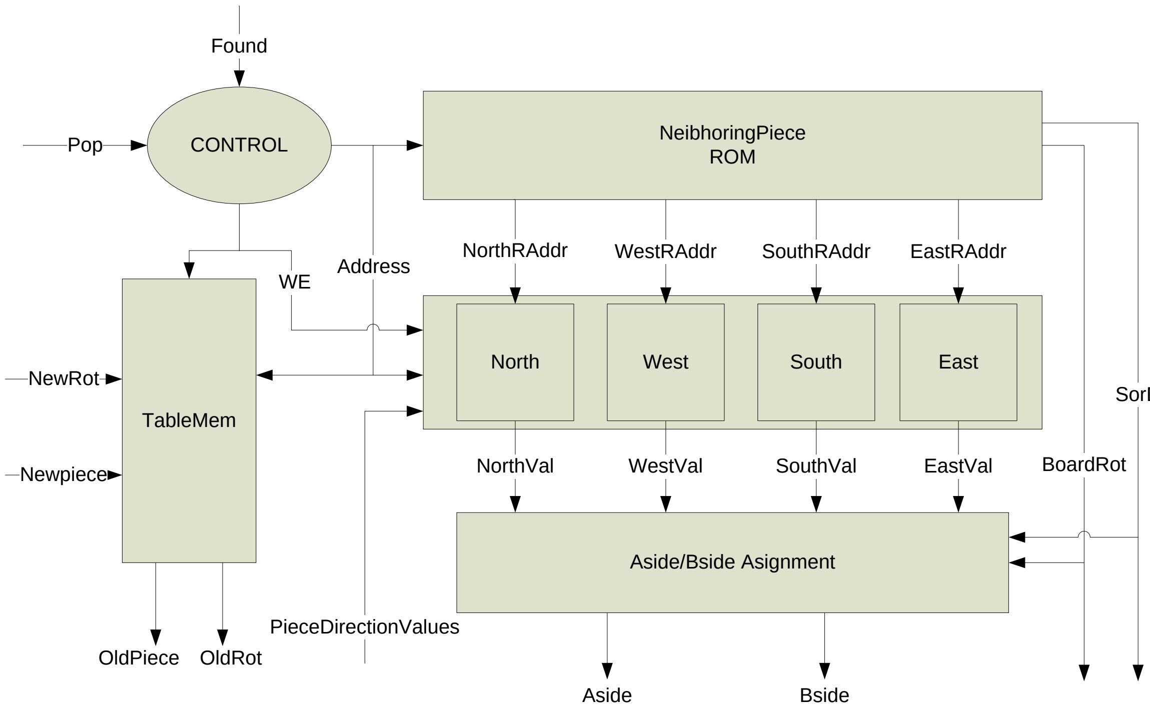

Figure 11. TABLE Architecture

The architecture of the TABLE (Figure 11) represents the Eternity II board. Its basic component is the TableMem that is the memory where the pieces are stored as they are placed on the board. Each slot contains the piece id and the rotation in which it is placed on the board (Table III). In this memory the board is rearranged with regard to the priority it is filled, e.g. the center (preplaced) piece is stored in address 0 the next piece (east of the center piece) is stored in address 1 and its north neighbor at address 2. Figure 12 shows the filling order of the board slots and on Figure 13 the rearranged board as it is saved on the TableMem is presented. The TableMem was created this way in order to minimize the cost of finding the next slot that needs to be filled. This way we need only one address pointer for the calculation of the next board slot that will be accessed. When a Push operation is executed only the address pointer needs to be increased by 1 and when a Pop operation is executed the pointer is decreased by 1. This memory has a width of 10 bit and a depth of 196.

TABLE III. TABLEMEM COMPONENTS

8 bit	2 bit
Piece id	Rotation

35	34	33	32	31	30
16	15	14	13	12	29
17	4	3	2	11	28
18	5	0	1	10	27
19	6	7	8	9	26
20	21	22	23	24	25

Figure 12. Board Filling Order

8
7
6
5
4
3
2
1
0

Figure 13. Board slots on TableMem

The implementation of the TableMem in the way described above created a difficulty on how the 4 neighbors of each board slot and their direction values would be accessed. A dedicated memory was created for this reason. When a new piece is requested the NeighboringPiece ROM is accessed. This ROM contains the address on TableMem of the adjacent pieces for each place on the board. It also contains the information for the direction where the place is (BoardRot) and if it is a piece with two neighbors or a cornering piece with one (SorD). The format these information are stored on NeighboringPiece ROM is shown on Table IV. This ROM was implemented in order to minimize the cost of the calculation of the adjacent pieces for each place on the board. This ROM is 35 bits wide and has a depth of 196, as the number of board slots.

TABLE IV. NEIGHBORINGPIECE ROM COMPONENTS

8 bit	8 bit	8 bit	8 bit	2 bit	1 bit
North neighbor	East neighbor	South neighbor	West neighbor	Board Rot	SorD

The address outputs from the NeighboringPiece ROM are forwarded to the direction value memory which consists of four independent memories. These memories are rearranged the same way as the TableMem memory in order to be able to save the direction values of a new piece at the same address as the piece itself. North has the north value of each piece as it is placed on the board, West the west value, etc. So the piece at the north of the new slot is accessed for its south value, the slot at the west for its west value etc. Four independent memories are needed because when a piece is pushed the address on these memories is the same as the TableMem address, but when they are accessed in order to acquire the information needed for a new piece, a different address for each memory is calculated by NeighboringPiece ROM, as four different board slots need to be accessed in order to acquire their direction values. In the end we have all the values of the neighboring pieces needed in order to ask for a new piece. These memories have a width of 5 bits and a depth of 196 each. It is also useful to note that these memories are implemented as distributed in order to have their output in the same clock cycle as the address. This does not have any effect on the critical path as it is bound on the PieceMem module.

The north, west, south, east values are calculated in the Aside/Bside Assignment module along with the BoardRot and SorD information. This module decides among the four neighboring piece adjacent values which are needed for the determination of the new piece. This is defined by the BoardRot signal, e.g. if the place where the new piece is about to be placed is at the north of the board, the east and south neighboring values are selected. The corner pieces (SorD) need one value (Aside) and the rest of the pieces both Aside and Bside.

The PieceMem answers with the new piece that is applicable for the board slot or with the Pop signal if there is no piece available with the requested side values. If a new piece is answered it is saved on the TableMem, along with its direction values on the direction values memory, and the next piece is requested. If the Pop signal is returned the last placed piece is popped and is sent along with the rotation it was placed on the board to the PieceMem in order for a new piece to take its place.

C. *Piece Memory Architecture*

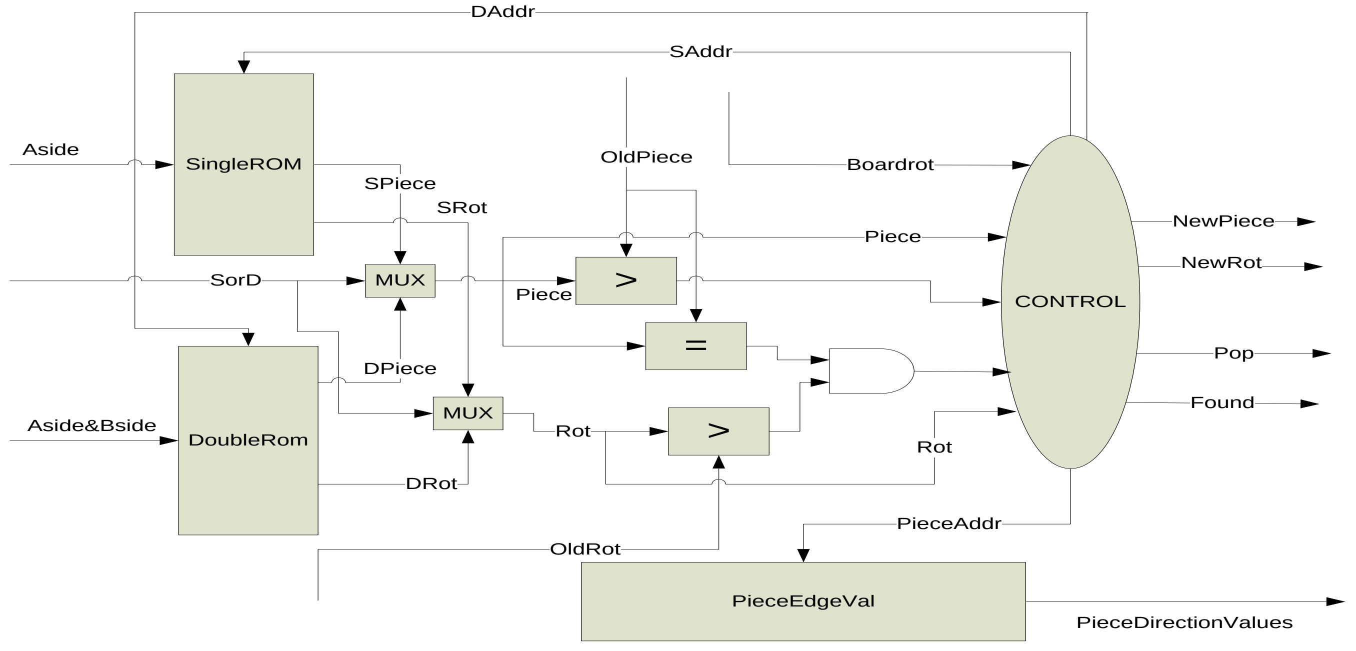

Figure 14. PieceMem Architecture

The PieceMem (Figure 14) is the module that contains all the 196 pieces and their direction values. The Table makes a piece request providing the appropriate information and PieceMem answers with the applicable new piece or the Pop signal if there is no available piece with the requested properties. All the pieces are contained in the SingleROM and the DoubleROM.

In SingleROM the pieces are saved with regard to each direction value of the piece. Also the rotation in which that value is at the north of the piece is saved. This memory is accessed when a corner piece is requested that needs only one side value. So this memory has a depth of 17, as the number of all the different color values, and width enough to save 49 pieces and their rotations (490 bits). Each color value is encountered on 44 to 49 pieces. The 5 bit address represents one of these 17 values. It outputs all the pieces that have an edge with the Aside value. The components of each slot of this memory are shown on Table V.

TABLE V. SINGLEROM AND DOUBLEROM COMPONENTS

8 bit – 2 bit	8 bit – 2 bit	8 bit – 2 bit	8 bit – 2 bit
Piece id – Piece rot	Piece id – Piece rot	Piece id – Piece rot	Piece id – Piece rot

In the DoubleROM the pieces are saved with regard to each piece's two adjacent edges. It represents a 17 by 17 array in each slot of which contains all the pieces that have the two values presented by the array addresses. These addresses are the Aside and Bside coming from the table. Each slot of this array may contain from 0 to 7 pieces with their rotations, making this memory 70 bit wide. It is implemented with a memory with 10 bit addressing where Aside&Bside are the address. The memory slots that don't represent a piece are idle. The components of each slot of this memory follow the same format as SingleROM and are shown on Table V.

These two memories were implemented in order to search the least amount of pieces possible in order to determine if a piece is applicable for the requested place on the table. So in order to search for a corner piece a maximum of 49 pieces need to be checked instead of all 196 and if a double edge piece is requested only 0-7 pieces need to be checked.

A register is implemented to mark if a piece is already used or not. The piece that is returned from PieceMem is marked as used and a piece is unmarked if it is the piece that was popped (oldpiece).

All the pieces that have the edge values requested are read from the appropriate memory. Each piece that has not already been placed on the table is checked. If the request follows a push operation the first piece found is returned. If the request follows a pop operation the new piece needs either to be equal to the previously placed piece and have a higher rotation value (some edge values are met more than once in some pieces) or have a higher id value. This is required in order to ensure that the same search space is not searched again. So, backtracking never leads to a re-visit of a previously visited node in the search space as following a Pop, the next piece has to have a higher id value.

If a piece is found it is forwarded to the TABLE along with its edge values that are read from the PieceEdgeVal module. These values are appropriately rotated in order to be correctly stored at the North, West, South and East memories of the TABLE. The components of this memory are shown on Table VI. This memory is implemented with logic cells (distributed memory) in order to be able to have these values the same clock cycle as the new piece that will be placed on the board. It causes a clock frequency drop and is part of the critical path, but the drop is not enough to justify the increase of the response of PieceMem by one clock cycle.

TABLE VI. PIECEEDGEVAL ROM COMPONENTS

5 bit	5 bit	5 bit	5 bit
North value	East value	South value	West value

This Module presented the most interest concerning which is the best approach for the piece searching operation. Three different approaches were followed with regard to the reply time of the PieceMem to the Table with a valid piece.

The first and obvious implementation was the one clock cycle response of the PieceMem. This architecture was able to check all the available pieces which had the specific attributes asked by the Table in one clock cycle. So this architecture had to include 49 greater than comparators for the piece id, 49 equal comparators for the piece id, 49 greater than comparators for the rotation of the piece as well as 49 AND gates. Also an arbiter is needed in order to choose the first valid piece from the 49 pieces if the table slot concerns a corner piece or the first valid piece from the 7 pieces otherwise. This implementation was very slow with regard to the clock frequency, which was less than 50MHz, and the resource utilization, as it occupied all the resources of the XC5V110T FPGA. It actually needs 115% of the FPGA and the CAD tool manages to lower the resources with the side effect of lowering the clock frequency. It utilizes a high amount of slices, more than 5% of the FPGA, for routing only operations. This implementation was the first fully working architecture that was downloaded and gave the first checkpoints of the search.

Because the above implementation did not have the expected performance a new approach was followed. After concerning that the corner slots, single edged pieces, on the board are far less than the double edged pieces (32 out of 196) the new implementation had to check only the double edged pieces in one clock cycle. This led to an architecture that had 7 greater than comparators for the piece id, 7 equal comparators for the piece id, 7 greater than comparators for the rotation of the piece as well as 7 AND gates. Also the Arbiter had to be a lot smaller. The search for a double edge piece takes one clock cycle to complete while the search for a single edge piece can take up to 7 clock cycles. This implementation can reach a clock frequency of 75MHz while utilizing more than 70% of the FPGA resources. The clock frequency increase provided a better performance although the PieceMem did not always respond in one clock cycle. This implementation was also downloaded and tested on actual hardware.

The last architecture, which is the one presented on Figure 14, has the best performance vs the resource utilization and also is the most simplistic. The simplistic characterization implies to the checking of 1 piece every clock cycle. The search can take from 1 to 49 clock cycles for a single edge piece and 1 to 7 clock cycles for a double edged piece. This architecture allowed the clock frequency to pass the 100MHz and thus allowing it to have the same performance as the first architecture. Also the resource utilization is 20 times less than the first architecture. This allowed the mapping of multiple EtSolver engines on a single FPGA. It is also useful to note that the critical resource moved from the slices to the Block RAMs. In order to balance it some memories that were implemented as Block RAMs, and in fact the SingleROM, had to be changed into Distributed Memories, causing a small drop of the clock frequency (it is still over 100MHz).

D. *Parallel Architecture*

In order to take advantage of the parallelization of the algorithm a parallel architecture was implemented (Figure 15). A large amount of independent EtSolver engines can be placed on a single FPGA.

These solvers are independent while searching. The initial configuration of the solvers is done by initializing a register for each one of them, which stores 196 pieces and their rotations as they are placed on the board. Each solver is initialized in a way that it has to search a different chunk of the search space. At the first initialization each initialization register is loaded with a different piece at the second board slot (the first slot is preplaced). 47 different pieces can be placed ath the second board slot. This way the search space is divided between the available EtSolver engines. The TableMem is then loaded by reading that register and the search continues from that point. If one engine finishes its chunk of the search space, its initialization register is then loaded with a different starting point, and more accurately from the point where the engine with the last chunk, in terms of search space, will have finished its search.

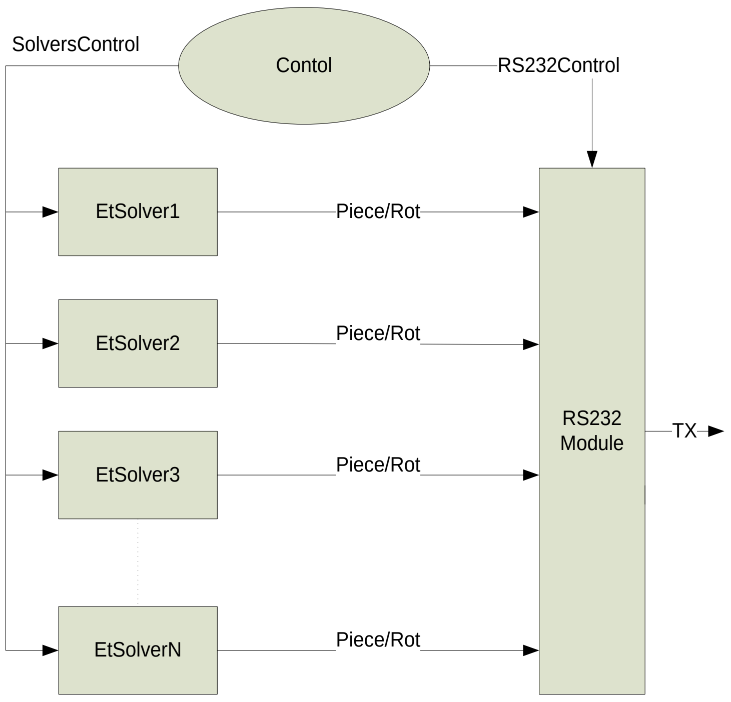

Figure 15. Parallel Architecture

A Control was needed in order to arrange the output from the RS232 module to the host PC, so that only one solver can send data at a time. This is used in order to send a solution (all 196 pieces placed) to the host PC and also checkpoints from each engine at regular time stamps. The checkpoints are needed in case a of a hardware failure, so that the search can resume from the last checkpoint. The checkpoints are saved on the host PC with regard to the engine they came from. In order to continue the search from the last checkpoint the initialization register is initialized by the checkpoint values.

E. *Implementation*

The Table below presents the resource utilization of a Xilinx Virtex 5 FPGA, the XC5V110T, for the three implementations of the PieMem. Also the performance is included in order to enlighten the best architecture.

TABLE VII. DIFFERENT PIECEMEM ARCHTECTURE COMPARISON

Architecture	**Architecture 1**	**Architecture 2**	**Architecture 3**	
			1 EtSolver engine	**22 EtSolver engines**
Slices	17253(99%)	13452 (77%)	885 (5%)	15458 (89%)
Block Rams	12 (8%)	12 (8%)	7 (4%)	143 (96%)
Clk Frequency	33Mhz	75Mhz	100MHz	75Mhz
Performance	4,1 MNodes/sec	5,4 MNodes/sec	4,1 MNodes/sec	67,5 MNodes/sec

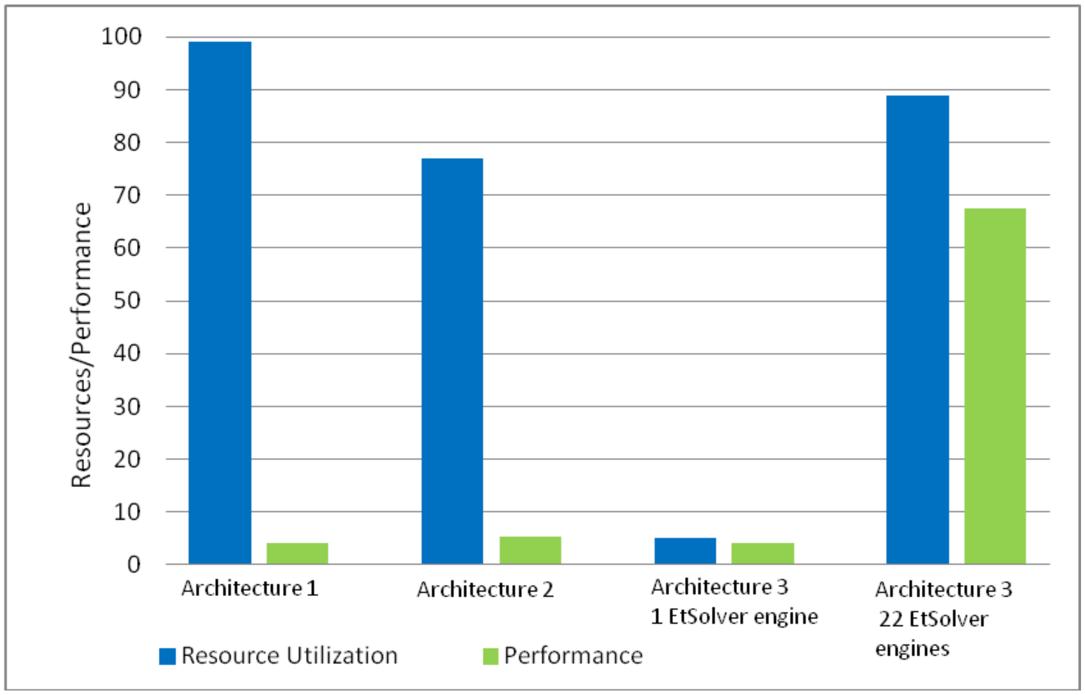

Figure 16. Performance vs Resource Utilization

As shown on Table VII and Figure 16 the first architecture has the least performance with regard to the resource utilization. The second architecture is a bit better both in terms of performance and in resource utilization. The performance increase comes from the clock frequency increase. The third architecture is the best performing. One engine has the same performance as the first architecture and at the same time utilizes a lot less resources. The resource utilization of a single EtSolver is very small (less than 5%). The simplicity of the algorithm, the implementation of the addressing of the Table memory and the direction value memories with a lookup table, NeighboringPiece ROM, are the main reasons that minimize the recourse utilization. This allowed the mapping of multiple EtSolver engines on a single FPGA, and in fact 22 of them on the XC5V110T. The clock frequency drop from 1 engine to 22 is due to the full utilization of the FPGA resources. The last architecture provided a performance increase of about 13 from the previous implementations.

The results presented here are measured based on actual clock frequencies available on the XC5V110T FPGA and not the optimal frequencies that are defined from the CAD tool. If the optimal clock frequencies are taken into account we can achieve an increase in performance of about 10%. The estimation of the performance if the optimal clocks were used on our last architecture is shown on the next Table.

TABLE VIII. ESTIMATION OF PERFORMANCE FOR OPTIMAL CLOCK FREQUENCY

Architecture	**1 EtSolver engine**	**22 EtSolver engines**
Slices	885 (5%)	15458 (89%)
Block Rams	7 (4%)	143 (96%)
Clk Frequency	116MHz	83Mhz
Performance	4,8 MNodes/sec	74,7 MNodes/sec

VII. SPACE VS TIME TRADEOFFS

In this Chapter the Space (resource utilization) vs Time (clock cycles for one operation and clock frequency) tradeoffs will be addressed for the various implementations and experiments that were done until the last and better performing architecture.

As mentioned on the previous Chapter and more accurately at the Piece Memory architecture presentation the first architecture has the least performance with regard to the resource utilization. It utilizes all the FPGA resources and has a very low clock frequency due to the long combinatorial paths produced by the need of searching between 49 pieces in one clock cycle. The second architecture is a bit better both in terms of performance and in resource utilization. The decrease of the pieces that need to be searched in one clock cycle from 49 to 7 decreases the resource utilization and allows a higher clock frequency, and as there is a small number of corner board slots that, also provides a performance increase. The third architecture is the best performing. One engine has the same performance as the first architecture and at the same time utilizes a lot less resources. In this architecture one piece is checked each clock cycle which decreases the resource utilization at 5% of the FPGA and allows a clock frequency of more than 100MHz where the first architecture has less than 40MHz.

The software that was developed by Mr. Miltiadis Smerdis is the exact same implementation of the algorithm as the hardware. The clustering of the pieces is the same as the hardware, in order to minimize the search between the pieces for a certain board slot. The flexibility of software implementations allowed the implementation of a faster Pop operation. When a piece is Popped, the next piece from the appropriate list, that has a higher id value, is checked so that not all the pieces with the needed attributes are checked. This improvement was also implemented on hardware but it did not provide equal performance vs resource utilization. The resource utilization increased by 20% but the performance increased roughly 5% and thus this improvement was dropped.

While trying to improve the performance of the second Piece Memory architecture a new architecture was the result that could perform a Push or Pop operation in 2 clock cycles (2-9 clock cycles for corner board slots). This was done by using distributed memories at the Table that can be written and read at the same clock cycle. A distributed single port ROM was used for the NeighboringPiece ROM and a dual port RAM for the TableMem where one port was used for writing and the second for reading. This allowed the request for the next piece to be done in one clock cycle, which on the previous architecture needed three clock cycles when a new piece had to be pushed before asking for the next piece. This architecture was tested and while it was working as expected on behavioral simulation it could not pass the Place and Route operation of the CAD tool. Also the resource utilization on slices increased from 77% to 98% (from Synthesize report) basically from the implementation of the memories from Block RAMs to distributed memories. Finally the clock frequency had a small drop (from Synthesize report from 75MHz to 65MHz) due to the migration of the critical path from the Piece Memory to the Table module. This improvement was not tested on the last architecture as the slice utilization would increase drastically and the clock frequency would drop decreasing its advantage vs the previous architectures which is the small resource utilization and the higher clock frequency.

VIII. Host Software

It is beyond the scope of this work, however, we need to mention that checkpointing was implemented so that the actual runs would advance despite occasional power supply and glitch problems. As expected, I/O is negligible, and hence the serial port does not impair the performance in any significant manner.

On the host PC a Python script is running in order to read from the serial port the checkpoints as well as the solution of the puzzle, if it is found. A special control is implemented in hardware in order to coordinate which of the 22 engines fitted on the XC5V110T FPGA will sent its checkpoint. If two engines want to make a checkpoint at the same time, the first is chosen and the other will stall until the first one finishes. The pieces are sent in the form of integers, first the piece id followed by the pieces rotation. The python script has to read 196x2 integers as well as some control signals. The output of this software is shown on Figure 17.

```
New Checkpoint
12                      Engine id
180                     Pieces Placed
139                     Piece id
0                       Piece Rotation
182
1
0
0
151
2
242
3
230
3
255
0
```

\

Figure 17. Output from Serial Port

There was also the need of software that takes this output and forms the Table for testing reasons. The initialization of some of the memories as well as the designation of the direction values of the pieces were done by hand. So in order to test if things were correctly initialized, a software, that by taking the output from the FPGA is able to print the whole board, was implemented.

This software reads the piece and its rotation and by having available the piece’s direction values prints each piece and the four values around it with respect to the rotation. The result is a view of the puzzle as it is solved by the hardware. If a piece is not placed correctly the adjacent pieces direction values wont mach which indicates a problem on hardware design or a memory initialization issue. The output of this software is a txt file with the form of Figure 18.

```
        -3-                     -5-                     -2-                     -8-                     -15-
-4-     232-3     -17--17-      181-3     -13--13-      165-1     -12--12-      213-1     -6--6-        241-2     -5-
        -6-                     -5-                     -15-                    -6-                     -9-
        -6-                     -5-                     -15-                    -6-                     -9-
-7-     94-1      -15--15-      242-3     -9--9-        151-2     -14--14-      256-0     -14--14-      239-0     -11-
        -5-                     -6-                     -10-                    -5-                     -1-
        -5-                     -6-                     -10-                    -5-                     -1-
-3-     246-1     -2--2-        230-3     -17--17-      139-0     -13--13-      182-1     -3--3-        244-1     -11-
        -14-                    -11-                    -13-                    -17-                    -6-
        -14-                    -11-                    -13-                    -17-                    -6-
-7-     100-1     -14--14-      255-0     -14--14-      104-3     -16--16-      117-1     -14--14-      251-3     -15-
        -5-                     -2-                     -16-                    -2-                     -15-
        -5-                     -2-                     -16-                    -2-                     -15-
-1-     247-2     -14--14-      136-2     -11--11-      131-0     -5--5-        254-3     -11--11-      90-2      -2-
        -1-                     -16-                    -13-                    -2-                     -7-
```

Figure 18. Table Form Output

As noted on Chapter III, dedicated software responsible for the completion of the puzzle, by placing the 60 outer pieces, was created. After a study on the outer pieces, by Ms. Katerina Papayannaki and Mr. Antonis Papayannakis, the number of the outer pieces that have on their inner board edge each direction value was counted. It is mentioned on Chapter I that only 17 of the 22 different colors are met on the inner pieces and the rest 5 are met only on the border pieces. The border pieces have one side grey colored, meaning it is the side that connects to the border. Its adjacent edges are colored with one of the five colors that are dedicated for the outer pieces. The opposite of the grey side has a color value between the 17 colors that are met in the inner pieces. The number of pieces that each of the 17 colors was met was counted and the result is the following Table.

TABLE IX. OUTER PIECES INNER SIDE VALUE COUNT

Direction Value	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Num of Pieces	6	4	4	2	3	6	4	1	3	3	3	1	3	2	4	5	2

So in order to minimize the run time of the software that is responsible for filling the outer pieces, software was created that checks if the inner board solution meets the criteria stated by the outer pieces. It takes as input the table form output (Figure 18) and creates a table, similar to Table IX, by counting the number of pieces, which are at the edge of the inner board, that have each of the different edge values. Then this table is compared with Table IX and if they don't match it means that the inner board solution is not valid. If the solution is marked as non valid the software that fills the outer pieces does not run.

After the filtering of the 196 piece solutions by the previous software only the solutions that could create a valid completed board are sent to the outer pieces filling software.

IX. Performance Evaluation

The Table below presents the results from actual runs on hardware on the XC5V110T FPGA vs the software implemented by Miltiadis Smerdis.

TABLE X. HARDWARE VS SOFTWARE

Architecture	Performance	SpeedUp
Smerdis Software on Intel Xeon 2,66GHz	26 MNodes/sec	1
1 EtSolver engine	4,1 MNodes/sec	1/6,3
22 EtSolver engines	67,5 MNodes/sec	2,6

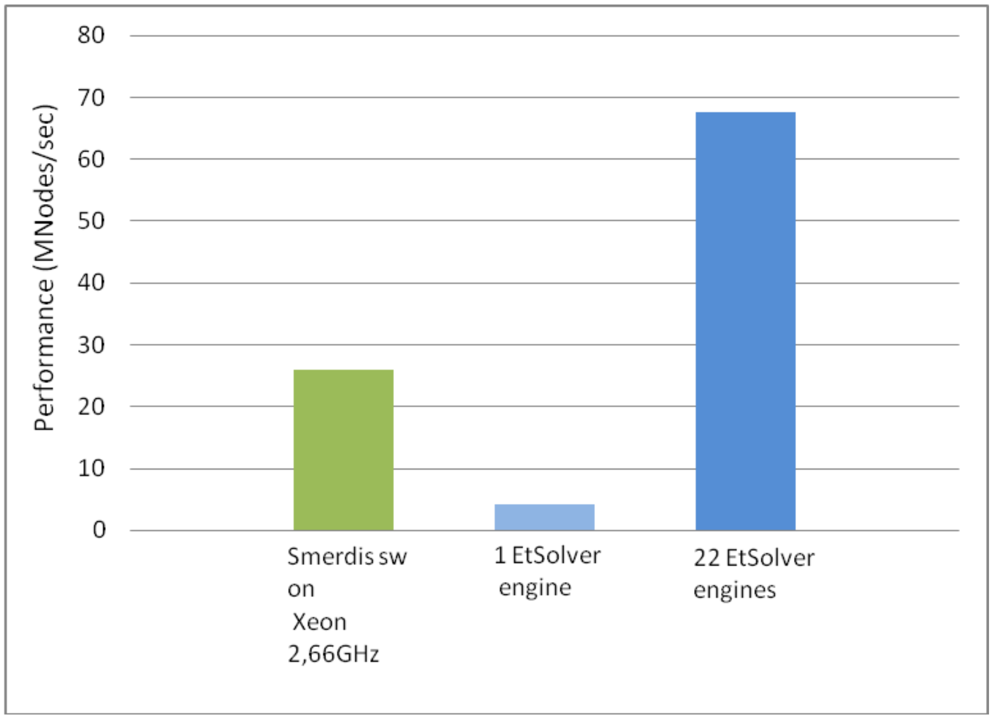

Figure 19. Hardware vs Software

We note that we used our own optimized software, which roughly corresponds to the hardware architecture, as Internet-distributed codes did not seem to have quite the same performance. To be precise, and although no published results exist, from our experimental work we gleaned that the best software has roughly 20MNodes/sec on a high-end workstation, whereas our own optimized software version visits 26MNodes/sec. As mentioned above this software was developed by Mr. Miltiadis Smerdis. The software is the exact same implementation of the algorithm as the hardware. The clustering of the pieces is the same as the hardware, in order to minimize the search between the pieces for a certain board slot. The flexibility of software implementations allowed the implementation of a faster Pop operation but as mentioned on Chapter VI the hardware implementation did not provide equal performance vs resource utilization. In the end the hardware has to check all the pieces with those attributes in order to choose the next piece for that slot, even pieces with lower id values.

As it is shown on Table X and Figure 19 one EtSolver has significant SpeedDown vs the software. This was an expected result as the FPGA implementation has more than 25 times lower clock frequency. The FPGA implementation overcomes this disadvantage by increasing the number of calculations per clock cycle and by exploiting parallelization. As a result we end up with a 2,6 SpeedUp with 22 EtSolver engines.

We also need to compare our work against the work presented by Vladimír Kašík on [18]. For this reason the EtSolver had to be mapped on the XUP board with a XC2VP30 FPGA. The Fast BT can push/pop a piece in every clock cycle while EtSolver needs multiple cycles for each operation. The Fast BT can achieve a performance of 16M(push or pop)/sec which is about 8MNodes/sec, as only push operations count as visited nodes. As it is shown on Table XI and Figure 20 one EtSolver engine (with a 100MHz clock frequency) has half the performance of the Fast BT (with a 16MHz clock frequency) but utilizes a lot less resources which allowed the mapping of 7 EtSolver engines (with a 75MHz clock frequency) on the XC2VP30 FPGA. As a result a SpeedUp of 2,7, vs the Fast BT, can be achieved on the same hardware device.

TABLE XI. ETSOLVER VS FAST BT

Architecture	Resource utilization		Performance	SpeedUp
	Slices	Block RAMs		
Fast BT	12060(88%)	0(0%)	8 MNodes/sec	1
1 EtSolver engine	1718(13%)	13(9%)	4,1 MNodes/sec	0,51
7 EtSolver engines	12026(88%)	91(68%)	21,5 MNodes/sec	2,7

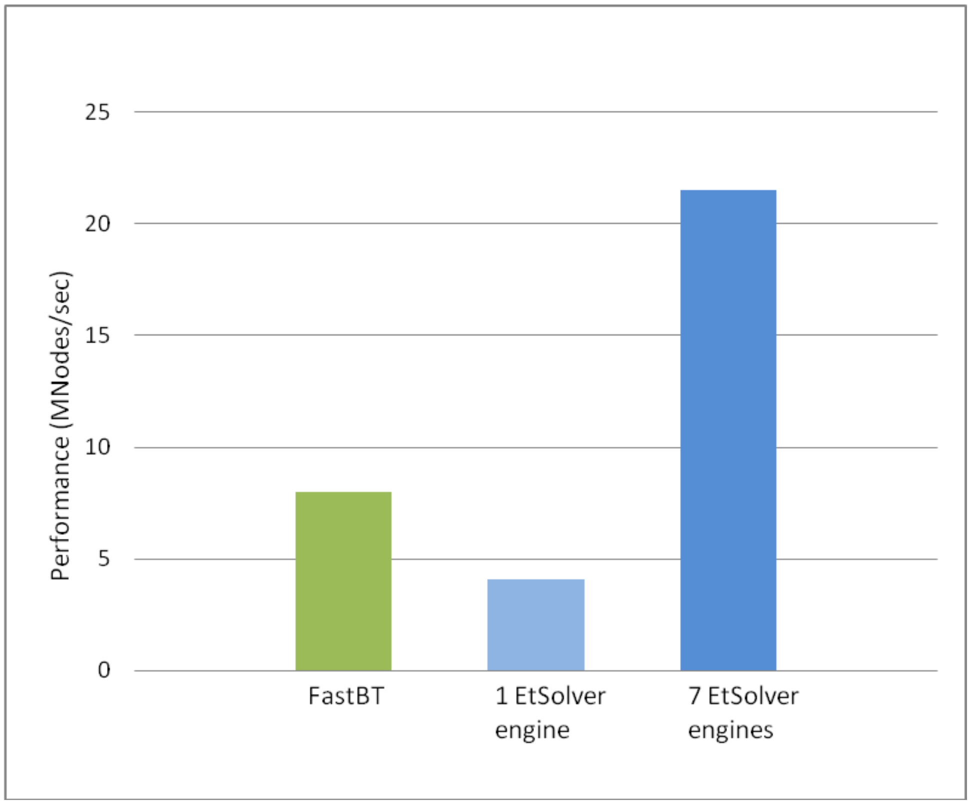

Figure 20. EtSolver vs FastBT

After a 3 months actual calculation time on the hardware the best available solution contained 187/196 pieces giving a score of 346/364 number of satisfied junctions.

X. Conclusions

The work presented in this paper shows a complete architecture for the Eternity II puzzle, with the implementation of a non-trivial DFS algorithm on hardware, and its 22-fold parallelization on a single Virtex 5 FPGA. Actual runs on hardware show an overall system performance SpeedUp of 2.6 vs. the equivalent software running on a high-end workstation. The hardware optimizations led to the implementation of the optimized software. These results are very promising because they show that exhaustive searches with recursive algorithms and non-trivial data structures can be well-addressed by FPGAs vs. general-purpose computers.

It is also useful to note that the design needs a very large amount of time in order to complete the search space (about 10^{108} years with 1 FPGA). The only chance to get a solution is if we are lucky and we initialize one of the 22 engines with a puzzle that happens to be near the solution, e.g. provide the correct first 50 pieces.

REFERENCES

[1] Eternity II official site : http://www.eternityii.com

[2] http://en.wikipedia.org/wiki/Eternity_II_puzzle

[3] Yasuhiko Takenaga, Toby Walsh. "TETRAVEX is NP-complete". Information Processing Letters 99 (2006), pp: 171–174, 2006.

[4] Pierre Schaus, Yves Deville. "Hybridization of CP and VLNS for Eternity II". Actes JFPC 2008.

[5] Wim Vancroonenburg, Tony Wauters, Greet Vanden Berghe. "A two phase hyper-heuristic approach for solving the Eternity II puzzle". Proceedings of the International Conference on Metaheuristics and Nature Inspired Computing edition, Oct 2010.

[6] Wei-Sin Wang, Tsung-Che Chiang. "Solving Eternity-II puzzles with a tabu search algorithm". Proceedings of the International Conference on Metaheuristics and Nature Inspired Computing edition, Oct 2010.

[7] V. Sklyarov. "FPGA-based implementation of recursive algorithms". Microprocessors and Microsystems, Special Issue on FPGAS: Applications and Designs, vol. 28/5-6, pp.197-211, 2004.

[8] V. Sklyarov. "Hierarchical finite-state machines and their use for digital control". IEEE Transactions on VLSI Systems, Vol 7, No 2, 1999, pp. 222-228.

[9] V. Sklyarov, I. Skliarova, and B. Pimentel. "FPGA-based implementation and comparison of recursive and iterative algorithms". Proceeding of FPL'05, Tampere, Finland, 2005, pp.235-240.

[10] V. Sklyarov, I. Skliarova. "Recursive and Iterative Algorithms for N-ary Search Problems". 2006, in IFIP International Federation for Information Processing, Volume 218, Professional Practice in Artificial Intelligence, eds. J. Debenham, (Boston: Springer), pp. 81-90.

[11] S. Ninos, A. Dollas. "Modeling recursion data structures for FPGA-based implementation". Proceedings, International Symposium on Field Programmable Logic (FPL), pp. 11-16, Heidelberg, 2008.

[12] Jorge Munoz, German Gutierrez, and Araceli Sanchis." Evolutionary Genetic Algorithms in a Constraint Satisfaction Problem: Puzzle Eternity II". Bio-Inspired Systems: Computational and Ambient Intelligence, Lecture Notes in Computer Science, 2009, Volume 5517/2009,pp. 720-727.

[13] Marijn J.H. Heule. “Solving edge-matching problems with satisfiability solvers”. Proceedings, Second International Workshop on Logic and Search (LaSh 2008), pp. 88-102. University of Leuven.

[14] A. Dollas, E. Sotiriades, A. Emmanouelides. “Architecture and Design of GE1, a FCCM for Golomb Ruler Derivation”. Proceedings, International Symposium on FPGA’s for Custom Computing Machines, FCCM 98, Napa, California, pp. 48-56, April 1998.

[15] E. Sotiriades, A. Dollas, P. Athanas. “Hardware - Software Codesign and Parallel Implementation of a Golomb Ruler Derivation Engine”. Proceedings, International Symposium on Field Programmable Custom Computing Machines,FCCM 2000, pp. 227-235. IEEE Computer Society, 2000.

[16] P. Malakonakis, E. Sotoriades, A. Dollas. “GE3: a single FPGA client-server architecture for Golomb ruler derivation”. Proceedings, Field-Programmable Technology, pages 470-473, Beijing 2010.

[17] M. Platzner. "Reconfigurable accelerators for combinatorial problems". Computer , vol.33, no.4, pp.58-60, Apr 2000.

[18] . “Acceleration of Backtracking Algorithm with FPGA”. Proceedings, Applied Electronics (AE), 2010 International Conference, pp.1-4, Sept. 2010.

[19] Erik D. Demaine, Martin L. Demaine. “Jigsaw Puzzles, Edge Matching, and Polyomino Packing: Connections and Complexity”. Graphs and Combinatorics, volume 23 (Supplement), June 2007, pages 195–208. Special issue on Computational Geometry and Graph Theory: The Akiyama-Chvatal Festschrift.

[20] Solomon W. Golomb. “Polyominoes: Puzzles, Patterns, Problems, and Packings”. Princeton University Press, 2nd edition, 1994.

[21] Y-X. Zhao, M-C. Su, Z-L. Chou, and J. Lee. “A puzzle solver and its application in speech descrambling”. Proceedings of the 2007 WSEAS International Conference on Computer Engineering and Applications, Gold Coast, Australia, January 17-19, 2007.

[22] K. Kimoto, H. Tsuji, Y. Murai, S. Tokumasu. "A solution of three-dimensional polyomino packing problems". IEEE International Conference on Systems, Man and Cybernetics, 2007, ISIC. , pp.3725-3730, 7-10 Oct. 2007.

[23] M.J. Anwar, M. Ishtiaq, M.A. Iqbal, M.A. Jaffar. "Block-based digital image watermarking using Genetic Algorithm". 6th International Conference on Emerging Technologies (ICET), 2010 , pp.204-209, 18-19 Oct. 2010.

[24] B. J. Brown, C. Toler-Franklin, D. Nehab, M. Burns, D. Dobkin, A. Vlachopoulos, C. Doumas, S. Rusinkiewicz, T. Weyrich. “A system for high-volume acquisition and matching of fresco fragments: Reassembling Theran wall paintings”. ACM Transactions on Graphics, 27(3), 2008.

[25] G. Papaioannou, E.-A. Karabassi, T. Theoharis. "Virtual Archaeologist: assembling the past". Computer Graphics and Applications, IEEE , vol.21, no.2, pp.53-59, Mar/Apr 2001.

[26] F. Kleber, M. Diem, R. Sablatnig. "Torn Document Analysis as a Prerequisite for Reconstruction". 15th International Conference on Virtual Systems and Multimedia, 2009, VSMM '09, pp.143-148, 9-12 Sept. 2009.

[27] F. Hoshi, Y. Murai, H. Tsuji, S. Tokumasu. "A new deterministic algorithm for two-dimensional rectangular packing problems based on polyomino packing models". 2010 IEEE International Conference on Systems Man and Cybernetics (SMC), pp.2760-2766, 10-13 Oct. 2010.

[28] P. F. Gorder. "Top7: From computer-aided design, a new protein". Computing in Science & Engineering, vol.6, no.2, pp. 6-11, Mar-Apr 2004.

[29] Z. Liangjia, Z. Zongtan, H. Dewen. "Globally Consistent Reconstruction of Ripped-Up Documents". IEEE Transactions on Pattern Analysis and Machine Intelligence, vol.30, no.1, pp.1-13, Jan. 2008.

[30] X. Ma, F. Lombardi. "Combinatorial Optimization Problem in Designing DNA Self-Assembly Tile Sets". 2008 IEEE International Workshop on Design and Test of Nano Devices, Circuits and Systems, pp.73-76, 29-30 Sept. 2008.

[31] “The Xgrow simulator”. : http://www.dna.caltech.edu/Xgrow

[32] P. W. K. Rothemund, N. Papadakis, E. Winfree. “Algorithmic selfassembly of DNA Sierpinski triangles”. PLoS Biology, vol. 2, no. 12, p.e424, 2004.