# Implementing Propositional Networks on FPGA

Cezary Siwek[1], Jakub Kowalski[1⋆], Chiara F. Sironi[2⋆⋆], and Mark H. M. Winands[2]

[1] Institute of Computer Science, Faculty of Mathematics and Computer Science
University of Wrocław, Wrocław, Poland
ave@cezar.info, jko@cs.uni.wroc.pl

[2] Games & AI Group, Department of Data Science and Knowledge Engineering
Maastricht University, Maastricht, The Netherlands
{c.sironi,m.winands}@maastrichtuniversity.nl

**Abstract.** The speed of game rules processing plays an essential role in the performance of a General Game Playing (GGP) agent. Propositional Networks (propnets) are an example of a highly efficient representation of game rules. So far, in GGP, only software implementations of propnets have been proposed and investigated. In this paper, we present the first implementation of propnets on Field-Programmable Gate Arrays (FPGAs), showing that they perform between 25 and 58 times faster than a software-propnet for most of the tested games. We also integrate the FPGA-propnet within an MCTS agent, discussing the challenges of the process, and possible solutions for the identified shortcomings.

**Keywords:** General Game Playing · Field-Programmable Gate Arrays · Propositional networks · Monte Carlo Tree Search

## 1 Introduction

The aim of General Game Playing (GGP) [7] is to develop a program that can play any arbitrary game at an expert level, given only its rules. Moreover, these rules are previously unknown, and an agent has a limited time to process them before the game begins. During the game, the time is also constrained, with usually only a few seconds available to choose a move.

In GGP, it is impossible for the designers of the program to embed in the agent existing knowledge about the game, as it is in the case of chess, checkers, Go, and other standard AI challenges. As such, with the goal to create a universal algorithm performing well in various situations and environments, the domain has been identified as a new grand challenge of Artificial Intelligence [6], and a special logic-based Game Description Language (GDL) has been designed to describe any deterministic, turn-based, finite game with perfect information [11].

⋆ Supported in part by the National Science Centre, Poland under project number 2015/17/B/ST6/01893.

⋆⋆ Supported by the Netherlands Organisation for Scientific Research (NWO) under the GoGeneral project, grant number 612.001.121.

**Table 1.** Description of GDL keywords. $?f$ represents a fact.

| Keyword | Description | Keyword | Description |
|---|---|---|---|
| $role(?r)$ | $?r$ is a player in the game | $true(?f)$ | $?f$ is true in current state |
| $init(?f)$ | $?f$ is true in initial state | $next(?f)$ | $?f$ is true in next state |
| $terminal$ | current state is terminal | $does(?r, ?m)$ | $?r$ plays move $?m$ |
| $goal(?r, ?s)$ | $?r$ gets score $?s$ in current state | $legal(?r, ?m)$ | $?r$ can play move $?m$ in current state |

Because of the generality, GGP benefits algorithms that are knowledge-free. As a result, the most successful approaches are based on the Monte Carlo Tree Search (MCTS) [4], the algorithm that apart from GGP [5] has proven itself in Go [17] and many other domains [3].

As the strength of game-playing search algorithms is usually closely correlated with their performance, it is crucial for the game reasoners to be as fast as possible. When in 1997 Deep Blue defeated Gary Kasparov, it was partially because of the hardware accelerators – Application Specific Integrated Circuits designed specifically for this system [9].

In GGP, as the quality of results obtained by MCTS depends on a number of performed simulations, much attention has been devoted to improving the speed of GDL resolution engines. This includes mainly fast, logically-optimized interpreters and compilers to low-level languages [10, 19]. Propositional Networks (propnets) [14], are efficient representation of GDL reasoners, closely correlated to logic circuits. They can speed-up the state computation process by several orders of magnitude compared to non-optimized custom-made or Prolog-based GDL reasoners [18]; thus they are used by many successful GGP players.

In this paper, we present the first implementation of Propositional Networks on Field-Programmable Gate Arrays (FPGAs), the integrated circuits that can be reconfigured by the end-user. Thus, we were able to achieve performance impossible for the reasoners encoded as a software. Resulting FPGA-based reasoner computes game states mostly about 25-58 times faster than the optimized software propnet implementation described in [18].

To utilize this computational potential, we have implemented a working MCTS-based GGP player proof of concept, upon which we study and present shortcomings and effort required to construct a hardware-accelerated player.

## 2 Preliminaries

### 2.1 Game Description Language and Propositional Networks

GDL is a first order logic language proposed to represent game rules in GGP in a compact and modular format [11]. A state in GDL is represented as a set of true facts. Special keywords, described in Table 1, are used to define different game elements and the game dynamics. By processing the GDL game rules, a player is able to reconstruct the dynamics of a finite state machine for the game.

Propnets [14] are an alternative to GDL to represent the dynamics of a game, and any GDL game description can be converted into a propnet. Propnets are directed graphs where the components are either propositions or connectives.

Each component has incoming arcs from its input components and outgoing arcs to its output components. The truth value of a component depends on the truth value of its inputs and is propagated to its outputs.

There are four types of connectives: *and*, *or* and *not* logic gates, and *transitions*, identity gates that output their input value with one step delay. Propositions can be divided into three categories: *input*, that have no input components, *base*, that have one single *transition* as input, and all other propositions, identified as *view*. The truth values of *base* propositions represent the state of the game. Their input, the *transition*, controls their value for the next state. Having no inputs, *input* proposition have their value set by the game playing agent, that sets to true the one corresponding to the action he decides to play. *View* propositions express agents' goals, legal moves and terminality of game states.

A unique truth assignment to *base* propositions determines the unique truth values of *view* propositions. The combination of truth assignments to *base* and *input* proposition uniquely determines the truth assignment for the next state.

### 2.2 Monte Carlo Tree Search

MCTS [4] is a simulation-based search algorithm that incrementally builds a tree representation of the search space of the game. More precisely, it repeats the following four phases until a given search budget expires:

- Selection: the algorithm traverses the tree built so far. A *selection strategy* is used to choose which action to simulate in each visited node until a state not yet in the tree is reached. One of the most commonly used selection strategies is UCB1 [1], the same we use in our MCTS implementation.
- Expansion: the first visited state in the simulation that was not part of the tree yet, is added to the tree as a new node.
- Playout: starting from the state corresponding to the node added during expansion, a *playout strategy* is used to simulate the game until a terminal state or a certain depth is reached.
- Backpropagation: the result obtained at the end of the simulation is propagated back in the tree and used to update statistics about the visited moves.

### 2.3 Field-Programmable Gate Arrays

FPGAs are chips, whose logic is designed to be configured after they were manufactured or even embedded in the final product (hence *Field*). This allows fast prototyping of the Integrated Chips (ICs), creating small amounts of products with custom hardware, or even performing remote updates to the hardware in the end devices. FPGAs are made out of thousands of interconnected Universal Logic Modules (ULMs), which can be individually programmed to perform simple logic operations and arbitrarily connected with each other. For specialized operations, this allows for a significant increase of computational speed and IO bandwidth against implementation in software.

Desired structure and behavior of the FPGA is usually written in a Hardware Description Language (HDL), like Verilog or VHDL. It resembles classic programming with expressions, statements and datatypes, but the execution flow is parallel rather than sequential, and there are explicit constructs to handle time.

FPGAs are used in many domains, including communication, image processing, control engineering, networks, cryptography, mathematics, neuro-computing, etc. A comprehensive survey on FPGA applications can be found in [13].

Related research, mostly concerns using FPGAs to implement game engines, especially board games [12], to accelerate computations and thus improve the performance of the agents. This includes FPGA-based approaches to play, e.g., chess [2], Othello [20], and Go [8].

## 3 Methodology

We present our approach, that given an arbitrary GDL game generates FPGA-based reasoner and embeds it into the MCTS algorithm. The entire random playout phase, has been implemented within a reasoner component. This significantly reduces the number of tree-to-reasoner calls, reducing the overhead, and improving the overall performance of the system.

We based our solution on Propositional Networks. They are a fast reasoning mechanism on their own, and because their structure consists mainly of standard logic gates, we can almost directly mirror their computational logic in a hardware chip. This approach can increase performance by orders of magnitude because of zero computational overhead and simultaneous propagation of signals. The latter is essential, as the simulation speed is dictated by the clock frequency, which is in turn constrained by the longest component path, not the total number of propnet components.

FPGAs are especially well suited to be used as a GGP reasoner because their reprogramming capability allows for switching between various (previously known) games on the fly. The Cyclone V chip we are using is even more up to the task because it integrates a dual core ARM computer running GNU/Linux operating system that can communicate with the FPGA through fast shared memory. We run the player search algorithm on this ARM computer, and the intense reasoning computations are delegated to the FPGA-part of the chip.

### 3.1 From GDL to Verilog

Our system generates ready-to-synthesize Verilog code. We use previously prepared Verilog modules that implement the behavior of the propnet component types, a template for a whole propnet module, and the project into which the propnet module will be injected. Now, given GDL rules of a new game, we generate software propnet using the code from [18]. Then, for every component in this propnet, we create new instance belonging to one of the before-mentioned modules and make it a new node in our hardware propnet. When all components

are placed, we implement edges of the software propnet as wire connections in the HDL. We do this by BFS traversal of the underlying propnet graph.

The propnet meta-information contains information about the propnet structure (e.g. initial state, game state size) and describes game's legal moves, states, etc. We write this data to the propnet module and a separate XML file that will be later passed to the software side running on the ARM computer. Propnet graph and meta-information for the propnet controller logic are filled into the propnet module template, and resulting file is copied to the FPGA project.

Because of the complexity of the compilation process for the FPGA, and the fact that Intel's tools require to be run on an AMD64 PC, given GDL rules, our system waits until the image for the FPGA is provided from the computer controlled by a human. Thus, the current version cannot compete in a standard GGP match; however, the human does not make any contribution to the resulting image, and in principle, the process can be fully automated.

### 3.2 System Architecture

Figure 1 presents the overall architecture of our project. The ARM computer contains the high-level part of the system. It consists of GGP player, MCTS implementation, and driver library initialized with a game meta-information. It exchanges data with the FPGA board through the shared memory, containing four regions for communication with the propnet controller. Those are for: command queue (e.g., reset, execute $n$ random simulations, set return context); sending next states; sending next legal moves; and sending scores of the players.

The information flow on FPGA is presented on the lower half of the figure. The main components are propnet driver, responsible for proper data transmission, and the propnet itself, programmed as described in Subsection 3.1 and containing parts dedicated to communicating with the rest of the board.

### 3.3 MCTS Reasoner Implementation

Our goal is to implement a reasoner that can be effectively used by the MCTS algorithm. Thus, it needs to perform random simulations from an arbitrary game state to some terminal state, computing players' scores in this state.

The search algorithm works on the integrated ARM computer and interfaces with the FPGA via a driver library encoded in Java. For the FPGA to start playouts from a specific node, it has to switch context into the state corresponding to this node. Thus, we require from the MCTS tree implementation to store data representing the internal state of the FPGA propnet. This state is provided by the library during the MCTS expansion phase. The library exposes to MCTS three functions:

- **FPGAState** *getRootState*(): returns game tree root in FPGA encoding.
- **(list**<**legalMoves**>, **list**<(**FPGAState**, **jointMove**)>**)** *getNextStates*(**FPGAState** *state*): returns for a given state the list of legal moves for each role, and all the children states and edges going to them.

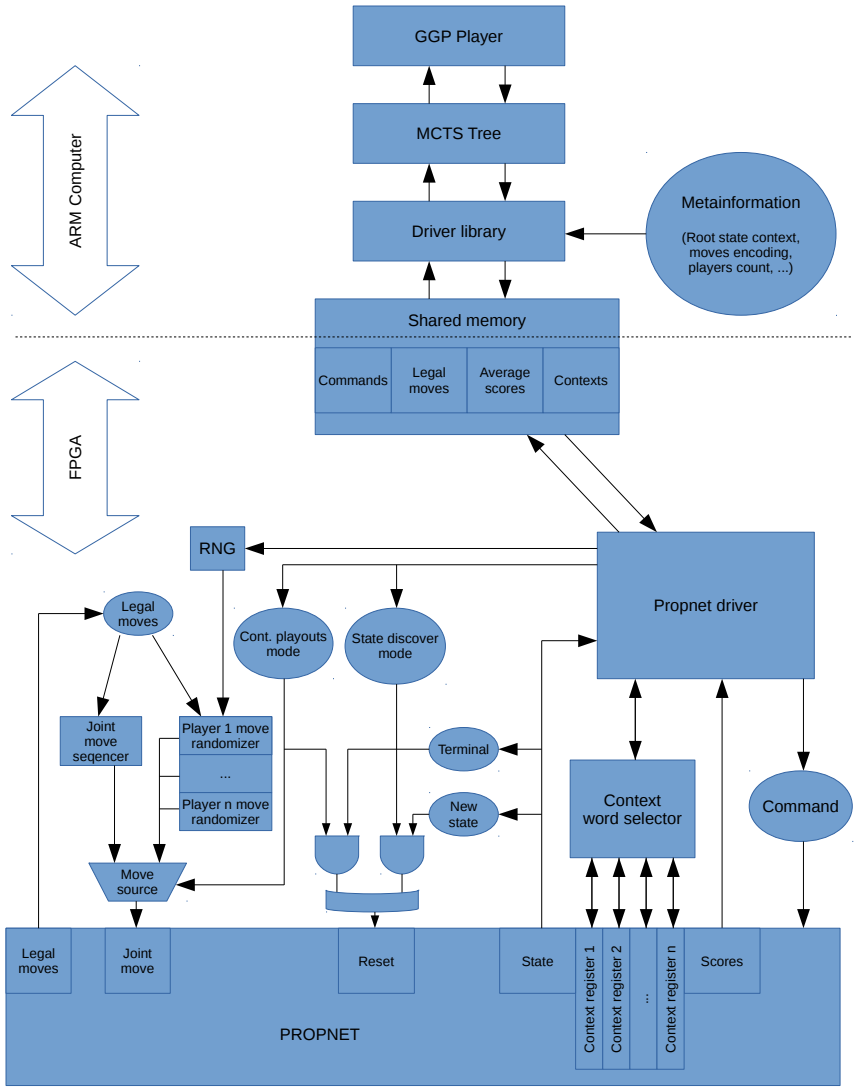


**Fig. 1.** System architecture.

– **list**<**long**> *getScores*(**FPGAState** *state*, **int** *n*): computes for each player the scores obtained during a batch consisting of $n$ random simulations.

Calling reasoner to calculate a single playout, which is standard for software propnets, is very inefficient in our FPGA-based architecture, mainly because of communication costs. In order to reduce the number of read-write cycles, we only provide interface for scheduling batched playouts. When simulating, MCTS uses the *getScores* function to request a specific number of playouts (it is an MCTS initialization parameter) and backpropagates the summarized scores.

### 3.4 State Computation

Each transition node has assigned a unique number, and thus the game state is coded as a bit vector, where $n$th bit corresponds to the value stored in the $n$th transition node. Since this can grow up to a few kilobytes, it is divided into 128-bit words when loaded from or stored in the shared memory. Now, when the library issues new playouts, the propnet driver module loads every context word into an appropriate context register in the propnet. Propnet reset, and every new game state evaluation, takes place in one clock cycle.

We have three modes that we use to control the behavior of the propnet module: *state discovery*, *context switching*, and *continuous playout*.

In the state discovery mode, all legal joint actions are iterated over by the move sequencer. For each joint action, after calculating the next state, the propnet driver starts forwarding context words (state representation) and the corresponding joint move into the shared memory. This can be a multi-step process, as the memory may force to stall sending the state until it is ready.

In the context switching mode, the propnet driver queries requested place in the shared memory for the context words, which are then written to the propnet's context registers.

In the continuous playout mode, the players' actions are continuously taken from the modules generating legal random actions, until a terminal state is reached. When that happens, the propnet module signals scores to the propnet driver and resets the internal propnet to the previously set context. To ensure generated actions are uniformly distributed, for each player, we randomize a number $i$ between 0 and the number of his legal actions, and loop through all his actions, reducing $i$ on set bits, until the $i$-th legal action is found.

## 4 Experiments

To evaluate the performance of the FPGA implementation of propnets we carried out two types of experiments. Firstly, we compare the speed of our propnets with one of the fastest software propnets reasoners [18] and also with the Java-based Prover from the GGP-Base package [15] used as a baseline. Secondly, we investigate how the obtained speed-up translates to the performance of an MCTS agent, focusing on analysis of influence of batch size to the number of MCTS node expansions and software MCTS operations overhead.

**Table 2.** Comparison of reasoners based on running Flat Monte Carlo algorithm. FPGA speed is equivalent to the clock frequency, which is probed in 1.0Mhz steps. FPGA chip utilization is the space required to fit the propnet on the board.

| Game | Speed (avg nodes/sec) | | | Initialization time | | #Propnet components | FPGA chip utilization |
|---|---|---|---|---|---|---|---|
| | FPGA | software | Prover | FPGA (min) | software (sec) | | |
| Horseshoe | 8,500,000 | 192,583 | 3,812 | 4:20 | 0.45 | 350 | 7% |
| Connectfour | 7,000,000 | 285,908 | 561 | 5:37 | 0.67 | 814 | 12% |
| Pentago | 7,000,000 | 119,111 | 342 | 5:20 | 2.70 | 1,291 | 13% |
| Jointconnectfour | 4,500,000 | 171,575 | 270 | 5:53 | 1.00 | 1,614 | 16% |
| Breakthrough | 1,400,000 | 38,015 | 601 | 12:03 | 1.35 | 17,752 | 72% |
| Reversi | 1,171,875 | 4,806 | 19 | 14:08 | 23.91 | 56,014 | 41% |

In our experiments we use TerasIC DE1-SoC board containing the Altera's Cyclone V series SoC: 5CSEMA5F31C6. The GGP player, search algorithm and communication with the reasoner are run by a computer embedded in the beforementioned SoC with ARM Cortex A9, Dual core @925Mhz with 1 GB RAM, running Debian 9 Strech 32-bit. The FPGA project compilation is performed on Intel Core i5-4670 with 16 GB DDR3 @1600Mhz RAM using Ubuntu 16.04 server 64-bit and Intel Quartus Prime Lite Edition 17.0 as FPGA compilation IDE. Software propnets and the GGP-Base Prover are tested on a Linux server consisting of 64 AMD Opteron 6174 2.2-GHz cores and 252 GB RAM.

### 4.1 Performance Comparison

To test the reasoner's performance, we use a Flat Monte Carlo Search, measuring the number of states visited during random playouts from the initial game state.

We compare results obtained for FPGA with other reasoners – software propnets and Prover. The overall results are presented in Table 2. They are based on 1 million simulations for FPGA, and more than 250 thousands simulations for the other reasoners (except 1000 simulations for Reversi). GDL descriptions of the games can be found in the Stanford Gamemaster repository [16].

As expected, the usage of hardware accelerator substantially increases the reasoner's efficiency. For all games except Reversi, the improvement factors are between 24.5 (Connect-Four) and 58 (Pentago). For Reversi, which produces the largest propnet among the tested games, FPGA-based reasoner computes states over 290 times faster. This example shows that smaller propnets do not necessarily imply smaller chip utilization.

The downside of moving from software to hardware is a considerable increase of initialization time. Instead of seconds it is about 5–6 minutes for small and medium games, and for large propnets it is almost a quarter. Such times exclude GGP players from being ready during their standard initialization clock. We discuss this issue in detail and present possible solutions in the next section.

### 4.2 MCTS Player Performance

Embedding an FPGA propnet reasoner into the MCTS involves delegating some computation time to the software responsible for managing the MCTS tree. The

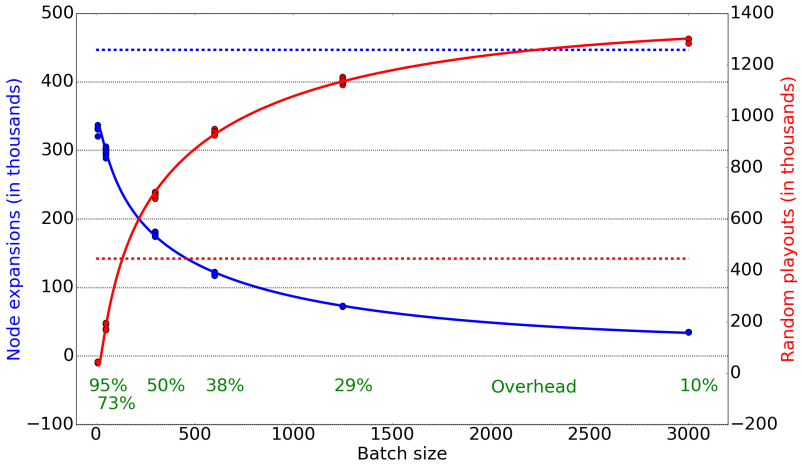


**Fig. 2.** Dependency between a batch size and the FPGA-based agent's performance measured by a number of nodes expansions, number of computed playouts, and the overhead (percent of time spent in an MCTS tree). Data was measured for batches of size 10, 50, 300, 600, 1500, and 3000; 10 runs for each test.

longer the time, the more overhead is observed, and the results are getting worse compared to the zero-overhead situation from the previous experiment.

Increasing batch size makes the evaluation of expanded nodes more reliable, yet significantly reduces their number. Reduced batches, instead, lead to more frequent calls to software part of the algorithm, increasing the overhead.

Figure 2 presents the data we gathered for Pentago (for other games the charts look similar). For both games we run 10 matches against the random player, considering only 10 first turns of each match. The number of node expansions obtained by the software player (using the same, yet non-batched MCTS, so number of playouts and node expansions are the same) is provided as a baseline.

Despite the overhead, it is possible to adjust batch so the FPGA player performs much more playouts than software-only agent. However, it is impossible to reach the same size of MCTS tree, which significantly influences the performance. This can be solved by implementing multithreaded MCTS, embedding MCTS in the FPGA, or using hardware with shorter communication latency.

## 5 Discussion

Let us analyze the time-profile of an initialization process. The FPGA initialization time currently prevents the described solution to be embedded in a competition-ready GGP player. However, there are possibilities for reducing it significantly. The first phase is a software propnet construction. The exact times have been presented in Table 2. For generating Verilog propnet module, we use

our own Java library, that requires up to 2 minutes and can be easily optimized. In our GGP player, the dynamic part of FPGA is a particular game's propnet, and the support structure remains constant. Preparing this support structure (compilation and fitting) requires 3 minutes, but can be reduced significantly using commercial tools, e.g., Intel Quartus Prime Pro, which allows caching part of the compilation process. The most time-consuming phase is propnet structure fitting, responsible for the physical placement of the logic modules on a chip. Its time depends mostly on the number of components, and requires solving computationally hard problems. Currently, it can take from 1 to about 30 minutes, depending on the game size.

Due to limited space on the FPGA chip, there is a hard limit on the game size we are able to handle. However, this size is not a direct result of the number of the propnet components, as it also heavily depends on the graph planarity and the optimizations performed by the synthesis toolchain. We can observe in Table 2 that for Reversi and breakthrough smaller initial propnet size lead to much higher chip utilization. Also, we would like to point out that the largest chip utilization we have observed is 72%, which allows to estimate the limit on the games we are able to hardware-accelerate using the described system.

We also have a limitation associated with memory block size that can be easily extended in the future. In the described implementation, the state size times number of legal joint moves in this state cannot exceed 32KB.

There is also a number of significant optimization improvements we can apply. For example, during gameplay, clock frequency has to be low because of long signal propagation paths within the game propnet. However, after the result is calculated and information exchange between the shared memory and propnet driver starts, frequency can be temporarily increased by an order of magnitude. This will make the process of writing data to the memory several times faster.

The usage of a PCI-E equipped FPGA-board would allow pushing the ARM computer out of the loop, removing the need to handle propnet on two machines and allowing FPGA board to talk directly to the main, much powerful, computer (which is necessary to reduce the influence of the MCTS tree overhead).

Currently, when MCTS has control, the reasoner is idle, waiting for the next task. As the reasoner operates independently of the ARM computer, it is possible to remove those pauses, by scheduling tasks ahead. From the MCTS point of view it can be managed as multithreaded simulations, e.g., by using virtual loses [17].

Summarizing, it is possible to create a fully functional GGP player that can successfully participate in the GGP competitions, although it requires heavy optimizations and top-level hardware and software. Still, for some large games, there is no guarantee that the initialization will finish before the imposed start time. This can be partially solved by storing compiled FPGA projects for already known games, allowing their fast retrieval when the same game is detected.

### 5.1 Future work

Although the hardware accelerators provide the highest computational efficiency, it comes with some drawbacks. However, as we have a complete implementa-

tion of the propnet in the Verilog, we can use the industrial-grade simulators and optimizers to run the propnet in software. This could lead to better optimized propnet structure and allow more straightforward embedding into the GGP player. To evaluate the usefulness of such simulated hardware propnets, we plan to implement them and compare their efficiency against the reference Java implementation and our FPGA-based reasoner.

Most MCTS implementations are based on the purely random playouts; however, multiple more sophisticated strategies have proven to be quite effective [5, 17]. We would like to implement and test such non-random simulations on the FPGA. This will complicate the board architecture and slow down the reasoner. Yet, it may be the only possibility to overcome certain limitations, and tackle games that cannot be solved by even extremely efficient brute force search.

In particular, because FPGAs have memory distributed around the entire chip, it is possible to locally keep track of state changes. Thus, once a player wins, we can memorize which propositions contributed to this, and create a heuristic state evaluation function that improves over time, similarly to some simulation control learning algorithms presented in [5].

## 6 Conclusions

In this paper, we present the first attempt to encode propnets, a successful computational representation in GGP, on a hardware chip. Because a GGP player has to handle any game encoded in GDL, we based our system on FPGAs, which allow us to reprogram our hardware reasoners and quickly switch between previously encountered games.

This is preliminary work that opens a new branch of GGP research, parallel to the improvement of software-based reasoners, which has been worked on for nearly a decade [19]. The approach we described is able to achieve from 25 up to 290 times improvement over the software propnets when comparing the number of visited states per time unit. Moreover, the ratio is considerably better for large games, the ones that are especially problematic for all kinds of software-based reasoners – even GDL compilers. We may conclude that FPGA-propnets are a faster alternative to software propnets for reasoning on game descriptions.

We also integrate the FPGA-propnet within an MCTS agent and discuss the difficulties that this entails. Using Pentago, we show how the communication between the FPGA-propnet and the software that manages the search introduces a considerable overhead. Because of this, our FPGA-propnet MCTS agent is not ready to participate in a GGP competition yet. However, we discuss various improvements that can solve the current shortcomings. This, together with the successful performance of the FPGA-propnet with respect to the software propnet when tested on their own, indicates that this research direction is promising.

Moreover, although we plan to enhance our system to handle more sophisticated AI approaches, we think that merging vanilla MCTS with computation power of hardware raises an interesting question about a gameplay level that can be achieved by using sheer brute force.

## References


1. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-time analysis of the multiarmed bandit problem. Machine learning **47**(2-3), 235–256 (2002)
2. Boulé, M., Zilic, Z.: An FPGA Move Generator For the Game of Chess. ICGA Journal **25**(2), 85–94 (2002)
3. Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A Survey of Monte Carlo Tree Search Methods. IEEE TCIAIG **4**(1), 1–43 (2012)
4. Coulom, R.: Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In: Computers and Games. pp. 72–83 (2007)
5. Finnsson, H., Björnsson, Y.: Learning Simulation Control in General Game Playing Agents. In: AAAI. pp. 954–959 (2010)
6. Genesereth, M., Love, N., Pell, B.: General Game Playing: Overview of the AAAI Competition. AI Magazine **26**, 62–72 (2005)
7. Genesereth, M., Thielscher, M.: General Game Playing. Morgan & Claypool (2014)
8. Haiying, G., Fuming, W., Wei, L., Yun, L.: Monte Carlo simulation of 9x9 Go game on FPGA. In: Conference on Intelligent Computing and Intelligent Systems. vol. 3, pp. 865–869 (2010)
9. Hsu, F.H.: Chess hardware in Deep Blue. Computing in Science Engineering **8**(1), 50–60 (2006)
10. Kowalski, J., Szykuła, M.: Game Description Language Compiler Construction. In: AI 2013: Advances in Artificial Intelligence, LNCS, vol. 8272, pp. 234–245 (2013)
11. Love, N., Hinrichs, T., Haley, D., Schkufza, E., Genesereth, M.: General Game Playing: Game Description Language Specification. Tech. rep., Stanford Logic Group (2008)
12. Olivito, J., Resano, J., Briz, J.L.: Accelerating Board Games Through Hardware/-Software Codesign. IEEE TCIAIG **9**(4), 393–401 (2017)
13. Romoth, J., Porrmann, M., Rückert, U.: Survey of FPGA applications in the period 2000–2015 (Technical Report) (2017)
14. Schkufza, E., Love, N., Genesereth, M.: Propositional Automata and Cell Automata: Representational Frameworks for Discrete Dynamic Systems. In: AI 2008: Advances in Artificial Intelligence, LNCS, vol. 5360, pp. 56–66 (2008)
15. Schreiber, S.: The general game playing base package. http://code.google.com/p/ggp-base/ (2013)
16. Schreiber, S.: Stanford Gamemaster. http://games.ggp.org/stanford/ (2016)
17. Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., Hassabis, D.: Mastering the game of Go with deep neural networks and tree search. Nature **529**, 484–503 (2016)
18. Sironi, C.F., Winands, M.H.M.: Optimizing propositional networks. In: Computer Games, CCIS, vol. 705, pp. 133–151 (2017)
19. Waugh, K.: Faster State Manipulation in General Games using Generated Code. In: IJCAI Workshop on General Intelligence in Game-Playing Agents (2009)
20. Wong, C., Lo, K., Leong, P.H.W.: An FPGA-based Othello endgame solver. In: Conference on Field-Programmable Technology, 2004. pp. 81–88 (2004)