Mobile Robot Path-planning Implementation in Software and Hardware

LUCIA VACARIU, FLAVIU ROMAN, MIHAI TIMAR,
TUDOR STANCIU, RADU BANABIC, OCTAVIAN CRET
Computer Science Department
Technical University of Cluj-Napoca
26-28 Baritiu Str., Cluj-Napoca
ROMANIA

http://www.cs.utcluj.ro/

Abstract: Search algorithms represent a useful and reliable technique in solving path finding, path planning and obstacle-avoidance types of problems that appear in mobile robots applications. The paper presents different implementations, both in software and in hardware, of the Breath-First (BF) Search Algorithm. The software implementation uses the standard type Intel Pentium 4 Processor and Java software environment. We also implemented the search algorithm in a hardware-based environment, built upon Xilinx FPGA core technologies. The comparison shows that the hardware implementation is an alternative to the classical software implementation that is a larger resources and time consumer.

Key-Words: Mobile Robot, Path-planning, Search algorithms, FPGA, VHDL

1 Introduction

The mobile robots need a control system for navigating in an environment [1]. A component of this navigation control system is the planning of the path the robot must follow. The path planning system determines the current position of the robot in the path, when the Start and the Goal position are known, with respect to the same reference system in the environment where the robot navigates. The environment is static or dynamic, with discrete or continuous representation.

To obtain the optimization of the path planning it is necessary to avoid obstacle collisions and to provide the shortest way or the shortest time from Start Point to Goal Point. With respect to this, there are different path planning implementations [2].

The search algorithms represent a useful and reliable technique in solving path-finding, path-planning and obstacle-avoidance types of problems. BF (Breath-First), DF (Depth First), A-star algorithms are successfully used to determine a valid path [3]. The characteristics of these algorithms allow them to be well fitted into mobile robots applications.

The diversity of applications for mobile robots and the diversity of environments where they navigate require in path planning a large amount of computer resources.

Another alternative to software implementation of algorithms is the use of digital devices Field Programmable Gate Arrays (FPGA) with very good performances in the processing capacity [4]. The reconfigurable logic devices may be used for repetitive and very much time consuming operations. Plus, the reconfigurable hardware provides great flexibility [5].

We used the BF algorithm to obtain the path of a mobile robot that navigates in a given static environment. We implemented and optimized the algorithm both in software and in hardware. We obtained results which prove that the execution times of the algorithm in hardware are much better than in software, using tests on the same environments.

The paper is structured as follows: section 2 deals with the overall description of BF used algorithm and its software implementation. Section 3 explains the hardware implementation chosen for the algorithm and presents the tests and experimental validation of the hardware implementation. Section 4 reports the comparative results obtained in the two implementations, and section 5 presents conclusions and ideas for future work.

2 Software Implementation

We used the Breath-First Search Algorithm to obtain the valid path for a mobile robot that navigates in a given static and discrete environment.

2.1 Breath-First Search Algorithm

The BF is a search facility developed in order to find a path from a certain point to another, in an environment which can be always a more or less particular representation of a (Nodes, Vertices) Graph [3].

The Breath-First is a technique for searching and returning a path from a given Starting Point to a given Goal Point. The algorithm guarantees finding a solution, if there exists one. As for the complexity, it is a linear algorithm with respect to the number of considered nodes, while the way it searches is based on maintaining a queue of all neighbors found to be accessible through means of vertices until the Goal is reached. Keeping a similar queue, in which for each node we keep the one from which our node was found as neighbor, does the reconstruction of the path.

We chose to use a typical, bidimensional, discrete environment, most often represented as a matrix of squares (cells), and denote the positions of the cells using the combination of the two coordinates. This way, the environment presents the cells as Nodes, and we consider the Vertex to be represented by the neighbors in the 4 directions (N, S, W, and E). There is no connection on diagonals (Fig.1).

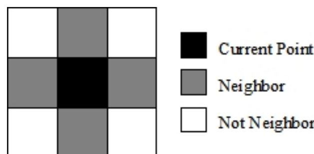

Fig.1 Neighbors scheme

The matrix contains free spaces, obstacles, the Starting Point (S) and Goal Point (G). It is assumed that the matrix is closed, bordered by obstacles (Fig.2).

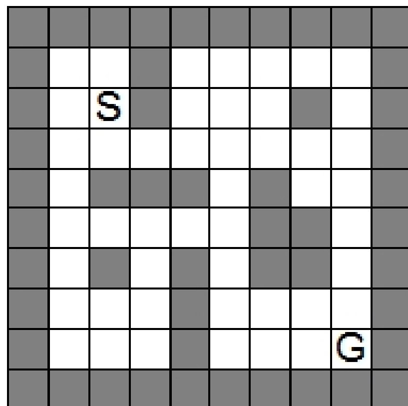

Fig.2 Sample matrix

Given a random map that respects the criteria mentioned above, a Starting Point and a Goal Point, the implementation of the Breath-First Search will return (if there exists) a valid path between the two points. The path will be marked, assuming that one can pass only through the free spaces and the directions of motion are only the ones stated. The algorithm is presented in Listing 1.

```
//BF Search Algorithm

queue q            // the process queue
point start        // starting point
point goal         // goal point
point crt          // current point
list nb            // list of free neighbors of current point
list visited       // list of booleans <=> "true" = "visited"
list prev          // list of marks, for each node we mark the
                   // node which "discovered" the current one

// beginning of algorithm

push(q,start)
prev[first] <- -1 // mark that there is no "prev" for first
set all visited <- "false"
visited[first] <- "true"

while (q not empty) execute
     crt <- pop(queue) //pop current node
     if (crt == goal)
          return (point,goal,last) //solution found
     else
          nb <- all FREE neighbors of crt //4 directions in our case
          for all items in nb
               if (not visited[item])
                    push(q,item)
                    last[item] <- crt
                    visited[item] <- "true"
               end if
          end for
     end if
end while

return "No solution" // if we arrive here the goal was not reached

// end of algorithm
```

Listing 1. BF Search Algorithm

The reconstruction of the solution is done by means of a special routine, which takes the list of previous elements and reconstructs the visited points of the map (Listing 2).

```
// Reconstructing the solution path

point start
point goal
list prev
point current

// beginning of code

current <- goal;

while (current != -1) execute // "-1" is "prev" of first
     mark current as "in the path"
     current <- prev[current]
end while

// end of code
```

Listing 2. Reconstruction of path

2.2 Implementation in Software

For the software implementation and experiments, we have chosen Java as Integrated Development Environment. The implementation required Object-Oriented techniques, Graphical User Interface and Data/Maps saving and exporting, the Java compiler and emulator [6], and the Eclipse IDE [7].

The procedure to measure the time interval in which the algorithm runs was to capture the system time and to compute the difference of the values before and after the execution of the algorithm routine (Listing 3).

```
Discrete d = new Discrete();  //space of matrix
d.readFromFile("ten.txt");    //acquire matrix

long time = System.currentTimeMillis();    //clock before
d = bf.startBFSearch(d);
long time2 = System.currentTimeMillis();   //clock after

elapsed = (time2 - time);                  //elapsed
new ShowPath(d).drawPath();                //map drawing
```

Listing 3. Execution time computing

We performed various tests including manually generated maps, random maps and small to large maps. We observed that the running time increases significantly for large maps. Each increase in the length of an assumed square matrix produces a 2nd order polynomial increase in the number of cells and spreading of the algorithm queue (which is the number of cells included in the queue but that are not on the path). Results prove very fast times and very short paths (80% of the times optimal) for small matrices (Fig.3).

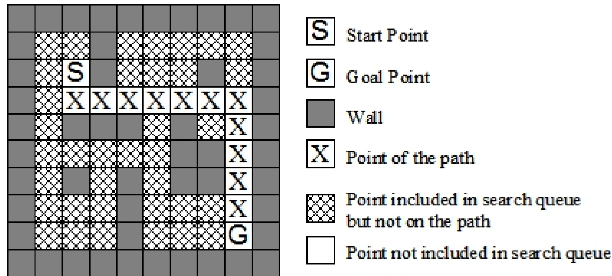

Fig.3 Result of BF Search

For very large dimensions of maps (e.g. 200x200 cells) we could not test the algorithm because the program required more virtual memory for processing than the operating system could give it. This is one of the limitations of the software implementation, another one being that for even larger maps (e.g. 1000x1000), the memory is insufficient even to hold the map data.

Also, a comparison table has been generated, based on the obtained results. The lower the spreading is, the shorter the time becomes. The time is given in milliseconds.

Width	Height	Total	Obstacles Fill	Spreading	Time (ms)
30	30	900	25%	High	10
30	30	900	50%	Medium	Almost 0
40	40	1600	25%	High	40
40	40	1600	50%	High	10
100	100	10000	25%	High	120
100	100	10000	50%	High	40

Table 1. Software BF Implementation Results

3 Hardware Implementation

Our hardware-based environment is built upon Xilinx FPGA core technologies, which is the well-known developing technology in reconfigurable-based computing functionalities.

3.1 Hardware environment

For the hardware implementation, we chose from the FPGA family, a Xilinx Virtex 2Pro Board (XC2VP30-FF896-6), manufactured by Digilent Inc. The board is equipped with VGA-output, used for visualization of test results on the monitor. It required Xilinx ISE 8.1 Environment [9], which was used for VHDL code synthesis, implementation and board programming.

A series of adaptations of the BF algorithm had to be done, in order to exploit the logic resources of the Virtex FPGA device. The input matrix is stored in BlockRAMs. Depending on the space available in the FPGA device, the process memory can be implemented inside or outside the chip (in the Virtex BlockRAMs or in an external dynamic RAM).

All components have been described in parameterizable VHDL code. Thus, the design becomes portable on any hardware support system.

3.1.1 Design of components

The hardware solution is based on a structural description with component-style design. It uses interconnected components, each of them performing a certain task. The most important component is the BF component, which implements the algorithm (Fig.4).

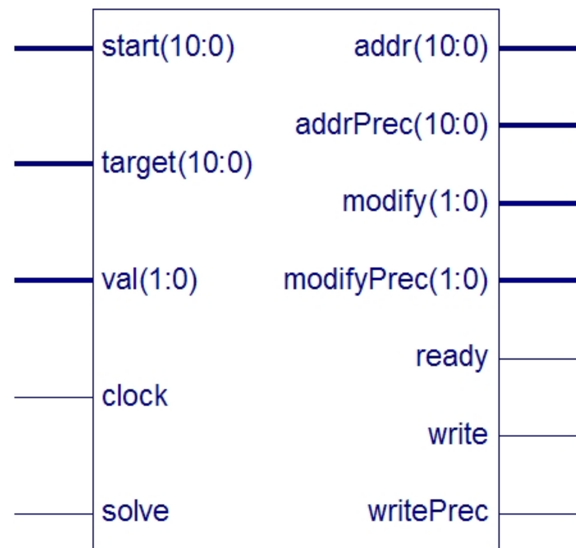

Fig.4 BF Component top view

The signals interact with the required memory for applying the BF algorithm, while some of the signals (clock, solve as inputs and ready as output) are signals that belong to the hardware configurations and interconnections.

The MarkPath component is the one that implements the reconstruction of the path after the algorithm has been applied (Fig.5).

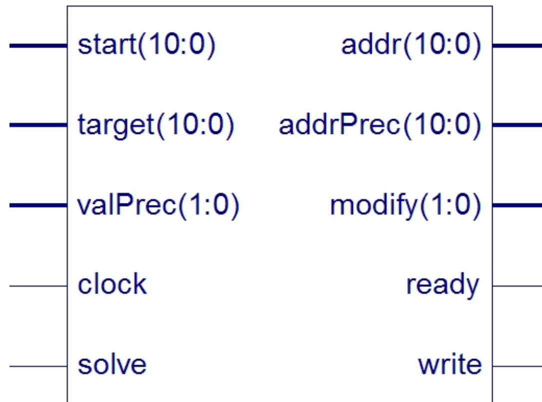

Fig.5 MarkPath Component

In hardware, no values initialization is made automatically, therefore it is required that before any algorithm is applied, the system is brought to a known state, which must be the initial state of the algorithm. Every signal must be initialized, and the memory map also. Therefore, we have designed a component that deals with all these and with other clock synchronization and enable / ready types of signals. The component is called CentralUnit (Fig.6).

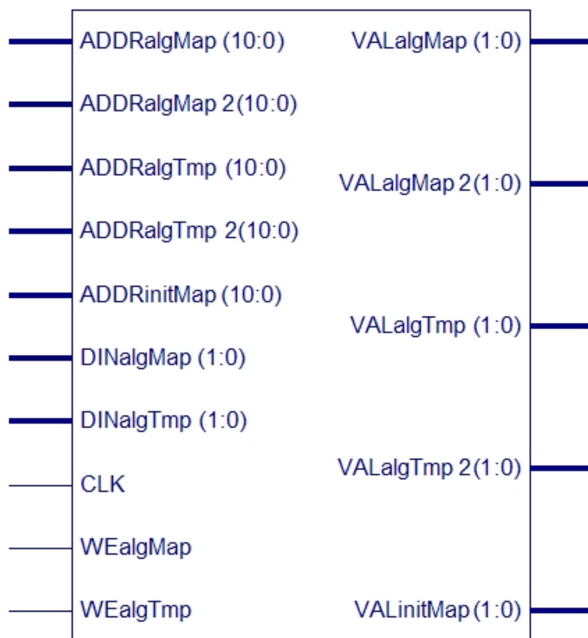

Fig.6 Central Unit Component

One of the components required for the integrated system to work is the CopyMem component, which reinitializes the matrix for the algorithm. This matrix will be usable for future implementation of other algorithms too (Fig.7).

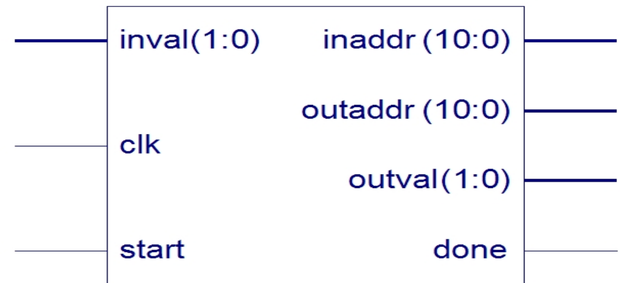

Fig.7 CopyMem Component

All these components are gathered into a larger one, which is a relatively simple Finite State Machine (FSM). This component supervises by its 4 states (Standby, Init, BF, and Path) the system's tasks.

The VGA module contains the necessary signals to synchronize the output on a regular monitor, it captures the value from the matrix and it outputs the color corresponding to the matrix value (Fig.8).

Fig.8 VGA Component

The block represents the VGA controller that provides the image on an 800x600 pixels screen display. It uses a 65 MHz clock and 60 ± 1 Hz refresh. This block generates the CRT-based HS and VS timing signals, and the R, G, B video data signals.

3.1.2 Integrated system design

The architecture of integrated system has been designed as the collection of components, connected to each other by signals, and with a few processes that handle the necessary synchronizations, command all the components, acquire signals from board inputs, and send data to outputs (monitor) (Fig.9).

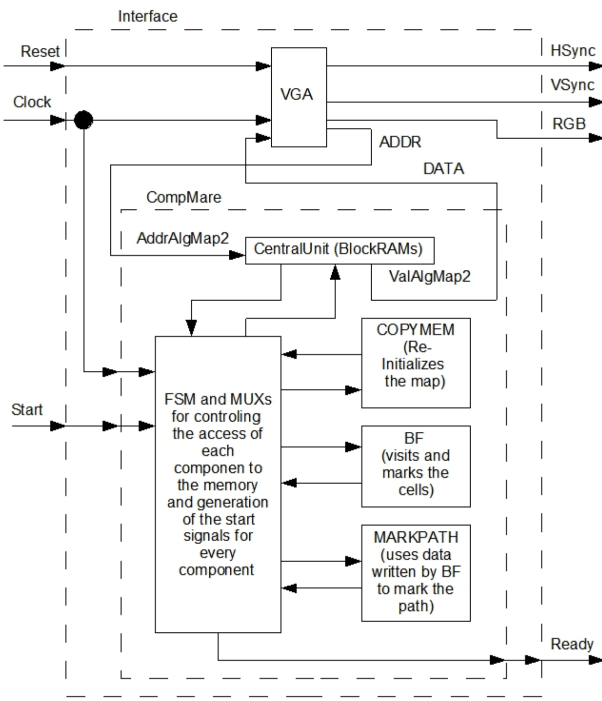

Fig.9 Integrated system

3.2 Testing and performance of hardware implementation

The hardware implementation was tested several times using different maps, the same that were used for the software implementation.

The results were photographed after the algorithm execution.

At the beginning, the report of the Xilinx synthesis process shows that the system can work at a maximum frequency of 185 MHz. We also tested the performances of our hardware implementation using the testing environment ModelSIM XE III 6.1 from Xilinx [10]. The waveforms generated helped us in choosing the working. We decided to start testing at 120 MHz frequency.

The images taken from the hardware implementation of the BF algorithm present the obtained map on the monitor. On the map, the path found and the dispersion are in the left side, and the directions chosen at each point, are in the right side. Fig. 10 to 14 show different map dimension used to verify that the BF algorithm obtains a valid path and to check how much time is required for that.

Figures prove that the algorithm performs very well in terms of path optimality, even if the spreading is almost every time very large.

Table 2 and Table 3 presents samples of algorithm execution times (in µs) in our hardware implementation at the 120 MHz and 150 MHz working frequency.

Fig.10. Result for 10x10 map dimension

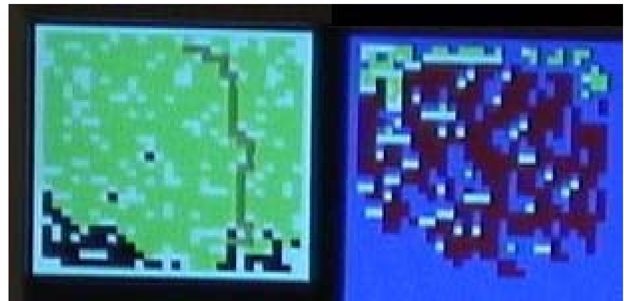

Fig.11. Result for 30x30 map dimension

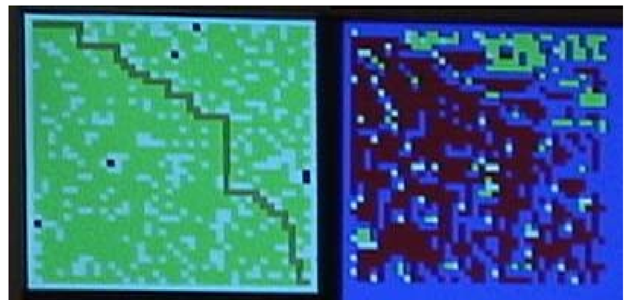

Fig.12. Result for 40x40 map dimension

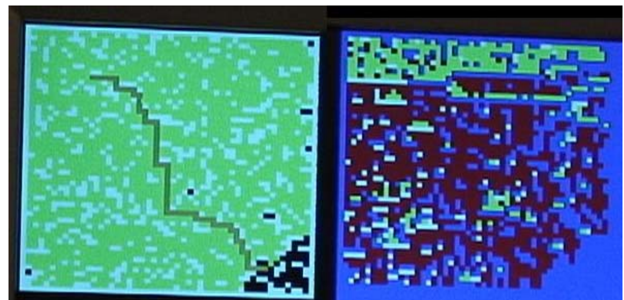

Fig.13. Result for 50x50 map dimension

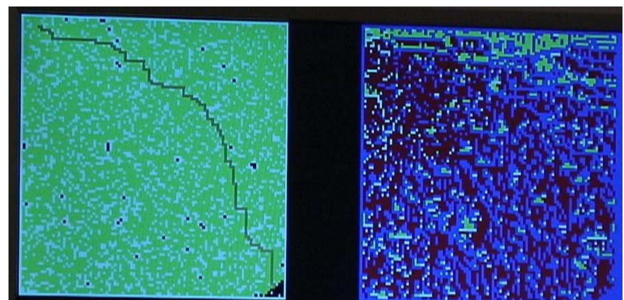

Fig.15. Result for 100x100 map dimension

Freq	Width	Height	Total	Obstacles Fill	Spreading	Time (µs)
120 Mhz	10	10	100	25%	High	2.78
	30	30	900	25%	High	34.95
	40	40	1600	50%	High	73.81
	50	50	2500	50%	High	112.15
	100	100	10000	50%	High	475.38

Table 2. Hardware BF Implementation Results at 120 MHz working frequency

Freq	Width	Height	Total	Obstacles Fill	Spreading	Time (µs)
150 Mhz	10	10	100	25%	High	2.35
	30	30	900	25%	High	28.2
	40	40	1600	50%	High	59.54
	50	50	2500	50%	High	90.14
	100	100	10000	50%	High	380.3

Table 3. Hardware BF Implementation Results at 150 MHz working frequency

The memory available in FPGA Virtex devices allows the implementations of large matrix (e.g. 100x100). Our maps use variable dimension matrices. Even for large maps, the FPGA circuit is used only at a small fraction of its capacity. But a lot larger maps cannot be stored into the BlockRAMs, considering the fact that we also need to keep the map of chosen directions, to be able to reconstruct the path. For these situations, an external Dynamic RAM memory can used.

4 Comparison Results

The different results between the two kinds of implementation appear especially for the run time of the algorithm. Generally, all of the measured times were at least two orders of magnitude better in hardware than in software.

By increasing the working frequency of the hardware clock, the execution time diminishes proportionally.

For very small dimension maps the software implementation is preferable because the time is small enough and the implementation is easy. For large dimension maps the hardware one is preferable because of much better running times.

The memory size used for maps in hardware implementation can be limited only to the amount of BlockRAM memory available in the Virtex devices. If a larger size memory is necessary, it can be implemented outside the chip, but in this case the working frequency will obviously be smaller than now.

The board with FPGA digital device is proper to use for mobile robot applications. It is now possible to take over a part of the necessary control system of the mobile robot skills.

5 Conclusion

Ours results demonstrate that the hardware-level solution for path-planning algorithms implementations is much faster and proves to be a serious alternative to usual software solutions.

The development of such hardware implementations remains quite difficult because of the lack of standards and the need to focus on every single detail of the implementation, normal for such a low-level approach.

Software implementations though provide flexible solutions, easy to implement, manage and maintain, and easy to modify, but much slower.

In our future research, we intend to make hardware implementations for other known path-planning algorithms, also based on the agent-oriented paradigm, and to use them for mobile robot navigation.

References:

[1] Siegwart, R., and Nourbakhsh, IR. *Introduction to Autonomous Mobile Robots*, The MIT Press, 2004.

[2] Arai T., Pagello, E., and Parker, LE. Advances in Multi-Robot Systems, *IEEE Transactions on Robotics and Automation,* Vol.18, No.5, 2002, pp. 655-661.

[3] Cormen, TH., Leiserson CE., Rivest, RL. *Introduction to Algorithms.* The MIT Press and McGraw-Hill, 1990.

[4] Cret, O., Mathe, Z., Grama, C., Vacariu, L., Roman, F., Darabant, A. Solving the Maximum Subsequence Problem with a Hardware Agents-based System. *WSEAS Transactions on Circuits and Systems*, Vol.5, No.9, 2006, pp. 1470-1478.

[5] Katz, RH., and Borriello G. *Contemporary Logic Design*. Benjamin Cunnings/Addison Wesley Publishing Company, 2005.

[6] The *Java API*, java.sun.com/reference/api

[7] The *Eclipse IDE*, www.eclipse.org

[8] *Digilent XUP/V2P board documentation,* www.digilentinc.com

[9] The *Xilinx ISE 8.1 Environment*, www.xilinx.com/support

[10] *ModelSIM XE III*, www.model.com