

# A Motion Planning Processor on Reconfigurable Hardware

Nuzhet Atay

Department of Computer Science and Engineering  
Washington University in St. Louis  
Email: atay@cse.wustl.edu

Burchan Bayazit

Department of Computer Science and Engineering  
Washington University in St. Louis  
Email: bayazit@cse.wustl.edu

**Abstract**—Motion planning algorithms enable us to find feasible paths for moving objects. These algorithms utilize feasibility checks to differentiate valid paths from invalid ones. Unfortunately, the computationally expensive nature of such checks reduces the effectiveness of motion planning algorithms. However, by using hardware acceleration to speed up the feasibility checks, we can greatly enhance the performance of the motion planning algorithms. Of course, such acceleration is not limited to feasibility checks; other components of motion planning algorithms can also be accelerated using specially designed hardware. A Field Programmable Gate Array (FPGA) is a great platform to support such an acceleration. An FPGA is a collection of digital gates which can be reprogrammed at run time, i.e., it can be used as a CPU that reconfigures itself for a given task.

In this paper, we study the feasibility of an FPGA based motion planning processor and evaluate its performance. In order to leverage its highly parallel nature and its modular structure, our processor utilizes the probabilistic roadmap method at its core. The modularity enables us to replace the feasibility criteria with other ones. The reconfigurability lets us run our processor in different roles, such as a motion planning co-processor, an autonomous motion planning processor or dedicated collision detection chip. Our experiments show that such a processor is not only feasible but also can greatly increase the performance of current algorithms.

## I. INTRODUCTION

In this paper we investigate the feasibility of a motion planning processor (MPP). We present a prototype processor and evaluate its performance against a high-end workstation. Such a processor has several advantages. Since it is a dedicated platform, it can be optimized to run the motion planning algorithms in the most efficient way. It may be attached to a host computer to reduce computation times. The cost of such processor would be the fraction of a general purpose CPU hence hundreds of them can be deployed in parallel to further increase the computational capabilities. It would also be a low-power unit reducing power consumption. It can be utilized as the control unit of an autonomous robot reducing its power consumption while increasing the computational efficiency. Such a processor would also help small robots, where the controller size needs to be smaller (such as the control of nano robots). Most important of all, by implementing such a processor on reconfigurable hardware and utilizing high level abstraction of the motion planning algorithms, the processor can be adapted and used in different domains.

Motion planning algorithms have applications in a wide range of domains, e.g., robotics, animation, biomedicine etc. The goal of a motion planning algorithm is to find a feasible path for the moving objects. By changing the feasibility criteria, the same motion planning algorithm can be used in different domains. For example, in probabilistic roadmap method (PRM) [1] being collision free can be selected as a feasibility criteria for robot navigation. The same PRM algorithm can also be used to find protein folding paths if the feasibility criteria is replaced by low-potential energy configurations [2]. Hence, the feasibility check is a fundamental operation in motion planning algorithms.

In addition to high level abstraction, the modern motion planning algorithms [1], [3]–[6], have another nice feature, inherited parallelism. In fact, motion planning algorithms generally have the parallelism in two levels. First individual parts of the algorithm can run in parallel (e.g., node generation in PRMs). Second, the feasibility check itself can be implemented in parallel (e.g., collision check between individual robot polygons and obstacle polygons can be run in parallel). This second level of parallelism has been recently realized in [7] where a graphics processing unit (GPU) is utilized to accelerate collision detection in motion planning. However, this approach still lacks the first level parallelism where individual components run in parallel. Even if multiple CPUs are used, other factors such as interruption from system kernel could reduce the efficiency. Instead, we propose a dedicated processor with the maximum parallelism at the each component of the system.

We have developed our prototype processor by using Field Programmable Gate Arrays (FPGAs). An FPGA is a collection of digital gates which can be reprogrammed at run time, i.e. it can be used as a CPU that reconfigures itself for a given task. FPGAs are continuously improving while suppressing the Moore's law. A recent analysis shows that it is expected that FPGA floating-point performance will overpass the general purpose CPUs by 2009 [8]. Although we have selected FPGA as our implementation platform, our design can be applied to other types of configurable hardware such as Application-Specific Integrated Circuits (ASIC), which are more expensive but faster.

Our motion planning processor is based on probabilistic roadmap method (PRM) because of their proven par-

allelism [9]. Since our aim is to evaluate the feasibility of motion planning processors, we have targeted the rigid body motion planning problem where comparison of different solutions is relatively easy. The object representations are sent to the motion planning processor through serial port and the processor returns a valid roadmap. The objects in the environment (obstacles and the robot) are general, non-convex objects represented with triangular meshes. Whenever it is possible, the processor executes node generation, collision check and local planning in parallel. We also take advantage of the reconfigurable nature of our motion planning processor. If it is required, the processor can work as dedicated collision detection chip and may serve as a feasibility checking co-processor for a more complex algorithm running on the host computer.

Although our current implementation is for rigid body motion planning, our processor can easily be modified to be used in high dimensional problems or in different domains. For example, by replacing the collision detection module with potential energy computation, we can build the roadmaps for proteins.

FPGAs are traditionally used as development and prototyping environment, but with the advances in their capabilities, they began to be used in many areas like telecommunications, automotive, networking etc. In this paper, we show that the robotics research can also gain from the advances of reconfigurable hardware. To the best of our knowledge, we are the first researchers to implement a dedicated motion planning processor. Using the current FPGA technology, we obtained speed-ups of upto 25 with respect to a Pentium 4 machine running at 3GHz with 1 GB memory. Higher speed-up can be reached by using faster reconfigurable systems such as ASIC (which typically runs 10 times faster than FPGAs).

In the next section we summarize related work. Section III present background information on PRMs and FPGAs. Sections IV and V describes our processor. Section VII shows our results and Section VIII concludes the paper.

## II. RELATED WORK

Amato and Dale's work [9] shows that probabilistic roadmap methods are embarrassingly parallel. In our design we have used a similar parallelism at each step of PRM algorithm. Gayle, et.al. recently achieved hardware acceleration in motion planning using graphics hardware (GPU) [7]. However, while utilizing parallel collision detection, other components of the motion planning algorithm still runs sequentially. Although not directly applied to motion planning, an increasing number of collision detection algorithms are achieving hardware acceleration using GPUs [10]–[20]. Recently there was another approach to gain hardware acceleration by implementing the collision detection using Application-Specific Integrated Circuit (ASIC) [21]. However, this design was not used in motion planning. To the best of our knowledge, we are the first researchers implementing a motion planning processor.

## III. BACKGROUND

### A. Probabilistic Roadmap Methods

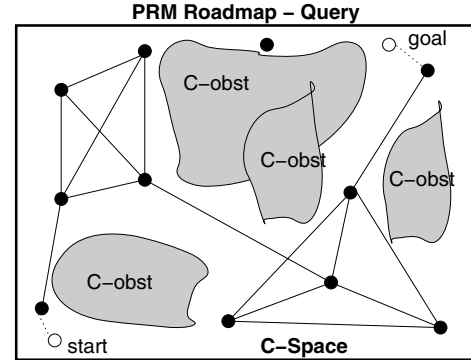


Fig. 1. A PRM roadmap in C-space.

PRMs work by sampling points ‘randomly’ from the robot’s configuration space (C-space), and retaining those that satisfy certain feasibility requirements (e.g., they must correspond to collision-free configurations of the movable object). Then, these points are connected to form a graph, or roadmap, using some simple planning method to connect ‘nearby’ points. During query processing, the start and goal are connected to the roadmap and a path connecting their connection points is extracted from the roadmap using standard graph search techniques (see Figure 1).

An algorithm for PRM can be summarized as the below:

#### PRMS: PROBABILISTIC ROADMAP METHODS

##### I. PREPROCESSING: ROADMAP CONSTRUCTION

1. NODE GENERATION (find collision-free configurations)
2. CONNECTION (connect nodes to form roadmap)  
(repeat as desired)

##### II. QUERY PROCESSING

1. CONNECT START/GOAL TO ROADMAP
2. FIND PATH IN ROADMAP BETWEEN CONNECTION NODES

PRMs have been shown to perform well in practice. In particular, after the roadmap is constructed during preprocessing, many difficult planning queries can be answered in fractions of seconds [1].

### B. Field Programmable Gate Arrays

A field-programmable gate array (FPGA) is an integrated circuit (IC) that can be programmed in the field after it is manufactured. FPGAs are similar in principle to, but have vastly wider potential application than programmable read-only memory (PROM) chips. FPGAs can be defined as reconfigurable processors that can be used for testing and implementing designs. Designs loaded on FPGAs are not final. As a result, the configuration of the circuit can be updated whenever needed.

A typical FPGA circuit (see Figure 2) consists of configurable logic cells (CLB) which are the primitive elements of FPGAs. Each CLB contains look-up tables (LUT) where combinatorial logic is stored. LUTs are memory elements

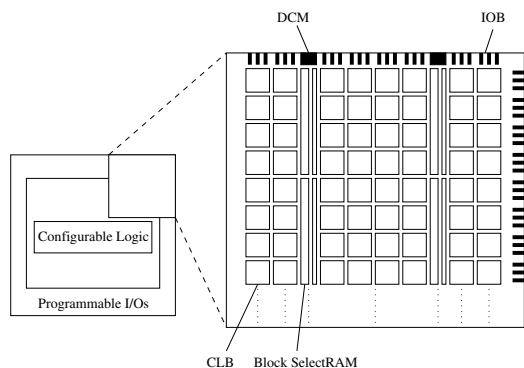


Fig. 2. A typical FPGA layout: configurable logic cells (CLBs), block RAMs, digital clocks (DCMs) and input-output buffers (IOBs).

which are addressed by inputs of circuit. Instead of rearranging gates according to design, the design is converted into LUTs.

To define the behavior of the FPGA it is required to use a Hardware Description Language (HDL) or a schematic designed using an electronic design automation tool. Either of these, when compiled, will generate a netlist, that can be mapped to the actual FPGA architecture. When done, the binary file generated is used to (re)configure the FPGA device. Common HDL's are VHDL and Verilog. A good introduction to FPGAs can be found in [22].

#### IV. SYSTEM OVERVIEW

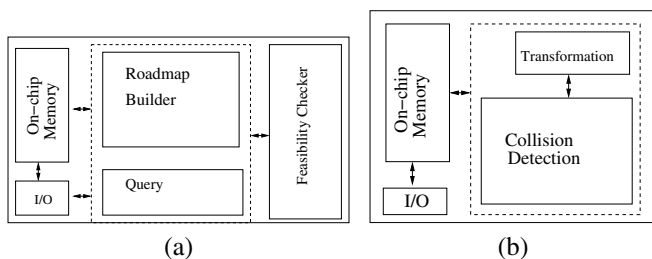


Fig. 3. Motion planning processor (MPP) in different configurations: (a) motion planning configuration, feasibility checker can be specified depending on the problem, e.g. collision detection or potential energy computation, (b) dedicated collision detection configuration.

The proposed motion planning processor has five major modules (Figure 3(a)): (i) *I/O* is responsible for communication between the host computer and the chip, (ii) *memory* stores the environmental models, (iii) *roadmap builder* builds a roadmap using the current models and feasibility criteria, (iv) *query* finds a path through roadmap and, (v) *feasibility checker* checks whether current configuration of the robot satisfies the feasibility constraint. This modular representation lets us switch individual components to reconfigure our hardware. For example, currently we use the collision detection for feasibility criteria, but in the future, we can switch the feasibility check module to potential energy computations, and we can use our motion planning processor for protein interactions. Similarly by removing roadmap builder and query modules, we can dedicate more space to the feasibility constraint checker and we can have a collision detection chip (see Figure 3(b)).

#### V. MOTION PLANNING PROCESSOR

The two most important modules of the motion planning processor are *Roadmap Building* and *Collision Detection*. Figure 4 shows the interaction of these two modules in more detail. Please note the high level of parallelism in the system. Next we will discuss the individual components.

##### A. Roadmap Building

*Roadmap Building Module* consists of *Node Generation* and *Node Connection Components*. *Node Generation Component* finds  $n$  collision free configurations and passes them to *Node Connection Component*. *Node Connection Component*, then, builds the roadmap by checking if the configurations are reachable from each other. The final roadmap can be passed to either *Query* module or the host computer.

1) *Node Generation*: After the environmental definitions are loaded, *Node Generation Module* starts finding free configurations. There are  $n_{rand}$  ( $n_{rand} = 10$ ) parallel random node generation circuits in *Node Generation Component*. Generating single random configuration is also done in parallel, i.e., six numbers for each degrees of freedom (dof) are generated in one clock cycle. Hence we can generate  $n_{rand}$  configurations in one clock cycle. After the configurations are generated they are sent to *Collision Detection Module*. *Node Generation Module* waits until the results are received back, then either stores a configuration in the buffer (if the configuration is free) or discards it (if it is in collision). This process is repeated until  $n$  free configurations are generated (see Figure 4).

2) *Node Connection*: At the end of the node generation, collision-free configurations are stored to be used as the nodes of roadmap. Next, *Node Connection Component* tries to connect each configuration to  $k$  ( $k = 5$ ) closest configurations (see Figure 4). This component has  $n_{closest}$  ( $n_{closest} = 10$ ) circuits to find the closest configurations. Once the  $k$  closest configurations are found, then connection between the current configuration and its  $k$  neighbors are checked using local planning circuits. For each closest configuration finding circuit, there are  $n_{lp}$  ( $n_{lp} = 1$ ) local planning circuits. Figure 5 shows in detail how we implemented these steps. First, edge-finder circuit reads one configuration, then finds  $k$  closest configurations. Next, it sends each pair to intermediate-point-finder circuit. This circuit finds the intermediate points between the pair (using a straight line). Each intermediate point is then send for feasibility check. If all the configurations are collision free, then this edge stored as collision free.

##### B. Collision Detection

*Collision Detection module* is responsible for checking the collision between robot at the specified configuration and the obstacles. This module first transforms the robot to the specified configuration, then checks intersections between triangles of the robot and obstacles in a parallel fashion. In order to transform the robot to specified configuration, first, a transformation matrix is computed for that configuration. This matrix is used to transform the robot triangles. These transformed triangles are stored in a buffer (FIFO) to be used

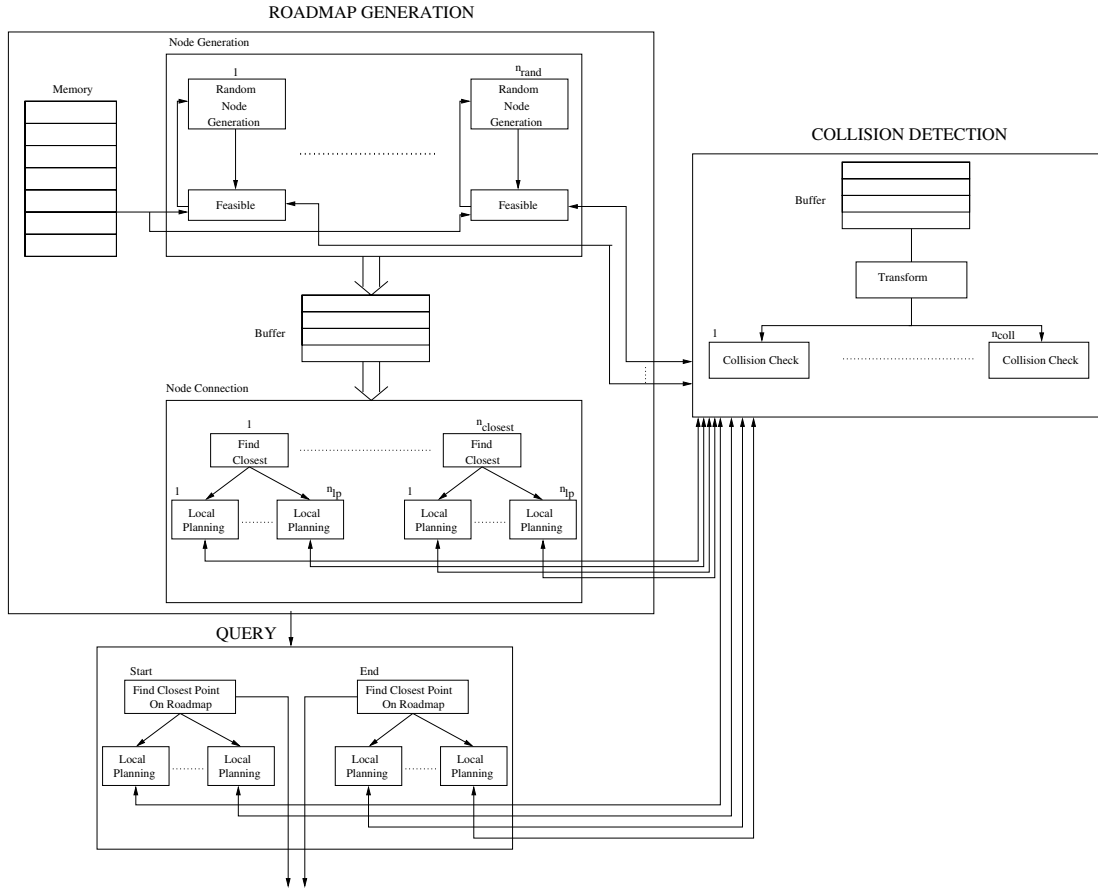


Fig. 4. PRM motion planning on Motion Planning Processor.

by collision detection circuit. By using a FIFO, dependency between two circuits is eliminated. This is important because collision detection is a time consuming process and with the help of FIFO, transformation matrix circuit does not have to wait for collision detection circuit to continue. As a result, after first transformed triangles are ready, transformation circuit computation overlaps with collision detection and its time does not increase computation time of circuit. (see Figure 6(a)).

Our triangle-triangle intersection test is based on the fast triangle-triangle intersection test described in [23]. Next, we will briefly summarize this algorithm and then show how it can be implemented in hardware.

**Fast Triangle-Triangle Intersection Detection.** This algorithm considers three cases: (i) triangles lie in the half-planes of each other, (ii) the triangles are coplanar, (iii) the triangles are not coplanar. It works in the following way:

- *Half-plane check:* If all the vertices of one triangle lies on the same half-space of the other triangle, there is no way these triangles intersects so just return *collision free*.
- *Coplanar:* If the triangles are coplanar, project them onto the axis-aligned plane where the areas of the triangles are maximized. Then do a two-dimensional triangle-triangle overlap test.
- *Not Coplanar:* If  $L$  is the line at the intersection of

two planes containing each triangle, both triangles are guaranteed to intersect with  $L$ . Find the intersection intervals for each triangle and check if they overlap (*collision*).

**Collision Detection Hardware.** Our hardware implementation closely matches the above algorithm. Figure 6(c) shows the internal structure collision detection circuits.

The collision detection circuit gets the next robot triangle ( $T_r$ ) from the buffer, and gets the next environmental triangle ( $T_e$ ) from the memory. It computes the plane equations for both triangles in parallel. Next, it checks if all the vertices of the  $T_r$  lies in one side of  $T_e$  (all vertices are checked in parallel). If that is the case, then there is no collision, so the circuit moves to the next environmental triangle. Otherwise, the circuit checks if the planes are coplanar. If that is the case, then it performs a triangle-triangle collision test in 2D. Otherwise, finds the line  $L$  at the intersection of the planes containing  $T_r$  and  $T_e$ . The final collision detection test is then to find the intersection of  $L$  and each triangle (in parallel) and check if they overlap.

Collision detection between the robot and the environment continues until either all triangle pairs are compared or one triangle-triangle intersection test returns collision. In order to increase the parallelism of the system, further collision

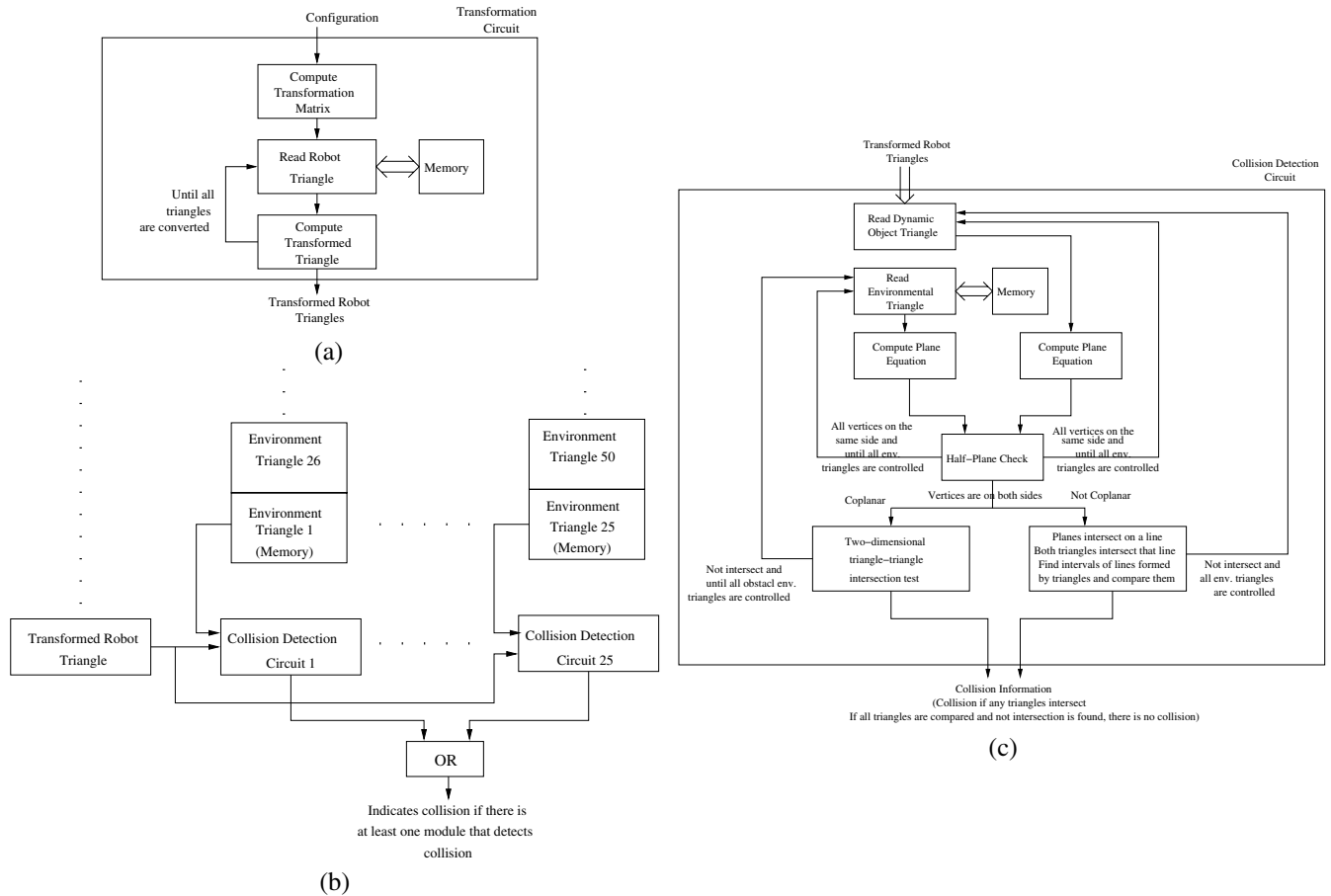


Fig. 6. Internal structure of collision detection module: (a) transformation circuit, (b) parallel collision detection circuits, (c) inside a collision detection circuit.

detection circuits can be added to the processor. The number of the total parallel collision detection circuits is only limited by the number of logic slices on the FPGA chips. When several collision detection circuits are employed in parallel, the transformed triangle  $T_r$  can be compared with several environmental triangles in parallel. A block diagram of this approach is given in Figure 6(b).

## VI. IMPLEMENTATION DETAILS

### A. I/O

In the current implementation, the host computer communicates with the motion planning processor using RS-232 serial port. This part is modular so it can be replaced by PCI interface for faster communication and easier installation into a computer. Input is used to get object representations as sets of triangles. Once the roadmap is formed, processor returns collision-free roadmap as a set of edges defined by endpoints of edges.

### B. Memory and Data Structures

The memory module of the collision detection processor is responsible from storing the object models, the configurations of the robot and results of the collision detection for each configuration. The objects are represented as triangular meshes. In

order to avoid costly floating point operations, we are using 32-bit fixed point arithmetic. Please note that this does not effect the performance of our processor since we normalize the coordinates before sending them to the processor. Each triangle is represented using 288 bits of data (each vertex is three 32-bit number, i.e., 96 bits, hence each triangle is  $96 \times 3$ ).

Instead of having one large memory, FPGAs usually have several small memories which are called Block RAMs. Each Block has data paths to CLBs. The advantage of using such a distributed memory is that several memory block can be accessed in parallel. However, the designer has to be careful to maintain data consistency when the data is distributed among the memory blocks. The amount of the data that can be transferred from memory module to the computational components at each clock cycle depends on the data width. In our implementation, each Block RAM has a data width of 36 bits. There are two data paths from each block, effectively doubling the data width to 72 bits. Hence we can get whole triangle data, including vertex points in 4 clock cycles. Instead of waiting 4 clock cycles, we distributed our triangle data to four Block RAMs resulting in one triangle read per clock cycle. Block RAMs are cascaded to obtain data width of 144 bits for one data path, and 288 for two paths.

One Block RAM has a capacity of 18K bits, so we can store

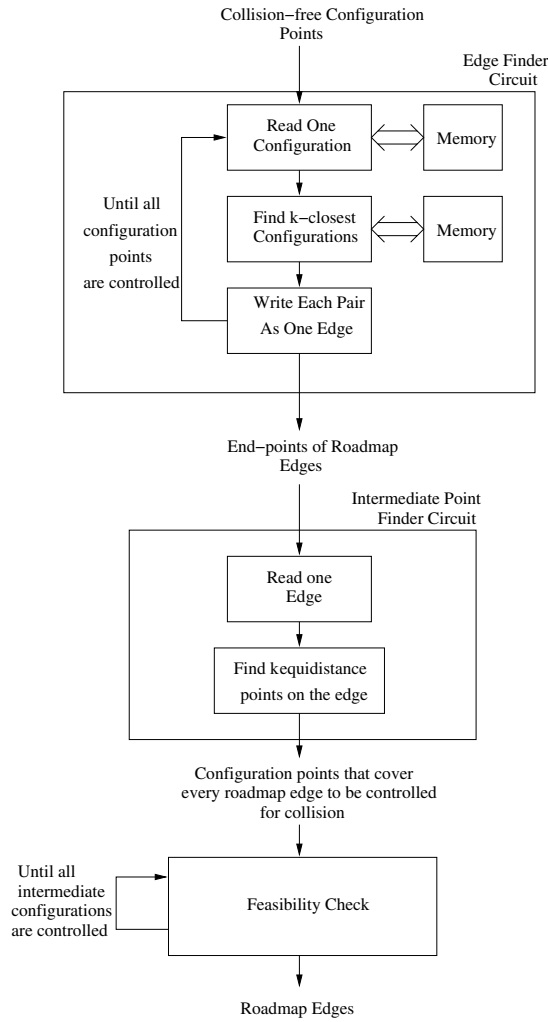


Fig. 5. Internal structure of finding closest configurations and local planning circuit.

up to 576 32-bit numbers, allowing 4 Block RAMs to store up to 256 triangles. Current FPGA chips can contain up to 336 Block RAMs which increases the total number of triangles to 86016.

### C. Design Issues.

We have used VHDL to develop our processor. Since trigonometric functions and multiplication are not directly supported in VHDL, we have used the Xilinx CoreGen tool to generate lookup tables (LUTs) that contain the results of trigonometric functions. We created a trigonometric module for sine and cosine functions which has symmetric output and uses distributed memory. When distributed memory is used for a LUT, the circuit has a latency of 2 clock cycles compared to 3 with Block RAM. Since speed is the most important issue in our problem, we preferred distributed memory. Input precision of 10 bits was specified, and the output precision was set to 32 bits. Similarly, the multiplications are also generated with CoreGen. It is a parallel signed multiplier with minimum pipelining and has a latency of 2 clock cycles. If maximum

pipelining were chosen, the latency would be 6. This circuit also uses LUTs constructed on distributed memory. Inputs are 32 bits wide and output is 64 bits wide which satisfies our 32-bit number representation. Division is a very costly operation in FPGA chip, which takes around 40 clock cycles. So, instead of division which is used in the last part of triangle-triangle intersection test, we used multiplication. Instead of dividing one part of an equation to a number, we multiplied other parts with that number. When we multiplied all equations to be compared, the result of comparison remained same, without the burden of division.

## VII. EXPERIMENTS

As mentioned before, our aim is to evaluate the feasibility of a motion planning processor and evaluate its performance. Specifically, we are interested in finding (i) if current FPGA technology can support our design, (ii) if there is a loss in accuracy due to fixed point arithmetic and trigonometric table lookups, (iii) how much the parallelism helps, (iv) how well our processor performs against a high end workstation.

In order to investigate these topics, we have designed four environments. Each environment is bounded by a  $240 \times 240 \times 240$  bounding box. Within this box, obstacles blocks of the size  $48 \times 48 \times 16$  are randomly placed. The robot is a block of size  $48 \times 24 \times 24$ . Each block is represented by 12 triangles. The environments are named  $Env_5, Env_{10}, Env_{15}$  and  $Env_{20}$  after the number of obstacles they contain (i.e., 5, 10, 15 and 20 obstacles). For each environment, obstacles are randomly placed. Figure 7 shows an example environment ( $Env_{15}$ ).

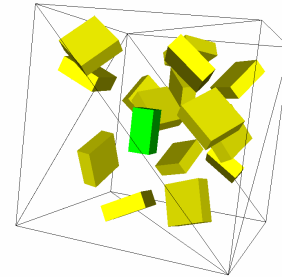


Fig. 7. Experimental environment with 15 obstacles. The robot is the long block at the center.

We have used our motion planning processor to generate roadmaps for each environment. Similarly we have used a workstation with Pentium-4 processor at 3 GHz with 1 GB memory to generate roadmaps for each environment using PRM. Both systems used the same parameters to generate the roadmaps. The workstation utilized one of the fastest collision detection algorithms, RAPID [24].

Using Xilinx ISE Foundation tool [25], we found out that our design can be loaded to Xilinx Virtex-4 XC4VLX200 chip. This chip allows us to create up to 25 collision detection circuits in parallel and can run our design at the clock rate of 50 MHz. Hence, we verified that current FPGA technology can support our processor.

Next, we have used ModelSim SE 6.1b by Mentor Graphics [26] to simulate our chip. ModelSim is an HDL (Hardware Description Language) high-end simulator. It takes the VHDL description of the design as well as input values, then compiles the design, and runs it in the defined clock frequency. Outputs and internal signals of circuit can be examined while simulation is running. Simulation runs circuit in real-time so timing is accurate, i.e. same results are obtained when design is loaded to FPGA chip. We have validated this by configuring our motion planning processor for collision detection and placing it on less capable Xilinx Virtex-4 XC4VLX25, and comparing the simulation times of collision detection to the real times on Virtex-4 XC4VLX25. Unfortunately, the slice size of Virtex-4 XC4VLX25 is not sufficient to run our motion planning processor, hence we run our experiments on the simulation.

In order to have a realistic comparison of our processor and high end workstation, both systems should use the same configurations to build the roadmap. For this purpose, we have selected 400 initial random configurations (300 colliding and 100 free) for each environment. Both MPP and the workstation started with these initial configurations and built the roadmaps.

In order to observe the effects of parallelism, we have configured MPP with single collision detection, 10 parallel collision detection and 25 parallel collision detection circuits. Figure 8(a) shows results for  $Env_5$  and  $Env_{20}$ . In this figure, number of parallel collision detection circuits are shown in the x-axis and the roadmap building time is shown in y-axis. This figure shows that the speed-up from single collision check to 10 parallel collision detection circuits are significant. But when we further increase the number of parallel collision detection circuits to 25, the speed-up with respect to 10 parallel collision detection circuits reduces to around 2.5. We believe as the parallelism increases, the speed-up will converge to the ratio between the number of collision detection circuits.

Next we have compared the roadmap building time of MPP to the Pentium-4 (P4) machine. Figure 8(b) shows the performance of MPP and P4 in four environments. Note that while a general purpose computer is significantly better than MPP with single collision detection circuit, MPP performs better as the number of collision detection circuits increase. In fact, in  $Env_5$  with 25 parallel collision detection, MPP performed 25 times faster than P4. Similarly in  $Env_{20}$  with 25 parallel collision detection, MPP performed 8 times faster than P4. This results show that MPP performs better than P4 in our experimental environments.

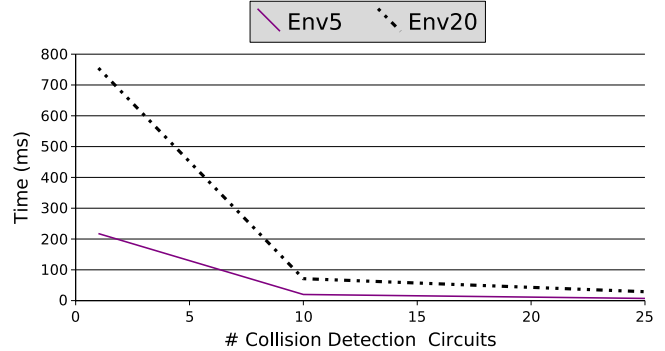
In order to validate the accuracy of MPP built roadmap, we compared the roadmaps generated by MPP and P4. We found out that both of them generated the same roadmaps. Hence, even though MPP used 32-bit fixed point arithmetic and table lookup values for trigonometry functions, there was no significant loss in accuracy.

Finally, we compared the performance of MPP and P4 in node generation. We have randomly generated 100 collision free configurations (i.e., if a random configuration is in collision, it is rejected a new configuration is generated). Figure 9 summarizes our results with 25 parallel collision detection

circuits in four different environments. As expected, MPP with 25 collision detection circuits performed better than the P4 chip.

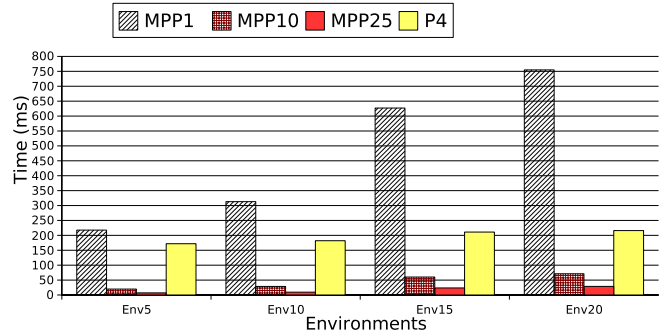
Our results are very promising and shows that it is not only feasible to design motion planning chips on reconfigurable hardware but also it is possible that such chips would run faster than current workstations.

Roadmap Building Time vs. # Parallel Collision Circuits



(a)

Roadmap Building times MPP vs. P4



(b)

Fig. 8. (a) Roadmap building times in four environments: (a) running times with respect to number of collision detection units, (b) roadmap building times for Motion Planning Processor (MPP) and Pentium 4 workstation. The MPP times are for single (MPP1), 10 (MPP10) and 25 (MPP25) parallel collision detections.

## VIII. CONCLUSION

In this paper we showed that it is feasible to design dedicated motion planning processors. Our design takes the advantage of inherited parallelism of motion planning and collision detection algorithms and can build a roadmap up to 25 times faster than a Pentium-4, 3Ghz CPU. The reconfigurable and modular nature of our motion planning processor also enables us to use our motion planning processor as dedicated collision detection chip. Our current chip uses fast triangle-triangle intersection test to check collision. Our future work includes implementing more advanced collision detection algorithms on the FPGA.

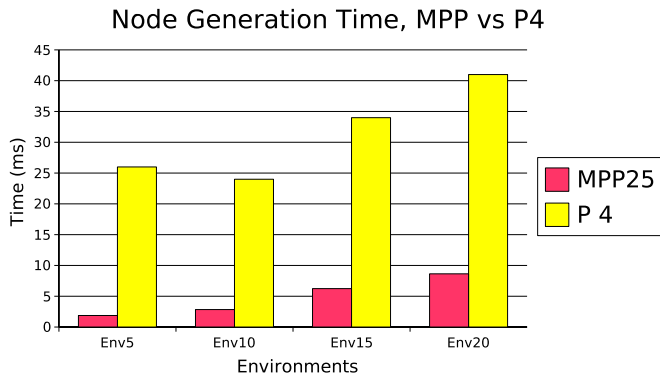


Fig. 9. Time to generate 100 collision free nodes in four environments, Motion Planning Chip (MPP25) with 25 parallel collision detection circuits vs. Pentium 4 Workstation.

## IX. ACKNOWLEDGEMENT

In our high-end workstation experiments, we have used a motion planning library developed by Parasol Laboratory at Texas A&M University. We would like to thank them for their generous help.

## REFERENCES

- [1] L. Kavraki, P. Svestka, J. C. Latombe, and M. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Trans. Robot. Automat.*, vol. 12, no. 4, pp. 566–580, August 1996.
- [2] N. M. Amato, K. A. Dill, and G. Song, "Using motion planning to map protein folding landscapes and analyze folding kinetics of known native structures," in *Proc. Int. Conf. Comput. Molecular Biology (RECOMB)*, 2002, pp. 2–11.
- [3] J. J. Kuffner and S. M. LaValle, "RRT-Connect: An Efficient Approach to Single-Query Path Planning," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, 2000, pp. 995–1001.
- [4] P. Bessiere, J. M. Ahuactzin, E.-G. Talbi, and E. Mazer, "The ariadne's clew algorithm: Global planning with local methods," in *Proc. IEEE Int. Conf. Intel. Rob. Syst. (IROS)*, vol. 2, 1993, pp. 1373–1380.
- [5] N. M. Amato, O. B. Bayazit, L. K. Dale, C. V. Jones, and D. Vallejo, "OBPRM: An obstacle-based PRM for 3D workspaces," in *Robotics: The Algorithmic Perspective*. Natick, MA: A.K. Peters, 1998, pp. 155–168, proceedings of the Third Workshop on the Algorithmic Foundations of Robotics (WAFR), Houston, TX, 1998.
- [6] D. E. Kodischek, "Robot planning and control via potential functions," in *The Robotics Review 1*, O. Khatib, J. J. Craig, and T. Lozano-Pérez, Eds. The MIT Press, 1989.
- [7] R. Gayle, W. Segars, M. Lin, and D. Manocha, "Path planning for deformable robots in complex environments," in *Robotics: Systems and Science*, 2005.
- [8] K. Underwood, "FPGAs vs. CPUs: trends in peak floating-point performance," in *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*. New York, NY, USA: ACM Press, 2004, pp. 171–180.
- [9] N. M. Amato and L. K. Dale, "Probabilistic roadmap methods are embarrassingly parallel," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, 1999, pp. 688–694.
- [10] G. Baciú and S. K. Wong, "Image-based techniques in a hybrid collision detector," in *IEEE Trans. on Visualization and Computer Graphics*, 2002.
- [11] N. K. Govindaraju, M. C. Lin, and D. Manocha, "Fast and reliable collision culling using graphics hardware," in *Virtual Reality Software and Technology (VRST)*, 2004.
- [12] N. K. Govindaraju, S. Redon, M. C. Lin, and D. Manocha, "Cullide: Interactive collision detection between complex models in large environments using graphics hardware," in *Graphics Hardware*, 2003.
- [13] A. Gress and G. Zachman, "Object-space interference detection on programmable graphics hardware," in *In SIAM Conf. on Geometric Design and Computing*, 2003.
- [14] B. Heidelberger, M. Teschner, and M. Gross, "Real-time volumetric intersections of deforming objects," in *Proc. of Vision, Modeling and Visualization*, 2003.
- [15] K. Hoff, A. Zaferakis, M. Lin, and D. Manocha, "Fast and simple 2d geometric proximity queries using graphics hardware," in *Proc. of ACM Symposium on Interactive 3D Graphics*, 2001, pp. 145–148.
- [16] P. M. Hubbard, "Collision detection for interactive graphics applications," in *IEEE Transactions on Visualization and Computer Graphics*, vol. 1, no. 3, 1995, pp. 218–230.
- [17] Y. J. Kim, M. A. Otaduy, M. C. Lin, and D. Manocha, "Fast penetration depth computation for physically-based animation," in *Proc. of ACM Symposium on Computer Animation*, 2002.
- [18] D. Knott and D. K. Pai, "Cinder: Collision and interference detection in real-time using graphics hardware," in *Proc. of Graphics Interface*, 2003.
- [19] K. Myszkowski, O. G. Okunev, and T. L. Kunii, "Fast collision detection between complex solids using rasterizing graphics hardware," in *The Visual Computer*, vol. 11, no. 9, 1995, pp. 497–512.
- [20] J. Rossignac, A. Megahed, and B. D. Schneider, "Interactive inspection of solids: cross-sections and interferences," in *Proceedings of ACM SIGGRAPH*, 1992.
- [21] G. Zachman and G. Knittel, "An architecture for hierarchical collision detection," in *The 11th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision '2003*, 2003, pp. 149–156.
- [22] H. Krupnova and G. Saucier, "FPGA technology snapshot: Current devices and design tools," in *11th IEEE International Workshop on Rapid System Prototyping (RSP 2000)*, 2000, pp. 200–2005.
- [23] T. Moller, "A fast triangle-triangle intersection test," *J. Graph. Tools*, vol. 2, no. 2, pp. 25–30, 1997.
- [24] "Rapid – robust and accurate polygon interface detection homepage," <http://www.cs.unc.edu/geom/OBB/OBBT.html>.
- [25] "Xilinx," <http://www.xilinx.com/>.
- [26] "Modelsim," <http://www.model.com/>.