USING FPGAs TO SOLVE THE HAMILTONIAN CYCLE PROBLEM

M. Serra
Dept. of Computer Science
Univ. of Victoria
Victoria, B.C., Canada

K. Kent
Dept. of Computer Science
Univ. of New Brunswick
Frederickton, N.B., Canada

ABSTRACT

The Hamiltonian Cycle (HC) problem is an important graph problem with many applications. The general backtracking algorithm normally used for random graphs often takes far too long in software. With the development of field-programmable gate arrays (FPGAs), FPGA-based reconfigurable computing offers promising choices for acceleration. This research exploits the idea of an instance-specific approach and proposes a system design based on a reconfigurable hardware implementation for solving the HC problems. In our implementation, only one FPGA is used, on an internal PCI-based board. The experimental results show that the reconfigurable hardware approach yields significant runtime speedups over the conventional approach, although the clock rate of the FPGA hardware is much slower than that of the workstation running the software solver.

1. INTRODUCTION AND CONTEXT

The focus of this research is to try and obtain an execution speed improvement for solving the Hamiltonian Cycle (HC) problem using Field Programming Gate Arrays (FPGAs) instead of using software alone, applied to general random graphs without any (helpful) constraints. The HC problem is a NP-Complete problem which involves detecting whether a graph has a cycle that contains all the vertices of the graph exactly once.

With the development of FPGAs, FPGA-based reconfigurable computing offers promising solutions for many applications, and many reconfigurable systems have been developed [1]. Most of them show the potential of high performance which is an order of magnitude higher than conventional approaches. Zhong et al. [3] described an instance-specific method for implementing the Boolean Satisfiability (SAT) solver circuits in the reconfigurable hardware, that is, using a template generator, and creating a circuit specific to the problem instance to be solved. Their experimental results show significant runtime speedups, although the clock rate of the reconfigurable hardware is 100 times slower than that of the workstation running the software solver. However, they used over 64 FPGAs and a parallel approach.

We propose an instance-specific HC solver accelerator which implements the backtracking algorithm to realize the HC solver circuits in one FPGA. In this approach, for each graph instance, the corresponding HC solver circuit is generated and implemented in the FPGA. We only use one FPGA and a regular microprocessor with a PCI bus.

1.2 FPGAs

A Field Programmable Gate Array, FPGA, consists of a two-dimensional array of uncommitted computational elements, Computational Logic Blocks (CLBs), with general interconnection resources. Both are user-programmable. We use the FPGAs from Xilinx and their CAD tools for design and synthesis. The major steps in the tools include:

- *Initial Design Entry.* Design the circuit using VHDL.
- *Logic Synthesis*. The final optimized logic network is decomposed, such that it can use the set of available CLBs.
- *Placement and Routing*. This step involves selection of CLBs and routing of the required connections among them.
- *Configuration.* The "bit file" obtained is the input to perform the physical "personalization" on the FPGA, and can be downloaded when commanded to initialize for execution.

1.2 Hamiltonian Cycles

A *graph* G = (V, E) consists of a set of *vertices* V = {v_1, v_2, …}, and a set of *edges* E = {e_1, e_2, …}, such that each edge e_k is identified with an unordered pair (v_i, v_j) of vertices. A *Hamiltonian Cycle* in a connected graph is defined as a closed walk that traverses every vertex of G exactly once, except the starting vertex, at which the walk also terminates. Given a graph, the main issues are to detect whether or not it contains a Hamiltonian Cycle, or to find all the Hamiltonian Cycles if any. Here we are only concerned about the former. From a computational point of view, it is a NP-Complete problem. We use a very general backtracking algorithm which makes no assumption about any graph characteristic, except that it is connected.

1.3 Reconfigurable Computing

One of the most fundamental trade-offs in the design of computing devices involves the balance between flexibility and efficiency. At one end of the spectrum are application specific integrated circuits (ASICs); at the other end of the spectrum are programmable processors. Reconfigurable computing can be viewed as hybrid, as it intends to achieve potentially much higher performance than processors, while maintaining a higher level of flexibility than ASICs. FPGAs are the main building blocks for a reconfigurable architecture. Here we use an FPGA for hardware acceleration of a graph problem, hoping to find a way to speed up the running time, using an approach which can in the future be used to other classes of computationally intense problems.

In general, a reconfigurable computing system contains both a microprocessor and the reconfigurable hardware. The processor executes the tasks that can be performed better in a general-purpose device due to their irregularity, or their data types, or their need for external communication; while the reconfigurable hardware uses FPGAs to implement custom algorithm-specific circuits for the tasks that can exploit their parallel nature, so that they can be performed at greater speed. Yet, unlike an ASICs approach, these systems remain flexible because the same custom circuitry for one task can be reused for a completely different task. The reconfigurable systems, therefore, combine the best of both general-purpose and custom-made solutions: flexibility and low cost in the first case, and speed in the latter. We use Compile-Time Reconfiguration, where the device is reconfigured once during the set-up of the system, and remains unchanged during the execution, until.it is reconfigured for a new application.

1.4 Instance-Specific Approach

In this research, we did not use the hardware/software codesign approach, where a problem is appropriately partitioned between software units, running on a general CPU, and hardware units, running on a specialized hardware. Instead, we move the whole algorithm to the reconfigurable hardware using an "instance-specific" approach.

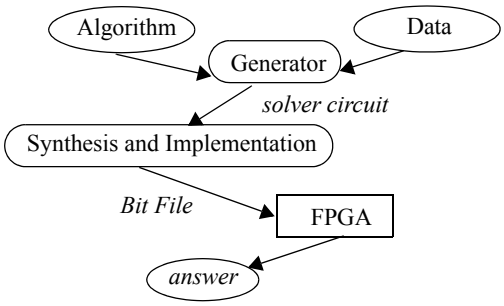

Figure 2. The instance-specific approach

In a general-purpose approach, the CPU reads instructions from a program loaded in memory. When changing the software instructions, the functionality of the system alters without changing the hardware. Once an algorithm is loaded into memory, it remains unchanged. and only the data changes depending on what kind of graph instance we want to solve. Conversely, in the instance-specific approach, the FPGA is tailored to an instance which consists of a specific algorithm and specific data, e.g. the backtracking algorithm and a graph instance in this project. A generator reads the algorithm and a graph instance, so that the solver circuit specific to the current graph instance can be generated. After synthesis and implementation, the bit file which can be downloaded onto the FPGA is generated. Then we reconfigure the FPGA to realize the solver circuit in hardware, and start the computation in the newly configured hardware.

In the instance-specific approach, the solver circuit must be generated for each graph instance, and we consider all the steps, including the solver circuit generation, synthesis and implementation, reconfiguration, and the FPGA execution as part of the total hardware running time. However, this is not necessary the case for other research [2, 3] where they gained significant hardware improvement by comparing only the FPGA execution time (not including time used for synthesizing and implementation) to the software running time.

2. THE ARCHITECTURE

The H.O.T Development System (DS) is a product from VCC (Virtual Computer Corp.). It is a standard commercial platform for the development and rapid prototypes of products using reconfigurable components. The H.O.T II PCI board resides in a regular Wintel workstation and communicates via a PCI bus. The board has the following components:

- one XC4062XLA FPGA with 2304 CLBs;
- Configuration Cache Manager (CCM);
- Two memory banks A and B with 4 MB of SRAM in total;
- A Configuration Flash and a Configuration Cache;

The FPGA communicates with the PCI bus and a VCC customized backend design which is inside the H.O.T. II Interface, through the Xilinx LogiCORE PCI Interface Macro (inside the H.O.T. II Interface). The customized backend design allows a user to communicate with the Configuration Cache Manager (CCM) and the two banks of memory. The CCM controls the Run-Time Reconfiguration (RTR), which allows a user to download or reload the design into the FPGA at execution time. The CCM can configure the FPGA from two on-board sources: the Configuration Flash and the Configuration Cache. The Configuration Flash contains the boot-up configuration for the system, which includes the configuration of the Xilinx LogiCore PCI Interface and the H.O.T II Interface. The Configuration Cache can hold three configurations and the user can load them into the Cache over the PCI Bus. The programmable clock can be programmed from 360KHz to 100MHz.

3. THE EXPERIMENTAL SETUP

The purpose of this project is to try and achieve hardware acceleration to find whether a graph has a Hamiltonian Cycle using an FPGA, instead of using software alone. The overall experiments involve two main parts: timing the software and timing the FPGA-based solver, in order to compare them. The experiments start with the generation of the adjacency matrix of a graph. The adjacency matrix is then fed into the hardware solver and the software solver separately, so that their running time can be compared.

Based on the definition of the Hamiltonian Cycle problem, we know that if a graph G contains any Hamiltonian Cycle(s), the degree of each vertex must be ≥ 2. Thus we decided to choose the graphs to experiment on in this project to be sparse ones, i.e. they have a relatively small number of edges and the degree of each vertex is ≥ 2. This

implies that we only attempt to solve the problem for "hard" instances.

Two types of graphs are chosen to experiment on: 1) Random graphs; 2) Complete graphs. Random graphs are chosen with the purpose of examining the acceleration of the hardware solver in general and finding out the CLB usage of them; complete graphs are used to find out the CLB usage per graph and per node. The random graphs are generated by a C program whose output is an adjacency matrix and which has two parameters: *graphSize* and *edgeProbability*. The former represents the total number of vertices and the latter controls the adjacency between any two different vertices by calling on *drand48*(), a standard C function which generates double-precision pseudo-random values that are uniformly distributed over the interval [0.0, 1.0] A complete graph is a simple graph in which there exists an edge between every pair of vertices and can be generated using the random graph software by specifying an edge probability of '1.

The Software Solver is a C program which implements the backtracking algorithm. The hardware solver contains 5 phases: (1) Generation of the VHDL description of a graph instance; (2) Generation of the Xilinx Foundation Project; (3) Loading the VHDL files into the Xilinx Foundation tool; (4) Synthesis; (5) FPGA configuration and solver circuit Execution. The first phase generates the VHDL files of the Hamiltonian Cycle Solver Circuit for a given graph G, which is done automatically by two Perl programs, which take as input an adjacency matrix and produce the first step towards the design of a netlist for a graph instance and its algorithm.

The solver circuit that is implemented on the FPGA consists of a network of intercommunicating finite state machines (FSMs), each corresponding to one vertex in G. At any point during the computation, exactly one FSM is active in the process of changing its state and output values; this FSM is said to be in control. The FSM corresponding to vertex *v* successively passes control to each of the FSMs corresponding to the vertices in the neighborhood of *v*. When a FSM has passed control to each of these neighboring machines, and none of them continued the computation, backtracking occurs. An FSM backtracks by entering the *idle* state and signaling back to the FSM from which it received control. At this point the latter machine passes control to another neighbor (or backtracks itself) and another branch of the computation tree is explored.

Figure 3 illustrates the structure of an arbitrary FSM, called R. The vertex corresponding to R has the following neighbors: NB_1, NB_2, …, NB_{d-1}, and NB_d. As shown in Figure 3, R has $d + 2$ states in total, where d states correspond to its d neighbors, and the other two states are *Idle* and *Stuck*. State *Idle* means R has not been activated yet, state *stuck* means all neighbors of R are activated. Once activated, R enters the next state along the state cycle such that the corresponding FSM is *idle*, and passes control to it, e.g. R is activated when it is at *Idle* state and the neighbor NB_1 has not been activated yet; then R passes control to the FSM that corresponds to NB_1. If no such state exists, state *stuck* is entered, which causes backtracking, i.e. at the following clock cycle, R returns to state *idle*.

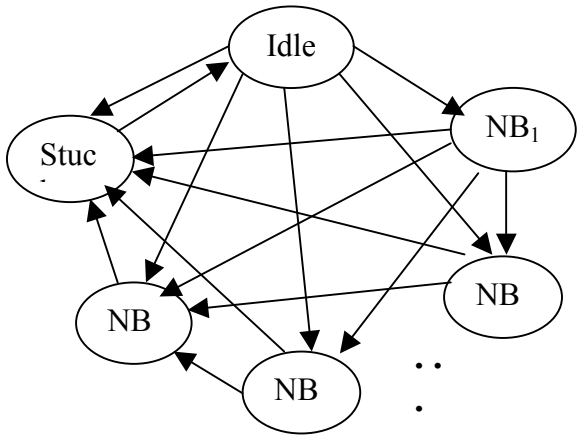

Figure 3. The structure of an arbitrary FSM

The computation terminates with *noHamiltonian* when the arbitrarily chosen initial FSM, i.e. the machine which was first given control, backtracks; or it terminates with *isHamiltonian* when all FSMs are in non-idle states, and the control machine represents a vertex that is adjacent to the initial vertex.

The Perl programs use the adjacency matrix as their input and follow the FSM paradigm just described to produce a set of VHDL files which describe the entire Hamiltonian Cycle Solver Circuit for the given graph. Thus the VHDL files represent a complete circuit for this instance. We then need to generate the VHDL code for the H.O.T. II interface, so that we can specify the connection between the user design, e.g. the Hamiltonian Cycle circuit, and the H.O.T. II interface. This has to be done for every graph instance. The files are now ready to be loaded into the Xilinx Foundation Tool, where they are compiled into an XNF (Xilinx Netlist Format) netlist of gates. The netlist is the input to the Xilinx Implementation Tool, which proceeds through its steps of translation, mapping, place & route, timing, configuration, execution. We emphasize again that all hardware timing results contain these configuration and synthesis phases.

4. EXPERIMENTAL RESULTS

The running times of the software solver and the hardware solver are compared in order to find out whether the FPGA-based solver wins or not. We present here a set of random graphs, because the software solver always wins for the complete graphs. The graph generation mechanism discards graphs that are trivially non_Hamiltonian, which refers to graphs with minimum degree less than 2 or those that are disconnected.

Table 1 lists only a subset of the random graphs used for this experiment, for lack of space here. The results for all graphs are available from the authors. The graphs shown here are the ones with 35 nodes (n = 35) and edge probability (p) ranging from 0.12 to 0.15. For each of these (n, p)

pair, 5 random graphs are generated, for example, v35p012.1 represents one of the graphs with 35 nodes and 0.12 edge probability. All times are given in seconds. The second column shows if the graph has a Hamiltonian Cycle or not: "1" for yes, "2" for no. The "Map time" in column 4 is the sum of the synthesis time and the implementation time. For the execution, the maximum clock rate over all circuits was 40MHz, so a very conservative clock rate of 16MHz was used for the computed average of the actual FPGA execution time, shown in column 5. The total hardware time needed for the FPGA-based solver is given in column 6. Thus we are comparing the software running time to the total hardware time which includes the mapping time, the FPGA reconfiguration time and the FPGA execution time. Column 7 shows when the FPGA-based solver wins as a ratio. The shaded rows indicate when the FPGA-based solver wins. There we see that the FPGA-based solver wins for most of the graphs with speedup ranging from 2 to 30. However for some of the graphs, e.g. the graphs v35p012.3 and v35p012.4, their FPGA execution time is much less than the software running time, but no acceleration is obtained for them because of the map-time overhead. Note that in some cases the software needed much more time for completion, but it was stopped executing once the hardware had finished.

We also computed the overall map time for random graphs, and found that the average map time for the biggest random graphs that fit into our small target FPGA is less than 1367 seconds, certainly a significant portion of the total hardware running time. However, the synthesis and implementation are done using the general CPU resources in the Xilinx Foundation tool, while the bit file is downloaded and executed in the target FPGA; thus the two processes can be done in parallel. Some map time can be saved by performing the synthesis and the implementation while the FPGA runs for another graph instance.

From the data above and the rest of the data collected, we can conclude:

- The software solver is more suitable for sparse random graphs with total number of vertices <= 20, because of the overhead mapping time for them in the hardware implementation.
- The FPGA-based solver has a better winning chance for sparse random graphs with total number of vertices >= 35, because the software usually takes far too long to solve those graph problems, and the mapping time for them becomes a small portion of the total hardware running time.
- Since the synthesis and the implementation for a graph instance can be done in parallel, using the CPU resource, with the execution in FPGA for another graph instance, we could only consider the FPGA execution time, and win for almost all graph instances.
- The CLB usage we plotted expands linearly with the size of the graphs.

5. SUMMARY

The Hamiltonian Cycle (HC) problem is not only a important graph problem, but also being widely used in applications such as telecommunication and computer networks. We proposed a new approach, namely, using an FPGA to solve the Hamiltonian Cycle problem. The system requirements for this new approach is low: only one FPGA is used. The experimental results show that the FPGA-based solver speeds up the runtime for sparse random graphs with total number of vertices >35, and has a winning potential over the conventional approach for the larger sparse random graphs. There is potential for further improvement as the capacity and performance of FPGA's improve. The HC-solver is a case study of a class of instance-specific reconfigurable hardware applications, which can be used to other classes of computationally intense problems.

1	2	3	4	5	6	7
graph		SW time	Map time	FPGA time	HW time	Speed up
v35p012.3	2	196	790	5.5	796	0.25
v35p012.4	1	315	898	9.2	907	0.35
v35p012.1	2	8516	920	245	1165	7.31
v35p012.5	2	22606	709	654	1363	16.59
v35p012.2	2	65298	1325	1866	3191	20.46
v35p013.3	2	563	813	16	829	0.68
v35p013.4	2	16782	972	482	1454	11.55
v35p013.5	2	220551	1117	6521	7638	28.88
v35p013.1	2	227186	1086	6601	7687	29.55
v35p013.2	2	303765	1137	8724	9861	30.80
v35p014.3	2	>691200	1217	360304	361621	>>
v35p014.1	2	>2246400	1302	637080	638382	>>
v35p014.2	2	>691200	1213	149401	150614	>>
v35p014.5	2	28290	1035	798	1833	15.43
v35p014.4	2	91334	968	2679	3647	25.05
v35p015.3	1	0.37	1018	0.04	1018	0.00
v35p015.5	1	289	1181	9	1190	0.24
v35p015.2	2	501	896	14	910	0.55
v35p015.4	1	2480	1170	73	1243	2.00

Table 1: Graphs with 35 nodes

6. REFERENCES

[1] List of FPGA-based Computing Machines. URL: http://www.io.com/~guccione/HW_list.html

[2] C. Plessl and M. Platzner, "Instance-Specific Accelerators for Minimum Covering", 1st International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA), pp. 85-91, June 200

[3] P. M. Zhong, Martonosi, P. Ashar, and S. Malik, "Using Configurable Computing to Accelerate Boolean Satisfiability", IEEE Transactions on CAD of Integrated Circuits and Systems, vol. 18, No. 6, June 1999.