Implementation of Recursive Search Algorithms in Reconfigurable Hardware

Iouliia Skliarova

University of Aveiro, Department of Electronics and Telecommunications, IEETA
3810-193 Aveiro, Portugal
iouliia@det.ua.pt

Abstract. By adapting to computations that are not so well supported by general-purpose processors, reconfigurable systems achieve significant increases in performance. A great deal of research effort in this area is aimed at accelerating the solution of combinatorial optimization problems. However, hardware description languages (such as VHDL) as well as system-level specification languages (such as Handel-C) that are usually employed for specifying the required functionality of reconfigurable systems do not provide a direct support for recursion. In this paper a method allowing recursive algorithms to be easily described in Handel-C and implemented in an FPGA (field-programmable gate array) is proposed. The recursive search algorithm for the knapsack problem is considered as an example. The required hardware support is provided by a recursive hierarchical finite state machine model.

1 Introduction

Although the concept of reconfigurable computing has been known since the early 1960s [1], it is only recently that technologies that allow it to be put into practice became available. The interest started at the beginning of the 1990s as FPGA (field-programmable gate array) densities broke the 10K logic gate barrier. Since then, reconfigurable computing became a subject of intensive research. For some classes of applications reconfigurable systems allow very good performance to be achieved compared to general-purpose computers. Other types of applications were mapped to reconfigurable hardware because it offers innovative opportunities to explore. According to the primary objective to be achieved, all these applications can be broadly divided into three categories: hardware emulation and rapid prototyping, evolvable hardware, and the acceleration of computationally intensive tasks.

The first category is hardware emulation and rapid prototyping [2]. It is quite difficult to know if a designed circuit is correct before its physical implementation. Traditionally, two basic approaches are employed for verification of complex projects: prototyping and software simulation. However, constructing a dedicated prototype is quite a costly and time-consuming task. Software simulation techniques are flexible but very slow. In such circumstances, the emulation of hardware circuits on field-programmable logic devices is considered to be a good alternative. Emulation is faster than software simulation and allows a design with large quantities of real-world data to be verified. Emulation is also more reliable than software in reflecting the characteristics of the final implementation, permitting performance to be evaluated more accurately. Recent commercially available FPGAs, such as Virtex-II Pro from Xilinx [3] and Stratix from Altera [4], incorporate large amounts of logic, arithmetic units (multipliers), embedded memory blocks, processor cores, etc., and are thus becoming an adequate platform for emulating complex systems.

The second category is evolvable hardware which comprises a variety of approaches to the design of electronic circuits using evolutionary techniques [5]. These approaches can be divided into two major groups. The first group covers direct evolution in existing components (usually in an FPGA), while the second includes trying to develop the desired functionality off-line and then implementing it in hardware. For such applications, the property of reconfiguration is essential.

The potential of evolvable hardware is quite extensive because theoretically it allows the construction of a circuit with a given specification, whose structure is previously unknown. This was shown in a number of papers presenting the results of evolution of circuits that are more efficient than any known conventional design [5]. Another interesting application of these techniques is auto-repair of autonomous systems, where repair by human experts is for some reason impossible (for example, systems used in space applications).

It should be noted that up to now, no sufficiently complex system has been constructed with the capabilities of rapid and efficient auto-optimization [6]. Nevertheless, evolvable hardware is an extremely promising and rapidly developing research direction, which is attracting more and more attention (as indicated by the increasing number of conferences in this area).

Finally, the last category, which is without doubt the prevalent one, is the acceleration of computationally intensive tasks. A common characteristic of these tasks is that they are very well suited to parallel implementations that take advantage of the basic capabilities of reconfigurable computing. Cryptography, signal and image processing are good examples in this category. These applications are characterized by large amounts of data to process and by inherent parallelism, and are suitable for pipelining. All these factors contribute to an increase in the performance of a reconfigurable system compared to a similar implementation on general-purpose computers.

Recently, a series of attempts have been made to accelerate applications that involve rather complex control flow. In this context special attention was given to problems in the area of combinatorial optimization. The most popular problems that have been analyzed are the Boolean satisfiability problem [7], the traveling salesman problem [8], and the covering problem [9]. Providing for solution of these problems with the aid of reconfigurable hardware is considered to be very advantageous because it allows the benefits, such as flexibility and speed, of both ASICs (application-specific integrated circuits) and general-purpose computers to be combined, and their weaknesses to be eliminated. Of course, in similar technology, FPGAs suffer a speed penalty of at least one order of magnitude compared to ASICs. However, FPGAs possess the flexibility and low development cost of software implementations. To enable an implementation based on reconfigurable hardware to outperform the equivalent software implementation, the following techniques are usually employed. First, primary functional units are constructed in such a way that they are optimized for particular operations, thus requiring fewer clock cycles. Second, the techniques of parallel processing and pipelining are employed. And finally, the memory organization is tailored to specific data sizes, thus speeding up data transfer.

2 Algorithms

The problems of combinatorial optimization arise in many application areas such as logic design, technical diagnostics, artificial intelligence, etc. Many of these problems are NP-hard [10], which means that in general, the execution time for a solution grows exponentially with the size of a problem instance. Of course, with the aid of reconfigurable hardware, we cannot

cancel out this effect of exponential growth, but we are able to delay it by enabling the primary operations of the respective algorithms to be executed more efficiently.

The exact algorithms that are employed in the area of combinatorial optimization are usually based on the generation and exhaustive examination of all possible solutions until a solution with a desired quality is found. The primary decision to be taken in this approach is how to generate the candidate solutions effectively. A widely accepted answer to this question consists of constructing a search tree [11], which enables all possible solutions to be generated in a well-structured and efficient way. The root of the tree is considered to be the starting point that corresponds to the initial situation. The other nodes represent various situations that can be reached during the search for results. The arcs of the tree specify steps of the algorithm that have been performed. In this case, recursive formulation of the algorithms is particularly appropriate.

Recently, commercial tools that allow reconfigurable digital circuits to be synthesized from *system-level specification languages* (SLSLs) such as Handel-C [12] and SystemC [13] have appeared on the market. In this area, C and C++ with application-specific class libraries and with the addition of inherent parallelism are emerging as the dominant languages in which system descriptions are provided. This fact allows the designer to work at a very high level of abstraction, virtually without worrying about how the underlying computations are executed. Consequently, even computer engineers with a limited knowledge of the targeted FPGA architecture are capable of producing rapidly functional, algorithmically optimized designs.

Although SLSLs are very similar to conventional programming languages there are a number of differences. In this paper we explore one of such differences, namely in the way in which recursive functions can be called. We consider Handel-C [12] as a language of study and the knapsack combinatorial problem [11] as an example.

It is known that functions in Handel-C may not be called recursively [12]. This can be explained by the fact that all logic needs to be expanded at compile time to generate hardware. Generally, recursion should be avoided when there is an obvious solution by iteration. There are, however, many good applications of recursion, as section 3 will demonstrate. This fact justifies a need of implementation of recursive functions on essentially non-recursive hardware. This involves the explicit handling of a recursion stack, which often obscures the essence of a program to such an extent that it becomes more difficult to comprehend. In the subsequent sections we will demonstrate an easy way of handling the recursion stack in Handel-C.

3 The Knapsack Problem

We have selected the knapsack problem for our experiments. There are numerous versions of the knapsack problem as well as solution methods. We will consider a 0-1 problem and a branch-and-bound method. A 0-1 problem is a special instance of the bounded knapsack problem. In this case there exist n objects, each with a weight $w_i \in Z^+$ and a volume $v_i \in Z^+$, i=0,…,n-1. The objective is to determine which objects should be placed in the knapsack so as to maximize the total weight of the knapsack without exceeding its total volume V. In other words, we have to find a binary vector $\mathbf{x} = [x_0, x_1, \ldots, x_{n-1}]$ that maximizes the objective function $\sum_{i=0}^{n-1} w_i x_i$ while satisfying the constraint $\sum_{i=0}^{n-1} v_i x_i \le V$.

A simple approach to solve the 0-1 knapsack problem is to consider in turn all 2^n possible solutions, calculating each time their volume and keeping track of both the largest weight

found and the corresponding vector **x**. Since each x_i, i=0,…,n-1, can be either 0 or 1, all possible solutions can be generated by a backtracking algorithm traversing a binary search tree in a depth-first fashion. In the search tree, a level i corresponds to the variable x_i and the leaves represent all possible solutions. This exhaustive search algorithm has an exponential complexity $\Theta(n2^n)$ (because the algorithm generates 2^n binary vectors and takes time $\Theta(n)$ to check each solution) making it unacceptable for practical applications. The average case complexity of the algorithm may be improved by pruning the branches that lead to non-feasible solutions. This can easily be done by calculating the current volume at each node of the search tree, which will include the volumes of the objects selected so far. If the current volume at some node exceeds the capacity constraint V, the respective branch does not need to be explored further and can safely be pruned away since it will lead to non-feasible solutions.

The pseudo-code of the employed algorithm is presented in Fig. 1. A simple backtracking algorithm involves two recursive calls responsible for exploring both the left and the right sub-trees of each node. Since one of the recursive calls is the last statement in the algorithm it can be eliminated as illustrated in Fig. 1.

```
x = 0; //current solution
opt_x = 0; //optimal solution found so far
opt_W = 0; //weight of the optimal solution
cur_V = 0; //volume of the current solution
level = 0; //level in the search tree

Knapsack (level, cur_V)
{
begin:
   if (level == n)
   {
      if ( sum_{i=0}^{n-1} w_i x_i > opt_W )
      {
         opt_W = sum_{i=0}^{n-1} w_i x_i ;
         opt_x = x;
      }
   }
   else
   {
      if ( (cur_V + v_level) ≤ V )
      {
         x_level = 1;
         Knapsack(level+1, cur_V + v_level);
      }
      x_level = 0; level++;
      goto begin; //instead of Knapsack(level+1, cur_V);
   }
```

Fig. 1. Pseudo-code of the algorithm employed for solving the knapsack problem

4 Hardware Implementation of Recursive Algorithms

For hardware implementation, the selected recursive algorithm has firstly been described with the aid of *Hierarchical Graph-Schemes* (HGS) [14]. The resulting HGS composed of two modules z_0 and z_1 is shown in Fig. 2a). The first module (z_0) is activated in the node *Begin* and terminated in the node *End* with the label a_1. The execution of the module z_0 is carried out in a sequential manner. Each rectangular node takes one clock cycle.

Any rectangular node may call any other module (including the currently active module). For instance, the node a_0 of z_0 invokes the module z_1. As a result, the module z_1 begins execution starting from the node *Begin* and the module z_0 suspends waiting for the module z_1 to finish. When the node *End* in the module z_1 is reached the control is transmitted to the calling module (in this case, z_0) and the execution flow is continued in the node a_1. The rhomboidal nodes are used to control the execution flow with the aid of conditions, which can evaluate to either *true* or *false*.

An algorithm described in HGS can be implemented in hardware with the aid of a *recursive hierarchical finite state machine* (RHFSM) [14]. Fig. 2b) depicts a structure of a generic RHFSM that can be used for any algorithm. The RHFSM includes two stacks (a stack of modules – *M_stack* and a stack of states – *FSM_stack*) and a combinational circuit (*CC*), which is responsible for state transitions within the currently active module (selected by the outputs of *M_stack*). There exists a direct correspondence between the RHFSM states and the node labels in Fig. 2a) (it is allowed to repeat the same labels in different modules). In the designed circuit, the *CC* is also employed for computing the solution of the knapsack problem.

The algorithm considered has been implemented and tested in an XC2S200 Spartan-II FPGA from Xilinx [3]. For experimental purposes the board RC100 [12] of Celoxica has been used. The stacks have been declared as Handel-C arrays of required dimensions (determined by the maximum number of levels n in the search tree). It should be more efficient to implement the stacks (by declaring them as dual-port RAMs) in embedded memory blocks (available in Spartan-II family FPGAs). This is possible since at most two stack locations are accessed in a single clock cycle (see Fig. 2). This issue will be addressed in future work.

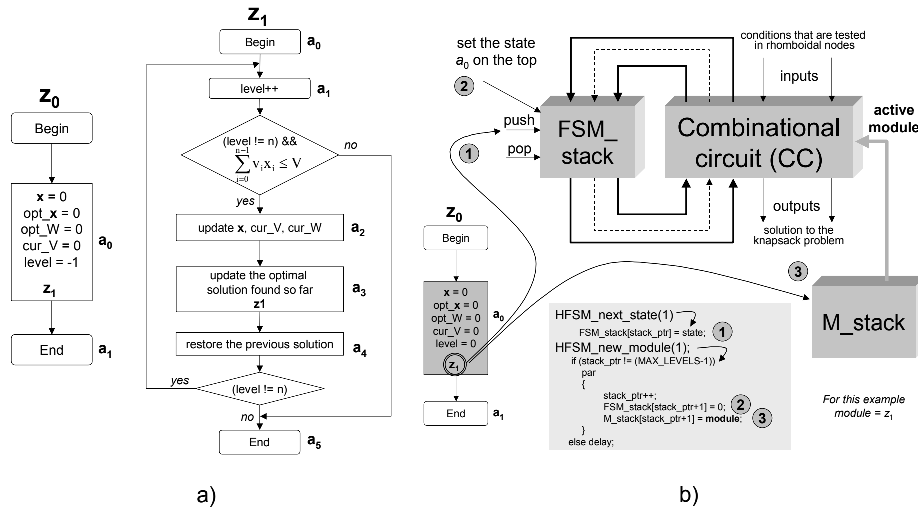

Fig. 2. a) An HGS describing the recursive search algorithm employed for solving the knapsack problem; **b)** An RHFSM with a Handel-C example of a new module invocation

5 Conclusion

In this paper a simple RHFSM-based method was described that allows recursion to be implemented in hardware. The proposed technique permits complex algorithms to be easily specified and realized on the basis of relatively simple circuits. The suggested design method

has been applied for solving the knapsack problem by a backtracking algorithm. The RHFSM model is also very useful for specifying control algorithms in an HDL (such as VHDL).

It should be noted that the objective of this work was not to achieve a significant performance improvement by implementing a computationally intensive algorithm in reconfigurable hardware. Therefore, no performance measurements have been executed up to now. Nevertheless this constitutes one of the ideas for future work. As evidenced by many experiments [11] the bounding function has a dramatic effect on the running time of the algorithm. Therefore one of our future goals is to implement the bounding function in hardware. Such an implementation can potentially overcome in performance a solution in software.

Acknowledgment

This work was partially supported by the Portuguese Foundation of Science and Technology under grant POSI/43140/CHS/2001.

References

1. G. Estrin, Reconfigurable Computer Origins: The UCLA Fixed-Plus-Variable (F+V) Structure Computer, *IEEE Annals of the History of Computing*, pp. 3-9, Oct./Dec. 2002
2. H. Krupnova and G. Saucier, FPGA-Based Emulation: Industrial and Custom Prototyping Solutions, in R. W. Hartenstein and H. Grunbacher, Eds., *Field-Programmable Logic and Applications*. *Lecture Notes in Computer Science*, vol. 1896. Springer, 2000, pp. 68–77
3. Xilinx. Data Sheets. [Online]. Available: http://www.xilinx.com/xlnx/xweb/xil_publications_index.jsp
4. Altera. [Online]. Available: http://www.altera.com/products/devices/dev-index.jsp
5. J. F. Miller, D. Job, and V. K. Vassilev, Principles in the Evolutionary Design of Digital Circuits – Part I, *Genetic Programming and Evolvable Machines 1*, 2000, pp. 7-35
6. J. Torresen, Possibilities and Limitations of Applying Evolvable Hardware to Real-World Applications, in R. W. Hartenstein and H. Grunbacher, Eds., *Field-Programmable Logic and Applications. Lecture Notes in Computer Science*, vol. 1896, Springer, 2000, pp. 230–239
7. I. Skliarova, A.B. Ferrari, Reconfigurable Hardware SAT Solvers: A Survey of Systems, *Proceedings of the 13th International Conference on Field-Programmable Logic and Applications* – FPL'2003, Lisbon, Portugal, Sept. 2003, pp. 468-477
8. I. Skliarova, A.B. Ferrari, FPGA-based Implementation of Genetic Algorithm for the Traveling Salesman Problem and its Industrial Application, *Proceedings of the 15th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems* - IEA/AIE'2002, Cairns, Australia, Jun. 2002, pp. 77-87
9. C. Plessl, M. Platzner, Instance-Specific Accelerators for Minimum Covering, Proc. of the 1st Int. Conf. on Engineering of Reconfigurable Systems and Algorithms, Las Vegas, USA, Jun. 2001, pp. 85-91
10. M. R. Garey and D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman and Company, San Francisco, 1979
11. D.L. Kreher and D.R. Stinson, Combinatorial Algorithms: Generation, Enumeration, and Search, CRC Press, 1999
12. Handel-C. [Online]: http://www.celoxica.com/
13. SystemC. [Online]: http://www.systemc.org/
14. V. Sklyarov, FPGA-based Implementation of Recursive Algorithms, *Microprocessors and Microsystems*, 28, 2004, pp. 197-211