Intelligent Systems Engineering with Reconfigurable Computing

Iouliia Skliarova
University of Aveiro, Department of Electronics and Telecommunications, IEETA, 3810-193 Aveiro, Portugal
iouliia@det.ua.pt,
WWW home page: http://www.ieeta.pt/~iouliia/

Abstract. Intelligent computing systems comprising microprocessor cores, memory and reconfigurable user-programmable logic represent a promising technology which is well-suited for applications such as digital signal and image processing, cryptography and encryption, etc. These applications employ frequently recursive algorithms which are particularly appropriate when the underlying problem is defined in recursive terms and it is difficult to reformulate it as an iterative procedure. It is known, however, that 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.

1 Introduction

Intelligent computing systems (ICS) comprising microprocessor cores, memory and reconfigurable user-programmable logic (usually, *field-programmable gate arrays* - FPGA) represent a promising technology which is well-suited for applications that require direct bit manipulations and are appropriate to parallel implementations, such as digital signal and image processing, cryptography and encryption, etc. In such ICS, the reconfigurable part is periodically modified in response to dynamic application requirements. Creation and validation of scalable, distributed ICS architectures requires a closely coordinated hardware and software development effort in the areas of FPGA-based accelerators, runtime control libraries and algorithm mapping [1]. This paper focuses on the algorithm mapping.

In ICS, real-world problems are formulated over simplified mathematical models, such as graphs, matrices, sets, logic functions and equations, to name a few.

Please use the following format when citing this chapter:

Skliarova, I., 2006, in IFIP International Federation for Information Processing, Volume 218, Professional Practice in Artificial Intelligence, eds. J. Debenham, (Boston: Springer), pp. 161–170.

Then, by applying mathematical manipulations to the respective model, a solution to the original problem is obtained. The involved mathematical manipulations differ according to the ICS type, but frequently they recur to combinatorics and require the solution of different combinatorial problems. Typical examples of these problems are finding the shortest or longest path in a graph, graph-coloring, Boolean function optimization, set covering, set encoding, etc.

It happens that many of the combinatorial problems of interest belong to the classes *NP-hard* and *NP-complete*, which implies that the relevant algorithms have an exponential worst-case complexity, imposing consequently very high computational requirements on the underlying implementation platform. This fact precludes the solution of many practical problems with conventional microprocessors. This is because conventional microprocessors are programmed with instructions selected from a predefined set, which are combined to encode a given algorithm. The use of conventional microprocessors is justified for problems where their performance is adequate because the design cost is very low. Besides, any required change in the algorithm can easily be incorporated in the respective implementation. However, since the conventional microprocessors are not optimized for solving the combinatorial problems, the resulting performance is very scant.

As opposed to the previous approach, a hardware-based solution can be tailored to the requirements of a given algorithm guaranteeing in this way an optimal performance. However, a specialized hardware circuit is only capable of executing a task, for which it has been designed, whereas a conventional microprocessor might be reutilized for different tasks via a simple modification of instruction sequence. This software/hardware compromise obligates designers to trade off between performance and flexibility.

The development of dedicated hardware systems for specific problems and domains involves considerable cost and design time. The experience shows that the resulting benefits are often scant and even non-existent, because of the current rate of evolution of conventional processor technology, which enforces supplanting specialized and optimized computing structures by those that are less efficient for a given application domain. Besides, the proper heterogeneity of combinatorial problems discourages from developing specialized hardware accelerators.

The invention of high capacity field-programmable logic devices, such as FPGA, set up an alternative method of computing. An FPGA is composed of an array of programmable logic blocks interlinked by programmable routing resources and surrounded by programmable input/output blocks. The logic blocks include combinational and sequential elements allowing both logic functions and sequential circuits to be implemented. The routing resources are composed of predefined routing channels interconnected by programmable routing switches. A logic circuit is implemented in an FPGA by distributing logic among individual blocks and interconnecting them subsequently by programmable switches. Recent FPGA incorporate also various heterogeneous structures, such as dedicated memory blocks, embedded processor cores, multipliers, transceivers, etc., which allow for the implementation of systems-on-chip.

The FPGA enable attaining both the hardware performance and the flexibility of software, since they can be optimized for executing a specific algorithm and reutilized for other algorithms via a simple reprogramming of their internal structure. As a result, ICS's engines can be constructed that are optimized for a given application via *reprogramming* the functionality of basic FPGA logic blocks, i.e.

without introducing any changes on the "hardware" level. Implementations based on reconfigurable hardware permit the execution of the relevant algorithms to be optimized with the aid of such techniques as parallel processing, personalized functional units, optimized memory interface, etc.

The applications, for which FPGA-based reconfigurable systems have been constructed, cover diversified domains such as image processing, video processing, search engines in genetic databases, pattern recognition, neural networks, high-energy physics, etc. The best performance was achieved for such applications that exhibited a high level of parallelism, had large amounts of data to process, and used a non-standard format of information representation. The combinatorial problems possess all these characteristics and are therefore eminently suitable to FPGA-based implementations. Such implementations, tailored at combinatorial problems, would allow the exponential growth in the computation time to be delayed, thus enabling more complex problem instances to be solved.

Recently, several attempts have been made to employ FPGA for solving complex combinatorial problems. Many of such problems are well suited for parallel and pipelined processing. Therefore, FPGA-based implementations can potentially lead to drastic performance improvements over traditional microprocessors. For instance, significant speedups have been shown for difficult instances of the Boolean satisfiability [2] and covering [3] problems comparing to a software solution.

Three different specification methods have been employed for the description of the respective hardware problem solvers: a schematic entry, a hardware description language, and a general-purpose programming language. The schematic-based approach is probably not appropriate because instead of thinking in terms of algorithms and data structures it forces the designer to deal directly with the hardware components and their interconnections. Contrariwise, the *hardware description languages* - HDLs (such as VHDL and Verilog) are widely used for specification of combinatorial algorithms [4] since they typically include facilities for describing structure and functionality at a number of levels, from the more abstract algorithmic level down to the gate level. The general-purpose programming languages, such as C and C++, have also been employed, being the respective descriptions transformed (by specially developed software tools [5]) to an HDL, which was used for synthesis. The higher portability and the higher level of abstraction of language-based specifications have determined their popularity and widespread acceptance.

Recently, commercial tools that allow digital circuits to be synthesized from *system-level specification languages* (SLSLs) such as Handel-C [6] and SystemC [7] 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.

Obviously, the higher level of abstraction leads to some performance degradation and not very efficient resource usage, as evidenced by a number of examples [8]. On the other hand SLSLs have many advantages such as portability, ease to learn (any one familiar with C/C++ will recognize nearly all features of SLSLs), ease of change and maintenance, and a very short development time. Therefore, we can expect that

as the tools responsible for generating hardware (more specifically, either an EDIF – *electronic design interchange format* file or an HDL file) from system-level source code advance, the SLSLs may become the predominant hardware description methodology, in the same way as general-purpose high-level programming languages have already supplanted microprocessor assembly languages.

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 [6] as a language of study and the knapsack combinatorial problem [9] as an example.

2 Functions in Handel-C

It is known that functions in Handel-C may not be called recursively [10]. This can be explained by the fact that all logic needs to be expanded at compile time to generate hardware. Three ways have been proposed to deal with recursive functions [10]:

- Using recursive macro expressions or recursive macro procedures. It should be noted that the depth of recursion must be determinable at compile time, therefore limiting the applicability of this method to rather simple cases. As an example, consider the problem of counting the number of ones in a Boolean vector. This can be accomplished with the aid of a recursive macro expression as shown in the code below. It should be noted that this is not a true recursion since the macro *count_ones* will be expanded as necessary and executed in just one clock cycle.

Example of a recursive macro expression used for calculating the number of ones in a Boolean vector:

```
unsigned 6 one = 1;
macro expr count_ones (x) = select ( width(x) == 0, 0,
            count_ones(x\\1) + (x[0] == 1 ? one : 0) );

void main()
{
   unsigned 32 vector;
   unsigned 6 number;

   vector = 0x1234abdf;
   number = count_ones(vector);
}
```

- Creating multiple copies of a function, for instance through declaring an array of functions. As in the previous case the number of functions required must be known at compile time.

- Rewriting the function to create iterative code. This is relatively easy if the function is calling itself (direct recursion) and the recursive call is either the first or the last statement within the function definition [11]. The recursive macro expression considered above can be rewritten as an iterative Handel-C function as shown below. This code is not equivalent to the code presented above since it will require 32 clock cycles (i.e. the number of bits in a vector) to execute.

Example of an iterative function used for calculating the number of ones in a Boolean vector:

```
void main()
{
   unsigned 32 vector;
   unsigned 6 number, i;

   par
   {  vector = 0x1234abdf;
      number = 0;
      i = 0;
   }
   while (i != width(vector))
      par
      {  number += (0 @ vector[0]);
         vector >>= 1;
         i++;
      }
}
```

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 because: 1) it represents a very large number of real-world problems; and 2) it provides very good teaching material. It is known that many engineering problems can be formulated as instances of the knapsack problem. Examples of such problems are public key encryption in cryptography, routing nets on FPGAs interconnected by a switch matrix [12], analysis of power distribution networks of a chip [13], etc. We consider the knapsack problem to be a good teaching material because the students are typically familiar with it from a course on data structures and algorithms and consequently we can use it in a 4th year reconfigurable computing course as an example for the design and implementation of hierarchical finite state machines (as described in section 4).

There are numerous versions of the knapsack problem as well as of the 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 $v_i \in Z^+$, $i=0,\ldots,n-1$. The objective is to determine what 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 the level i corresponds to 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 intermediate 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 below. 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 the code below.

Pseudo-code of the algorithm employed for solving the knapsack problem:

```
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_1 (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_1(level+1, cur_V + v_level);
          }

         x_level = 0;
         level++;
         goto begin;
          //instead of Knapsack_1(level+1, cur_V);
      }
}
```

4 Specification in Handel-C and 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. 1(a). The first module (z_0) is responsible for initialization of all the variables of the algorithm and 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. 1(b) depicts a structure of a generic RHFSM that can be used for implementation of any recursive 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 RHFSM states and node labels in Fig. 1(a) (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.

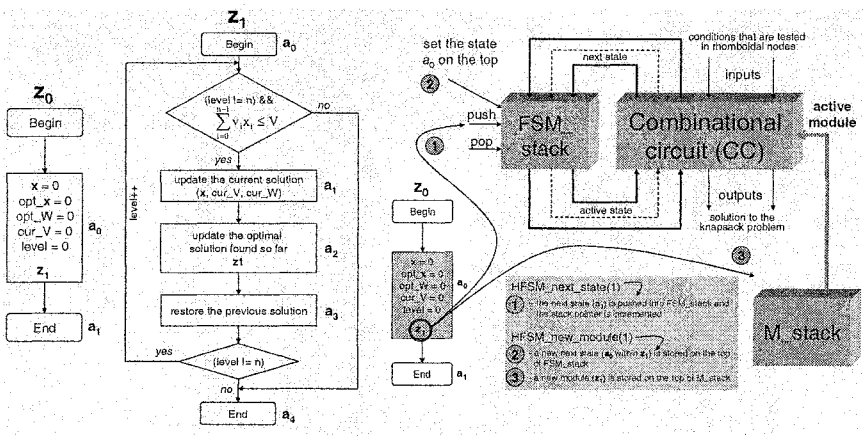

Fig. 1. (a) An HGS describing the recursive search algorithm employed for solving the knapsack problem; **(b)** Recursive hierarchical finite state machine with a Handel-C example of a new module invocation.

The code below demonstrates how to describe an RHFSM from Fig. 1 in Handel-C. The functions employed for the RHFSM management are presented in Fig. 2. Firstly, the RHFSM is reset and the node a_0 in the module z_0 is activated. This node calls the function *HFSM_next_state* in order to push the next state (a_1) into the *FSM_stack* (see Fig. 1(b)). In parallel, a new module z_1 is called with the aid of function *HFSM_new_module*. This function increments the stack pointer and stores at the top of stacks *M_stack* and *FSM_stack* a new module (z_1) and a node a_0, respectively.

The execution proceeds on module z_1 until *HFSM_end_module* function is called. This function decrements the stack pointer allowing the previously active module and the respective node within that module to be recovered.

The remaining functions are trivial and their implementation is shown in Fig. 2. Besides of state transitions, computations required by the algorithm from Fig. 1(a) are performed in each node. These computations are very simple and are not shown in the code below for the sake of simplicity.

Note that the presented code supports recursive calls. For such purposes, in the state a_2 of z_1, a new module z_1 is invoked. This causes the next state within the currently active module (a_3) to be pushed into the stack and the stack pointer to be incremented.

Handel-C function designed for solving the knapsack problem

```
void ExhaustiveSearch()
{  unsigned STATE_SIZE state;
   unsigned MODULE_SIZE module;
   unsigned 1 done;

   HFSM_reset();
   do
   {  par //initialize stacks
      {  module = get_module();
         state = get_state();  }

      switch(module)
      {
         case 0: //description of the module z0
           switch (state)
           { case 0: // state a0 - initialize variables
                par {  HFSM_new_module(1);
                       HFSM_next_state(1);
                    } break;
             case 1: //state a1
                HFSM_end_module();
           }     break;

         case 1: //description of the module z1
           switch (state)
           { case 0: // state a0
                if (/*condition*/) HFSM_next_state(1);
                else HFSM_end_module();
                break;
```

```
                case 1: //state a1 - update current solution
                  par {  //construct the current solution
                         HFSM_next_state(2);
                       } break;

                case 2: // state a2
                  par {  // update the best solution
                          HFSM_new_module(1); //recursive call
                         HFSM_next_state(3);
                       } break;

                case 3: //state a3
                  par {  //restore the previous solution
                         if (/*condition*/) HFSM_next_state(0);
                         else HFSM_end_module();
                       } break;
              }
          } done = test_ends();
    } while(!done);
}
```

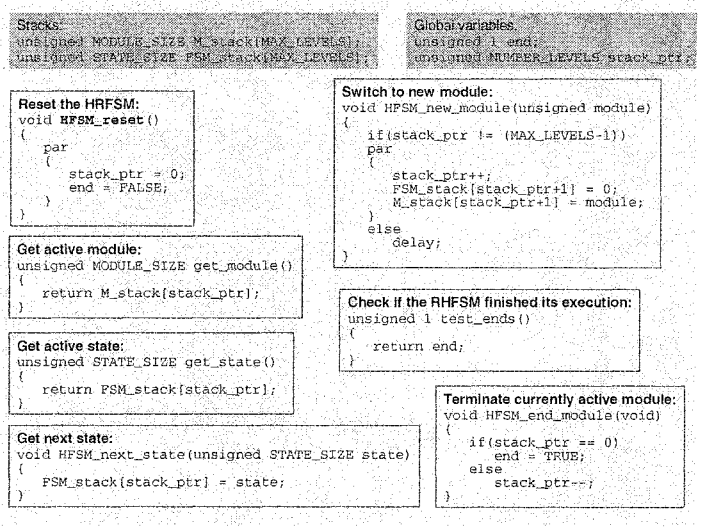

Fig. 2. Handel-C functions, which are responsible for resetting the RHFSM, controlling state transitions, performing hierarchical module calls and returns, etc.

The algorithm considered has been implemented and tested in an XC2S200 Spartan-II FPGA from Xilinx [15]. For experimental purposes the board RC100 [6] of Celoxica has been used. The stacks have been declared as Handel-C arrays of the 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.

5 Conclusion

Reconfigurable hardware supplies very vast opportunities for implementing effective ICS engines targeted at accelerating computing processes. Since these processes involve often recursively formulated functions, an efficient hardware implementation of recursion is needed. In this paper a simple RHFSM-based method was described that allows recursion to be easily implemented in hardware. The proposed technique permits complex algorithms to be 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. An RHFSM model is also very useful for specifying control algorithms in an HDL as shown in [14].

References

1. B. Schott, S. Crago, C. Chen, J. Czarnaski, M. French, I. Hom, T. Tho, and T. Valenti, Reconfigurable Architectures for System Level Applications of Adaptive Computing, [Online], Available: http://slaac.east.isi.edu/papers/schott_vlsi_99.pdf.

2. I. Skliarova and A.B Ferrari, Reconfigurable Hardware SAT Solvers: A Survey of Systems, *IEEE Trans. on Computers*, vol. 53, issue 11 (2004) 1449-1461.

3. V. Sklyarovand I. Skliarova, Architecture of a Reconfigurable Processor for Implementing Search Algorithms over Discrete Matrices, Proc. of Int. Conf. on Engineering of Reconfigurable Systems and Algorithms – ERSA (2003) 127-133.

4. P. Zhong, Using Configurable Computing to Accelerate Boolean Satisfiability, Ph.D. dissertation, Department of Electrical Engineering, Princeton University (1999).

5. O. Mencer and M. Platzner, Dynamic Circuit Generation for Boolean Satisfiability in an Object-Oriented Design Environment, Proc. of the 32nd Hawaii Int. Conf. on System Sciences (1999).

6. Handel-C, [Online], Available: http://www.celoxica.com/.

7. SystemC, [Online], Available: http://www.systemc.org/.

8. E.M. Ortigosa, P.M. Ortigosa, A. Cañas, E. Ros, R. Agís, and J. Ortega, FPGA Implementation of Multi-layer Perceptrons for Speech Recognition, Proc. of the 13th Int. Conf. on Field-Programmable Logic and Applications – FPL (2003) 1048-1052.

9. D.L. Kreher and D.R. Stinson, *Combinatorial Algorithms. Generation, Enumeration, and Search*, CRC Press (1999).

10. DK2, Handel-C Language Reference Manual, Celoxica Ltd (2003).

11. M. Wirth, *Algorithms and Data Structures*, Prentice-Hall, Inc. (1986).

12. A. Ejioui and N. Ranganathan, Routing on Field-Programmable Switch Matrices, *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 11, n. 2 (2003) 283-287.

13. M. Zhao, R.V. Panda, S.S. Sapatnekar, and D. Blaauw, Hierarchical Analysis of Power Distribution Networks, *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, n. 2 (2002) 159–168.

14. V. Sklyarov, FPGA-based Implementation of Recursive Algorithms, *Microprocessors and Microsystems*, n. 28 (2004) 197-211.

15. Spartan-IIE FPGA Family, [Online], Available: http://www.xilinx.com.