Recursive and Iterative Algorithms for N-ary Search Problems

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

Abstract. The paper analyses and compares alternative iterative and recursive implementations of N-ary search algorithms in hardware (in field programmable gate arrays, in particular). The improvements over the previous results have been achieved with the aid of the proposed novel methods for the fast implementation of hierarchical algorithms. The methods possess the following distinctive features: 1) providing sub-algorithms with multiple entry points; 2) fast stack unwinding for exits from recursive sub-algorithms; 3) hierarchical returns based on two alternative approaches; 4) rational use of embedded memory blocks for the design of a hierarchical finite state machine.

1 Introduction

Adaptive control systems (ACS) are capable to change their functionality without modifying physical components. In general this can be achieved with the aid of reprogrammable devices such as field programmable gate arrays (FPGA). A method for the design of ACS on the basis of hierarchical finite state machines (HFSM) was proposed in [1] and it makes possible to realize modular and hierarchical specifications of control algorithms based on such alternative techniques as iterative and recursive implementations [2]. It was shown that recursive implementations are more advantageous in hardware in terms of the execution time although they might require slightly more FPGA resources. This paper suggests further advances in scope of hierarchical, in general, and recursive, in particular, specifications (as well as the relevant implementations) and presents new arguments in favor of the results [1,2].

There exists a technique [3] that enables recursion to be implemented in hardware through establishing a special control sequence provided by a hierarchical finite state machine. The paper shows that the efficiency of this technique can be significantly improved through the use of the following novel methods: 1) supporting

Please use the following format when citing this chapter:

Sklyarov, V., Skliarova, I., 2006, in IFIP International Federation for Information Processing, Volume 218, Professional Practice in Artificial Intelligence, eds. J. Debenham, (Boston: Springer), pp. 81–90.

multiple entry points to sub-algorithms that are called recursively; 2) employing a fast unwinding procedure for stacks used as an HFSM memory; 3) establishing flexible hierarchical returns; 4) the rational use of embedded memory blocks for the design of HFSM stacks.

The remainder of this paper is organized in four sections. Section 2 discusses N-ary search problems that can be solved using either iterative or recursive techniques. Section 3 characterizes known results for the specification and implementation of hierarchical algorithms and suggests four methods for their improvement. Section 4 describes the experiments. The conclusion is in Section 5.

2 N-ary Search Problems

Computational algorithms for many search problems are based on generation and exhaustive examination of all possible solutions until a solution with the 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 an N-ary search tree [4], 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. At the beginning the tree is empty and it is incrementally constructed during the search process.

A distinctive feature of this approach is that at each node of the search tree a similar sequence of algorithmic steps has to be executed. Thus, either iterative or recursive procedures can be applied [2]. The only thing that is different from node to node is input data. This means that the entire problem reduces to the execution of a large number of repeated operations over a sequentially modified set of data.

Of course, exhaustive checking all possible solutions cannot be used for the majority of practical problems because it requires a very long execution time. That is why it is necessary to apply some optimization techniques that reduce the number of situations that need to be considered. In order to speed up getting the results various tree-pruning techniques can be applied.-

The other known method of improving the effectiveness of the search is a reduction [5], which permits the current situation to be replaced with some new simpler situation without sacrificing any feasible solution. However, reduction is not possible for all existing situations. In this case another method is used that relies on the divide-and-conquer strategy [4]. This applies to critical situations that have to be divided into N several simpler situations such that each of them has to be examined. The objective is to find the minimum number N.

Thus, an N-ary search problem can be solved by executing the following steps:

1. Applying reduction rules.
2. Verifying intermediate results, which permits to execute one of the following three sub-steps:
   - 2.1. Pruning the current branch of the algorithms and backtracking to the nearest branching point;
   - 2.2. Storing the current solution in case if this solution is the best;
   - 2.3. Sequential executing the points 3 and 4.

3. Applying selection rules (dividing the problem into sub-problems and selecting one of them).
4. Executing either the point 4.1 (for iterative algorithm) or the point 4.2 (for recursive algorithm).
   4.1. Executing the next iteration (see points 1-4) over the sub-problem.
   4.2. Recursive invocation of the same algorithm (see points 1-4) over the sub-problem.

Thus, either an iterative (see points 1-3, 4.1) or a recursive (see points 1-3, 4.2) algorithm can be executed and it is important to know which one is better.

Let us consider another example. Fig. 1 depicts a system, which receives messages from an external source. The messages have to be buffered and processed sequentially according to their priority. Each incoming message changes the sequence, because it has to be inserted in a proper position.

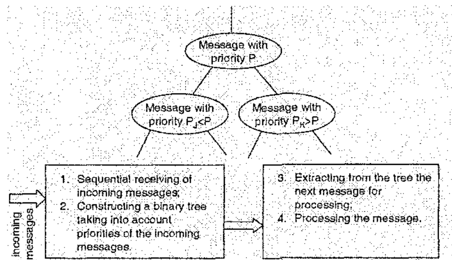

Fig. 1. Example of an adaptive embedded system

Let us assume that the following strategy is applied. All incoming messages are accommodated on a binary tree (N=2) whose nodes contain three fields that are: a pointer to the left child node, a pointer to the right child node, and the message identification, which is used to activate the respective processing procedure. The nodes are maintained so that at any node, the left sub-tree contains only messages that are less priority-driven (i.e. the messages that have smaller priority) than the message at the node, and the right sub-tree contains only messages that are more priority-driven. It is known that such a tree can be constructed and used for sorting various types of data [6] and for our particular example we can sort messages according to their priority. In order to build the tree for a given set of messages, we have to find the appropriate place for each incoming node in the current tree. In order to sort the messages, we can apply a special technique [6] using forward and backward propagation steps that are exactly the same for each node.

In case of modular specification the algorithm can easily be modified to change the sequence of processing. For example, messages with priorities less than some given value can be ignored; priorities can be flexibly changed dependently on some other external conditions, etc. Such changes do not require redesigning the complete algorithm and just a module responsible for task 2 in Fig. 1 has to be altered.

It is very important to note that hierarchical algorithms, in general, and recursive algorithms, in particular, can be very efficiently used for solving such problems and it was shown and proven in [2].

3 Specification and Implementation of Hierarchical Algorithms

3.1 Known Results

It is known [3] that hierarchical algorithms can be constructed from modules with the aid of the language called hierarchical graph-schemes (HGS). Recursion is provided through invocations of the same module. An HFSM permits execution of HGSs [3] and contains two stacks (see Fig. 2), one for states (*FSM_stack*) and the other for modules (*M_stack*). The stacks are managed by a *combinational circuit* (CC) that is responsible for new module invocations and state transitions in any active module that is designated by outputs of the *M_stack*. Since each particular module has a unique code, the same HFSM states can be repeated in different modules. Fig. 2 demonstrates how the HFSM executes an algorithm. Any non-hierarchical transition is performed through a change of a code only on the top register of the *FSM_stack* (see the example marked with •). Any hierarchical call alters the states of both stacks in such a way that the *M_stack* will store the code for the new module and the *FSM_stack* will be set to the initial state (normally to a_0=0…0) of the module (see the example marked with ■). Any hierarchical return just activates a pop operation without any change in the stacks (see the example marked with ♦). As a result, a transition to the state following the state where the terminated module was called will be performed. The stack pointer *stack_ptr* is common to both stacks. If the *End* node is reached when *stack_ptr*=0, the algorithm terminates execution.

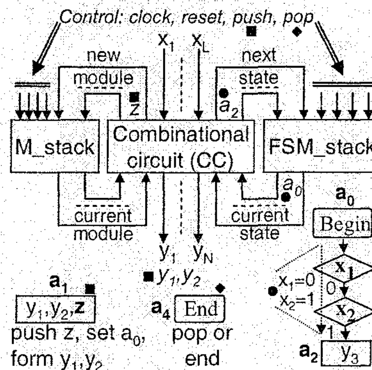

Fig. 2. Functionality of a hierarchical FSM

3.2 Novel Methods

Section 1 lists the proposed innovative facilities for HGSs and HFSMs. This subsection provides detailed explanations of these facilities augmented by examples that are illustrated through synthesizable **V**ery **H**igh Speed Integrated Circuits **H**ardware **D**escription **L**anguage (VHDL) specifications.

3.2.1. Providing Multiple Entry Points to Sub-algorithms

Fig. 3 demonstrates a fragment of a recursive sorting algorithm discussed in [2]. The two stacks used in a HFSM (such as depicted in Fig. 2) can be described in VHDL as follows (subscripts, such as 0 in a_0, are not allowed in VHDL but they have been used to provide consistency between VHDL codes and the relevant figures and textual descriptions):

```
process(clock,reset)                                    (1)
begin
   if reset = '1' then
       -- setting to an initial state and initializing
       -- if reset is active
    elsif rising_edge(clock) then
        if hierarchical_call = '1' then
          if -- test for possible errors
          else
              sp <= sp + 1;
              FSM_stack(sp+1) <= a0;          -- ref1
              FSM_stack(sp) <= NS;
              M_stack(sp+1) <= NM;
          end if;
        elsif hierarchical_return = '1' then
          sp <= sp - 1;                       -- ref2
        else      FSM_stack(sp) <= NS;
        end if;
   end if;
end process;
```

Here any module invocation/termination is indicated through a signal *hierarchical_call/hierarchical_return*; *sp* is a common stack pointer; *NS/NM* is a new state/module, a_0 is an initial state of each module. Indicators *ref1*, *ref2* will be used for future references.

The combinational circuit (CC) in Fig. 2 has the following skeletal code (template):

```
process (current_module,current_state,inputs)    (2)
begin
   case M_stack(sp) is
      when z1 =>
          case FSM_stack(sp) is
             -- state transitions in the module z1
             -- generating outputs for the module z1
          end case;
        -- repeating for all modules, which might exist
   end case;
end process;
```

As we can see from Fig. 3 any hierarchical module invocation, such as that is done in the node a_2, activates the same algorithm once again, starting from the node *Begin* (a_0). Skipping the node a_0 removes one clock cycle from any hierarchical call. However in this case the algorithm in Fig. 3 must have multiple entry points and a particular entry point will be chosen by the group of rhomboidal nodes enclosed in an ellipse. This possibility is provided by the additional tests performed in the nodes with hierarchical calls (such as a_2 and a_3 in Fig. 3). The following fragment demonstrates how these tests can be coded in VHDL for the state a_2.

```
when a2 => -- generating outputs and the signal
          -- hierarchical_call
       NM<=z1;
       if x3='1' then  NM_FS <= a1;  -- this is because
       -- x1 cannot be equal to 1 after the state a2
       elsif x2 ='0' then NM_FS <= a5;
       elsif x4 ='0' then NM_FS <= a2;
       else NM_FS <= a3;
       end if;
```

Here *NM_FS* is the first state of the next module. The line *ref1* in (1) has to be changed as follows:

```
FSM_stack(sp+1) <= NM_FS;
```

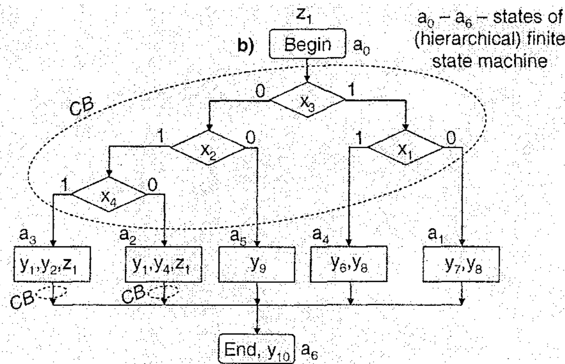

Fig. 3. HGS with multiple entry points provided through inserting the control block (CB) in the nodes with hierarchical calls

3.2.2. Fast Stack Unwinding

Since there is just the node *End* after a_2 and a_3, hierarchical activation of any one of the nodes a_1, a_4, a_5 (see Fig. 3) leads to termination of the algorithm. To implement this termination in [2] the lines in (1)

```
if hierarchical_return = '1' then
          sp <= sp - 1;
```

are executed repeatedly until the pointer *sp* receives the value assigned at the beginning of the algorithm (see the line `if reset = '1'` in (1) for our example). In the general case, this value is assigned during the first call of the respective module (such as that depicted in Fig. 3) followed by subsequent recursive invocations of the module. Repeated execution of the line `sp <= sp - 1;` requires multiple additional clock cycles. To eliminate these redundant clock cycles the proposed method of fast stack unwinding is employed. The line *ref2* in (1) is changed as follows:

```
sp <= sp - unwinding;
```

where the signal *unwinding* is calculated as

```
unwinding <= sp - saved_sp + 1;
```

and `saved_sp <= sp` at the first invocation of the module. Thus, redundant clock cycles for hierarchical returns will be avoided.

3.2.3. Execution of Hierarchical Returns

Hierarchical calls in [2] are carried out as follows:

```
if hierarchical_call = '1' then
         -- error handling
   sp <= sp + 1;
   FSM_stack(sp+1) <= a0;
   FSM_stack(sp) <= NS; -- ***
   M_stack(sp+1) <= NM;
```

The line indicated by asterisks sets the code of the next state during a hierarchical call. As a result, after a hierarchical return the top register of the *FSM_stack* contains the code of the proper HFSM state (i.e. no additional clock cycle is required). Since the next state is determined before the invocation of a module (such as z_1), the latter cannot affect the state transition, i.e. any possible change of the conditions x_1-x_4 in z_1 cannot alter the previously defined next state. For our example this does not create a problem. However, for many practical applications it is a problem and it must be resolved. The following code gives one possible solution:

```
if rising_edge(clock) then
       if hierarchical_call = '1' then
          -- error handling
             sp <= sp + 1;
             FSM_stack(sp+1) <= NM_FS;
             M_stack(sp+1) <= NM;
```

After a hierarchical return from *NM*, the code above sets *FSM_stack* to the state where the hierarchical call of *NM* was executed. This enables us to provide correct transitions to the next state because all logic conditions that might be changed in the called module *NM* have already received the proper values. However, this gives rise

to another problem; namely it is necessary to avoid repeating invocation of the same module *NM* and iterant output signals. The following code overcomes the problem:

```
if rising_edge(clock) then                              (3)
       if hierarchical_call = '1' then
           -- error handling
               sp <= sp + 1;
               FSM_stack(sp+1) <= NM_FS;
               M_stack(sp+1) <= NM;
       elsif hierarchical_return = '1' then
           sp <= sp - 1;
           return_flag <= '1';
       else      FSM_stack(sp) <= NS;
                    return_flag <= '0';
       end if;
```

The signal *return_flag* permits module invocation and output operations to be activated during a hierarchical call and to be avoided during a hierarchical return. Indeed, the *return_flag* is equal to 1 only in a clock cycle when the signal *sp* is decremented (see the code (3) above). As soon as the currently active module is being terminated, the control flow will be returned to the point from which this module was called. Thus, the top of the *M_stack* will contain the code of the calling module and the top of the *FSM_stack* will store the code of the calling state. The *return_flag* enables us to eliminate the second call of the same module (and the second activation of the relevant output signals). This is achieved with the aid of the following lines that have to be inserted in the code (2):

```
when state_with_module_call => NS <=
                  -- testing the conditions and
                  -- computation of the next state
    if return_flag = '0' then
      hierarchical_call <= '1';
      -- specifying outputs
      NM <= -- assigning the next module
        else
          hierarchical_call <= '0';
          outputs <= (others => '0');
    end if;
```

Finally, the proposed technique permits logic conditions to be tested after terminating the called module, which might alter these conditions.

3.2.4. Using Embedded Memory Blocks

Synthesis of HFSMs from the specifications considered above has shown that stack memories (see Fig. 2) are very resource consuming. However, for implementing the functionality (1) embedded memory blocks can be used (such as that are available for FPGAs [1]). It should be noted that the majority of recent microelectronic

devices offered on the market either include embedded memory blocks (such as FPGAs with block RAMs) or allow these blocks to be used.

4. Implementation Details and the Results of Experiments

Alternative iterative and recursive implementations for various problems were analyzed and compared in [2]. Two types of recursive calls were examined, namely for cyclic and binary (N-ary) search algorithms. The relevant comparative data were analyzed through modeling in software and synthesis and implementation in hardware from system-level (Handel-C) and RTL (VHDL) specifications. One of the results was the following: using recursive algorithms in hardware for problems of N-ary search seems to be more advantageous comparably with iterative implementations.

This paper suggests methods for further improvements of the results [2], which makes possible to demonstrate new advantages of recursive algorithms over iterative algorithms. The first column of Table 1 lists HFSMs for the problems P_1, P_3 and P_4 considered in [2] and for each of them shows the results of different implementations, where N_s is the number of the occupied FPGA slices and N_{clock} is the number of the required clock cycles. Note that the maximum allowed clock frequency, obtained with the aid of implementation tools, is practically the same for all columns related to the same problem.

Table 1. The results of experiments with HFSMs

Problem from [2]	N_s/N_{clock}			
	Implementation [3]	Block RAM	Distributed RAM	Multiple entries for [3]
P_1	192/72	50/72	53/72	189/59
P_3	68/62	17/62	19/62	67/51
P_4	49/11	15/11	16/11	47/9

Analysis of Table 1 demonstrates significant advantages of the proposed methods. Note that block and distributed RAM can also be used for synthesis from HGSs with multiple entry points. This gives nearly the same number of slices as for the columns *Block RAM* and *Distributed RAM*. Thus, combining different methods proposed in this paper permits to achieve the best results.

Table 2 lists the same examples P_1, P_3, P_4 that have been considered in [2] and shows (comparing with [2]) the percentage reduction in either hardware resources (FPGA slices) for the columns marked with N_s, or the number of clock cycles for the columns marked with N_{clock}. All the conditions for providing experiments are the same as in [2]. The synthesis and implementation of circuits from specification in VHDL were done in Xilinx ISE 8.1 for xc2s400e-6ft256 FPGA (Spartan-IIE family of Xilinx) available on the prototyping board TE-XC2Se [7]. Note once again that advantages of all columns (i.e. *Block RAM*, *Distributed RAM* and *Multiple entries*) can be combined. For example, considering for the problem P_1 HGSs with multiple entry points and using block RAMs for the implementation of HFSM enable us to

decrease resources in 24% and reduce the number of cycles in 18%. Hence, in addition to comparison performed in [2], the results of this paper present further advantages which can be gained for modular algorithms in general, and recursive algorithms in particular over non-modular iterative implementations.

Table 2. Experiments with examples from [2]

Problem from [2]	N_s/N_{clock} (%)		
	Block RAM	Distributed RAM	Multiple entries
P_1	24/0	24/0	0/18
P_3	12/0	12/0	0/5
P_4	7/0	7/0	0/4

5 Conclusion

In this paper alternative recursive and iterative implementations of algorithms for N-ary search problems have been analyzed. Such algorithms are frequently used for describing functionality of adaptive control systems. From the results of [2] we can conclude that recursive implementations are more advantageous in hardware in terms of the execution time although they might require slightly more FPGA resources. The paper suggests four methods for further improvements in the specification, synthesis and hardware implementation of hierarchical, in general, and recursive, in particular, algorithms. To clarify their use in practical projects many useful fragments of synthesizable VHDL code are presented. The experiments described in the paper have proven significant advantages of the proposed methods comparing with other known results.

References

[1] V. Sklyarov, Models, Methods and Tools for Synthesis and FPGA-based Implementation of Advanced Control Systems, Proceedings of ICOM'05, Kuala Lumpur, Malaysia, 2005, pp. 1122-1129

[2] V. Sklyarov, I. Skliarova, and B. Pimentel, FPGA-based implementation and comparison of recursive and iterative algorithms, Proceeding of FPL'2005, Tampere, Finland, 2005, pp. 235-240.

[3] V. Sklyarov, Hierarchical Finite-State Machines and Their Use for Digital Control, *IEEE Transactions on VLSI Systems*, vol. 7, no 2, pp. 222-228, 1999.

[4] I. Skliarova and A.B. Ferrari, The Design and Implementation of a Reconfigurable Processor for Problems of Combinatorial Computation, *Journal of Systems Architecture*, Special Issue on Reconfigurable Systems, vol. 49, nos. 4-6, 2003, pp. 211-226.

[5] A.D. Zakrevski, *Logical Synthesis of Cascade Networks* (Moscow: Science, 1981).

[6] B.W. Kernighan and D.M. Ritchie, *The C Programming Language* (Prentice Hall, 1988).

[7] Spartan-IIE Development Platform, Available at: www.trenz-electronic.de.