Acceleration of Linear Block Code Evaluations Using New Reconfigurable Computing Approach

Hidehisa Nagano, Takayuki Suyama and Akira Nagoya
NTT Communication Science Laboratories
2-4 Hikaridai, Seika-cho, Soraku-gun, Kyoto 619-0237 JAPAN
Tel: +81-774-93-{5276, 5272, 5270}
Fax: +81-774-93-5285
{nagano, suyama, nagoya}@cslab.kecl.ntt.co.jp

Abstract

This paper presents an approach to performing applications using reconfigurable computing (RC). Our RC approach is achieved by effective use of design automation systems. Logic circuits specialized for each individual application task are automatically implemented on FPGAs. Such circuits can quickly perform tasks that are time-consuming for general purpose computers. Decoding of binary linear block codes for the evaluation is taken up as an example application. Experimental results show that the time for decoding of the code specific decoding circuit implemented on FPGAs, in which computations are executed in parallel, is much shorter than that of the software decoder.

1 Introduction

Recently, due to advances in Field Programmable Gate Array (FPGA) technologies, users can easily create their original logic circuits and reconfigure them. In addition, as these extensions of FPGAs and increasing demands for high performance computing, reconfigurable computing (RC) systems using FPGAs are receiving much attention [1]. Many RC systems adapted to target applications have been proposed and developed, such as reconfigurable coprocessors and special–purpose machines.

In this paper, we present an RC approach to performing applications using logic synthesis systems. In our approach, we use the circuit specialized not for the application but for an individual application instance that is the task which users want to perform. These circuits utilizing features of individual instances are automatically designed and implemented by using design automation systems and the reconfigurability of FPGAs. In this paper, the evaluation of binary linear block codes is taken up as an example application in order to confirm the effectiveness of our approach. Generally, in this evaluation, each of many codes is evaluated by computer simulation. This simulation is to decode many times using the software decoder, but it takes for a long time. Therefore, we accelerate these simulations by using decoders implemented on FPGAs. These decoders are automatically designed for each of codes and implemented by utilizing the reconfigurability of FPGAs. Experimental results show that decoders on FPGAs can significantly reduce the simulation time by utilizing the parallel computability of the decoding procedure.

The rest of this paper is organized as follows: Section 2 describes our RC approach generally. In Section 3, our approach to code evaluations is discussed. Section 4 gives experimental results for the example application. We conclude this paper in Section 5.

2 RC Approach

In this section, we generally describe our RC approach to performing applications. Many ordinary RC systems (e.g. reconfigurable coprocessors and special-purpose machines) are reconfigured for the target applications. We call this RC approach the application specific approach. On the other hand, our RC approach reconfigures FPGAs for an instance (an individual task of users) of the application. We call our RC approach the instance specific approach. Fig. 1 shows the distinguishing characteristics of our approach. In our approach, first, we give an application instance to the Circuit Generator Program (CGP). The CGP analyzes the instance and automatically generates a high level behavioral description of a circuit specialized for the instance. This description is written in an HDL. Then, we use high level logic synthesis systems and design automation systems in order to implement the circuit on FPGAs.

For our approach, a CGP for the target application should be developed first. Next, whenever an instance is given, the circuit specialized for the instance should be synthesized. This approach, however, has an advantage. The CGP can analyze specific features of a given instance (e.g. parallel computability) and generate a hardware description of a circuit exceedingly utilizing such features. Thus, the circuit specialized for an instance can achieve higher performance. For the problem discussed in Section 3, performing an instance is time-consuming. In addition, many instances must be performed. Therefore, this advantage of our approach is significant.

3 Acceleration of Linear Block Code Evaluations

In this section, we discuss the acceleration of the evaluation of binary linear block codes. This application is a good example for showing the effectiveness of our RC approach. First, the evaluation of binary linear block codes is described. Then, we explain the architecture of the circuit performing an instance, and the developed CGP is described.

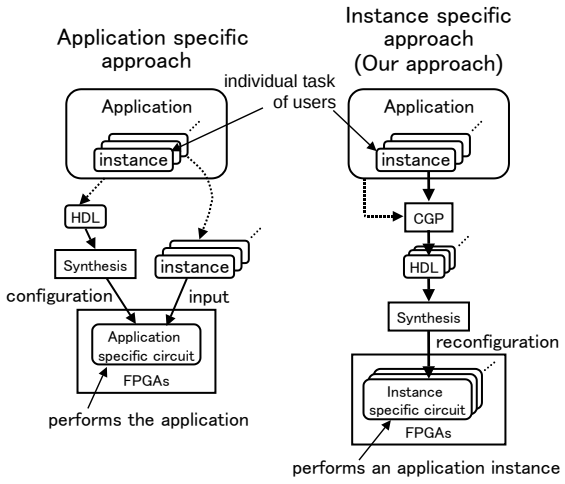

Fig. 1: RC approach to the application.

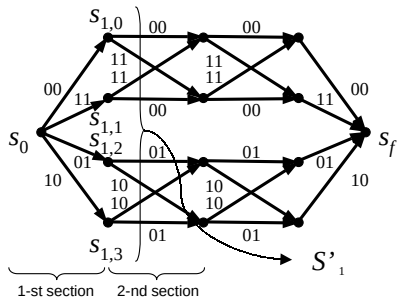

Fig. 2: 4-section trellis diagram for the first-order RM code of length 8.

3.1 Features of Linear Block Code Evaluations

In this paper, we take up simulations of decoding binary linear block codes for the evaluation which suppose soft decision maximum likelihood (SDML) decoding. These simulations are to decode for many received sequences assuming the real data transmission. However, these simulations have two troublesome features.

(1) The simulation for a code takes for a long time (e.g. several hours or days) to obtain statistically reliable results, because the simulation using the software decoder executed on a general purpose computer, which cannot utilize the parallel computability of the decoding procedure, is time-consuming.

(2) Simulations for many codes should be done in order to find good codes suitable for particular situations.

These features, however, can be overcome by our RC approach discussed in Section 2. In order to reduce the evaluation time, we use the SDML decoder implemented on FPGAs instead of the software decoder. The decoder on FPGAs specialized for each code is automatically implemented.

3.2 SDML Decoding

Here, we explain SDML decoding for a binary linear block code using its L-section trellis diagram [2]–[4]. The binary phase shift keying modulation, the 8-level quantization and the additive white Gaussian noise channel are supposed [5].

Let N be a multiple of L. The L-section trellis diagram for a binary linear block code of length N is the state transition diagram of a finite state automaton accepting all codewords of the code. A branch of the diagram has a branch label representing N/L code bits. For example, the 4-section trellis diagram for the first-order Reed-Muller (RM) code of length 8 is shown in Fig. 2. In this trellis, two adjacent states are connected by a single branch with a branch label representing two code bits. The initial state is denoted s_0 and the final state is s_f. For an L-section trellis diagram, sub-graphs are called sections as shown in Fig. 2. Let i be an integer ($1 \leq i \leq L$). S'_i denotes the set of all states at the end of the i-th section. For convenience, $S'_0 = \{s_0\}$ and $S'_L = \{s_f\}$. The states in S'_i are numbered as $S'_i = \{s_{i,0}, s_{i,1}, \ldots, s_{i,|S'_i|-1}\}$. A set of all branches between the same two adjacent states in a section is called a *parallel branch set*, and the set of all branch labels of a parallel branch set is called a *parallel label set*.

SDML decoding is resolved into the shortest path problem in the trellis. Let the branch label of a branch b in the i-th section be denoted l_b ($l_b = (l_{b,1}, l_{b,2}, \ldots, l_{b,N/L})$). $\boldsymbol{r} = (r_1, \cdots, r_N)$ is the 8-level quantized receive sequence. (i.e. r_j is an integer and $0 \leq r_j \leq 7$ for $1 \leq j \leq N$.) The branch metric of b, denoted $BM(b)$, is calculated as follows.

$$BM(b) = \sum_{j=1}^{N/L} f(l_{b,j}, r_{\frac{N}{L}(i-1)+j}),$$

$$f(x,y) = \begin{cases} 7-y & if \quad x=1 \\ y & if \quad x=0. \end{cases}$$

For a path from s_0 to $s_{i,j}$, its path metric is the sum total of the branch metrics of all branches on the path. The state metric of $s_{i,j}$, denoted $SM(s_{i,j})$, is the smallest (shortest) path metric of path metrics of all paths from s_0 to $s_{i,j}$. SDML decoding is to find the path giving $SM(s_f)$ and choose the branch label sequence of it as the decode word. The standard decoding procedure is as follows.

(Step 1) For each of all parallel branch sets, calculate branch metrics and find the branch with the smallest branch metric in the parallel branch set.

(Step 2) From the 1-st section to the L-th section, calculate the state metrics of all states in S'_i after finding the state metrics of all states in S'_{i-1} in the following manner.

For a state $s_{i,j}$ in S'_i,

$$SM(s_{i,j}) = \min_{s_{i-1,k} \in CS(s_{i,j})} (SM(s_{i-1,k}) +$$

$MB(s_{i-1,k}, s_{i,j})$).

If the path through $s_{i-1,k}$ gives $SM(s_{i,j})$, the state number k is stored at $s_{i,j}$.

(Step 3) Choose the label sequence of the path that gives $SM(s_f)$ as the decode word. □

In Step 2, $CS(s_{i,j})$ is the set of states in S'_{i-1} that are connected to $s_{i,j}$, and $MB(s_{i-1,k}, s_{i,j})$ is the smallest branch metric between $s_{i-1,k}$ and $s_{i,j}$ calculated in Step 1. Note that for any two parallel label sets, p and p', in a section, $p \cap p' = \phi$ or $p = p'$ [4]. Therefore, calculations in Step 1 for parallel branch sets in a section with the same parallel label set are done only once.

3.3 Architecture of SDML Decoder

Here, we explain the architecture of the SDML decoder implemented on FPGAs.

3.3.1 Structure and Behavior

First, we explain the structure and behavior of the SDML decoder shown in Fig. 3. The decoder is constructed from hardware components stated in Section 3.3.2. BRANCH_SET$_{i,j}$ is the hardware component that calculates the branch metrics of a parallel label set, denoted $p_{i,j}$, in the i-th section with the input receive sequence and outputs the smallest branch metric. These correspond to calculations for a parallel branch set in Step 1 stated in Section 3.2. STATE$_{i,m}$ is the hardware component for the state $s_{i,m}$. STATE$_{i,m}$ calculates path metrics by adding input state metrics and branch metrics in sequence and finds $SM(s_{i,m})$ as Step 2 in Section 3.2. BRANCH_SET$_{i,j}$ is connected to STATE$_{i,m}$ if a parallel branch set with the parallel label set $p_{i,j}$ in the i-th section is connected to the state $s_{i,m}$. STATE$_{i,m}$ and STATE$_{i-1,m'}$ are connected if $s_{i,m}$ and $s_{i-1,m'}$ are connected by branches in the trellis. Calculations in Step 1 of the decoding procedure can be executed in parallel for all parallel label sets in all sections. In order to utilize this parallel computability, all BRANCH_SETs for all sections work in parallel. In Step 2, state metrics can be found in parallel for all states in S'_i after finding all state metrics of states in S'_{i-1}. For utilizing this parallel computability, STATEs for states at the end of each section work in parallel. These parallel executions reduce the decoding delay. They are achieved by implementing the circuit specialized for an instance code. The trellis structure is code specific.

3.3.2 Submodules for Decoder

Here, hardware components (which we call submodules) of the SDML decoder are explained.

1. BRANCH_SET$_{i,j}$ is the submodule that calculates the branch metrics of a parallel label set $p_{i,j}$ in the i-th section with the input receive sequence and find the smallest branch metric as Step 1 stated in Section 3.2. After finding the smallest branch metric, BRANCH_SET$_{i,j}$ holds it and the number of branch label giving it. For requirements from connected submodules (e.g. STATE$_{i,m}$), BRANCH_SET$_{i,j}$ outputs the smallest branch metric of $p_{i,j}$. These functions of BRANCH_SET$_{i,j}$ are constructed from an adder, a comparator, registers holding the metric and the label number and a small functional unit to generate branch labels.

2. STATE$_{i,m}$ is the submodule for the state $s_{i,m}$. STATE$_{i,m}$ calculates path metrics by adding input state metrics and branch metrics in sequence and finds $SM(s_{i,m})$ as Step 2 in Section 3.2. After finding $SM(s_{i,m})$, STATE$_{i,m}$ holds $SM(s_{i,m})$ and the previous state number giving it. The number of the parallel label set between these states is also stored. For requirements from connected submodules (e.g. STATE$_{i+1,m'}$), STATE$_{i,m}$ outputs $SM(s_{i,m})$. These functions of STATE$_{i,m}$ are constructed from registers, an adder and a comparator.

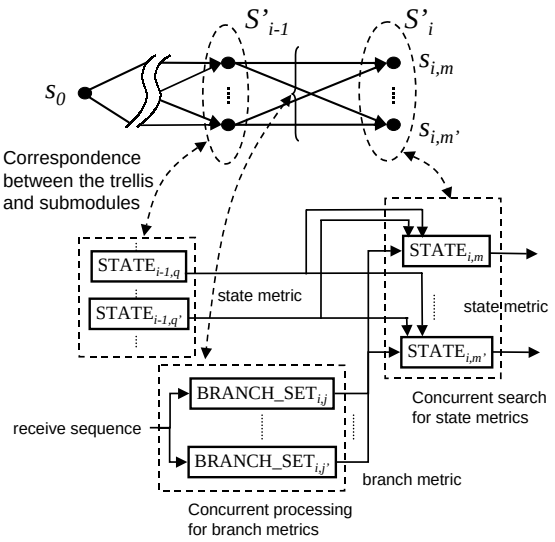

Fig. 3: Structure and behavior of the decoder.

3.4 CGP

We have developed a CGP which generates behavioral descriptions of decoders for given codes. Each decoder description is code specific and utilizes parallel computability based on the code specific trellis structure previously stated. In addition, the CGP is indispensable to using respective decoders for the evaluation of many codes while reconfiguring FPGAs. We use an HDL called SFL [6] for the descriptions of decoders. SFL is a purely behavioral and object oriented language not mixed with connection descriptions. For the SFL description, submodules stated in Section 3.3.2 are presented as objects and the complex code specific connections in Fig. 3 are presented as simple behaviors of objects. Therefore, SFL lessen the burden of developing a CGP which flexibly generates code specific descriptions.

Now, we explain the developed CGP. A binary linear block code is defined by its generator matrix.

Table 1: Evaluation results.

Code		RM(32, 6)	RM(32, 26)
FPGAs	$CYCLE$	38	185
	CLK (MHz)	11.5	6.08
	$DELAY$ (μs)	3.31	30.4
Software	$DELAY$ (μs)	76.2	211.5

Given a generator matrix (text file) and a parameter L, the CGP analyzes the L-section trellis structure and determines submodules used in the behavioral description of the decoder. Next, the CGP generates the SFL description of code specific submodules used in the description. Finally, the behavioral description following the decoding procedure stated in Section 3.2 is generated. Parallel executions stated in Section 3.3.1 are presented by parallel activation of objects in SFL.

4 Experiments

Our top down implementation flow of decoders is shown in Fig. 4. Assuming the FPGAs are Altera FLEX 10K, we evaluated decoders on FPGAs with Altera's FPGA mapping tool, MAX+plus II. A high level logic synthesis system called PARTHENON[6] was used to synthesize SFL descriptions of decoders. Decoding delays (the time for decoding a codeword) of decoders on FPGAs were evaluated by comparison with software decoders. For software decoders, all steps of the decoding procedure stated in Section 3.2 is executed in sequence. The software decoder is written in C language and executed on Sun Ultra2 (Ultra Sparc 200MHz). The first-order RM code of length 32, denoted RM(32,6), and the third-order RM code of the same length, denoted RM(32,26) were taken up as example codes and the 4-section trellis diagrams for these codes were used.

The experimental results are shown in Table 1. For decoders on FPGAs, $CYCLE$, CLK and $DELAY$ denotes the number of clock cycles required for decoding a codeword, the clock frequency and the decoding delay, respectively. $DELAY$ of the software decoder is also shown. From this table, we see that $DELAY$ of the decoder on FPGAs for RM(32,6) is about 1/23 of that of the software decoder. The same value for RM(32,26) is approximately 1/7. As a result, using the decoder implemented on FPGAs for each code instead of the software decoder reduces the simulation time.

These experiments are for the early stage of our work. The parallel computability of the decoding procedure not stated in Section 3.3.1 still remains. Utilizing it, decoders on FPGAs can achieve more high performance.

5 Conclusion

In this paper, we have presented an approach to performing applications using RC. Our RC approach is realized by using design automation systems and brings high performance computing. By developing a CGP, logic circuits specialized for each application instance are automatically implemented on FPGAs. Such circuits can perform an instance fast, which are time-consuming for general purpose computers. Experimental results show that an example application, the evaluation of binary linear block codes, can be performed extremely fast by using code specific circuits.

We are planning on tackling other problems to confirm the effectiveness of our approach and applying dynamically reconfigurable computing.

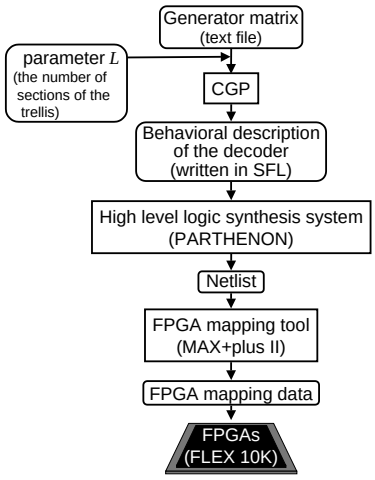

Fig. 4: Implementation flow.

References

[1] Miyazaki T., "Reconfigurable Systems: A Survey," *Proc. of ASP-DAC '98*, pp. 447–457, Feb. 1998.

[2] Wolf, J., "Efficient Maximum Likelihood Decoding of Linear Block Codes Using a Trellis," *IEEE Trans. Inf. Theory*, vol. IT-24, no. 1, pp. 76–80, Jan. 1978.

[3] Forney, G.D., Jr., "Coset Codes—Part II: Binary Lattices and Related Codes," *IEEE Trans. Inf. Theory*, vol. 34, no. 5, pp. 1152–1187, Sep. 1988.

[4] Kasami, T., Takata, T., Fujiwara, T. and Lin, S., "On Structural Complexity of the L-section Minimal Trellis Diagrams for Binary Linear Block Codes," *IEICE Trans. Fundamentals*, vol. E76-A, no. 9, pp. 1411–1421, Sep. 1993.

[5] Clark, G.C., Jr. and Cain, J.B., *Error-Correction Coding for Digital Communications*, Plenum Press, 1981.

[6] Nakamura, Y., Oguri, K., Nagoya, A., Yukishita, M. and Nomura, R., "High-level synthesis design at NTT Systems Labs," *IEICE Trans. Inf & Syst.*, vol. E76-D, no. 9, pp. 1047–1054, Sep. 1993.