Brief Papers

C-Based Complex Event Processing on Reconfigurable Hardware

Hiroaki Inoue, Takashi Takenaka, and Masato Motomura

***Abstract*— This brief presents an efficient complex event-processing framework, designed to process a large number of sequential events on field-programmable gate arrays (FPGAs). Unlike conventional structured query language based approaches, our approach features logic automation constructed with a new C-based event language that supports regular expressions on the basis of C functions, so that a wide variety of event-processing applications can be efficiently mapped to FPGAs. Evaluations on an FPGA-based network interface card show that we can achieve 12.3 times better event-processing performance than does a CPU software in a financial trading application.**

***Index Terms*— Circuit synthesis, computer languages, pipeline processing, reconfigurable architectures.**

I. Introduction

Recent trends in real-time application domains, such as financial trading, network surveillance, healthcare, and supply chain management, require processing of high volumes of time-series events in order to extemporarily extract meaningful information. Complex event processing (CEP) [1] is a new computing paradigm that responds to such application requirements. By definition, CEP generates complex events (i.e., useful information) from a sequence of real-time events and allows events to be both filtered with user-defined patterns and transformed into new data so that applications will be able to quickly and easily handle the events and data.

Many software-based CEP systems support new event languages [2], [3] as extensions of the structured query language (SQL) used in database management systems (DBMSs). SQL is a domain-specific declarative language widely used in DBMSs, and enables applications to handle a large volume of data items so as to be efficiently implemented with simple operators. In such systems, an event is defined as data with multiple fields (a tuple). The two most powerful functions that such CEP event languages support are: 1) regular expressions with Kleene closure (e.g., $*$ or $+$ in regular expressions) and 2) user-defined aggregation functions. A regular expression provides a concise and flexible means for matching sequential events. In particular, Kleene closure enables a finite yet unbounded number of time-series events to be efficiently handled. In addition, user-defined aggregation functions help achieve high throughput with algorithm-oriented applications, such as change-point analysis and cryptography.

However, existing software-based CEP systems, which achieve sophisticated event processing with SQL-based declarative languages, suffer from poor event-processing performance (at most, 500 000 events/s) [4]. For example, in financial trading markets, the Options Price Reporting Authority has, in fact, announced that event traffic will reach 6.537 mega events/s (1.57 Gb/s) by July, 2012 [5]. Typically, events are sent in small user datagram protocol (UDP) packets in order to reduce the latency required for event processing. Once the arrival rate of the events exceeds a certain threshold, the systems are unable to sustain their event processing since UDP packets begin to be dropped [6].

One promising approach would seem to be the use of reconfigurable hardware, such as field-programmable gate arrays (FPGAs), in order to accelerate event processing. The authors of [7] have proposed an epoch-making FPGA-based CEP system that employs an in-house SQL compiler and have achieved 1 Gb/s event-processing performance. While the FPGA-based CEP system [7] supports regular expressions with Kleene closure, it fails to support any aggregation functions. This is because the SQL compiler has an inability to handle the hardware complexity that would be required for support of aggregation functions executed along regular expressions with Kleene closure.

This brief reports a new design methodology that develops a novel hardware-friendly C-based event language. This C-based approach offers two major benefits: 1) high-throughput regular expressions with Kleene closure and 2) support of various aggregation functions. This is possible because recent advances in high-level synthesis industry tools, such as Impulse C and NEC CyberWorkBench, allow source codes written in C to be directly converted to fully-optimized hardware description language (HDL) source codes.

The remainder of this brief is structured as follows. Section II depicts our motivating example, Section III introduces our C-based framework, Section IV presents the results of our evaluation, and Section V summarizes our work.

II. Motivating Example

One of our target applications is financial trading, which requires real-time processing with respect to various stock prices obtained from stock exchanges. Fig. 1 shows a simple stock price analysis. Here, we use the opening prices of NEC Japan stock in the morning and afternoon sessions from January 4 (the beginning of this year's trading) to January 31, 2011. A smoothing operation is often first conducted in order to eliminate fluctuations in stock prices. After that, a change-point analysis, such as data mining, is often performed to the smoothed line in order to extract useful information. The information can be used to conduct an advanced analysis. In this example, we have used a moving average of four stock prices as a smoothing operation and used detection of local maxima and minima as a change-point analysis.

III. C-Based CEP Language

The idea behind our C-based language is the use of regular expressions. Then, each element of a regular expression is a function written in C. This means that our language gives a new feature to the ANSI C specification since it enables C-based aggregation functions to be invoked along a regular expression. Concepts behind

Manuscript received November 16, 2011; revised March 14, 2012; accepted April 22, 2012. Date of publication May 16, 2012; date of current version April 22, 2013.

H. Inoue and T. Takenaka are with the System IP Core Research Laboratories, NEC Corporation, Kawasaki 211-8666, Japan (e-mail: h-inoue@ce.jp.nec.com; takenaka@aj.jp.nec.com).

M. Motomura is with the Graduate School of Information Science and Technology, Hokkaido University, Sapporo 060-0814, Japan (e-mail: motomura@ist.hokudai.ac.jp).

Digital Object Identifier 10.1109/TVLSI.2012.2197230

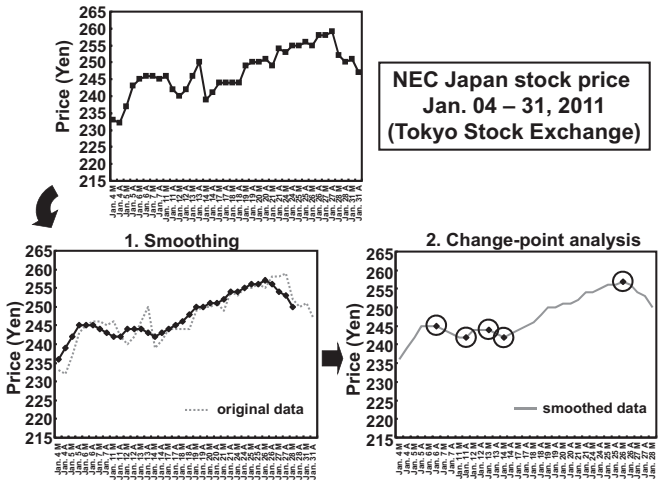

Fig. 1. Motivating example—stock price analysis.

(1) EVENT_RULE macro:

`EVENT_RULE (<rname>, <rule>, <initial>, <final>)`	
`<rname>`	**The name of this event rule**
`<rule>`	**Event regular expression**
`<initial>`	**Initial statement**
`<final>`	**Final statement**

(2) Regular expression definition:

`<fname>`	**Function; the return value is *boolean***
`(r)`	**Grouping; bypass default binding**
$r_1\ r_2$	**Sequence; r_1 followed by r_2**
r_1 `|` r_2	**Choice; either r_1 or r_2**
`r+`	**Closure; one or more repetitions of r**

(3) Function definition:

`bool <fname> ( evin_t ev, evarg_t *arg ){}`	
`<fname>`	**Function name**
`ev:`	**Input event**
`arg:`	**Data shared among functions**

Fig. 2. C-based event language.

the PATTERN clause of SASE+ [2] and the MATCH-RECOGNIZE clause of the ANSI draft [3] served as references.

A. Language Overview

Fig. 2 presents an overview of our C-based event language. The language has three user-defined data structures: evin_t, evout_t, and evarg_t. While structure evin_t contains tuples of an input event, structure evout_t contains tuples of an event that is output when a specified regular expression matches a sequence of events. Structure evarg_t contains data shared among functions. Since arrays can be used in structure evarg_t, it can use array elements to store values contained in individual streams, for the purpose of stream partitioning.

A user simply writes an EVENT_RULE macro that has four arguments: <rname>, <rule>, <initial>, and <final>. <rname> indicates the name of the event rule. <initial> is an initial statement that sets initial values in an argument used in the first function used. <final> is a final statement that outputs the results obtained in the last function used as an output event. <rule> describes a regular expression to be used for event processing. Like traditional regular expression syntaxes, ours has four basic rules: grouping (), sequence r_1r_2, choice $r_1|r_2$, and (Kleene) closure $r+$. Here, zero or more repetitions of r, known as $r*$, can be expressed with a combination of a choice and a closure (e.g., $ab* = a|ab+$).

```
typedef struct { uint32_t price; uint32_t time } evin_t;
typedef evin_t  evout_t;
typedef evout_t evarg_t;

EVENT_RULE("Smoothing", "F0 F1 F1 F2",
           "evarg.price = evarg.time = 0;",
           "evout.price = evarg.price;
            evout.time  = evarg.time;");

bool                              bool
F0(evin_t ev, evarg_t *arg)       F2(evin_t ev, evarg_t *arg)
{                                 {
  arg->price = ev.price;            arg->price =
  arg->time  = ev.time;             (arg->price + ev.price)/4;
  return 1                          return 1
}                                 }

bool
F1(evin_t ev, evarg_t *arg)
{
  arg->price += ev.price;
  return 1
}
```

Fig. 3. Example source code in smoothing operation.

The elements of regular expressions are C-based functions, referred to as <fname>, whose return values are Boolean. Each function has two arguments: an input event (ev) and a pointer to shared data (arg). It uses the two to return true when a specified condition is a match. An input event is given to all functions at the same time. When all return values of functions invoked along a regular expression are true, the entire regular expression is a match. The most important point here is that changed arg values will propagate along a specified regular expression, with each function (starting with the second function) in the chain receiving values that have been modified by the previous function.

Fig. 3 shows an example source code in a smoothing operation. In this example, input event ev, output event evout, and argument arg each contain two data items: time and price. Here, we use a function-based regular expression F0F1F2F3 in order to calculate a moving average of four stock prices as a smoothing operation. Function F0 stores the time and price of the current event in argument arg; function F1 adds the price of the current event to that of the previous event; and function F2 calculates a moving average, dividing the sum of the prices of the current event and the previous event by 4. Here, function F1 is used twice. In a change-point analysis operation [8], input event ev and output event evout each contain two data items: time and price. Argument arg contains three data items: trend (for the direction of the smoothed polygonal line), last_time, and last_price (i.e., time and price data for the previous event). Here, we use a function-based regular expression (C0|C1)+C2 in order to detect local maximums and minimums for our change-point analysis. Function C0 stores the current event to argument arg when the price of the event is equal or greater than that of the previous event. Function C1 does the same operation when the price of the current event is equal or less than that of the previous event. The closure of either function C0 or C1 will result in repeated invoking of corresponding functions. Function C2 either outputs a local maximum when the price of a current event is less than that of the previous event after function C0 has been invoked, or outputs a local minimum when the price of a current event is greater than that of the previous event after function C1 has been invoked.

B. Logic Construction

Our novel method makes it possible for the event-processing logic described in our event language to be systematically constructed with four rules (see Fig. 4). The idea behind it is the logic synthesis of regular expressions with data paths.

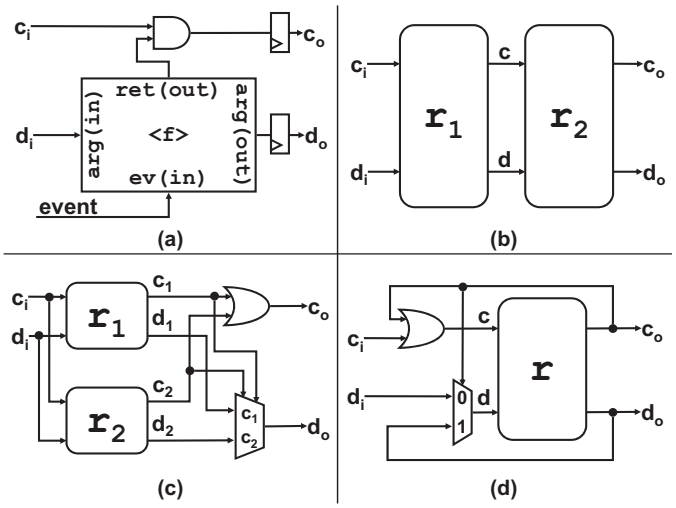

Fig. 4. Logic construction method. (a) Function: $<f>$. (b) Sequence: $r_1 r_2$. (c) Choice: $r_1|r_2$. (d) Closure: $r+$.

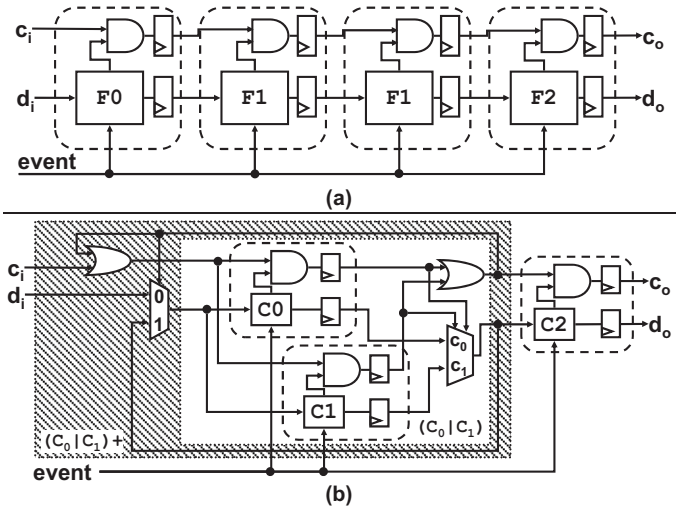

Fig. 5. Logic examples for smoothing and change-point analysis. (a) Smoothing. (b) Change point analysis.

In other words, while our method constructs a nondeterministic finite automaton (NFA) structure for return values of defined functions by using methods based on those in [9], it correctly connects arguments among defined functions on the NFA structure in a pipelined way. Grouping requires no construction rules since it changes only the operator bindings.

Function $<f>$ [see Fig. 4(a)] is simply replaced by a synthesizable function used in a behavioral description language. A logical AND operation is performed on the return value of function $<f>$ and the value c_i of the previous control path, yielding, via a flip-flop, the value c_o for the next control path. Function $<f>$ inputs an event and the value d_i of the previous data path, and outputs, via a flip-flop, calculated data as the value d_o for the next data path.

Sequence $r_1 r_2$ [see Fig. 4(b)] simply connects: 1) the values c_i and d_i of the previous control and data paths to the corresponding input values of rule r_1; 2) two output values c and d of rule r_1 to the corresponding input values of rule r_2; and 3) the two output values of rule r_2 to the values c_o and d_o for the next control and data paths. This construction rule is equivalent to an assignment statement in C.

Choice $r_1|r_2$ [see Fig. 4(c)] connects the values c_i and d_i of the previous control and data paths to the corresponding input values in rules r_1 and r_2. A logical OR operation on the value c_1 of the control path of rule r_1 and the value c_2 of the control path of rule r_2 is performed, yielding the value c_o for the next control path. The value d_o for the next data path is selected from either d_1 or d_2 on the basis of values c_1 or c_2 of the control paths of rules r_1 and r_2. Although the number of selection options is considerable, our current option is as follows: if c_2 is true, d_o is d_2. Otherwise, d_o is d_1. This construction rule is equivalent to an if statement in C.

Closure $r+$ [see Fig. 4(d)] connects two output values of rule r to the values c_o and d_o for the next control and data paths. Although the number of repetition options is considerable, our repetition option employs the longest match rule: if c_o is true, the input values c and d of rule r are connected to the values c_o and d_o for the next control and data paths. Otherwise, they are connected to the values c_i and d_i of the previous control and data paths. This construction rule is equivalent to a while statement in C.

Fig. 5 illustrates the synthesized logic of smoothing and change-point analysis operations. In the smoothing operation [see Fig. 5(a)], functions F0, F1, and F2 are simply connected in series, using the sequence construction rule along a function-based regular expression F0F1F1F2. In the change-point analysis operation [see Fig. 5(b)], functions C0, C1, and C2 are connected with the sequence, choice, and closure construction rules along a function-based regular expression (C0|C1) + C2. In both examples, the initial value c_i of the control path is always true, and the initial value d_i of the data path is assigned by initial statement <initial>. Moreover, when c_o is true, the final value d_o of the data path is assigned to an output event evout by final statement <final>. While high-level synthesis tools help optimize the contents of C functions included in a function-based regular expression, our method connects the functions along the regular expression in a pipelined way, as shown in Figs. 4 and 5. It is difficult for industry tools to support the same functionality because our function-based regular expressions are beyond the scope of the ANSI C specification.

IV. Evaluation

We have used an actual 20-Gb/s FPGA-based network interface card (NIC). To the best of our knowledge, this is the first report of an over-10-Gb/s evaluation environment. We have designed an in-house tool that converts a function-based regular expression included in our event language to a C code. Then, we have utilized NEC CyberWorkBench in order to compile both the generated C code and C functions to HDL codes. In addition, we have used Xilinx ISE 11.4 in order to compile the HDL codes.

A. Implementation

We have implemented both smoothing and change-point analysis operations with all necessary peripheral circuits on our target NIC. For verification, we have used the NEC Japan stock information (see Section III) as events. The date of each event is replaced by a sequence number in order to continuously input events at the cycle level. As shown in Fig. 6, the logic successfully detects five change points on a server. The number of occupied slices with respect to the operations increases only 2.7% while the peripheral circuits occupy 30% on the FPGA.

B. Performance Speedup

We have evaluated how much our FPGA framework makes it possible to achieve better performance than does CPU software in

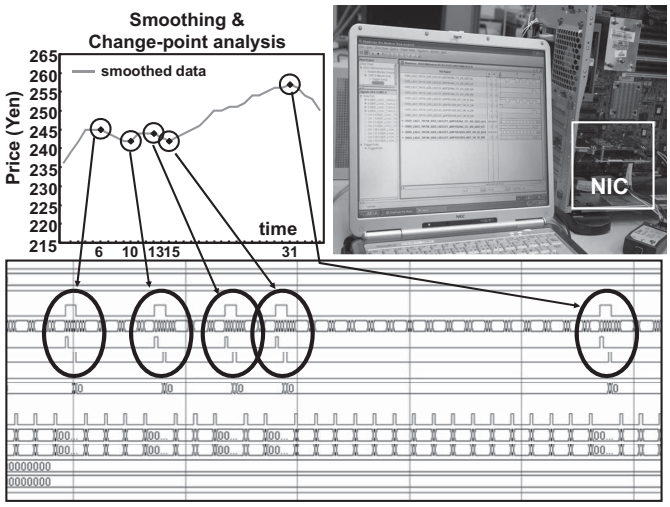

Fig. 6. Implementation of our motivating example.

our motivating example. In order to measure the naive CPU software performance, we have newly implemented C source codes that directly execute the operations of our motivating example on an OS. In addition, for reference, we have evaluated a simple, famous trading benchmark, referred to as volume-weighted average price (VWAP) [10], which means the ratio of the price traded to the total volume traded over a particular time. This benchmark simply requires an aggregation function without any regular expressions. In this evaluation, we have used an Intel Xeon CPU running at 3.3 GHz with a 12-MB cache. Compared with CPU software, our work achieves 12.3 times better performance with respect to our motivating example and 7.2 times better with respect to the VWAP benchmark. These speedups come from the fact that our C-based event language enables the two applications to be directly mapped to an FPGA in a pipelined way. It should be noted that, for fair comparison, CPU software performance is measured without any data transfer latency from an NIC since the data transfer latency of a few microseconds results in significant overheads in CPU software.

V. Conclusion

The requirements for fast CEP will necessitate hardware acceleration that uses reconfigurable devices. Key to the success of our work is logic automation generated with our C-based event language. With this language, we have achieved both higher event-processing performance and higher flexibility for application designs than those with SQL-based CEP systems. In a financial trading application, we have, in fact, achieved 12.3 times better event-processing performance on an FPGA-based NIC than does CPU software. In future work, we intend to implement complicated examples that require support for multiple streams.

References

[1] S. Charkravarthy and Q. Jiang, *Stream Data Processing: A Quality of Service Perspective*. New York: Springer-Verlag, Apr. 2009.

[2] D. Gyllstrom, J. Agrawal, Y. Diao, and N. Immerman, "On supporting Kleene closure over event streams," in *Proc. IEEE 24th Int. Conf. Data Eng.*, Apr. 2008, pp. 1391–1393.

[3] F. Zemke, A. Witkowski, and M. Cherniak, "Pattern matching in sequences of rows," American National Standards Institute, Washington, DC, Tech. Rep. ANSI NCITS H2-2006-nnn, Mar. 2007.

[4] M. R. Mendes, P. Bizarro, and P. Marques, "A performance study of event processing systems," in *Performance Evaluation and Benchmarking*, vol. 5895. New York: Springer-Verlag, 2009, pp. 221–236.

[5] OPRA. (2011, Jan. 19). *Updated Traffic Projections 2011 & 2012*, Chicago, IL [Online]. Available: http://www.opradata.com/specs/upd_traffic_proj_11_12.pdf

[6] R. Mueller, J. Teubner, and G. Alonso, "Streams over wires—A query compiler for FPGAs," in *Proc. Int. Conf. Very Large Data Bases*, vol. 2. Aug. 2009, pp. 229–240.

[7] L. Woods, J. Teubner, and G. Alonso, "Complex event detection at wire speed with FPGAs," in *Proc. Int. Conf. Very Large Data Bases*, vol. 3. Sep. 2010, pp. 660–669.

[8] H. Inoue, T. Takenaka, and M. Motomura, "20 Gb/s C-based complex event processing," in *Proc. IEEE Int. Conf. Field Program. Logic Appl.*, Sep. 2011, pp. 97–102.

[9] R. Sidhu and V. Prasanna, "Fast regular expression matching using FPGAs," in *Proc. IEEE Symp. Field-Program. Custom Comput. Mach.*, Apr. 2001, pp. 227–238.

[10] B. Johnson, *Algorithmic Trading and DMA: An Introduction to Direct Access Trading Strategies*. Myeloma, U.K.: Myeloma Press, Feb. 2010.

Reduced-Complexity LCC Reed–Solomon Decoder Based on Unified Syndrome Computation

Wei Zhang, Hao Wang, and Boyang Pan

***Abstract*— Reed–Solomon (RS) codes are widely used in digital communication and storage systems. Algebraic soft-decision decoding (ASD) of RS codes can obtain significant coding gain over the hard-decision decoding (HDD). Compared with other ASD algorithms, the low-complexity Chase (LCC) decoding algorithm needs less computation complexity with similar or higher coding gain. Besides employing complicated interpolation algorithm, the LCC decoding can also be implemented based on the HDD. However, the previous syndrome computation for 2^{η} test vectors and the key equation solver (KES) in the HDD requires long latency and remarkable hardware. In this brief, a unified syndrome computation algorithm and the corresponding architecture are proposed. Cooperating with the KES in the reduced inversion-free Berlekamp-Messy algorithm, the reduced-complexity LCC RS decoder can speed up by 57% and the area will be reduced to 62% compared with the original design for $\eta = 3$.**

***Index Terms*— Algebraic soft-decision (ASD) decoder, low-complexity Chase (LCC) decoding, Reed–Solomon (RS) codes, unified syndrome computation (USC).**

I. Introduction

Reed–Solomon (RS) codes are widely employed as the error control code in numerous digital communication and storage systems. Berlekamp developed the first practical decoding procedure for RS codes in 1968 [1]. In recent years, the error control capability was improved by Koetter and Vardy [2], by incorporating the reliability information from the channel into the algebraic soft-decision (ASD) decoding process. Among all ASD algorithms, the low-complexity Chase (LCC) decoding needs to interpolate 2^{η} test vectors with maximum multiplicity one [3]. Hence, the LCC decoding has less computation complexity and similar or higher coding gain compared with other ASD algorithms.

Usually, the interpolation is the common method in the LCC decoding algorithm to get the error locator and the evaluator polynomial. By applying the re-encoding and the coordinate transformation technique, the number of points needs to be interpolated can be reduced to $n-k$ for an (n, k) RS code [4]–[6]. Much work has been

Manuscript received January 2, 2012; revised March 16, 2012; accepted April 22, 2012. Date of publication May 30, 2012; date of current version April 22, 2013.

The authors are with the School of Electronic Information Engineering, Tianjin University, Tianjin 300222, China (e-mail: tjuzhangwei@tju.edu.cn; tjuwanghao@tju.edu.cn; panboyang@tju.edu.cn).

Digital Object Identifier 10.1109/TVLSI.2012.2197030