High Level Synthesis with Stream Query to C Parser:

Eliminating Hardware Development Difficulties for Software Developers

†Eric Shun Fukuda ‡Takashi Takenaka ‡Hiroaki Inoue
*Hideyuki Kawashima †Tetsuya Asai †Masato Motomura

†Graduate School of Information Science and Technology
Hokkaido University
Sapporo, Hokkaido, 060–0814 Japan
{fukuda@lalsie., motomura@, asai@}ist.hokudai.ac.jp
‡NEC Corporation
Kawasaki, Kanagawa, 211–8666 Japan
{takenaka@aj, h-inoue@ce}.jp.nec.com
*Graduate School of Systems and Information Engineering
University of Tsukuba
Tsukuba, Ibaraki, 305–8573 Japan
kawasima@cs.tsukuba.ac.jp

Abstract— Recently, reconfigurable hardware is attracting wide attention as a stream processing platform for its high performance and power efficiency. To allow many software engineers to benefit from reconfigurable hardware, high level synthesis tools have been actively developed. Although these tools have enormously reduced the amount of work and difficulties, the users still need hardware development knowledge. In this paper, we introduce a method that parses SQL queries into high-level-synthesis-intended C codes. Our experiments using a dynamically reconfigurable hardware that features a high level synthesis tool showed that the hardware's potential was fully extracted and the developer writing the SQL queries does not need hardware development knowledge.

I. Introduction

The amount of information over the Internet has been explosively increasing in recent years. To deal with this so-called "Infoplosion [3]," parallel distributed computing has been effectively adopted. However, the ever-rising power consumption of parallel distributed computing has become a serious and immediate issue.

In view of this situation, reconfigurable hardware is attracting attention as a solution for reducing the power consumption[6, 4]. Although using a dedicated hardware for a specific task has always been effective for reducing the power consumption, the application was limited due to its high manufacturing cost. Now that reconfigurable hardware lowered the cost for developing dedicated hardware dramatically, it is applied to many kinds of tasks.

Stream processing is one of the fields that dedicated hardware can be highly effective. Since stream processing requires continuous observation over information streams, and streams often have a very high bit rate, advantages of dedicated hardware such as high throughput and low power consumption is highly beneficial for stream processing. Therefore, stream processing by reconfigurable hardware is one of the active research fields recently.

A major problem of reconfigurable hardware accelerated stream processing is that it is difficult for software engineers to use. Using a reconfigurable hardware requires hardware development knowledge which is entirely different from software development knowledge. Generally, however, complicated algorithms for stream processing are developed by software engineers. Therefore, in order for reconfigurable hardware acceleration for stream processing to be widely used, it is essential to enable software engineers to design hardware.

High level synthesis (HLS) is one of such technologies. It reduces the development cost of hardware by synthesizing hardware configurations from software code such as C. However, as HLS design experts are fairly well aware of, and as revealed in detail in a recent study [1], hardware development knowledge is still needed even when using HLS tools.

Another approach to enable software engineers to design hardware is to use a language that is more specific to an application such as SQL-based stream language [6]. Although this approach restricts the application of the product hardware, the developer needs less knowledge about hardware development compared to HLS. Since this approach is intended to compile SQL queries directly to

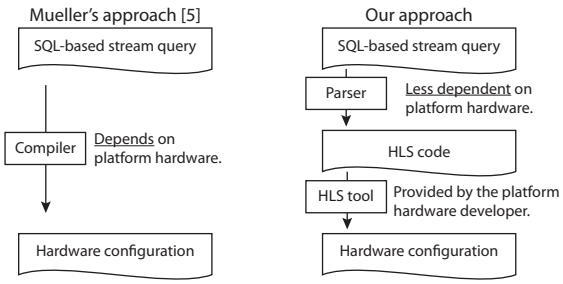


Fig. 1. Comparison of conventional and our approaches.

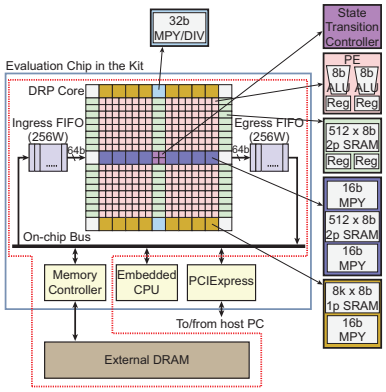


Fig. 2. Architecture of DRP.

hardware configurations, however, developing such compiler would take much effort to support various reconfigurable hardware.

In order to overcome these difficulties of HLS and SQL-to-hardware compiler, a method that uses both of these in combination has been proposed [7]. This approach first uses a parser that converts an SQL query to an HLS code written in C, and then uses an HLS tool that compiles the code to a hardware configuration. Although this method was proposed as a part of a larger system that uses an FPGA, it successfully reduced the workload of development of SQL-to-hardware compiler by giving over the hardware configuration process to the HLS tool, rather than doing it by hand.

Since the HLS tool undertakes the hardware specific configuration, this method should be able to be applied to other hardware. Therefore, in this paper, we try to apply this method to another reconfigurable hardware, Dynamically Reconfigurable Processor (DRP), and focus on compiling basic SQL queries that were used in [6] as our primary evaluation.

Our contributions in this paper are as follows:

- We evaluate how well the SQL-to-C parser extracts DRP's potential.
- We verify whether the SQL-to-C parser provides SQL-to-hardware compiler with portability to DRP by using it in combination with an HLS tool.
- We point out what should developers of SQL-to-C parser be aware of when porting the parser to another environment.

Schema:

```
CREATE INPUT STREAM Trades (
  Seqnr  int,         -- sequence number
  Symbol string(4),   -- valor symbol
  Price  int,         -- stock price
  Volume int)         -- trade volume
```

Q_1:

```
SELECT Price,Volume
  FROM Trades
 WHERE Symbol="UBSN"
  INTO UBSTrades
```

Q_2:

```
SELECT Price,Volume
  FROM Trades
 WHERE Symbol="UBSN" AND Volume>100000
  INTO LargeUBSTrades
```

Q_3:

```
SELECT count() AS Number
  FROM Trades [SIZE 600 ADVANCE 60 TIME]
 WHERE Symbol="UBSN"
  INTO NumUBSTrades
```

Q_4:

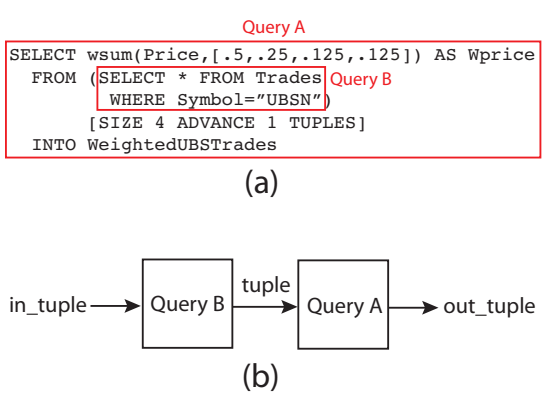
```
SELECT wsum(Price,[.5,.25,.125,.125] AS Wprice
  FROM (SELECT * FROM Trades
          WHERE Symbol="UBSN")
       [SIZE 4 ADVANCE 1 TUPLES]
  INTO WeightedUBSTrades
```

Fig. 3. Example queries and schema of incoming stream.

II. Related Work

Mueller et al. proposed a system called *Glacier*, which compiles SQL-based stream queries to high-throughput hardware configurations [6]. This work took five basic queries (four of which, Q_1 to Q_4, are listed in Fig. 3) as application examples, and used FPGA as its hardware platform. It essentially proposed how to "map" each of SQL primitives to a corresponding hardware template, and then connects them as they are specified in queries provided to the system. This idea was extended in many ways and became the basis of several related works such as [4].

One of the authors of this paper, on the other hand, has proposed an advanced framework for compiling SQL based continuous query with user-defined C/C++ functions. The system realizes 20Gbps bit stream processing on FPGA [2] and exploits HLS not only for compiling the user-defined C/C++ functions but also the C codes that are parsed from standard SQL queries by their original parser. Our work tries to apply this parsing method to DRP, jointly using the HLS tool bundled to it.




```
Trades queryA (Trades in_tuple)
{
  tuple = some_query_processingA(in_tuple);
  return tuple;
}

Trades queryB (Trades in_tuple)
{
  tuple = some_query_processingB(in_tuple);
  return queryA(tuple);
}

void main()
{
  /* Pipeline loop */
  while (1) {
    Trades in_tuple = receive_tuple();
    out_tuple = queryB(in_tuple);
    send_tuple(tuple)
  }
}
```

(c)

Fig. 4. C function calls for nested queries.

III. Evaluation Platform

As a case study, we used Dynamically Reconfigurable Processor (DRP), a processor that is commercially available [5]. DRP's architecture is shown in Fig. 2. It has a dedicated development environment which features an HLS tool that enables developers to design hardware in C. We chose DRP as our evaluation platform for two reasons: it features a state-of-the-art HLS tool which makes DRP probably the most easiest hardware platform for software engineers to utilize, and its evaluation kit is provided as an ExpressCard which allows handy evaluation.

IV. SQL-based Stream Processing Language

In our work, we use SQL-based stream processing language as an application description language. Fig. 3 shows some example queries. The schema in Fig. 3 specifies the fields that stream named "Trades" has. The tuples from the stream are processed according to the queries (Fig. 3, Q_1 to Q_4) which consists of the following clauses:

- `SELECT` clause specifies the fields of the outgoing stream.
- `FROM` clause specifies the input stream.
- `WHERE` clause specifies the conditions for selecting the tuples.
- `INTO` clause specifies the name of the outgoing stream.

Additionally, some queries have aggregation functions in their `SELECT` clauses (Q_3 and Q_4 in Fig. 3). Aggregation functions calculate some measures from a certain range of tuples in the stream, which is specified by a window written in the `FROM` clause with its size and sliding interval (e.g. Q_3 counts the number of tuples whose symbol is "UBSN" within 600 seconds, and Q_4 calculates the weighted sum of the stock prices from the previous four tuples).

V. Our Approach

Generally, hardware development procedure can be divided into two stages; the first is to fix the processing architecture which involves specifying the I/O or where to pipeline, and the second is to arrange the wires, registers and memories so that the circuit meets the requirements such as delay or resource amount.

As well as [7] the parser converts SQL to HLS code written in C, and then the generated HLS code is compiled to hardware configuration by HLS tool (Fig. 1). In other words, the compilation process is divided into two stages:

1. Parser: specifies the architecture that is suited to SQL-based stream processing.
2. HLS tool: synthesizes and optimizes hardware configurations.

In this work, we use an existing HLS tool that is bundled with DRP, therefore hereafter in this paper, we look only into the parser.

A. Generalization of Queries into Abstract Hardware

We first generalize the query structure to abstract hardware modules in order to allow any queries to be converted into hardware configurations.

First of all, whether there are any sub queries inside the query or not is determined. To find a sub query, we look inside the `FROM` clause. If there is a sub query (Fig. 4a), there is going to be two query modules between the input and output (Fig. 4b), otherwise, there will be only one query module.

The query module is described in Fig. 5a. It consists of three parts, selection module, slide timing module and aggregation module. However, if the query does not use aggregation, the query module only has the selection module. The output of selection module is sent out as an output tuple in such case (e.g. Q_1 and Q_2). The selection module asserts the selected signal if the

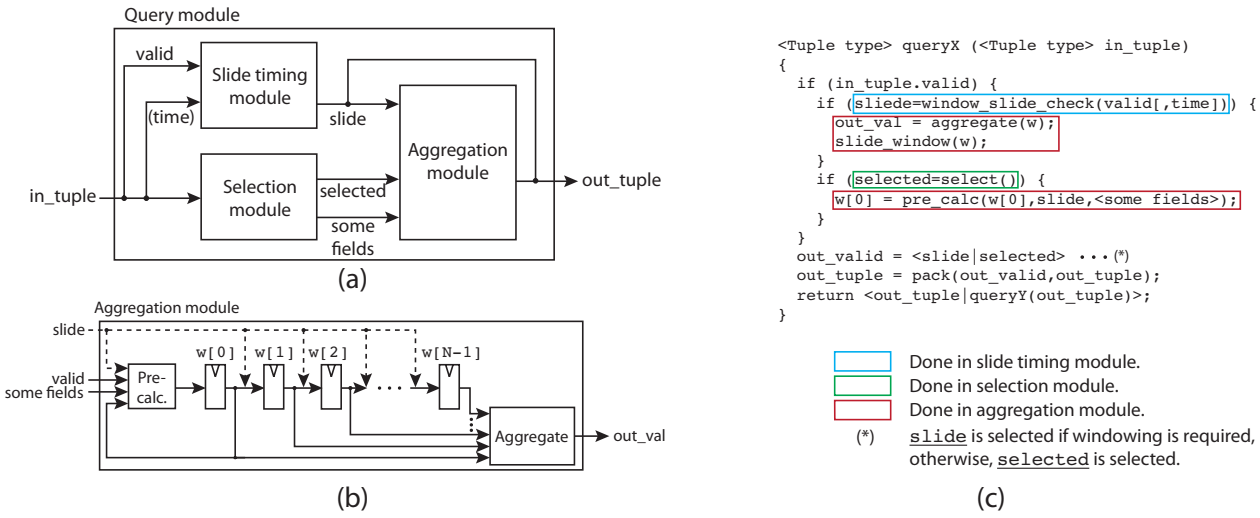


Fig. 5. Query module architecture and its code.

tuple satisfies the conditions specified in `WHERE` clause (e.g. `Symbol="UBSN" AND Volume>100000`, in Q_2), or otherwise negate it. The slide timing module monitors the incoming tuples and notifies the aggregation module of the slide timing by asserting the "slide" signal. The slide timing can be detected by counting the valid tuples, when tuple-based windowing is used, or by monitoring the timestamp of the incoming tuples, when time-based windowing is used.

The aggregation module consists of three components, a shift register (`w[0]` to `w[N-1]`), an aggregation unit and a pre-calculation unit (Fig. 5b). The shift register holds the values that are used by the aggregation unit. The number of registers is determined by dividing the window size by sliding interval. When the registers receive "slide" signal from the slide timing module, each register sends its content value to the next register. The aggregation unit collects the values from the shift register and outputs the aggregation result. Pre-calculation unit updates the content of `w[0]` whenever there is a valid input tuple. What the pre-calculation unit does depends on the aggregation specified in the query. For example, when `count(·)` is specified (Q_3), the pre-calculation unit increments the value in `w[0]`, initializing it by zero when "slide" is asserted. Aggregation unit, then, collects the values in the registers and adds them up.

B. Mapping Abstract Hardware to C Code

In the generated C code, the main function first calls the function that corresponds to the most inner query (Fig. 4c), giving the input tuple that was received in `receive_tuple(·)`. The query function calls the query function that corresponds to the next inner query in the return statement after some operations to the query. The most outer query function returns the tuple itself after some operations to it. Finally, when the returned tuple reaches the main function, it becomes an output (`send_tuple(·)`).

The functionalities of the query modules are mapped to C code as shown in Fig. 5c. The operations such as slide timing module checking the timing and asserting the "slide" signal, or selection module checking the conditions specified in the `WHERE` clause of the query and asserting the "selected" signal, are mapped as conditions in **if** statements. This is because whether the operations in aggregation units are executed depends on these conditions, and therefore, the operations that depend on these conditions are executed under the corresponding **if** statements. If the query does not require windowing, the operations of slide timing module and aggregation module are omitted in the code.

C. Shallow Hardware Optimization in C Code

As shown in Fig. 4, there is a pipelining directive just before the while loop in the main function. Thanks to the HLS tool, all a parser developer has to do in order to pipeline the loop is to write this directive. However, before we pipeline the loop, some preparations must be done to the code. First, loops cannot have inner loops. We use another kind of directive to unroll the inner loops. This directive is also specified right before the loop, and the HLS tool will automatically unrolls the loop. The other preparation for the loop is to design the hardware architecture, which is already done in the abstract hardware, so that it can be efficiently pipelined. In our code, there is a "valid" flag in the tuple with the intention of doing this. The "valid" flag enables the hardware to receive and send tuples at a constant speed, which leads to making the pipelined loop efficient.

Another optimization done in the code is to burst access the memory. The DRP evaluation kit we used not having an network interface, it has to send and receive tuples to and from the external DRAM. The HLS tool we used has a featured function (which is hidden in `receive_tuple(·)` and `send_tuple(·)`) to do this. This feature requires a predictive numbers of input and output tuples, therefore the optimization we made in order to efficiently pipeline also profit this optimization. Note that this is the only optimization we made that is specific to DRP.

VI. Evaluation

We compared the performance of the codes that were directly written in HLS C, and the codes generated by our parser from Q_1 to Q_4 (Fig. 6). We did not optimize the codes directly written in HLS C because optimization would be difficult for software engineers who generally do not have hardware development skills. Since the intended users of our parser are those who do not have hardware development knowledge, it is fairer to compare with non optimized HLS codes.

Synthesis was done by *DRP tool* which is a development tool suite bundled with the DRP evaluation kit. The synthesis tool included in *DRP tool* is based on CyberWorkBench which was developed by NEC [9]. The generated C codes are fully capable of synthesizing into hardware configurations unless the resource usage exceeds what is available on DRP[8]. DRP can be driven at various clock speeds and DRP tool has a functionality to search the optimal clock speed for the application. Each result shown in Fig. 6 was measured at such clock speed.

For reference, we measured the throughput of Intel Core i5-2520M processor (2.5GHz) running Q_1 (horizontal line in Fig. 6). The figure shows that the throughput of HLS C codes written without hardware development knowledge is about the same as that of the CPU. When using our parser, the throughput was more than twice as fast as the CPU. However, when we consider the power efficiency, assuming that the DRP consumes 500 mW and CPU 5 W, the DRP was 24 times more efficient than the CPU.

As long as the queries consist of `SELECT`, `FROM`, `WHERE`, and `INTO` clauses and the hardware resources are sufficient, the DRP's throughput will remain as high as shown in Fig. 6 because the system can be pipelined as shown in Fig. 4. However, when a query contains `GROUP BY` or `JOIN` clauses, which are outside the scope of this paper, the throughput tends to go down vastly. [6] suffers from reduction of throughput when dealing with `GROUP BY` clause because it requires a CAM to implement a grouping functionality. `JOIN` is a very difficult operation that many works have been seeking its efficient implementation on hardware[1, 4, 6]. Building translation functionalities of these clauses into our system is an important part of our future work.

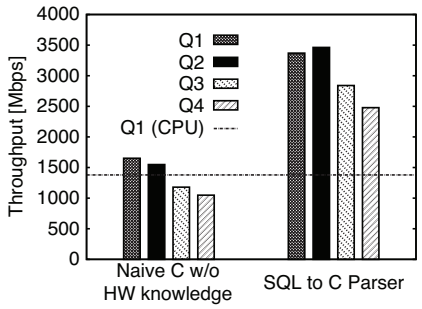


Fig. 6. Throughput comparison between naive C written without hardware development knowledge and parsed C.

VII. Discussion

One of the metrics that signifies the usage of a parser, besides the absolute performance of the resulting hardware, is how well the hardware's potential is extracted. To evaluate such significance, we measured the bandwidth usage of the generated hardware. Table I shows how efficiently the input bandwidth of DRP was used. Input bandwidth is the most critical limitation that constrains the overall throughput. Therefore, the input bandwidth usage can be a good barometer for evaluating how efficient the hardware is used. According to the table, over 90% of the potential of DRP was extracted by using the parser, whereas only 50% of the potential was extracted without it.

The cost-consuming and hardware dependent low level operations such as wiring or scheduling were delegated to the HLS tool. There were only three optimizations that had to be done by the parser that involve hardware development knowledge: 1) specifying which loop to be pipelined, 2) enabling the memory burst accessing option, and 3) providing the basic architecture in order to pipelining and memory burst accessing to be effective. Among these, the burst memory accessing option was the only optimization that was hardware specific in the architectural level. This means that the parser can be used in various hardware architectures as long as the HLS tool provides the abstracting function for controlling the I/O. Since a controller of the I/O is one of the most difficult component to design in lower levels and therefore requires hardware development knowledge, and highly depends on the hardware architecture, it can be said that the parser is providing a good portability.

VIII. Conclusion

In this paper, we showed that the concept of SQL-to-C HLS based compiler, proposed in [7] using an FPGA, was effective on DRP which features a dynamically reconfigurable architecture. The HLS C code was compiled to hardware configuration by a state-of-the-art proprietary

TABLE I
USAGE OF INPUT BANDWIDTH.

Query	Hand written C	SQL to C Parser
Q_1	49.6%	90.8%
Q_2	50.4%	91.7%
Q_3	49.8%	96.5%
Q_4	33.5%	96.8%

HLS tool that is customized to DRP. The results of the evaluation show that the parser enables software engineers to develop stream processing hardware that is 24 times power efficient than a common CPU, or twice as fast as directly written HLS C code without any hardware development knowledge.

The SQL-to-C parser provided portability to SQL-to-hardware compiler to work on DRP, by delegating the cost-consuming low level optimization of DRP to the HLS tool, and utilizing the high level function for controlling the I/O which is dependent to DRP.

The optimizations except the I/O done by the parser do not depend on DRP at the architectural level as long as the synthesized hardware configuration does not violate the hardware limitation. Therefore, the only aspect of DRP that a developer of a SQL-to-C parser should be aware of is the I/O. This tendency can be generalized to various kinds of reconfigurable hardware.

The limitation of this work is that the sample queries we evaluated were rather simple. We will improve our parser and consider larger or more complex queries that include grouping functionality.

ACKNOWLEDGEMENT

We are deeply grateful to Koichiro Furuta, Taro Fujii, Takeshi Inuo, and Takao Toi at Renesas Electronics Corporation for their helpful discussion and support. This work was partially supported by JSPS Grant-in-Aid for Challenging Exploratory Research (No. 24650033).

REFERENCES

[1] E. S. Fukuda, H. Kawashima, H. Inoue, T. Fujii, K. Furuta, T. Asai, and M. Motomura. C-based adaptive stream processing on dynamically reconfigurable hardware: a case study on window join. In *Proceedings of the 9th international conference on Reconfigurable Computing: architectures, tools, and applications (ARC)*, 2013.

[2] H. Inoue, T. Takenaka, and M. Motomura. 20Gbps C-based complex event processing. In *Proceedings of the 2011 21st International Conference on Field Programmable Logic and Applications (FPL)*, 2011.

[3] M. Kitsuregawa. Challenge for info-plosion. In *Proceedings of the 18th international conference on Algorithmic Learning Theory (ALT)*, 2007.

[4] T. Miyoshi, H. Kawashima, Y. Terada, and T. Yoshinaga. A coarse grain reconfigurable processor architecture for stream processing engine. In *Proceedings of the 2011 21st International Conference on Field Programmable Logic and Applications (FPL)*, 2011.

[5] M. Motomura. A dynamically reconfigurable processor architecture. *Microprocessor Forum*, 2002.

[6] R. Mueller, J. Teubner, and G. Alonso. Streams on wires - a query compiler for FPGAs. *Proceedings of the VLDB Endowment*, Vol. 2, No. 1, 2009.

[7] T. Takenaka, M. Takagi, and H. Inoue. A scalable complex event processing framework for combination of SQL-based continuous queries and C/C++ functions. In *Proceedings of the 2012 22nd International Conference on Field Programmable Logic and Applications (FPL)*, 2012.

[8] T. Toi, T. Awashima, M. Motomura, and H. Amano. Time and space-multiplexed compilation challenge for dynamically reconfigurable processors. In *IEEE International Midwest Symposium on Circuits and Systems*, 2011.

[9] K. Wakabayashi and B. C. Schafer. *"All-in-C" Behavioral Synthesis and Verification with CyberWorkBench*, pp. 113–127. Springer Netherlands, 2008.