Introducing a New Language for Stream Applications

Mohamad Dabbagh
Department of Computer Science
Shabestar Branch
Islamic Azad University
Shabestar, Iran
mohamad_dbgh@yahoo.com

***Abstract*— Stream programs represent an important class of high-performance computing. These programs are rich in parallelism and can be naturally targeted to distributed and multi-core architectures. Since computer architectures gradually become multi-core, there is a pressing need to provide an efficient programming language that supports all aspects of parallelism in the streaming applications. In this paper, we introduce a new flexible stream programming language, called FSPL. The FSPL, is an architecture-independent programming language designed for high-performance streaming application development. It aims to improve programmer productivity and program efficiency within the streaming domain. In the FSPL language, each program is a collection of independent filters which communicate by the means of data channels. This model lends itself naturally to concurrent and efficient implementations on modern multiprocessors. One of most significant features supported in FSPL is that when you define a filter, it is not needed to specify the amount of data produced and consumed by that filter.**

Keywords: Data channel; filters; FSPL; Stream Programming;

I. INTRODUCTION

As applications designed around some notion of streams continue to increase, there is a need to provide better software support in the form of flexible languages modeled around the concept of streams of data. Using the notion of streams makes it possible for programmers to structure programs in ways that provides the compiler with enough information about parallelism, program- and data-flows, which the compiler can make use of in order to produce efficient translations to parallel machines.

By streams we mean the continuous one directional flow of data in any format as shown in "Fig. 1" and streaming applications are applications that process single or multiple streams in the incoming and outgoing directions. Examples include digital signal processing in radar systems, or in a software FM radio [1], communication protocols, speech coders, audio beam-forming, image processing, cryptographic kernels, and network processing.

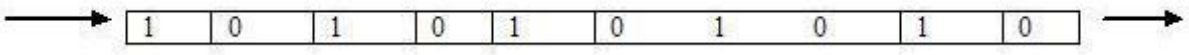

Figure 1. Binary streams of 1's and 0's in the direction of the arrow

Streaming applications are generally compute intensive and demand real time processing of the data they receive thereby, making it imperative that processing is done in the most efficient manner. They are suitably mapped to parallel architectures where most memory operations are localized in the processing units and the notion of global variables do not exist. This is very difficult to deal with in languages that schedule tasks sequentially. Most of the high level languages in use today fall into this category e.g. C, C++ etc. They are optimized for general purpose application programming on machines that have centralized memory architectures and therefore need a great deal of difficult programming to express the parallelism that is inherent in today's parallel architectures.

Using them on parallel and reconfigurable architectures with distributed memory systems, limits the gains in efficiency, speed and lower power consumption that are expected from these architectures. Moreover, general purpose languages do not provide a natural and intuitive representation of streams, thereby having a negative effect on readability, flexibility, and programmer productivity.

These problems have made it necessary to rely on hardware implementations, like application specific integrated circuits (ASIC), field programmable gate arrays (FPGA) and special purpose digital signal processing (DSP) hardware, in areas that require industrial high performance processing. However, these come with a major shortcoming, namely, lack of flexibility [2].

For instance, in third generation (3G) radio base stations, considering the expected long life cycles that are often required, it is not desirable to encapsulate critical functions into hardware since minor changes could require a complete removal of the hardware modules involved. Software modules make it possible to integrate added functionalities or improve on algorithms with minimal hardware changes [2]. Therefore, to reduce cost, speed up design time and improve on the ability to provide for the customer's various needs while reducing down time required for upgrades, there is need to use commercially available reconfigurable processors while developing compilers that can exploit the parallelism that these processors provide and at the same time avoiding performance penalties.

On the other hand, despite the prevalence of stream applications, there is surprisingly little language and compiler support for practical, large-scale stream programming. Of course, the notion of a stream as a programming abstraction has been around for decade, and a number of special-purpose stream languages have been designed. Many of these languages and representations are elegant and theoretically sound, but they often lack features and are too inflexible to support straightforward development of modern stream applications, or their implementations are too inefficient to use in practice. In a stream programming language, a program is a collection of filters connected by data channels. Each filter is a functional unit that consumes data from its input channels and produces results on its output channels. In their purest form, stream programming languages are ideally suited to parallel implementations as the output behavior of a filter is deterministic function of the data on its input channels and its internal state. As filters are independent and isolated from one another, they can be scheduled in parallel without concern about data races or other concurrent programming pitfalls that plague shared memory concurrent programs. The appeal of this model is evidenced by a number of stream programming languages and systems include Borealis [3], Cg [4], StreamIt [5] and Brook [6]. These languages have a long lineage which can be traced back to Wadge and Ashcroft's Lucid [7] data flow language and, to some extent, to the Esterel and Lustre family of synchronous languages [8], [9].

All reasons mentioned above, motivate us to design and implement a Flexible Stream Programming Language, hereafter referred to as FSPL. FSPL is a language and compiler specifically designed for modern stream programming. The FSPL language has a main goal: to provide high-level stream abstractions that improve programmer productivity and program efficiency within the streaming domain.

The remainder of this paper is organized as follows. In section 2, we characterize the domain of streaming programs that motivates the design of FSPL. In section 3, we provide a detailed description of the FSPL language. We present a detailed example of a lexical analyzer implemented with FSPL in section 4 and describe related work in section 5 and finally section 6 contains our conclusions.

II. STREAMING APPLICATION DOMAIN

The applications that make use of a stream abstraction are diverse, with targets ranging from embedded devices, to consumer desktops, to high-performance servers. However, we have observed a number of properties that these programs have in common-enough so as to characterize them as belonging to a distinct class of programs, which we will refer to as streaming applications. The following are the salient properties of a streaming application, independent of its implementation:

- Large streams of data. Perhaps the most fundamental aspect of a stream program is that it operates on a large (or virtually infinite) sequence of data items, hereafter referred to as a data stream. Data streams generally enter the program from some external source, and each data item is processed for a limited time before being discarded. This is in contrast to scientific codes, which manipulate a fixed input set with a large degree of data reuse.
- Independent stream filters. Conceptually, a streaming computation represents a sequence of transformations on the data streams in the program. We will refer to the basic unit of this transformation as a filter: an operation that – on each execution step – reads one or more items from an input stream, performs some computation, and writes one or more items to an output stream. Filters are generally independent and self-contained, without references to global variables or other filters. A stream program is the composition of filters into a stream graph, in which the outputs of some filters are connected to the inputs of others.
- A stable computation pattern. The structure of the stream graph is generally constant during the steady-state operation of the program. That is, a certain set of filters are repeatedly applied in a regular, predictable order to produce an output stream that is a given function of the input stream.
- Sliding window computations. Each value in a data stream is often inspected by consecutive execution steps of the same filter, a pattern referred to as a sliding window. Examples of sliding

windows include moving averages and differences; error correcting codes; bio-sequence analysis; natural language processing; image processing (sharpen, blur, etc.); motion estimation; and network packet inspection.

- Occasional modification of stream structure. Even though each arrangement of filters is executed for a long time, there are still dynamic modifications to the stream graph that occur on occasion. For instance, if a wireless network interface is experiencing high noise on an input channel, it might react by adding some filters to clean up the signal; or it might re-initialize a sub-graph when the network protocol changes from 802.11 to Bluetooth.
- Occasional out-of-stream communication. In addition to the high-volume data streams passing from one filter to another, filters also communicate small amounts of control information on an infrequent and irregular basis. Examples include changing the volume on a cell phone, printing an error message to a screen. These messages are often synchronized with some data in the stream–for instance, when a frequency hopping radio changes frequencies at a specific point of the transmission.
- High performance expectations. Often there are real-time constraints that must be satisfied by stream programs; thus, efficiency (in terms of both latency and throughput) is of primary concern. Additionally, many embedded applications are intended for mobile environments where power consumption, memory requirements, and code size are also important.

III. FSPL OVERVIEW

FSPL is an architecture-independent programming language intended for large-scale and high-performance streaming application development. This language introduces abstractions, such as having functions implemented in filters, hierarchically structured streams that are designed to facilitate modularity, programmer productivity, and flexibility. FSPL aims to allow programmers to easily and naturally express their design of streaming applications.

This language is inspired from the StreamIt language [5], but is not identical. The FSPL language, incorporates new ideas that have been judged to be necessary but have not been supported in StreamIt.

The major limitation of StreamIt is that it requires filters to have static input and output rates. That is, the number of data items peeked, popped, and pushed by each filter must be constant from one invocation of the filter to the next [5]. So, applications such as compression and lexical analyzer that have dynamic flow rates could not be easily implemented with this language.

In the FSPL language, when you define a filter, it is not needed to specify the amount of data produced and consumed by that filter. This feature leads programmer to develop more sophisticated applications in an easily and efficient manner.

Another restriction of StreamIt is that input and output channels are created implicitly whenever filters want to communicate with each other [5]. In our language, FSPL, channels are independent from filters and programmer can define them explicitly whenever he/she wants. Then, filters can read data from or write data to these channels.

A. Filters

In the FSPL language, a program is a collection of computational unit that operates on a large sequence of data. The basic unit of computation in FSPL is called filter. Indeed, filter reads one or more data item(s) from its input channel (consuming data), performs some operations, and writes one or more data item(s) to its output channel (producing data). Filters are atomic and independent, so they have the potential to execute in parallel.

In FSPL, filter is defined using the module keyword which must be followed by a module name. The body of a filter includes four main parts in its complete form:

- Declaring input and output ports along with variables declaration if needed.
- Defining constructor of the module. Constructor name should be the same as the module name.
- Defining destructor of the module. Destructor should start with ~ sign and its name should be the same as the module name.
- The main part of the module, which is responsible for performing computations, defined using the process keyword. Each module should have at most one process.

A basic filter declaration in FSPL looks like “Fig. 2”.

```
module  Module_Name
{
        //ports declaration
        //variables declaration
        Module_Name()
        {
           //constructor
        }
         ~Module_Name()
        {
           //destructor
        }
        process
        {
           //read data from input port
           //computations
           //write data to output port
        }
}
```

Figure 2. General format of filters in FSPL

FSPL's representation of a filter is an improvement over general-purpose languages. In a procedural language, the analog of a filter is a block of statements in a complicated loop nest. This representation is unnatural for expressing feedback and parallelism that is inherent is streaming systems. Also, there is no clear abstraction barrier between one filter and the other one, and high-volume stream processing is muddled with global variables and control flow. The loop nest must be rearranged if the input or output ratios of a filter changes, and scheduling optimizations further inhibit the readability of the code. In contrast, FSPL places the filter in its own independent unit, making explicit the parallelism and inter-filter communication while hiding the grungy details of scheduling and optimization from the programmer.

FSPL language very naturally allows also hierarchical system design. Each filter can be specified as a network of filters. This approach facilitates modularity, where the internal specification of any filter can be modified without impacting other filters.

B. Channels

In the FSPL language, we introduce channel variables in order to establish a communication between modules. These channel variables is used for transferring data items between any two filters. The role of the channel for establishing communication between two modules is shown in "Fig. 3".

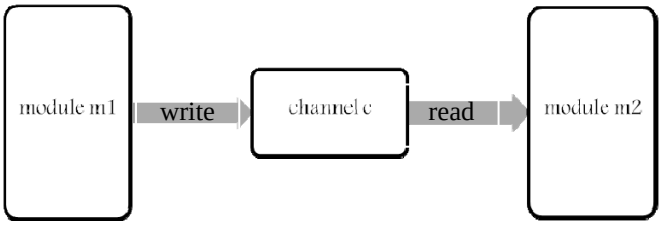

Figure 3. The role of the channel in FSPL

As can be seen, a channel variable receives data from the writer module and transferring its value to reader module. In other words, the producer filter writes data items into the channel, afterward the consumer filter reads the same data items from the same channel.

In FSPL, channel is defined using the *channel* keyword followed by the data type of channel variable and an identifier indicates the channel's name. As an example, *channel char c* defines a character channel named c.

Two operators are supported on these channels; '!>' and '!<' . The former is used for reading data items from the channel and the latter is used for writing data items into the channel.

C. Filter's Input and Output Ports

As we mentioned, in the FSPL language, we can define filter's input and output ports explicitly. The input port is defined using the *in* keyword followed by the data type of the port and an identifier indicates the name of

the input port. As an example, *in integer input_1* indicates an input port named input_1 which holds integer values. Similarly, the output port is defined using the *out* keyword followed by the data type of the port and an identifier indicates the name of the output port. As an example, *out integer output_1* indicates an output port named output_1 which holds integer values.

D. The Operators of Input and Output Ports

There are two explicit operators in FSPL for reading and writing values from and to input and output ports denoted '!>' and '!<' respectively that can be defined informally as follows.

If *input* is a an input port and *ch* is a variable then the command *ch !> input* is interpreted informally as: read the first value from the input port named input and assign this value to variable *ch*. If *input* is empty then execution of this command is delayed. Similarly, If *output* is a an ouput port and *ch* is a variable then the command *output !> ch* is interpreted informally as: read the value of *ch* variable and write this value to the output port named *output*. It should be noted that the reading and writing operations from and to input and output ports must be defined in the process of the module.

E. Data Types in FSPL

This section describes the various data types used in the FSPL programs. Data types are passed between modules and also can be used as local variables. Data types are always created atomically. They are of fixed size, and are generally statically allocated. The following data types exist in FSPL (see TABLE I).

TABLE I
PRIMITIVES DATA TYPES IN FSPL

Name	Description	Size
char	character	1byte
short	short integer	2bytes
int	integer	4bytes
long	long integer	4bytes
bool	Boolean value. It can take one of two values: true or false.	1byte
float	floating point number	4bytes
double	Double precision floating point number.	8bytes

The operators that are supported on these primitive data types are the same as they are used in c language.

F. Variable Declarations in FSPL

The variable declaration in FSPL is like variable declaration in c language which is defined using data type followed by an identifier. A variable declaration may declare one or multiple variables, possibly with initializers; multiple variables are separated with commas.

Also, we can declare another data types such as arrays and struct in FSPL if needed. Their syntax is the same as c language.

IV. DETAILED EXAMPLE

As we described in the previous sections, the most important feature supported in FSPL is that it is suitable for developing applications with dynamic rates of data production and assumption. Because when programmer defines a filter, it is not needed to specify the amount of data produced and consumed by that filter.

One of these applications is lexical analyzer. It is the first phase of a compiler. Lexical analyzer takes in a sequence of characters at its input and produces a sequence of tokens at its output (see "Fig. 4"). The critical point we should consider is that lexical analyzer has not any information about the length of a token; one token may consist of 20 characters while another consists of 5 characters and etc. but we can easily implement this application in FSPL.

The program for implementing lexical analyzer in FSPL is a collection of five filters named read, tokenizer, write, combination and main. These filters are represented in "Fig. 5". Read filter takes in the stream of characters from input file and produces the stream of characters on its output port. Tokenizer filter reads stream of characters from its input port, converts them into a token and writes stream of token into its output ports. Write filter reads stream of token from its input port and writes stream of tokens on output file. In order to implement this program hierarchically, we define a filter named combination which encapsulates read and tokenizer filter into one distinct filter. Main filter is responsible for connecting filters, scheduling filters and finally executing program.

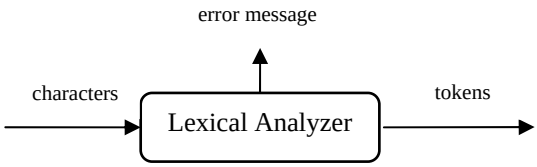

Figure 4. The lexical analyzer

```
module READ
{
        out char output;
        FILE *fr;
        READ()
        {
                fr = fopen("input.txt","r");
        }
        ~READ()
        {
                fclose(fr);
        }
        process
        {
                char ch = getc(fr);
                output !< ch;
        }
}

module Tokenizer
{
        in char input;
        out char output[1000];
        int i;
        char arr[1000];
        Tokenizer()
        {
                i = 0;
        }
        ~Tokenizer()
        {
        }
        process
        {
                char ch;
                ch !> input;
                if( ch != '')
                {
                        arr[i] = ch;
                        i++;
                }
                else
                {
                        arr[i++] = NULL;
                        i = 0;
                        output !< arr;
                }

        }
}

module WRITE
{
        in char input[1000];
        FILE *fw;
        WRITE()
        {
                fw = fopen("output.txt","w+");
        }
        ~WRITE()
        {
                fclose(fw);
        }
        process
        {
```

```
                char arr[1000];
                arr !> input;
                fputs(arr,fw);
                fprintf(fw,"\n");
        }
}

module COMBINATION
{
        out char output;
        COMBINATION()
        {
                channel char c;
                module READ r;
                module Tokenizer t;
                r !< c;
                t !> c;
                t !< output;
        }
}

module main
{
        main()
        {
                channel char ch[1000];
                module COMBINATION c;
                module WRITE w;
                c !< ch;
                w !> ch;
        }
}
```

Figure 5. Implementing lexical analyzer in FSPL

V. Related Work

There has been a wealth of research of various stream languages and projects. This section will introduce some of the other projects.

A large number of programming languages have included a concept of a stream; see [10] for a survey. Synchronous languages such as LUSTRE [9], Esterel [8], and Signal [11] also target the embedded domain, but they are more control-oriented than StreamIt [5] and are not aggressively optimized for performance. Sisal (Stream and Iteration in a Single Assignment Language) is a high performance, implicitly parallel functional language [12]. The Distributed Optimizing Sisal Compiler [12] considers compiling Sisal to distributed memory machines, although it is implemented as a coarse-grained master/slave runtime system instead of a fine-grained static schedule.

Ptolemy [13] is a simulation environment for heterogenous embedded systems, including Synchronous Data Flow that is similar to static-rate stream graphs in StreamIt. SDF programs, however, do not include the peeking and messaging constructs of StreamIt. In SDF languages, actors are the active computational elements (Filters). SDF computation model does not impose structure on the program. All actors are allowed to have multiple input and output channels.

Cg is a system for programming graphics hardware. The Cg language is based on both the syntax and philosophy of C. In particular Cg is intended to be general purpose, rather than application specific, and is a hardware oriented language. Cg also adopts a few features from C++ and Java. But unlike these languages Cg is intended to be a language for programming in the small rather than programming in the large [14].

The StreamC language in combination with C++ is to be used for writing programs that utilize the Imagine stream processing system. StreamC includes commands for transferring streams of data to and from the Imagine system and between Imagine processors, for reading and writing microcontroller variables, and for executing kernels [15].

VI. Conclusion

Parallel processing is one of most important issue in computer science. There are many approaches to parallelizing programs such as parallelizing loops or developing advanced parallel compilers. But using stream programming paradigm, programmer writes his code implicitly parallel.

In this paper, we introduced a flexible stream programming language, called FSPL. The FSPL is an independent-architecture programming language for high performance stream applications. FSPL aims to allow programmers to easily and naturally express their design of streaming applications.

The most important ideas implemented in FSPL is that when you define a filter, it is not needed to specify the amount of data produced and consumed by that filter. The advantage of using this idea is flexibility for developing stream applications.

REFERENCES

[1] *The StreamIt Cookbook*, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge. MA 02139, Nov. 2004. [Online]. Available: http: //cag.csail.mit.edu/streamit/index.shtml,accessed,November-11-2006.

[2] J. Bengtsson, "*Thesis for the degree of licentiate of engineering, efficient implementation of streaming applications on processors arrays, technical report*," School of Information Science, Computer and Electrical Engineering, Halmstad University, Tech. Rep., 2006.

[3] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. *The Design of the Borealis Stream Processing Engine.* In Second Biennial Conference on Innovative Data Systems Research (CIDR 2005), Asilomar, CA, January 2005.

[4] Soga A., Shiba M., and Kawamoto T., 2008, *An attempt of real time Cg control with multi touch devices*, In the proceedings of the 2008 ACM symposium on virtual reality software and technology, ACM, New York, USA.

[5] W. Thies, M. Karczmarek, and S. Amarasinghe. *Streamit: A language for streaming applications.* In International Conference on Compiler Construction (CC'02), Apr. 2002.

[6] Liao S., Du Z., Wu G. and Lueh G., 2006, *Data and computation transformations for Brook streaming applications on multiprocessors*, Proceedings of the international symposium on code generation and optimization, pages 196-207, IEEE computer society, Washington DC.

[7] E. A. Ashcroft and W. W. Wadge. *Lucid, a non-procedural language with iteration*. Communications of the ACM, 20(7):519–526, July 1977.

[8] F. Boussinot and R. De Simone. *The ESTEREL language*. Proc. IEEE, 79(9):1293–1304, Sept. 1991.

[9] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. *LUSTRE: A declarative language for programming synchronous systems*. In Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages (POPL), pages 178–188, Munich,West Germany, Jan. 21–23, 1987. ACM SIGACT-SIGPLAN, ACM Press.

[10] Robert Stephens. *A Survey of Stream Processing*. Acta Informatica, 34(7), 1997.

[11] Thierry Gautier, Paul Le Guernic, and Loic Besnard. *Signal: A declarative language for synchronous programming of real-time systems*. Springer Verlag Lecture Notes in Computer Science, 274, 1987.

[12] "J. Gaudiot, W. Bohm, T. DeBoni, J. Feo, and P. Mille". *The Sisal Model of Functional Programming and its Implementation*. In Proceedings of the Second Aizu International Symposium on Parallel Algorithms/Architectures Synthesis, 1997.

[13] Edward A. Lee. *Overview of the Ptolemy Project*. UCB/ERL Technical Memorandum UCB/ERL M01/11, Dept. EECS, University of California, Berkeley, CA, March 2001.

[14] William R. Mark, R. Steven Glanvile, Kurt Akeley, Mark J. Kilgard, "*Cg: A system for Programming Graphics Hardware in C-Like Language*", proceedings of the international conference on computer graphics and interactive techniques, San Diego, California, 2003.

[15] Abhishek Das, William J. Dally, Peter Mattson, "*Compiling for stream processing*", proceedings of the 15th international conference on parallel architectures and compilation techniques, Seattle, Washington, USA, 2006.