A LANGUAGE FOR STREAMS ON RECONFIGURABLE MANY-CORES

Gordon Ononiwu
Department of Electrical/Electronic Engineering,
Federal University of Technology, Owerri, Imo State.
NIGERIA.
ononiwugordon@yahoo.co.uk

ABSTRACT

Portable stream programming language (PSPL) is a language for stream based application programming on reconfigurable many-core architectures. It provides the programmer with useful domain specific primitives that allow for bitwise operations on data. The first step in its development has been completed. The syntax has been fixed and a parser has been provided for the front end of the PSPL compiler. The scanner and the parser where generated using automatic tools (scanner and parser generators) which rely on complex mathematical algorithms for their generation. Abstract syntax (data structures that preserve the source program so that program structure is evident) has been implemented for the parser using a "syntax separate from interpretation" style of programming. Tests have been carried out to ensure that the correct data structures are generated. The final outcome is a parser that other phases of the compiler can depend on for onward transmission of the source program in an unambiguous manner. The development of subsequent phases of the compiler will form the next logical step in the process of transforming PSPL to a standalone language for computing on reconfigurable many-core architectures.

Keywords: Reconfigurable, Baseband, Parser, Streams, Abstract, syntax, many-core

INTRODUCTION

As the number of applications modeled around some notion of streams continues to rise, there is a need to provide better programming support in the form of portable languages. Using the notion of streams makes it possible for programmers to structure programs in ways that provide the compiler with enough information about parallelism, program- and data-flows, and reconfiguration, which the compiler can make use of in order to produce efficient translations to reconfigurable many-cores.

By streams we mean the continuous one directional flow of data in any format and streaming applications are applications that process single or multiple streams in the incoming and outgoing directions at predefined flow rates. An example of such an application can be seen from the processing that goes on in third generation (3G) radio base stations (RBS) (Bengtsson, 2006), digital signal processing (DSP) in radar systems, or in a software defined FM radio (MA, 2007).

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 eg. Many-cores.

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. 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.

Copyright © 2012 SAVAP International
www.savap.org.pk

However, these hardwired implementations come with a major drawback, namely, lack of flexibility. 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 in algorithms often 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 (Bengtsson, 2006).

Therefore, to reduce cost, speed up design time and improve on the ability to provide for the user's various needs while reducing down time required for upgrades, there is need to use commercially available reconfigurable many-cores while developing compilers that can exploit the parallelism that these processors provide. At the same time avoiding performance penalties and allow for reconfiguration.

THE PROBLEM

For baseband applications, such as those found in radar systems and baseband processing in RBS, there is the need to provide type primitives and operators that can express bit-level manipulations in a decent manner (Bengtsson, 2006). The lack of suitable functions for bit-wise operations on data in earlier stream based programming languages like StreamIT, StreamC, and Streambits has been the major limitation to designer productivity in the streaming domain.

PSPL is to a large extent based on StreamIT but is not identical. It provides all the advantages that are inherent in earlier stream compilers, such as having functions implemented in filters, using pipelines to describe the data flow graphs, and giving room for multiple concurrent pipelines as the need may arise, but also incorporates new ideas for the bit-wise manipulation of data, that have been judged to be necessary but have not been supported in streamIT (Bengtsson, 2006). PSPL also aims to provide a means of passing on reconfiguration parameters to the compiler.

In this paper we outline a series of steps taken in moving PSPL towards being a standalone language. To be a standalone language, we need to compile the PSPL code directly to a back end of any many-core architecture. In doing this the following tasks have been achieved:

a. Fixing the syntax of PSPL. There were challenges here due to the fact that the types of PSPL are parameterized and that programs are organized as stream transformations processing streams.
b. Designing the abstract syntax (AS), a data structure to be used by the rest of the compiler in processing applications written in PSPL.
c. Writing a parser to read files of text containing PSPL programs and producing an abstract syntax representation of the program.

The rest of the paper is organized as follows: Section 3 provides background information relating to compilers, streams and stream processing systems and work that has already been done in developing a number of stream languages. Section 4 introduces the PSPL prototype specification which was first implemented using legal java syntax. Section 5 discusses the implementation. Section 6 discusses the results and Section 7 concludes and discusses future work to be done.

BACKGROUND

This section introduces compilers, stream processing and the stream application domain, earlier work done in producing suitable stream languages and domain specific concepts that will enhance the suitability of the PSPL language.

Compiler

A compiler is large, complex software machinery that uses advanced algorithms to translate a high level language into a form of machine code that can be understood by a processor. The compiler has to distinguish between correct and incorrect codes of a language, generate an intermediate representation for the language (IR), generate correct machine codes and organize memory for the variables used. Modern compilers are organized in several phases that operate on successive abstract products from one phase to the next. This phase like structure is done to make the complex program modular.

Stream Processing System (SPS)

The history of streams and stream processing is centered on the notion of a stream processing system (SPS). An SPS is any collection of modules or blocks that compute in parallel, and communicates via channels. A typical SPS module is typically divided into three classes: sources that pass data into systems; filters that perform atomic computations; and sinks that pass data out of the system. Examples of SPS are dataflow systems, reactive systems, specialized functional and logic programming with streams, synchronous concurrent algorithms, signal processing systems, and certain classes of real-time systems.

A stream transformer (ST) is defined as an abstract system that takes a set of x streams as input and produces a set of y streams as output (Stephens, 1997). SPS can be considered as a parallel implementation of an ST specification, and stream processing can be defined as the study of both STs and SPSs.

Research into this branch of computer science began in the early 1960s with the study of data flow analysis used to evaluate potential concurrency in computations. In 1974, the first data flow language, Lucid, was conceived (Stephens, 1997). Examples of other data flow languages that came thereafter are LUSTRE (Caspi *et al.,* 1987), LAPSE (Gurd *et al.,*1981) and MAD (Foley, 1989). In 1985 the first paper on synchronous concurrent algorithms (SCAs), and reactive systems was released. Reactive systems, together with signal processing networks and synchronous dataflow networks have been the stimuli for a large body of research into stream processing.

The 1980s also saw the use of stream processing in hardware design. A language, Daisy, was used in applicative stream processing for the design and synthesis of hardware. In the 90s, reactive systems continued to be an intensive area of research. The 90s also saw the concentration of research into the study of the compositional properties of STs and into further developing the theoretical foundations of stream processing (Stephens, 1997).

The Stream Application Domain

A lot of applications make use of a stream abstraction, ranging from embedded applications for hand-held computers, cell phones, digital signal processors (DSP's), high performance applications such as intelligent software routers, GSM base stations, HDTV editing consoles (Thies *et al.,* 2001) and consumer desktops.

As array structured parallel and reconfigurable processors become the widely used implementation strategy for computationally demanding high performance applications, the variety in the applications using the stream paradigm will increase. This calls for an efficient approach to handling component and architecture abstraction through the use of an application programming interface (API).

Many-cores require portable compilers that can exploit their parallelism in a most efficient manner, and at the same time, provide the programmer with a level of abstraction from the particular target architecture.

StreamIT

We provide only a brief overview of the StreamIt language. For a detailed description see [MA, 2007; Thies *et al.,* 2001; Thies *et al.,* 2002). The StreamIt language as it is currently implemented makes use of legal java syntax. It is a portable language for high performance signal processing applications. It was designed for communication exposed machines such as the RAW. The basic construct of the language is made up of filters, splitJoins, Feedback loops, and pipelines.

StreamBITS

StreamBits is a prototype language for baseband application programming (Bengtsson, 2006). This prototype was designed because of the need to meet up with some of the short comings of earlier stream languages. Therefore, it was aimed at;

a. Providing program structures that are natural to use for the definition of abstract components.

Copyright © 2012 SAVAP International
www.savap.org.pk

b. Offering primitive types and operators that allow the programmer to efficiently express application specific computations.
c. Expressing bit-level and data-parallel operations more efficiently without considering machine specific details in algorithmic implementation.

The primary structure of the language is to a large extent based on the StreamIt language structure, but it is not considered by the designers to be identical to StreamIt. The basic stream language constructs are the filter and pipeline. The filter construct is where the computations are performed, while the pipeline construct is used to organize filters and other stream components in a daisy chain format. Components are added by the add (component) command. Dual tapes of streams are supported in this language unlike in StreamIt, where components can only operate on a single Stream tape. One tape consists of a data Stream while the other tape is for the configuration Stream. This tape is used to pass stream reconfiguration parameters to components throughout the application (Bengtsson, 2006).

THE PSPL PROTOTYPE SPECIFICATION

The PSPL prototype evolved from the StreamBits prototype. With PSPL, improvements have been made by introducing constructs that allow for greater flexibility at bit level manipulation and stream mapping. A complete PSPL program consists of a main () function (sequential section implemented using a language subset based on C) and a stream program (parallel data flow section). The main function is used for control and provides functions for interaction with the hardware such as memory management, I/O, peripheral devices etc. The stream program provides the constructs for stream manipulation and mapping and implements the processing of compute intensive kernel functions. The main function and the stream program must be defined inside a program declaration. The Stream program implements the structure of a directed flow graph. Vertices in the graph correspond to filters that perform some computation. As in StreamBits, the filters are connected together in pipelines and with an added functionality namely, a switch. Switches are tools used to split and join pipelines into concurrent streams of data and provide mapping functions for more transparent stream manipulations. These splits and joins can be combined to form feedback and stream forwarding loops.

IMPLEMENTATION

This section describes the techniques that were used in generating an abstract syntax representation of the PSPL language. It is the first step towards translating the language into executable code.

Grammars

A language is a meaningful set of sentences, each sentence comprising of a sequence of words, each word comprising of a sequence of symbols or characters. Every formal language has its grammar and we can define the grammar as a set of rules that must be followed in order to make distinctions between correct and incorrect sentences. Therefore, we say that the grammar describes the language. For a programming language there is the added requirement that the grammar has to be context free in order to allow for easy parsers.

The context free grammar (CFG) of a language is a set of rules describing how to form sentences in the language. The CFG consists of four parts T, NT, S, and P.

a. T, the set of terminal symbols representing words of the language (tokens).
b. NT, the set of non-terminal symbols (syntactic categories e.g. the types of sentences or sentence category).
c. S, the start symbol or Goal which is a non-terminal standing for the syntactic category whose sentences we are describing.
d. P, the set of productions. Each non-terminal is mapped to a production

If we have a rule like:

AB → aba AB

|aba

|

;

AB → aba AB, is the production (P) and derives sentences built by the word aba

Followed by another AB which could be an aba or just an empty entity. AB is the non-terminal (NT) while aba is the terminal (T). The start symbol (S) is AB. We also introduce additional rules in order to reduce ambiguity and have one parse tree per given sentence. This is done to ensure that the writer will be sure of how the compiler interprets the sentence and it also reduces the need for too many parentheses.

Scanner

A scanner is a program that scans the sequence of characters presented to the compiler identifying the words of the language. Such words can comprise of the keywords, identifiers, punctuations, operators etc. Separators like blank spaces between the words (often called white spaces), comments and newlines are discarded. It would further complicate the parser if white spaces and comments where to be accounted for and this is the main reason why there is a separation between both phases (Appel & Palsberg, 2002).

Scanner generators take in the regular expressions; apply complex algorithms to convert them first to Nondeterministic Finite Automata (NFA), and finally to Deterministic Finite Automata (DFA). It is from the DFA that the words of the language are picked out since every regular expression has a DFA that recognizes its language. JFlex (Klein, 2005) and JavCC (Appel & Palsberg, 2002)are scanner generators that generate lexical analyzers written in java. We used JFlex because it is based on java.

Parser

A parser is the second phase of the compiler and is built from the CFG. The parser takes in the set tokens generated by the scanner and matches it to the CFG, evaluates the matching attributes, and generates an abstract representation of the grammar.

The question is how does a parser go about doing all this? To answer this we need to understand how the parser works. The symbols appear to the parser as a sequence of tokens, some of them with values attached to them and it is the job of the parser to build a parse tree based on what it gets. The parse tree is generated by connecting each symbol to the one that derived it based on the rules of the grammar.

Parser Generator

There are many parser generators that generate parsers written in java, some of the well-known ones are CUP, Antlr, JavaCC , SableCC, Coco/R, BYACC/Java, Jikes, and Jacc parser generators (Jones, 2002). For this project we used Jacc. We are interested in the following (but not limited to these) differences between Jacc and the other parser generators mentioned above;

a. It a pure Java implementation that is portable and runs on any Java platforms.
b. It generates a bottom-up/shift reduce parser with disambiguating rules.
c. Modest additions to help users understand and debug generated parsers, including: HTML outputs, and tests for conflicts.

Semantic Actions

Each terminal and non-terminal may be associated with its own semantic value and each non-terminal with its own syntactic category. For instance, for the rule:

AB → aba AB

The semantic actions must return a value whose type is associated with AB. To do this, it needs an abstract syntax representation of the parsed sections of the phrase. This provides the parser with data structures that can be used to store source code attributes in a way that ensures that the phrase structure of the source code is evident. This works recursively until we eventually get a value that represents the start symbol or non-terminal AB. The compiler uses the abstract syntax to build a parse tree with the start symbol as its root, the non-terminals as the nodes of its branches, and the terminals as its leaf's. The compiler will need concrete classes that will be used to construct the parse tree from the abstract syntax.

Copyright © 2012 SAVAP International
www.savap.org.pk

ISSN-L: 2223-9553, ISSN: 2223-9944
Vol. 3, No. 2, September 2012

RESULTS

The implementation of the PSPL parser involved the design of the BNF, fixing the syntax for PSPL, the implementation of the specification files for both JFlex and Jacc which resulted in a scanner and a parser, and the design of the abstract classes that will be used by the parser to generate an AST for any program written in PSPL. These abstract classes are contained in two packages; syntaxtree and visitor. The following sections will outline some of the results.

The BNF

While fixing the grammar, it was broken into smaller parts by the introduction of non-terminal symbols and the introduction of three main groupings, namely: Expressions, Statements and Declarations. Expressions are entities that have a value and are usually associated with a type. The following are a list of the non-terminals that were used to simplify the grammar:

Streamprogram (Start symbol), ObjectDec, PipelineDec, FilterDec, SwitchDec, SwitchDecList, ObjectType, ObjectTypeDecList, ObjectTypeDec, MapStm, MapStmList, Switches, SwitchStm, SwitchStmList, InitDec, ProcedureDec, ProcedureDecList, StreamDec, StreamDecList, StreamType, Formal, FormalList, Src, SrcList, Dst, DstList, Sorc, Dest, Type, Stm, StmList, VarDec, VarDecList, Exp, ExpList, Index expression, BitvecExpression, BitvecExpressionList, Identifier, Integer Literal, Float Literal

Operators that are supported by PSPL are shown as follows:

|| << >> % ++ -- -> => == += .* ./ .+ .- .'' : != ^ ~ - / * = && & | > < [ ] { } ( ) ; . ,! ? +

These are further grouped into operators and unary operators as follows:

Operators op || << >> % .* ./ .+ .- = && & | > < - / * +

Unary Operators Unop + - ^ ~ ++ -- .' ' !

Parsing

All the classes in the visitor package and the sytaxtree package were compiled. Also, the files psplLexer.java, psplParser.java and psplMain.java were also compiled and tested.

CONCLUSION

A parser has been provided for PSPL. The syntax had to be fixed in order to make it easy to parse and tests were carried out to ensure that the desired outcomes were achieved. Also, abstract syntax was implemented for the parser using a "*syntax seperate from interpretation*" style of programming. The parser uses the abstract syntax to preserve source programs in concrete data structures that are used by later phases of the compiler for synthesis and translation to machine code. Tests were carried out to ensure that the correct abstract syntax trees are being generated as desired.

REFERENCES

Appel, A.W. and Palsberg, J. (2002). *Modern compiler implementation in Java*, 2nd ed. CAMBRIDGE: Cambridge University Press.

Bengtsson, J. (2006). "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.

Buck, I. (2003, Oct.) Brook specification v0.2. <http://merrimac.stanford.edu/brook/brookspec-v02.pdf> [June 16, 2007]

Caspi, P., Pilaud, D., Halbwachs, N. and Plaice, J. (1987). *Lustre: A declarative language for programming synchronous systems*. pp. 178–188.

Foley, J. (1989). "Manchester dataflow machine: Benchmark test evaluation report, "Department of Computer Science, University of Manchester, Manchester, UK, Technical Report UMCS-89-11-1, Gurd, J.R., Clauert, J.R.W. and Kirkham, C.C. (1981). *Generation of dataflow graphical object code for the lapse programming language*, in Proceedings of the Conference on Analyzing Problem Classes and Programming for Parallel Computing (CONPAR '81), ser. LNCS, W. H¨andler, Ed., vol. 111. N¨urnberg, FRG: Springer, pp. 155–168.

Jones, M. (2002) Jacc: Just another compiler for java, a reference manual and user guide. Department of Computer Science, Engineering, Stanford University, Stanford, CA 94305.

Klein, G. (2005, July) Jflex user's manual. <http://www.jflex.de/manual.pdf> [July 28, 2007]

Stephens, R. (1997). A survey of stream processing. *Acta Informatica*, vol. 34, no. 7, pp. 491–541.

The Stream It Cookbook, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge.MA 02139, Nov. 2004. http://cag.csail.mit.edu/streamit/index.shtml Accessed August 11, 2007.

Thies, W., Karczmarek, M. and Amarasinghe, S. (2002). StreamIt: A language for streaming applications, Lecture Notes in Computer Science, vol. 2304, pp. 179.

Thies, W., Karczmarek, M., Gordon, M., Maze, D., Wong, J., Hoffmann, H., Brown, M. and Amarasinghe, S. (2001). *Streamit: A compiler for streaming applications*. http://citeseer.ist.psu.edu/515289.htm http://www.lcs.mit.edu/publications/pubs/pdf/MIT-LCS-TM-622.pdf

Copyright © 2012 SAVAP International
www.savap.org.pk