Static Analysis of Behavioural Properties of Stream Synchronisers

Anna Tikhonova

March 25, 2014

Contents

1 Introduction

For years, processor manufacturers have delivered increases in clock rates, so that single-threaded code executed faster on newer processor without any modifications. While manufacturing technology still improves, physical limitations of semiconductor-based electronics have become a major concern of design. In order for the processors to continue to improve in performance, multi-core design has become necessary [1].

Unlike the increase in clock frequency for previous-generation processors, this shift in hardware design does not provide automatic benefits for software. The improvement in performance gained by the use of a multi-core processor depends significantly on the programming concepts, tools and infrastructure used for software implementation. Some parallel programming models such as OpenMP [2], MPI [3], Cilk Plus [4] were developed for multi-core platforms. However, in most cases these models require massive software re-engineering to take advantage of multi-core processing.

The main challenge in engineering of correct concurrent code is managing concurrency. Concurrency management includes ensuring the correct sequencing of the interactions between different computational executions, and coordinating access to resources that are shared among executions. The complexity comes from the fact that these actions are mixed with computations.

Software parallelization is a significant topic of ongoing research. One of the research directions that addresses issues of efficient multi-core programming and difficult concurrency management is coordination programming. The idea of coordination programming is to represent a program as a set of computational processes and to specify relations between them in special coordination language. Potentially, this model allows execution of every single process on a dedicated processor core. However, in real-world applications, a large number of concurrent processes and their communication facilities have to share a very limited amount of resources. Thus, the issue of application progress becomes quite complicated. In order to boost application performance a programmer has to tune it manually.

A new programming language called Astra*Kahn* is described in [5]. Astra*Kahn* attempts to combine coordination programming with stream processing to provide a self-regulatory concurrency mechanism. The goal of the project is to provide an environment for development of scalable concurrent applications that does not require manual tuning efforts.

2 Background and Related Work

This section is an overview of relevant studies for both stream processing and coordination aspects of Astra*Kahn*.

2.1 Stream Processing

Stream processing research, particularly the study of stream processing systems, has a long history. In general, a stream processing system can be described as a system comprised of a set of processing components that compute in parallel and communicate data via channels. Channels pass data as infinite sequences; these infinite sequences are referred to as streams.

Examples of stream processing research are dataflow systems, reactive systems, signal processing systems, etc. An overview of the hystorical development and the discussion of the different techniques for streams programming is presented in the survey by R. Stephens [6].

Stephens identifies that the first type of stream processing systems are dataflow systems. The first dataflow programming language Lucid [7] was designed in 1970s. In Lucid, each variable is an infinite stream of values. Computation is carried out by defining transformation functions that process these streams. Lucid is possibly the first language to introduce the idea of a block that transforms input message sequences into output sequences.

In 1974 G. Kahn published his paper [8] outlining the semantics of a simple language for parallel programming. In his work Kahn suggests a distributed model of computation where a group of deterministic sequential processes communicate via unbounded FIFO channels. In this model a program is represented as a directed graph whose vertices are prefix monotonic[1] computational processes and edges are stream-carrying channels with the following assumptions:

- Channels are the only way for processes to communicate;
- Channels transmit messages within a finite time;
- At any given time a process is either performing computation or waiting for messages on one of its input channels;
- Each process is a sequential program.

Kahn proved that the resulting process network exhibits deterministic behaviour, i.e. its output does not depend on computation or communication delays. This model is now referred to as Kahn Process Network (KPN). Astra*Kahn* is based on KPNs.

[1]Process is prefix monotonic when it takes partial input stream - prefix - in order to produce partial output stream, i.e. for prefix monotonic process $f(x)$ and prefix $p : f(p) = r$ it is true that $f(p||t) = r||T$

One of the latest KPN interpretations in programming languages is StreamIt [9]. In StreamIt, basic computational unit is called *filter*. A filter is a user-defined single-input, single-output block that translates input data sequences to output sequences. Notable feature of StreamIt is that it imposes structuring on applications with the following structural primitives:

- *pipeline* specifies sequential composition of filters,
- *splitjoin* specifies parallel composition of filters,
- and *feedbackloop* provides a way to create loop constructs in stream graph.

Recall that KPN model is theoretical and works well under interpretation when resources are unlimited. In real world, concurrent processes and their communication facilities have to share a very limited pool of resources. Thus, several refinements to KPNs are needed to address resource limitations.

The fundamental technique to deal with the issue is load balancing. Several static and dynamic approaches were suggested for load balancing in StreamIt applications. Static load balancing implies partitioning of a StreamIt program into a set of balanced patritions. Static strategies introduce heuristic approach [9], as well as use of ILP solvers [10], approximation alghorithms [11] and machine learning [12]. Dynamic load balancing technique proposed in [13] takes advantage of filters' statelessness[2]. The approach is based on communication pressure that emerges from the channels' boundedness. Here communication pressure characterises the channel occupancy of some computational process and feedback from its co-located computational processes.

Astra*Kahn* proposes demand-driven self-regulatory mechanism. It is based on the same concept of communication pressure with a refinement to achieve fine-grained parallelism.

In Astra*Kahn* predecessor S-Net [14] the problem with resource management persists. A programmer has to tune S-Net application manually.

2.2 Coordination programming

The coordination aspect of Astra*Kahn* is related to significant amount of prior work in data-driven coordination. Main existing coordination technologies are described in details in the survey [15] by G. Papadopoulos and F. Arbab.

Unlike a great many approaches to coordination programming, coordination and computation in Astra*Kahn* are completely separate. This is achieved by introduction of a special type of process called a synchroniser. A synchroniser can be thought of as a facility that guarantees data availability on computational process inputs.

S-Net is also the case. S-Net utilises *synchrocells* in order to control synchronisation. Synchrocells are the only means in S-Net to combine messages from several channels into a single record; they only realise a functional mapping from input streams to the output stream. In constrant to synchrocells, synchronisers allow complex synchronisation patterns. They are programmed in a dedicated programming language which is a part of Astra*Kahn*.

Another related coordination technology is the language Reo [16]. Like Astra*Kahn* or S-Net, Reo is exogenous in terms of coordination. Reo's emphasis is on complex coordinators, or *connectors*, and their composition out of simpler ones, rather than on the components that communicate via these connectors.

A channel is an atomic connector in Reo. A channel is undirected, i.e. its ends may be either *source* or *sink*. Source and sink are connected to producer and to consumer nodes respectively. Channels in Reo are typed, however, no fixed set of types is assumed. Channels coincide in another fundamental entity called a *node*. Reo defines sets of operations on channels and nodes, see [16] for reference.

[2]Regardless of input sequence A, stateless filter output sequence does not depend on input sequences that came before A.

Complex connector in Reo is represented as undirected graph contisting of channels and nodes. C. Baier et al. introduce *constraint automata* and propose them as an operational model for component connectors in Reo [17]. Concerns of equivalence or containment checks and verification are addressed in this paper as well as in [18].

3 The AstraKahn approach to Streaming Networks

Astra*Kahn* is an attempt to provide programming paradigm based on Kahn's model of process networks. Program in Atra*Kahn* is represented as a directed graph of computational processes connected with edges that are stream-carrying channels. Detailed description of Astra*Kahn* language is given in [5].

Though Astra*Kahn* preserves certain properties of KPNs, it provides the following refinements:

complete separation of coordination logic from computations This refinement leaves computational components stateless and opens opportunities for easier parallelisation. Coordination logic is expressed in special vertices called *synchronisers*. *Synchronisers* are programmed in dedicated programming language;

self-regulatory concurrency mechanism based on the concept of communication pressure. The mechanism address the issue of application progress under the interpretation when resources are limited. Behavioural classification of computational components is provided along with this mechanism;

separation into independent layers communicating by means of interfaces This refinement addresses standard engineering issues such as abstraction, encapsulation and hierarchical development.

As the result of refinement, Astra*Kahn* is seen as a construction of three independent layers:

- Topology and Progress Layer (TPL) defines the topology and provides concurrency regulatory mechanism
- Constraint Aggregation Layer (CAL) insures type safety all over network with data constraints provided by each component
- Data Instrumentation Layer (DIL) manages data distribution and concurrent memory access.

3.1 Coordination and Synchronisation

As already mentioned before, Astra*Kahn* defines a special type of vertex called a synchroniser. The intention of synchronisers is to separate coordination code from actual computation. Synchronisers are programs written in a dedicated language which is a part of the Astra*Kahn* paradigm. The formal syntax of the language and examples are given in [5].

From mathematical point of view, a synchroniser is represented as a non-deterministic finite state machine with conditional transitions. A transition fires if and only if corresponding predicate is true. This predicate can be used to check for

- presence of a message in a certain channel,
- certain structure of an incoming message,
- value of C-style conditional expression on special *state* variables.

State variables are global to all states of a synchroniser. Since finite state machines read one symbol at a time, they do not require parallel access management. *State* variables evolve with C-style assignment expressions that execute when the associated transition takes place.

When a synchroniser is in the state in which it accepts only messages from certain input channels, other input channels are blocked. Blocked channels preserve messages until synchroniser makes a transition to another state in which it accepts messages from this channel, thus unblocking it.

A synchroniser always has a start state and assumes cyclic behaviour. A correctly programmed synchroniser loops infinitely around its start state.

Also, synchronisers provide internal facilities to store messages. These facilities are called *store* variables. A *store* variable is associated with channel it stores message from. All store variables lose their associated values whenever the synchroniser reaches its start state again.

3.2 Pressure Propagation in Astra*Kahn*

Astra*Kahn*'s self-regulating concurrency mechanism relies on the concept of communication pressure. Communication pressure is a dynamic characteristic of a channel that indicates demand of messages in a certain part of streaming network.

Channels in Astra*Kahn* are bounded FIFO queues. When a consumer node processes messages at a lower rate than they are produced, the storage capacity of connecting channel descreases. This descrease can be quantified by the number of messages in the queue. This number defines *positive* pressure in a channel. Positive pressure becomes *critical* when there is no storage for a fresh FIFO element. Channels under critical positive pressure become blocked. A blocked channel does not accept any further input, thus it directly affects a producer node. If the producer cannot output a message into a channel, then it is suspended. This can lead to a pressure increase in the corresponding input channel and eventual blocking of this channel.

In such a way positive pressure propagates across the network. In case of sufficiently large difference of pressure on input and output channels of a node, Astra*Kahn*'s coordinator can run several copies of this node in parallel. Increase of the processing rate of a node decreases pressure at this point. In Astra*Kahn* this form of adjustment is called *proliferation*.

When a consumer node processes messages at higher rate than they are produced, there are no messages in a connecting channel waiting to be consumed. This demand for messages is quantified by the concept of *negative* pressure. Since synchronisers are the only stateful vertices in Astra*Kahn*, they are the only vertices within the network that induce negative pressure. A synchroniser assumes zero processing latency. Thus, negative pressure can be thought of as the number of messages that the synchroniser would accept immediately. Messages are accepted immediately when one of the synchroniser input channels is blocked due to unavailability of messages on another. For example, this situation occurs when a synchroniser performs the zip operation and one of the producer nodes supplies messages more slowly.

Also, negative pressure can be exerted by a network consumer if the production rate of the whole network is insufficient. In an Astra*Kahn* network, negative pressure propagates exclusively via synchronisers. In contrast to positive pressure, negative pressure is not associated with a physical state change of a channel; it is rather a mechanism that a TPL implementation can consider to adjust pressure in a more efficient way.

4 Motivation and Problem Statement

The long-term goal of the Astra*Kahn* project is to provide an environment for development of scalable concurrent applications that does not require manual tuning efforts. An Astra*Kahn* approach to adaptive concurrency relies on the concept of communication pressure propagation across a network. Since pressure propagation directly affects proliferation levels of certain vertices, effectiveness of concurrency self-regulation depends largely on correctness of pressure propagation strategy. An important part of this strategy is pressure propagation through synchronisers because, unlike other vertices in Astra*Kahn*, they induce negative pressure that represents a demand for messages in a certain part of a network.

This project is focused on synchroniser analysis and development of supporting tools. Since the synchroniser is programmed in a dedicated language, a compiler for this language is needed. Target architecture assembly generation is of no concern within the project, thus, the compiler translates a given source code into a C program and then calls available C compiler to generate an executable file. Also, the compiler provides various syntactic and semantic checks.

In order to propagate pressure through a synchroniser, Astra*Kahn* coordinator needs to know its *transfer function*. The transfer function describes relations between demands of messages on a synchroniser's input and output channels. Derivation of the transfer function is a concern of static analysis, in particular, execution path analysis. Because synchronisers are non-deterministic, there may be potentially an infinite number of possible execution paths and therefore more than one transfer function. It may be impossible to determine an exact execution path at compile time, because it may depend on incoming message content. If synchroniser has an inner loop, the transfer function must consider the number of iterations in this loop. Thus, transfer function derivation relates to induction variable analysis.

These issues are subject to synchroniser analysis in the project. Once a set of transfer functions is obtained, it is up to TPL how to choose a particular transfer function and propagate pressure through a synchroniser. Also, synchroniser analysis includes passport generation for Constraint Aggregation Layer.

5 Project Plan

5.1 Research Plan

The research timeline is suggested as follows:

- Phase I (3 months)
    - A literature review on stream processing and coordination programming
    - Understanding of computation model and basics of concurrency regulatory mechanism in Astra*Kahn*
    - Development of synchroniser compiler frontend
- Phase II (3 months)
    - Development of a transfer function derivation and pressure propagation techniques for synchronisers
    - Prototyping of the technique and the compiler backend
    - Demonstration of running example
- Phase III (6 months)
    - Development of the compiler backend
    - Further development, testing and improvement of the transfer function derivation and pressure propagation techniques
    - Thesis writing

5.2 Research Training and Development

I have attended several general research trainings. During *Thesis, What Thesis?* it was explained what the thesis and the dissertation actually are and what a well-written dissertation should contain. In this session many useful tips for paper writing were also covered.

During *Literature Search and Keeping Up-to-Date* session several techniques to search online databases in a more structured and sophisticated way in order to improve search results were demonstrated. Also, an overview of online services helping keep literature up-to-date was provided.

During *Initial Registration Assessment* training the process of initial registration was explained. The session covered the university requirements for structure of initial assessment report and what information should be included in each chapter.

Aside of my research activities as a part of FSF community I plan to participate in GNU Tools Cauldron 2014 Workshop. This is to get an update of best practices with GNU tools and compilation technology.

I also attend weekly group English classes for PhD/research students to improve my writing skills.

6 Conclusion and Progress

The contribution of the research is a technique for pressure propagation through synchronisers. This mechanism is an important part of concurrency self-regulating mechanism in Astra*Kahn*. An approach to the problem will be provided in accordance with runtime system regulation policies and a tool for pressure propagation support on runtime will be implemented. The chosen pressure propagation strategy will be tested with a runtime system prototype to decide on its usability for adaptive concurrency regulation. Depending on the result, synchronisation model nondeterminism restrictions and/or strategy changes can be provided to enable efficient concurrency self-regulation on runtime.

During the first three months of the research, synchroniser structure and properties were studied. As the result, a compiler frontend was implemented[3] in the OCaml programming language. It includes parsing of synchroniser source code, basic syntactic and semantic checks, as well as building internal structures that are the basis for further analysis algorithms implementation.

References

[1] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs Journal*, 30(3), 2005.

[2] OpenMP Architecture Review Board. OpenMP application program interface version 4.0, 2013.

[3] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.0, 2012. Chapter author for Collective Communication, Process Topologies, and One Sided Communications.

[4] Intel Corporation. Intel cilk plus language extension specification version 1.2, 2013.

[5] Alex Shafarenko. Astrakahn: A coordination language for streaming networks. *CoRR*, abs/1306.6029, 2013.

[6] Robert Stephens. A survey of stream processing. *Acta Inf.*, 34(7):491–541, 1997.

[7] Edward A. Ashcroft and William W. Wadge. Lucid, a nonprocedural language with iteration. *Commun. ACM*, 20(7):519–526, 1977.

[3] Private git repository is at https://github.com/atikhono/aksync.

[8] Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[9] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman P. Amarasinghe. A stream compiler for communication-exposed architectures. In *ASPLOS*, pages 291–303, 2002.

[10] Manjunath Kudlur and Scott Mahlke. Orchestrating the execution of stream programs on multicore platforms. *SIGPLAN Not.*, 43(6):114–124, June 2008.

[11] Sardar M. Farhad, Yousun Ko, Bernd Burgstaller, and Bernhard Scholz. Orchestration by approximation: Mapping stream programs onto multicore architectures. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 357–368, New York, NY, USA, 2011. ACM.

[12] Zheng Wang and Michael F.P. O'Boyle. Partitioning streaming parallelism for multi-cores: A machine learning based approach. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 307–318, New York, NY, USA, 2010. ACM.

[13] Rebecca L. Collins and Luca P. Carloni. Flexible filters: Load balancing through backpressure for stream programs. In *Proceedings of the Seventh ACM International Conference on Embedded Software*, EMSOFT '09, pages 205–214, New York, NY, USA, 2009. ACM.

[14] Clemens Grelck, Sven-Bodo Scholz, and Alexander V. Shafarenko. A Gentle Introduction to S-Net: Typed Stream Processing and Declarative Coordination of Asynchronous Components. *Parallel Processing Letters*, 18(2):221–237, 2008.

[15] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. In *ADVANCES IN COMPUTERS*, pages 329–400. Academic Press, 1998.

[16] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[17] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan Rutten. Modeling component connectors in reo by constraint automata. *Sci. Comput. Program.*, 61(2):75–113, July 2006.

[18] Bahman Pourvatan, Marjan Sirjani, Hossein Hojjat, and Farhad Arbab. Symbolic execution of reo circuits using constraint automata. *Sci. Comput. Program.*, 77(7-8):848–869, July 2012.