Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Pipelined fission for stream programs with dynamic selectivity and partitioned state

B. Gedik*, H.G. Özsema, Ö. Öztürk

Department of Computer Engineering, Bilkent University, 06800, Ankara, Turkey

HIGHLIGHTS

- Formalizes the pipelined fission problem for streaming applications.
- Models the throughput of pipelined fission configurations.
- Develops a three-stage heuristic algorithm to quickly locate a close to optimal pipelined fission configuration.
- Experimentally evaluates the solution and demonstrate its efficacy.

ARTICLE INFO

Article history:
Received 19 February 2015
Received in revised form
2 December 2015
Accepted 3 May 2016
Available online 14 May 2016

Keywords:
Data stream processing
Auto-parallelization
Pipelining
Fission

ABSTRACT

There is an ever increasing rate of digital information available in the form of online data streams. In many application domains, high throughput processing of such data is a critical requirement for keeping up with the soaring input rates. Data stream processing is a computational paradigm that aims at addressing this challenge by processing data streams in an on-the-fly manner, in contrast to the more traditional and less efficient store-and-then process approach. In this paper, we study the problem of automatically parallelizing data stream processing applications in order to improve throughput. The parallelization is automatic in the sense that stream programs are written sequentially by the application developers and are parallelized by the system. We adopt the asynchronous data flow model for our work, which is typical in Data Stream Processing Systems (DSPS), where operators often have dynamic selectivity and are stateful. We solve the problem of *pipelined fission*, in which the original sequential program is parallelized by taking advantage of both *pipeline parallelism* and *data parallelism* at the same time. Our pipelined fission solution supports *partitioned stateful* data parallelism with dynamic selectivity and is designed for shared-memory multi-core machines. We first develop a cost-based formulation that enables us to express pipelined fission as an optimization problem. The bruteforce solution of this problem takes a long time for moderately sized stream programs. Accordingly, we develop a heuristic algorithm that can quickly, but approximately, solve the pipelined fission problem. We provide an extensive evaluation studying the performance of our pipelined fission solution, including simulations as well as experiments with an industrial-strength DSPS. Our results show good scalability for applications that contain sufficient parallelism, as well as close to optimal performance for the heuristic pipelined fission algorithm.

1. Introduction

We are experiencing a data deluge due to the ever increasing rate of digital data produced by various software and hardware sensors present in our highly instrumented and interconnected world. This data often arrives in the form of continuous streams. Examples abound, such as ticker data [41] in financial markets, call detail records [7] in telecommunications, production line diagnostics [3] in manufacturing, and vital signals [35] in healthcare. Accordingly, there is an increasing need to gather and analyze data streams in near real-time, detect emerging patterns and outliers, and take automated action. Data stream processing systems (DSPSs) [37,20,36,1,5] enable carrying out these tasks in a natural way, by taking data streams through a series of analytic operators. In contrast to the traditional store-and-process model of data management systems, DSPSs rely on the process-and-forward model and are designed to provide high throughput and timely response.

* Corresponding author.
E-mail addresses: bgedik@bilkent.edu.tr (B. Gedik), habibe.ozsema@bilkent.edu.tr (H.G. Özsema), ozturk@bilkent.edu.tr (Ö. Öztürk).

http://dx.doi.org/10.1016/j.jpdc.2016.05.003
0743-7315/© 2016 Elsevier Inc. All rights reserved.

Since performance is one of the fundamental motivations for adopting the stream processing model, optimizing the throughput of stream processing applications is an important goal of many DSPSs. In this paper, we study the problem of *pipelined fission*, that is automatically finding the best configuration of combined pipeline and data parallelism in order to optimize application throughput. Pipeline parallelism naturally occurs in stream processing applications [39]. As one of the stages is processing a data item, the previous stage can concurrently process the next data item in line. Data parallelization, aka. *fission*, involves replicating a stage and concurrently processing different data items using these replicas. Typically, data parallelism opportunities in streaming applications need to be discovered (to ensure safe parallelization) and require runtime mechanisms, such as splitting and ordering, to enforce sequential semantics [33,15].

Our goal in this paper is to determine how to distribute processing resources among the data and pipeline parallel aspects within the stream program, in order to best optimize the throughput. While pipeline parallelism is very easy to take advantage of, the amount of speed-up that can be obtained is limited by the pipeline depth. On the other hand, data parallelism, when applicable, can be used to achieve higher levels of scalability. Yet, data parallelism has limitations as well. First, the mechanisms used to establish sequential semantics (e.g., ordering) have overheads that increase with the number of replicas used. Second, and more importantly, since data parallelism is applied to a subset of operators within the chain topology, the performance is still limited by other operators for which data parallelism cannot be applied (e.g., because they are arbitrarily stateful). The last point further motivates the importance of pipelined fission, that is the need for performing combined pipeline and data parallelism.

The setting we consider in this paper is multi-core shared-memory machines. We focus on streaming applications that possess a *chain topology*, where multiple stages are organized into a series, each stage consuming data from the stage before and feeding data into the stage after. Each stage can be a *primitive* operator, which is an atomic unit, or a *composite* [22] operator, which can contain a more complex sub-topology within. In the rest of the paper, we will simply use the term operator to refer to a stage. The pipeline and data parallelism we apply are all at the level of these operators.

Our work is applicable to and is designed for DSPSs that have the following properties:

- **Dynamic selectivity**: If the number of input data items consumed and/or the number of output data items produced by an operator are **not** fixed and may change depending on the contents of the input data, the operator is said to have dynamic selectivity. Operators with dynamic selectivity are prevalent in data-intensive streaming applications. Examples of such operators include data dependent filters, joins, and aggregations.
- **Backpressure**: When a streaming operator is unable to consume the input data items as fast as they are being produced, a bottleneck is formed. In a system with backpressure, this eventually results in an internal buffer to fill up, and thus an upstream operator blocks while trying to submit a data item to the full buffer. This is called backpressure, and it recursively propagates up to the source operators.
- **Partitioned processing**: A stream that multiplexes several sub-streams, where each sub-stream is identified by its unique value for the partitioning key, is called a *partitioned stream*. An operator that independently processes individual sub-streams within a partitioned stream is called a *partitioned operator*. Partitioned operators could be stateful, in which case they maintain independent state for each sub-stream. DSPSs that support partitioned processing can apply fission for partitioned stateful operators—an important class of streaming operators [34,4].

There are several challenges in solving the pipelined fission problem we have outlined. First, we need to formally define what a valid parallelization configuration is with respect to the execution model used by the DSPS. This involves defining the restrictions on the mapping between threads and parallel segments of the application. Second, we need to model the throughput as a function of the pipelined fission configuration, so as to compare different pipelined fission alternatives among each other. Finally, even for a small number of operators, processor cores, and threads, there are combinatorially many valid pipelined fission configurations. It is important to be able to quickly locate a configuration that provides close to optimal throughput. There are two strong motivations for this. The first is to have a fast edit–debug cycle for streaming applications. The second is to have low overhead for dynamic pipelined fission, that is being able to update the parallelization configuration at run-time. Note that, the optimal pipelined fission configuration depends on the operator costs and selectivities, which are often data dependent, motivating dynamic pipelined fission. In this paper, our focus is on solving the pipelined fission problem in a reasonable time, with high accuracy with respect to throughput.

Our solution involves three components. First, we define valid pipelined fission configurations based on application of *fusion* and *fission* on operators. Fusion is a technique used for minimizing scheduling overheads and executing stream programs in a streamlined manner [25,13]. In particular, series of operators that form a *pipeline* are fused and executed by a dedicated thread, where buffers are placed between successive pipelines. On the other hand, using fission, series of pipelines that form a *parallel region* are replicated to achieve data parallelism.

Second, we model concepts such as operator compatibility (used to define parallel regions), backpressure (key factor in defining throughput), and system overheads like the thread switching and replication costs (factors impacting the effectiveness of parallelization), and use these to derive a formula for the throughput.

Last, and most importantly, we develop a heuristic algorithm to quickly locate a pipelined fission configuration that provides close to optimal performance. The algorithm relies on three main ideas: The first is to form regions based on the longest compatible sequence principle, where compatible means that a formed region carries properties that make it amenable to data parallelism as a whole. The second is to divide regions into pipelines using a greedy bottleneck resolving procedure. This procedure performs iterative pipelining, using a variable utilization-based upper bound as the stopping condition. The third is another greedy step, which resolves the remaining bottlenecks by increasing the number of replicas of a region.

We evaluate the effectiveness of our solution based on extensive analytic experimentation. We also use IBM's SPL language and its runtime system to perform an empirical evaluation. Our SPL-based evaluation shows that we can quickly locate a pipelined fission configuration that is within 5%–10% of the optimal using our heuristic algorithm.

In summary, we make the following contributions:

- We formalize the pipelined fission problem for streaming applications that are organized as a series of stages and can potentially exhibit dynamic selectivity, backpressure, and partitioned processing.
- We model the throughput of pipelined fission configurations and cast the problem of locating the best configuration as a combinatorial optimization one.
- We develop a three-stage heuristic algorithm to quickly locate a close to optimal pipelined fission configuration and evaluate its effectiveness using analytical and empirical experiments.