PIPELINED FISSION FOR STREAM PROGRAMS WITH DYNAMIC SELECTIVITY AND PARTITIONED STATE

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

By
Habibe Güldamla Özsema
December, 2014

PIPELINED FISSION FOR STREAM PROGRAMS WITH DYNAMIC SELECTIVITY AND PARTITIONED STATE
By Habibe Güldamla Özsema
December, 2014

We certify that I have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. Dr. Buğra Gedik (Advisor)

Assoc. Prof. Dr. Özcan Öztürk (Co-Advisor)

Prof. Dr. Özgür Ulusoy

Asst. Prof. Dr. Gültekin Kuyzu

Approved for the Graduate School of Engineering and Science:

Prof. Dr. Levent Onural
Director of the Graduate School

ABSTRACT

PIPELINED FISSION FOR STREAM PROGRAMS WITH DYNAMIC SELECTIVITY AND PARTITIONED STATE

Habibe Güldamla Özsema
M.S. in Computer Engineering
Advisor: Asst. Prof. Dr. Buğra Gedik
Co-Advisor: Assoc. Prof. Dr. Özcan Öztürk
December, 2014

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 this thesis, we study the problem of automatically parallelizing data stream processing applications 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, 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 and data parallelism at the same time. Our 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 to express pipelined fission as an optimization problem. The bruteforce solution of this problem takes a very long time for moderately sized stream programs. Accordingly, we develop a heuristic algorithm that can quickly, but approximately, solve this problem. We provide an extensive evaluation studying the performance of our solution, including simulations and experiments with an industrial-strength Data Stream Processing Systems (DSPS). Our results show good scalability for applications that contain sufficient parallelism, closeness to optimal performance for the algorithm.

Keywords: Data Stream Processing, Parallelization, Pipelining, Fission.

ÖZET

DEVİNGEN SEÇİCİ VE BÖLÜMLÜ DURUMSAL VERİ KATARI PROGRAMLARI İÇİN ARDIŞIK DÜZENLENMİŞ FİZYON

Habibe Güldamla Özsema
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Y. Doç. Dr. Buğra Gedik
Tez Eş Danışmanı: Doç. Dr. Özcan Öztürk
Aralık, 2014

Günümüzde, çevrimiçi veri katarı formatında bulunan kullanılabilir dijital bilgi gittikçe artan bir orana sahiptir. Birçok uygulama alanında, bu tür verilerin yüksek üretilen iş kapasiteli olarak işlenmesi, yükselen girdi oranlarına ayak uydurmak için kritik bir gerekliliktir.Veri katarı işleme, bu zorluğu veriyi annda işleme tarzı ile ele almayı amaçlayan bir hesaplama örneklemidir.

Bu tez, veri katarı işleme uygulamalarının otomatik bir şekilde paralelleştirilme problemini, üretilen işi arttırarak nasıl çözüleceğini gösterir. Paralelleştirme işlemi, veri katarı uygulamalarının, uygulama geliştiricileri tarafından sırasıyla yazılması ve sistem tarafından paralelleştirilmesi şeklinde otomatiktir. Bu tezde, devingen seçici ve durumsal işleçler kullanılan eşzamansız veri akş modeli benimsenmiştir. Ardışık düzenlenmiş fizyon problemi, orijinal sıralı programda ardışık düzenlenmiş ve veri paralelleştirmesinden faydalanarak çözülmüştür. Ardışık düzenlenmiş fizyon çözümü, bölümlü durumsal veri paralelleştirmeyi desteklemektedir ve paylaşımlı bellekli çok çekirdekli makineler için tasarlanmıştır.

İlk olarak ardışık düzenlenmiş fizyon problemi, maliyet tabanlı formülasyonla optimizasyon problemine indirgenmiştir. Bu problemin kapsamlı çözümü çok zaman aldığı için, bu problemi hızlı ve yaklaşık olarak çözen bulgusal çözüm önerilmiştir. Tezde önerilen yaklaşımın, simülasyonlarla ve endüstriyel Veri Katarı İşleme Sistemleri (VKİS) ile kapsamlı olarak değerlendirilmesi yapılmıştır. Elde edilen sonuçların, yeterli paralelleştirme içeren programlar için iyi bir ölçeklenebilirlik ve optimum performansa yakınlık sağladığı görülmüştür.

Anahtar sözcükler: Veri Katarı İşleme, Paralelleştirme, Ardışık Düzenleme,Fizyon.

Acknowledgement

I would like to express my special appreciation and thanks to my advisor Asst. Prof. Dr. Buğra Gedik not only for giving me invaluable support in supervision of this thesis, but also for his guidance and support to keep me motivated all the time.

I would also like to thank my advisor Assoc. Prof. Dr. Özcan Öztürk for introducing me with parallel systems in the first place and his help to get the opportunity to carry out my thesis in Bilkent University.

I would like to thank to Prof. Dr. Özgür Ulusoy and Asst. Prof. Dr. Gültekin Kuyzu for kindly accepting to spend their valuable time to review and evaluate my thesis.

I would like to express my thanks to STM AS for permitting and supporting me to continue my M.S. education.

I also thank to Scientific and Technical Research Council of Turkey (TÜBİTAK) for financially supporting me in my graduate education and MS thesis.

I am grateful to Ceren and Berk Can for their endless and unconditional friendship.

Last but not the least, I would like to express my appreciation to my mother Nevin for supporting me and encouraging me with her wishes. I would also thank to my sister Gülperi as she has been a tremendous mentor for me.

Contents

List of Figures

List of Tables

Chapter 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 [1] in financial markets, call detail records [2] in telecommunications, production line diagnostics [3] in manufacturing, and vital signals [4] 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) [5, 6, 7, 8, 9] 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.

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 thesis, 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 [10]. 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 [11, 12].

Our goal in this thesis 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 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 thesis 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 [13] operator, which can contain a more complex sub-topology within. In the rest of the thesis, 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 [14, 15].

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 thesis, 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 [16, 17]. 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 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 to 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.

The rest of this thesis is organized as follows. Chapter 2 lays the necessary background for our work. Chapter 3 presents our model for capturing the throughput of a given parallelization configuration that involves pipeline as well as data parallelism. It also formalizes our problem as a combinatorial optimization one. Chapter 4 presents our heuristic solution to the problem of quickly finding a close to optimal pipelined fission configuration. Chapter 5 presents our evaluation, including analytical as well as empirical results. Chapter 6 overviews related work and Chapter 7 concludes the thesis.

Chapter 2

Background

In this section, we summarize the terminology used for the pipelined fission problem, and outline a system execution model that will guide the problem formulation and solution used in the rest of the thesis.

2.1 Terminology and Definitions

A stream graph is a set of operators connected to each other via streams. As mentioned earlier, we consider graphs with chain topology in this work. Figure 2.1 summarizes the terminology used to define our pipelined fission problem.

There are two operator properties that play an important role in pipelined fission, namely selectivity and state.

- Selectivity of an operator is the number of items it produces per number of items it consumes. It could be less than one, in which case the operator is selective; it could be equal to one, in which case the operator is one-to-one; or it could be greater than one, in which case the operator is prolific.
- State specifies whether and what kind of information is maintained by the operator across firings. An operator could be stateless, in which case it does not maintain any state across firings. It could be partitioned stateful, in

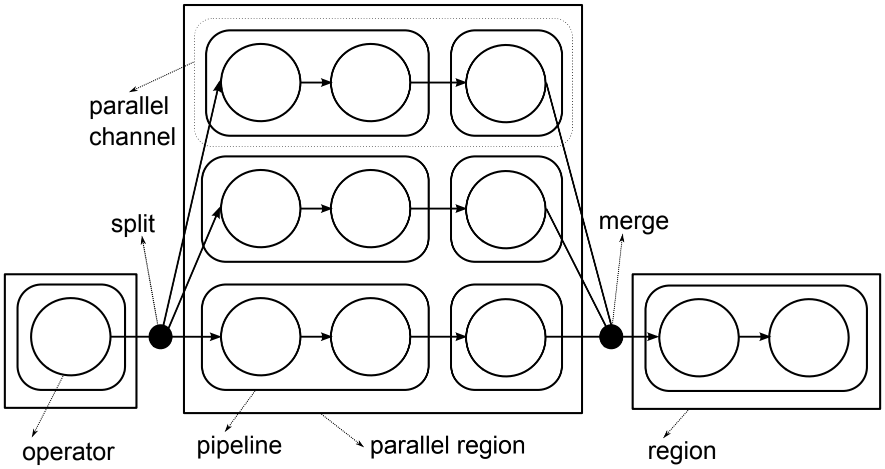


Figure 2.1: Pipelined fission terminology.

which case it maintains independent state for each sub-stream determined by a partitioning key. Finally, an operator could be stateful without a special structure.

We name a series of operators fused together as a pipeline. A series of pipelines replicated as a whole is called a parallel region. Series of pipelines that fall between parallel regions form simple regions. Each replica within a parallel region is called a parallel channel. A parallel channel contains replicas of the pipelines and operators of a parallel region. In order to maintain sequential program semantics under selective operators, split and merge operations are needed before and after a parallel region, respectively. The split operation assigns sequence numbers to tuples and distributes them over the parallel channels, such as a hash-based splitter for a partitioned stateful parallel region. The merge operation unions tuples from different parallel channels and orders them based on their sequence numbers. A parallel region cannot contain an arbitrarily stateful operator [11] and thus such regions are formed by stateless and partitioned stateful operators.

2.2 Execution model

A distributed stream processing middleware typically executes data flow graphs by partitioning them into basic units called processing elements. Each processing element contains a sub-graph and can run on a different host. For small and medium-scale applications, the entire graph can map to a single processing element. Without loss of generality, in this thesis we focus on a single multi-core host executing the entire graph. Our pipelined fission technique can be applied independently on each host when the whole application consists of multiple distributed processing elements.

In this thesis, we follow an execution model based on the SPL (Stream Processing Language) runtime [18], which has been used in a number of earlier studies as well [10, 11, 12, 19, 17]. In this model, there are two main sources of threading, which contribute to the execution of the stream graph. The first one is operator threads. Source operators, which do not have any input ports, are driven by their own operator threads. When a source operator makes a submit call to send a tuple to its output port, this same thread executes the rest of the downstream operators in the stream graph. As a result, the same thread can traverse a number of operators, before eventually coming back to the source operator to execute the next iteration of its event loop. This behavior is because the stream connections in a processing element are implemented via function calls. Using function calls yields fast execution, avoiding scheduler context switches and explicit buffers between operators. This optimization is known as operator fusion [16, 17].

The second source of threading is threaded ports. Threaded ports can be inserted at any operator input port. When a thread reaches a threaded port, it inserts the tuple at hand into the threaded port buffer, and goes back to executing upstream logic. A separate thread, dedicated to the threaded port, picks up the queued tuples and executes the downstream operators. In pipelined fission, we use threaded ports to ensure that each pipeline is run by a separate thread. For instance, in Figure 2.1 there are 8 threads, where the scheduling of threads to the processor cores is left to the operating system.

The goal of our pipelined fission solution is to automatically determine a parallelization configuration, that is the pipelines, regions, and number of replicas, so as to maximize the throughput.

Chapter 3

Model

In this section, we model the pipelined fission problem and present a brute-force approach to find a parallelization configuration that maximizes the throughput.

3.1 Application Model

We start with modeling the application topology, the operators, and the parallelization configuration.

Topology. We consider applications that have a chain topology. The operators that participate in the chain can be composite and have more complex topologies within, as long as they fit into one of the operator categories described below.

Let $O = \{o_i \mid i \in [1..N]\}$ be the set of operators in the application. Here, $o_i \in O$ denotes the i^{th} operator in the chain. o_1 is the source operator and o_N is the sink operator. For $1{<}i{\leq}N$, operator o_i has o_{i-1} as its upstream operator and for $1{\leq}i{<}N$, o_i has o_{i+1} as its downstream operator. Figure 3.1 shows an example chain topology with $N = 5$ operators.

Operators. For $o \in O$, $k(o) \in \{\mathrm{f}, \mathrm{p}, \mathrm{s}\}$ denotes the operator kind: f is for stateful, p is for partitioned stateful, and s is for stateless. For a partitioned stateful operator o (that is $k(o) = \mathrm{p}$), $a(o)$ specifies the partitioning key, which is a set of stream attributes. $s(o)$ denotes the selectivity of an operator, which can

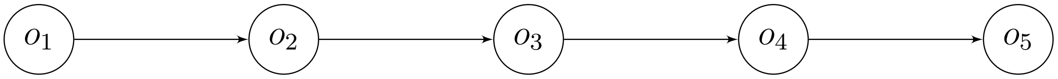


Figure 3.1: A chain topology with 5 operators.

go over 1 for prolific operators — operators that can produce one or more tuples per input tuple consumed.

For $o \in O$, $f\langle o\rangle : \mathbb{N}^+ \rightarrow \mathbb{R}$ is a base scalability function for operator o. Here, $f\langle o\rangle(x) = y$ means that x copies of operator o will raise the throughput to y times the original, assuming no parallelization overhead. We have $k(o_i) = \text{s} \Rightarrow f\langle o_i\rangle = f_l$, where f_l is the linear scalability function, that is $f_l(x) = x$. In other words, for stateless operators, the base scalability function is linear. For partitioned stateful operators, including parallel sources and sinks, bounded linear functions are more common, such as:

$$f_b(x; u) = \begin{cases} x & \text{if } x \leq u \\ u & \text{otherwise} \end{cases}$$

Here, $f_b(; u)$ is a bounded linear scalability function, where u specifies the maximum scalability value. For partitioned stateful operators, the size of the partitioning key's domain could be a limiting factor on the scalability that could be achieved. For parallel sources and sinks, the number of distinct external sources and sinks could be a limiting factor (e.g., number of TCP end points, number of data base partitions, etc.).

Parallelization configuration. Let us denote the set of threads used to execute the stream program as $T = \{t_i \mid i \in [1..|T|]\}$. The number of replicas for operator $o \in O$ is denoted by $r(o) \in \mathbb{N}^+$. Note that, we have $k(o) = \text{f} \Rightarrow r(o_i) = 1$, as stateful operators cannot be replicated.

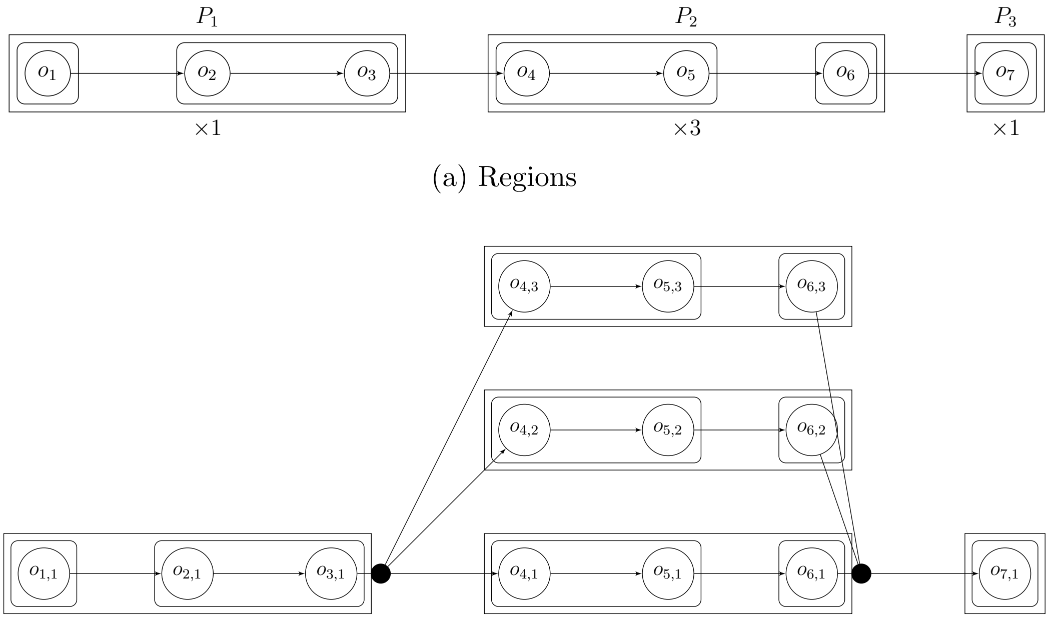
Let us denote the j^{th} replica of an operator o_i as $o_{i,j}$ and the set of all operator replicas as $V = \{o_{i,j} \mid o_i \in O \wedge j \in [1..r(o_i)]\}$. We define $m : V \rightarrow T$ as the operator to thread mapping that assigns operator replicas to threads. $m(o_{i,j}) = t$ means that operator o_i's j^{th} replica is assigned to thread t. There are a number of rules about this mapping that restrict the set of possible mappings to those that are consistent with the execution model we have outlined earlier. We first define additional notation to formalize these rules.

Given $O' \subseteq O$, we define a boolean predicate $L(O')$ that captures the notion of a sequence of operators. Formally, $L(O') \equiv \{o_{i_1}, o_{i_2}\} \subset O' \Rightarrow \forall_{i_1 \leq i \leq i_2}, o_i \in O'$. There are two kinds of sequences we are interested in. The first one is called a non-replicated sequence and is defined as $L_s(O', r) \equiv L(O') \wedge \forall_{o \in O'}, r(o) = 1$. In a non-replicated sequence, all operators have a single replica. The second is called a replicated sequence and is defined as $L_p(O', r) \equiv L(O') \wedge \forall_{o \in O'}, (k(o) \neq \mathrm{f} \wedge r(o) = l) \wedge \bigcap_{o \in O', k(o) = \mathrm{p}} a(o) \neq \emptyset$. That is, a group of operators are considered a replicated sequence if and only if they form a sequence, they do not include a stateful operator, they all have the same number of replicas, and if there are any partitioned stateful operators in the sequence, they have compatible partitioning keys[1]. We will drop r, that is the function that defines the replica counts for the operators, from the parameter list of the sequence defining predicates, L_s and L_p, when it is obvious from the context.

With these definitions, we list the following rules for the operator replica to thread mapping function, m:

- $m(o_{i_1,j_1}) = m(o_{i_2,j_2}) \Rightarrow j_1 = j_2$. I.e., operator replicas from different channels are not assigned to the same thread. Here, channel corresponds to the replica index.
- $t = m(o_{i_1,j}) = m(o_{i_2,j}) \Rightarrow \exists\, O' \,\text{s.t.}\, \{o_{i_1}, o_{i_2}\} \subseteq O' \subset O \wedge (L_s(O') \vee L_p(O')) \wedge (\forall_{o_i \in O'}, m(o_{i,j}) = t)$. I.e., if two operator replicas are assigned to the same thread, they must be part of a replicated or non-replicated sequence and all other operator replicas in between these two on the same channel should be assigned to the same thread.
- $m(o_{i_1,j}) = m(o_{i_2,j}) \Rightarrow \forall_{l \in [1..r(o_{i_1})]}, m(o_{i_1,l}) = m(o_{i_2,l})$. I.e, if two operator replicas are assigned to the same thread, their sibling operator replicas should share their threads as well. For instance, if $o_{1,1}$ and $o_{2,1}$ both map

[1] In practice, there is also the requirement that these keys are forwarded by the other operators in the sequence [11], but such details do not impact our modeling.



(a) Regions

(b) Pipelines

Figure 3.2: Regions and pipelines.

to t_1, then their siblings $o_{1,2}$ and $o_{2,2}$ should share the same thread, say t_2.

Regions and Pipelines. The above rules divide the program into regions and these regions into sub-regions that we call pipelines, as shown in the example in Figure 3.2.

In this example, we have 3 parallel regions: $P_1 = \{P_{1,1}, P_{1,2}\}$, $P_2 = \{P_{2,1}, P_{2,2}\}$, and $P_3 = \{P_{3,1}\}$. The first region P_1 has a single replica, that is $r(P_1) = 1$ and it consists of two pipelines, namely $P_{1,1}$ and $P_{1,2}$. The first pipeline has a single operator inside, whereas the second one has two operators. Concretely, we have $P_{1,1} = \{o_1\}$ and $P_{1,2} = \{o_2, o_3\}$. The second region has $r(P_2) = 3$, as there are 3 parallel channels, and it consists of 2 pipelines, namely $P_{2,1}$ and $P_{2,2}$. We have $P_{2,1} = \{o_4, o_5\}$ and $P_{2,2} = \{o_6\}$. Finally, the third region is $P_3 = \{P_{3,1}\}$, where $r(P_3) = 1$ and $P_{3,1} = \{o_7\}$.

Given the thread mapping function m and the replica function r, the set of regions formed is denoted by $\mathcal{P}(m, r)$ or $\mathcal{P}$ for short. To find the first region, $P_1 \in \mathcal{P}$, we start from the source operator o_1 and locate the longest sequence of operators $O' \subset O$ s.t. $o_1 \in O' \wedge (L_s(O') \vee L_p(O'))$. We can apply this process successively, starting from the next operator in line that is not part of the current set of regions, until the set of all regions, $\mathcal{P}$, is formed. The pipelines for a given region are similarly formed by grouping operators whose first replica are assigned to the same thread.

For each pipeline $P_{i,j} \in P_i \in \mathcal{P}$, there are $r(P_i)$ replicas and a different thread executes each pipeline replica. Then the total number of threads used is given by $\sum_{P_i \in \mathcal{P}} r(P_i) \cdot |P_i|$. In the example above, we have 9 threads and 13 operator replicas.

3.2 Modeling the Throughput

Our goal is to define the throughput of a given configuration $\mathcal{P}$. Once the throughput is formulated, we can cast our problem as an optimization one, where we aim to find the thread mapping function (m) and the operator replica counts (r) that maximize the throughput.

To formalize the throughput, we start with a set of helper definitions. We denote the kind of a region as $k(P_i)$, and define:

$$k(P_i) = \begin{cases} \text{f} & \text{if } \exists\, o_k \in P_{i,j} \in P_i \text{ s.t. } k(o_k) = \text{f} \\ \text{s} & \text{if } \forall_{o_k \in P_{i,j} \in P_i}\, k(o_k) = \text{s} \\ \text{p} & \text{otherwise} \end{cases} . \tag{3.1}$$

We denote the selectivity of a pipeline $P_{i,j}$ as $s(P_{i,j}) = \prod_{o_k \in P_{i,j}} s(o_k)$, the selectivity of a region P_i as $s(P_i) = \prod_{P_{i.j} \in P_i} s(P_{i,j})$, and the selectivity of the entire flow $\mathcal{P}$ as $s(\mathcal{P}) = \prod_{P_i \in \mathcal{P}} s(P_i)$. We denote the cost of a pipeline as $c(P_{i,j})$ and define it as:

$$c(P_{i,j}) = \sum_{o_k \in P_{i,j}} s_k(P_{i,j}) \cdot c(o_k). \tag{3.2}$$

Here, $s_k(P_{i,j}) = \prod_{o_l \in P_{i,j}, l<k} s(o_l)$ is the selectivity of the sub-pipeline up to and excluding operator o_k.

Region throughput. We first model a region's throughput in isolation, assuming no other regions are present in the system. Let $R(P_i)$ denote the maximum input throughput supported by a region under this assumption. And let $R_j(P_i)$ denote the output throughput of the first j pipelines in the region assuming the remaining pipelines have zero cost. Furthermore, let $R(P_{i,j})$ denote the input throughput of the pipeline $P_{i,j}$ if all other pipelines had zero cost (making it the bottleneck of the system).

We have $R_0(P_i) = \infty$ and also for $j > 0$:

$$R_j(P_i) = \begin{cases} s(P_{i,j}) \cdot R_{j-1}(P_i) & \text{if } R_{j-1}(P_i) < R(P_{i,j}) \\ s(P_{i,j}) \cdot R(P_{i,j}) & \text{otherwise} \end{cases}. \tag{3.3}$$

In essence, Equation 3.3 models the backpressure. If the input throughput of a pipeline, when considered alone, is higher than the output throughput of the sub-region formed by the pipelines before it, then the latter throuhgput is used to compute the pipeline's output throughput when it is added to the sub-region. This represents the case when the pipeline in question is not the bottleneck. The other case is when the pipeline's input throughput, when considered alone, is lower than the output throughput of the sub-region formed by the pipelines before it. In this case, the former throughput is used to compute the pipeline's output throughput when it is added to the sub-region. This represents the case when the pipeline in question is the bottleneck within the sub-region.

The throughput of a pipeline by itself, that is $R(P_{i,j})$, can be represented as:

$$R(P_{i,j}) = (c(P_{i,j}) + h(P_{i,j}))^{-1}, \tag{3.4}$$

where $h(P_{i,j})$ is the cost of switching threads between sub-regions, defined as:

$$h(P_{i,j}) = \delta \cdot (\mathbf{1}(j{>}1) + \mathbf{1}(j{<}|P_i|) \cdot s(P_{i,j})). \tag{3.5}$$

Here, δ is the thread switching overhead due to the queues involved in-between. The input overhead is incurred for the pipelines except the first one, and the output overhead is incurred for the pipelines except the last one.

With these definitions at hand, we can define the input throughput $R(P_i)$ as the output throughput of the region divided by the region's selectivity. That is:

$$R(P_i) = R_{|P_i|}(P_i)/s(P_i). \tag{3.6}$$

Parallel region throughput. The next step is to compute the throughput of a parallel region. For that purpose, we first define an aggregate scalability function $f\langle P_i \rangle$ for the region P_i as:

$$f\langle P_i \rangle(x) = \min_{o_k \in P_{i,j} \in P_i} f\langle o_i \rangle(x). \tag{3.7}$$

The aggregate scalability function for a region simply takes the smallest scalability value from the scalability functions of the constituent operators within the region.

We denote the parallel throughput of a region P_i as $R^*(P_i)$ and define it as follows:

$$R^*(P_i) = \left(c_p \cdot \log_2(r(P_i)) + \frac{1}{R(P_i) \cdot f\langle P_i \rangle(r(P_i))} \right)^{-1}. \tag{3.8}$$

Here, c_p is the replication cost factor for a parallel region. Recall that a parallel region needs to reorder tuples. In the presence of selectivity, this often requires attaching sequence numbers to tuples and re-establishing order at the end of the parallel region. The re-establishment of order takes time that is logarithmic in the number of channels, per tuple. However, such processing typically has a low constant compared to the cost of the operators.

Let $R^+(P_i)$ be the parallel throughput of the region when it is considered within the larger topology that contains the other regions, albeit assuming that all other regions have zero cost. We have:

$$R^+(P_i) = \left(h(P_i) + \frac{1}{R^*(P_i)} \right)^{-1}. \tag{3.9}$$

Here, $h(P_i)$ is the cost of switching threads between regions. We have:

$$h(P_i) = \delta \cdot (\mathbf{1}(i > 1) + \mathbf{1}(i < |\mathcal{P}|) \cdot s(P_i)). \tag{3.10}$$

Throughput of a program. Given these definitions, we are ready to define the input throughput of a program, denoted as $R(\mathcal{P})$. We follow the same approach as we did for regions formed out of pipelines.

Let us define the output throughput of the first k regions as $R_k(\mathcal{P})$, assuming the downstream regions have zero cost. We have $R_0(\mathcal{P}) = \infty$, and for $i > 0$:

$$R_i(\mathcal{P}) = \begin{cases} s(P_i) \cdot R_{i-1}(\mathcal{P}) & \text{if } R_{i-1}(\mathcal{P}) < R^+(P_i) \\ s(P_i) \cdot R^+(P_i) & \text{otherwise} \end{cases}. \tag{3.11}$$

By dividing the output throughput of the program to its selectivity, we get:

$$R(\mathcal{P}) = R_{|\mathcal{P}|}(\mathcal{P})/s(\mathcal{P}). \tag{3.12}$$

Bounded throughput. So far we have computed the unbounded throughput. In other words, we have assumed that each thread has a core available to itself. However, in practice, there could be more threads than the number of cores available. For instance, replicating a region with 3 pipelines 3 times will result in 9 threads, but the system may only have 8 cores. However, replicating the region 2 times will result in an underutilized system that has only 6 threads and thus not all cores can be used.

Let C denote the number of cores in the system. We denote the bounded throughput of a program with parallelization configuration of m (the thread mapping function) and r (the operator replica counts) as $R(\mathcal{P}(m, r), C)$. The bounded throughput is simply computed as the unbounded throughput divided by the utilization times the number of cores. Formally,

$$R(\mathcal{P}, C) = R(\mathcal{P}) \cdot \frac{C}{U(\mathcal{P})}. \tag{3.13}$$

Here $U(\mathcal{P})$ is the utilization for the unbounded throughput. Equation 3.13 simply scales the unbounded throughput by multiplying it with the ratio of the maximum utilization that can be achieved (which is C) to the unbounded utilization. We assume that the cost due to scheduling of threads by the operating system is negligible. For instance, if the unbounded throughput is 3 units, but results in a utilization value of 6 and the system has only 4 cores, then the bounded throughput is given by $3 \cdot (4/6) = 2$ units.

The computation of the utilization, $U(\mathcal{P})$, is straightforward. We already have a formula for the input throughput of the program, which can be used to compute the input throughputs of the parallel regions and the pipelines. Multiplying input throughputs of the pipelines with with the pipeline costs would give us the utilization, after adding the overheads for the thread switching and scalability. Overall utilization can be expressed as:

$$\begin{aligned} U(\mathcal{P}) = R(\mathcal{P}) \cdot \sum_{1 \leq i < |\mathcal{P}|} s_i(\mathcal{P}) \cdot \big(c_p \cdot \log_2(r(P_i)) + \\ h(P_i) + \sum_{1 \leq j < |P_i|} s_j(P_i) \cdot (h(P_{i,j}) + c(P_{i,j})) \big). \end{aligned} \tag{3.14}$$

Here, $s_j(P_i) = \prod_{1 \leq l < j} s(P_{i,l})$ is the selectivity of the region P_i up to and excluding the j^{th} pipeline, and $s_j(\mathcal{P}) = \prod_{1 \leq l < j} s(P_l)$ is the selectivity of the program $\mathcal{P}$ up to and excluding the j^{th} region.

Optimization. Our ultimate goal is to find $\text{argmax}_{m,r} R(\mathcal{P}(m,r), C)$, where m is subject to the rules we have outlined earlier. One way to solve this problem is to combinatorially generate all possible parallel configurations. This can be achieved via a recursive procedure that takes as input the maximum number of threads, say M, and the set of operators O, and generates all valid parallelization configurations of the operators that uses at most M threads. Let us denote the set of configurations generated by such a generator as $D(O, M)$. Then we can compute $\text{argmax}_{(m,r) \in D(O,M)} R(\mathcal{P}(m,r), C)$ as the optimal configuration. There are two problems with this approach. First, and the more fundamental one, is that, the computation of $D(O, M)$ takes a very long time even for a small number of threads and operators; and this time grows exponentially, since the number of variations increase exponentially (both with increasing number of operators and maximum number of threads). Figure 3.3 shows the number of parallel configurations as a function of the number of operators in the chain and the maximum number of threads used. Second, we need to pick a reasonable value for M, which is typically greater than C. It can be taken as a constant times the number of cores, that is $k \cdot C$. Unfortunately, using a large constant will result in

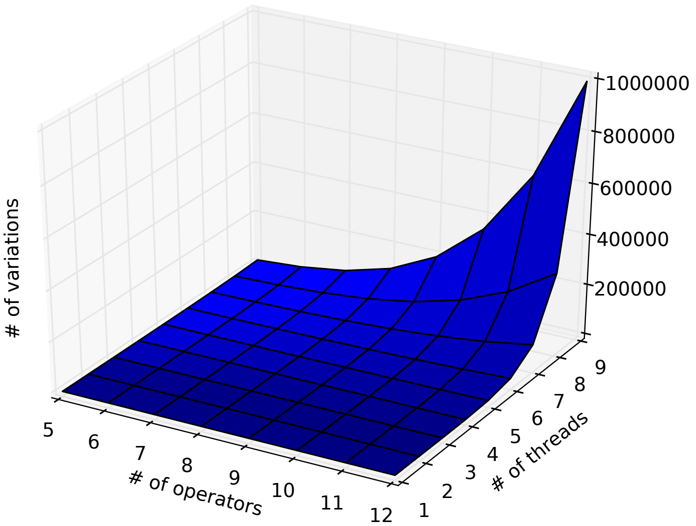


Figure 3.3: Number of parallel program configurations as a function of number of threads (M) and number of operators ($|O|$).

an excessively long running time for the configuration generation algorithm. On the other hand, using a small constant will have the risk of finding a sub-optimal solution.

Chapter 4

Solution

In this section we present an algorithm to quickly solve the pipelined fission problem that was formalized in Chapter 3. Our algorithm is heuristic in nature and trades off throughput optimality to achieve reasonable performance in terms of solution time. Despite this, our results, presented later in Chapter 5, show that not only does our algorithm achieve close to optimal throughput, but also it outperforms optimal versions of fission-only and pipelining-only alternatives.

```
Algorithm 1: PipelinedFission($O$,$M$,$\alpha$)
Data: $O$: operators (with their costs, $c$; selectivities, $s$; and state kinds, $k$), $M$: number of cores, $\alpha$:
      fusion cost threshold
Result: Pipelined fission configuration
$R \leftarrow$ ConfigureRegions($O, \alpha$)                        ▷ Configure regions
$t \leftarrow 0$; $P \leftarrow \varnothing$; $r \leftarrow \varnothing$                          ▷ Initialize best settings
for $i \in 0.1 \cdot [0..10]$ do                               ▷ Range of utilization scalers
    $P' \leftarrow$ ConfigurePipelines($R, i \cdot M$)                ▷ Configure pipelines
    $r' \leftarrow$ ConfigureReplicas($R, P', M$)                ▷ Configure # of replicas
    $t' \leftarrow$ ComputeTput($R, P', r'$)                      ▷ Compute the throughput
    if $t' > t$ then $t \leftarrow t'$; $P \leftarrow P'$; $r \leftarrow r'$
return $\langle R, P, r \rangle$                                 ▷ Return the final configuration
```

4.1 Overview

Algorithm 1 presents our solution, which consists of three phases, namely (*i*) region configuration, (*ii*) pipeline configuration, and (*iii*) replica configuration.

The region configuration phase divides the chain of operators into chains of regions. This is done based on the compatibility of the successive operators in terms of their state, while avoiding the creation of small regions that cannot achieve effective parallelization. The second and third phases are used to configure pipeline and data parallelism, respectively. That is, pipeline configuration creates pipelines within regions, and replica configuration determines the number of replicas for the regions. These two phases are run multiple times, each time with a different amount of CPU utilization reserved for them, but always summing up to the number of CPUs available in the system. In particular, we range the fraction of the CPU utilization reserved for pipelining from 0% to 100%, in increments of 10%. The reason for running the pipeline and replica configuration phases with differing shares of CPU utilization is that, we do not know, apriori, how much parallelism is to be reserved for pipelining versus how much for fission, in order to achieve the best performance with respect to throughput. Among the multiple runs of the second and the third phases, we pick the one that gives the highest throughput as our final pipelined fission solution. Internally, pipeline configuration phase and replica configuration phase work similarly. In pipeline configuration, we repeatedly locate the bottleneck pipeline and divide it. In replica configuration, we repeatedly locate the bottleneck region and increase its replica count. In what follows, we further detail the three phases of the algorithm.

4.2 Region Configuration

Algorithm 2 presents the region configuration phase, where we divide the chain of operators into a chain of regions. The algorithm consists of two parts. In the first part, we form effectively parallelizable regions. This may leave out some operators unassigned. In the second phase, we merge the consecutive unassigned operators into regions as well.

The first for loop in Algorithm 2 represents the first step. We form regions by iterating over the operators. We keep accumulating operators into the current

Algorithm 2: CONFIGUREREGIONS(O,α)

```
Data: $O$: operators, $\alpha$: fusion cost threshold
Result: Regions
$R \leftarrow \{\}$                                   ▷ The list of regions that will hold the final result
$C \leftarrow \{\}$                                   ▷ The list of operators in the current potential region
for $i \leftarrow 1; i \leq |O|; i \leftarrow i + 1$ do        ▷ For each operator
    ▷ The current region borrows the operator's properties
    if $k(o_i) \neq \mathrm{f} \wedge (|C| = 0 \vee k(C) = \mathrm{s})$ then
        $k(C) \leftarrow k(o_i)$                      ▷ Update the region's kind
        if $k(o_i) = \mathrm{p}$ then                 ▷ If $o_i$ is partitioned
            $a(C) \leftarrow a(o_i)$                  ▷ Update current region's key
    ▷ The current region stays partitioned, possibly with a broadened key
    else if $k(o_i) = \mathrm{p} \wedge a(o_i) \subseteq a(C)$ then
        $a(C) \leftarrow a(o_i)$                      ▷ Update current region's key
    ▷ The current region and the operator are incompatible
    else if $k(o_i) \neq \mathrm{s}$ then
        if COMPUTECOST($C$) $> \alpha$ then           ▷ Region is costly enough
            $R \leftarrow R \cup \{C\}$               ▷ Materialize the region in $R$
            $m(o) \leftarrow 1, \forall_{o \in C}$    ▷ Mark region's operators as assigned
        $C \leftarrow \{\}$                           ▷ Reset the current region
        if $k(o_i) \neq \mathrm{f}$ then              ▷ Parallelizable operator
            $i \leftarrow i - 1$                      ▷ Redo iteration with empty current region
        continue
    $C \leftarrow C \cup \{o_i\}$                     ▷ Add the operator to the current region
▷ Handle the pending region at loop exit
if COMPUTECOST($C$) $> \alpha$ then                   ▷ Region is costly enough
    $R \leftarrow R \cup \{C\}$                       ▷ Materialize the region in $R$
    $m(o) \leftarrow 1, \forall_{o \in C}$            ▷ Mark region's operators as assigned
▷ Merge all consecutive unassigned ops to a region
$C \leftarrow \{\}$                                   ▷ Reset the current region
for $i \leftarrow 1; i \leq |O|; i \leftarrow i + 1$ do        ▷ For each operator
    if $m(o_i) = 1 \wedge |C| > 0$ then               ▷ We have a complete run
        $R \leftarrow R \cup \{C\}$                   ▷ Materialize the region in $R$
        $C \leftarrow \{\}$                           ▷ Reset the current region
    else                                              ▷ Run of unassigned operators continues
        $C \leftarrow C \cup \{o_i\}$                 ▷ Add the operator to the current region
return $R$                                            ▷ The final set of regions
```

region, as long as the operators are not stateful or incompatible. Stateless operators are always compatible with the current region. Partitioned stateful operators are only compatible if their key is the same as the key of the active region so far, or broader (has less attributes). In the latter case, the region's key is updated accordingly. When an incompatible operator is encountered, the current region that is formed so far is completed. However, this region is discarded if its overall cost is below the fusion cost threshold, α. The motivation behind this is that, if a region is too small in terms of its cost, parallelization overhead will dominate and effective parallelization is not attainable.

Once a region is completed, the algorithm continues with a fresh region, starting from the next operator in line (the one that ended the formation of the former

Algorithm 3: ConfigurePipelines(R,M)

```
Data: $R$: regions, $M$: number of cores
Result: Pipeline configuration
$P \leftarrow R$                                          ▷ Initialize the set of pipelines to regions
▷ Find the bottleneck pipeline ($C$), and compute the total utilization ($U$)
$\langle C, U \rangle \leftarrow$ FindBottleneckPipeline($R, P$)
while $U \leq M$ do                                       ▷ System is not fully utilized
    ▷ Find the best split for the pipeline (maximizes throughput)
    $o_i \leftarrow \text{argmax}_{o_k \in C}$ ComputeTput($\{o_j \in C \mid j<k\}, \{o_j \in C \mid j \geq k\}$) $C_0 \leftarrow \{o_j \in C \mid j < i\}$ ▷ First half of the best split
    $C_1 \leftarrow \{o_j \in C \mid j \geq i\}$           ▷ Second half of the best split
    if ComputeTput($\{C_0, C_1\}$) $\leq$ ComputeTput($C$) then
        break                                             ▷ No further improvement is possible
    $P \leftarrow P \setminus \{C\} \cup \{C_0, C_1\}$     ▷ Split the pipeline
    $\langle C, U \rangle \leftarrow$ FindBottleneckPipeline($P$)   ▷ Re-eval. for next iter.
return $P$                                                ▷ The final set of pipelines
```

region). The first step of the algorithm ends, when all operators are processed. In the second step, the operators that are left without a region assignment are handled. Such operators are either stateful or cannot form a sufficiently costly region with the other operators around them. In the second step, consecutive operators that are not assigned a region are put into their own region. However, these regions cannot benefit from parallelization in the pipeline and replica configuration phases that are described next.

4.3 Pipeline Configuration

Algorithm 3 describes the pipeline configuration phase. We start with each region being a pipeline and iteratively split the bottleneck pipeline. The FindBottleneckPipeline procedure is used to find the bottleneck pipeline. This procedure simply computes the unbounded throughput of the program as new pipelines are successively added, using the formalization from Chapter 3, and selects the last pipeline that resulted in a reduction in the unbounded throughput as the bottleneck one. It then reports this bottleneck pipeline, together with the the utilization of the current configuration. If the utilization is above or equal to the total utilization reserved for the pipeline configuration phase (recall Algorithm 1), then the iteration is terminated and the pipeline configuration phase is over. Otherwise, i.e., if there is room available for an additional pipeline, we find

Algorithm 4: CONFIGUREREPLICAS(R, P, M)

```
Data: $R$: regions, $P$: pipelines, $M$: number of cores
Result: Set of number of replicas of each region
$r[C] \leftarrow 1, \forall_{C \in R}$                              ▷ Initialize the replica counts to 1
▷ Find the bottleneck region ($C$), and compute the total utilization ($U$)
$\langle C, U \rangle \leftarrow$ FINDBOTTLENECKREGION$(R, P, r)$
while $U \leq M$ do                                                  ▷ System is not fully utilized
    $t \leftarrow$ CALCULATETPUT$(R, P, r)$                          ▷ Baseline throughput
    $r[C] \leftarrow r[C] + 1$                                       ▷ Increse the channel count
    if $t \geq$ CALCULATETPUT$(R, P, r)$ then                        ▷ Throughput decreased
        $r[C] \leftarrow r[C] - 1$                                   ▷ Revert back
        break                                                        ▷ No further improvement is possible
    $\langle C, U \rangle \leftarrow$ FINDBOTTLENECKREGION$(R, P)$   ▷ Re-eval. for next iter.
return $r$                                                           ▷ Return the replica counts
```

the best split within the bottleneck pipeline. This is done by considering each operator as a split point and picking the split that provides the highest unbounded throughput. However, if the unbounded throughput of this split configuration of two consecutive pipelines is lower compared to the original single pipeline (which might happen for low cost pipelines due to the impact of thread switching overhead), we again terminate the pipeline configuration phase. This is because, if the bottleneck pipeline cannot be improved, then no overall improvement is possible.

4.4 Replica Configuration

Algorithm 4 describes the replica configuration phase. It is similar in structure to the pipeline configuration phase. However, it works on regions, rather than pipelines. It iteratively finds the bottleneck region and increases its replica count. The FINDBOTTLENECKREGION procedure is used to find the bottleneck region. This procedure simply computes the unbounded throughput of the program as new regions are successively added, using the formalization from Chapter 3, and selects the last region that resulted in a reduction in the unbounded throughput as the bottleneck one. It then reports this bottleneck region, together with the utilization of the current configuration. If the utilization is above the number of CPUs available, then the iteration is terminated and the replica configuration phase is over. Otherwise, i.e., if there is room available for an additional parallel channel, we increment the replica count of the bottleneck region. However, if the unbounded throughput of this parallel region with an incremented replica count has a lower unbounded throughput compared to the original parallel region (which might happen for low cost regions due to the impact of replication cost factor and thread switching overhead), we again terminate the region configuration phase. This is because if the bottleneck region cannot be improved, then no overall improvement is possible.

Chapter 5

Evaluation

In this section, we evaluate our heuristic solution and showcase its performance in terms of the achieved throughput, as well as the time it takes to locate a parallelization configuration. We perform two kinds of experiments. First, we evaluate our pipelined fission solution using model-based experiments under varying workload and system settings. Second, we evaluate our algorithm using stream programs written in IBM's SPL language [18] and executed using the IBM InfoSphere Streams [20] runtime.

In our experiments, we compare our solution against four different approaches, namely: optimal, sequential, fission-only, and pipelining-only.

- Sequential solution is the configuration with no parallelism.
- Optimal solution is the configuration that achieves the maximum throughput among all possible parallel configurations.
- Fission-only solution is the configuration that achieves the highest throughput among all possible parallel configurations that do not involve pipeline parallelism (that is, each parallel channel is executed by a single thread).
- Pipelining-only is the solution with the highest throughput among all possible parallel configurations that do not involve data parallelism.

Name	*Range*	*Default Value*
Operator cost mean	[50, 250]	200
Operator cost stddev	-	100
Number Of operators	[1,8]	8
Number Of cores	[1, 12]	4
Selectivity mean	[0.1,1]	0.8
Selectivity stddev	-	0.4
Stateless operator fraction	[0,0.8]	0.4
Stateful operator fraction	[0,0.8]	0.4
Partitioned stateful operator fraction	[0,0.8]	0.2
Thread switching overhead	[10, 210]	1
Replication cost factor	[10, 190]	50

Table 5.1: Experimental parameters: default values and ranges for model based experiments

5.1 Experimental Setup

For the model-based experiments, we used the analytical model presented in Chapter 3 to compare alternative solutions. The five alternative solutions we study were all implemented in Java. The SPL experiments rely on the parallelization configurations generated by these solutions to customize the runtime execution of the SPL programs. The SPL programs are compiled down to C++ and executed on the Streams runtime [20].

We describe the experimental setup for the model based experiments in Table 5.1. Each model based experiment was repeated 1000 times, whereas SPL based experiments were repeated 50 times. All experiments were executed on a Linux system with 2 Intel Xeon E5520 2.27GHz CPUs with a total of 12 cores and 48GB of RAM.

We cover the determination of the thread switching overhead and replication cost factor for the SPL experiments later in Section 5.3.

5.2 Model-based experiments

Streaming applications contain operators with diverse properties. Accordingly, the throughput of the topology is highly dependent on the properties of the operators involved. Hence, we evaluate our solution by varying operator selectivity, operator cost, and operator kind with respect to state. In addition, a variety of other factors impact the throughput of the topology, among which four most important ones are replication cost factor, thread switching overhead, number of cores, and the number of operators. Accordingly, we also perform experiments on these.

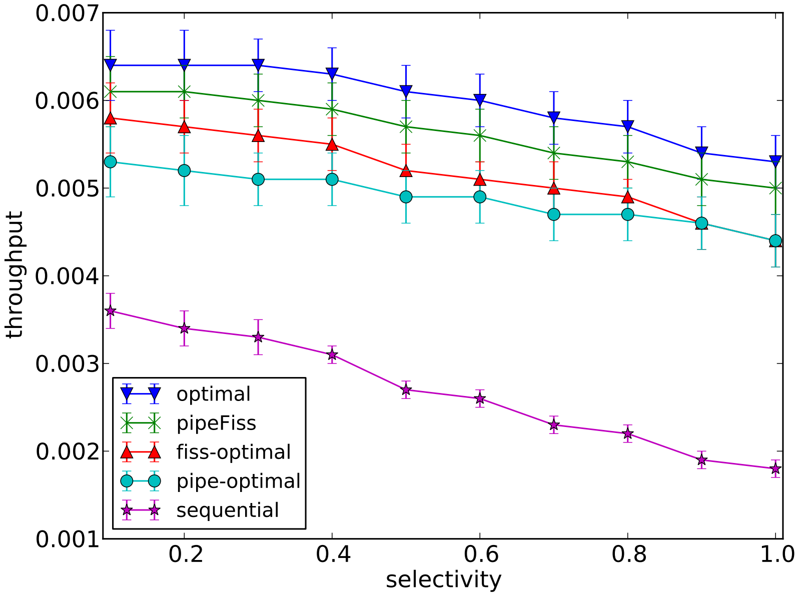


Figure 5.1: The impact of selectivity.

5.2.1 Operator Selectivity

The impact of operator selectivity on the performance of our solution is shown in Figure 5.1. The figure plots the throughput (y-axis) as a function of the mean operator selectivity (x-axis) for different approaches. We observe that for the entire range of selectivity values, our solution outperforms the fission-only and pipelining-only optimal approaches, and provides up to 2.7 times speedup in throughput compared to the sequential approach. The throughput provided by our approach is also consistently within 5% of the optimal solution, for the entire range of selectivity values. Interestingly, we observe that the pipelining-only approach provides reduced performance compared to fission-only approach, for low selectivity values. This is because with reducing selectivity, the performance impact of the operators that are deeper in the pipeline reduces, which takes away the ability of pipelining to increase the throughput (as speedup due to pipelining is limited by the pipeline depth).

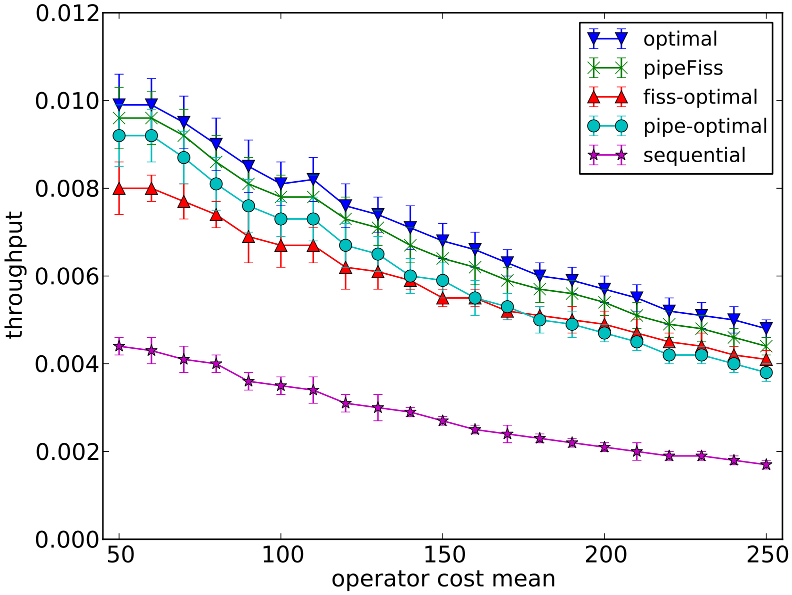


Figure 5.2: The impact of operator cost.

5.2.2 Operator Cost Mean

The impact of operator cost on the performance of our solution is shown in Figure 5.2. The figure plots the throughput (y-axis) as a function of the mean operator cost (x-axis) for different approaches. Again we observe that the pipelined fission solution is quite robust, consistently outperforming fission-only and pipelining-only optimal solutions, and staying within 5% of the optimal solution. It achieves up to 2.5 times speedup in throughout compared to the sequential approach. One interesting observation is that, for smaller mean operator cost values, the performance of the fission-only approach is below the pipelining-only approach, but gradually increases and passes it as the mean operator cost increases. The reason is that, the fission optimization has a higher overhead due to the replication cost factor, and thus, for small operator costs, it is not beneficial to apply fission. As the operator cost increases, fission becomes more effective.

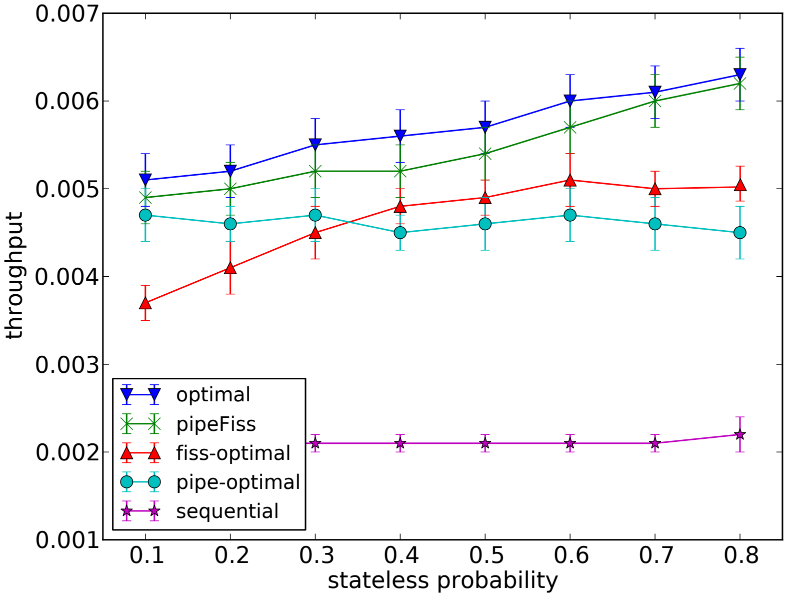


Figure 5.3: The impact of the operator kind.

5.2.3 Operator Kind

The impact of the operator kind on the performance of our solution is shown in Figure 5.3. Recall that operators can be stateful, stateless, or partitioned stateful. Figure 5.3 plots the throughput (y-axis) as a function of the fraction of stateless operators (x-axis) for different approaches. While doing this, we keep the fraction of partitioned stateful operators fixed at 0.2. We observe that the percentage of stateless operators do not impact the pipelining-only solution. The reason is that pipeline parallelism is applicable for both stateful and stateless operators. On the other hand, fission-only solution improves as the percent of the stateless operator increases. The reason is that data parallelism is not applicable for stateful operators. We also observe that our pipelined fission solution stays close to the optimal throughout the entire range of the stateless operator fraction. Again, pipelined fission clearly outperforms pipelining-only and fission-only approaches.

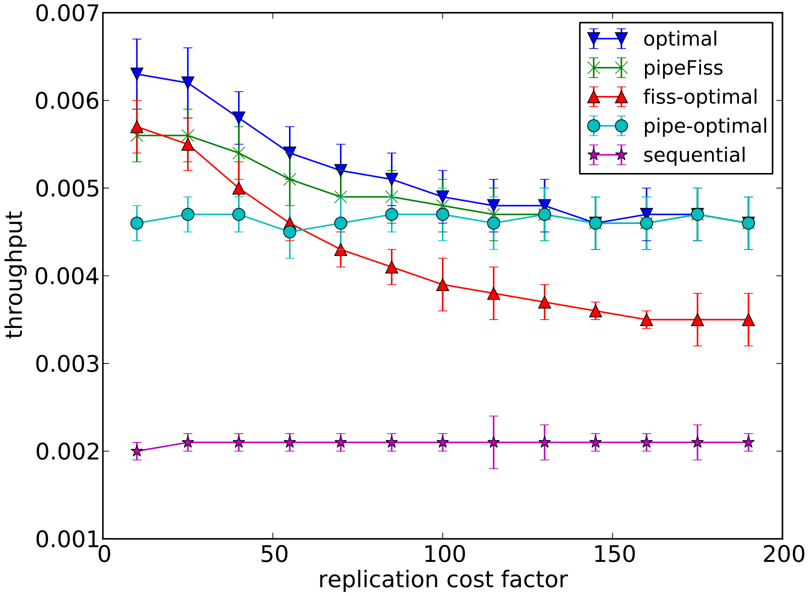


Figure 5.4: The impact of replication cost factor.

5.2.4 Replication Cost Factor

The impact of the replication cost factor on the performance of our solution is shown in Figure 5.4. The figure plots the throughput (y-axis) as a function of the replication cost factor (x-axis) for different approaches. The results indicate that our solution considerably outperforms pipelining-only and fission-only approaches, providing up to 25% higher throughput compared to pipelining-only optimal approach and up to 30% higher throughout compared to fission-only optimal approach. Note that our pipelined fission approach provides performance as good as fission-only optimal approach when the replication cost factor is close to 0 and as good as pipelining-only optimal approach when the replication cost factor is very high. In effect, our solution switches from using fission to using pipelining as the replication cost factor increases. We also observe that the optimal solution's throughput advantage is bigger for small replication cost factors, yet the gap with pipelined fission quickly closes as the replication cost factor increases. In the SPL based experiments presented later, we show that for realistic replication cost factors, our solution provides performance that is very close to

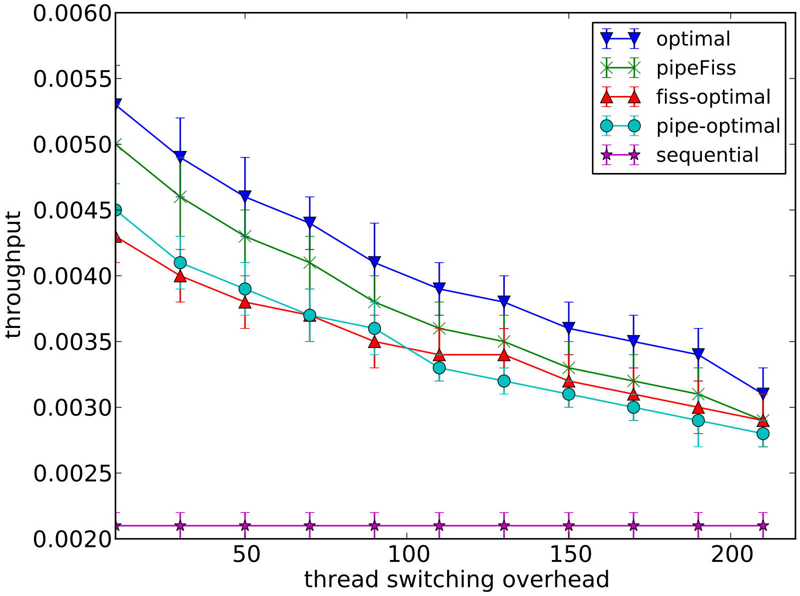


Figure 5.5: The impact of the thread switching overhead.

the optimal.

5.2.5 Thread Switching Overhead

The impact of the thread switching overhead on the performance of our solution is shown in Figure 5.5. The figure plots the throughput (y-axis) as a function of the thread switching overhead (x-axis) for different approaches. We observe that the throughput of all solutions, except the sequential one, decreases as the thread switching overhead increases. It is an expected result as all solutions benefit from parallelism via using threads, except the sequential solution. Again, our pipelined fission solution outperforms pipelining-only and fission-only optimal solutions, and is able to stay close to the optimal performance throughout the entire range of thread switching overhead values. We also observe that as the thread switching overhead increases, all approaches start to get closer in terms of the throughput. This is due to the reducing parallelization opportunities, as a direct consequence of the high thread switching overhead values.

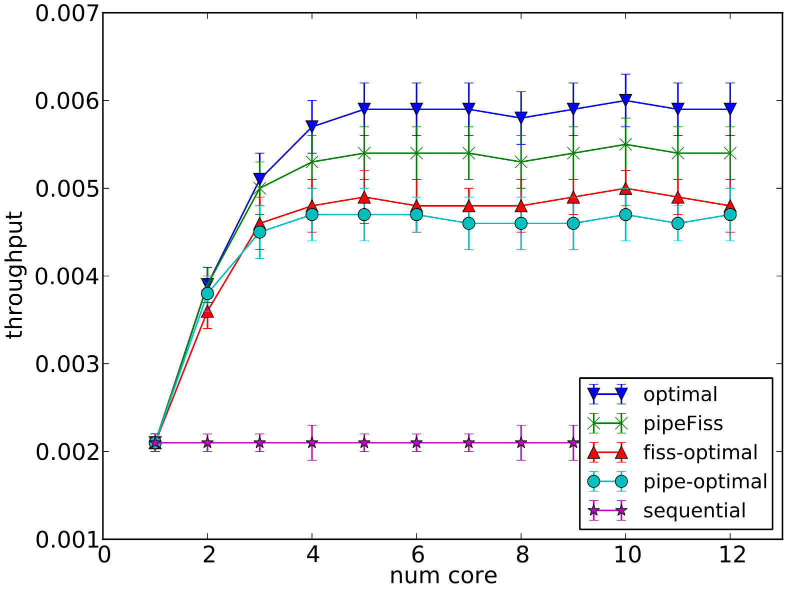


Figure 5.6: The impact of the number of cores.

5.2.6 Number of Cores

The impact of the number of cores on the performance of our solution is shown in Figure 5.6. The figure plots the throughput (y-axis) as a function of the number of cores (x-axis) for different approaches. We observe that for all approaches, the throughput only increases to a certain degree, after which it stays flat. There are two reasons for not being able to achieve linear speedup: i) not all operators are parallelizable, ii) the thread switching and replication cost factor introduce overheads in parallelization. We again observe that our pipelined fission approach outperforms the pipelining-only and fission-only optimal solutions. With increasing number of cores, the gap between the optimal approach and alternatives increases, as the search space gets bigger. However, since the throughput flattens quickly, the increase in the gap stops early. For instance, our approach stays within 8% of the optimal.

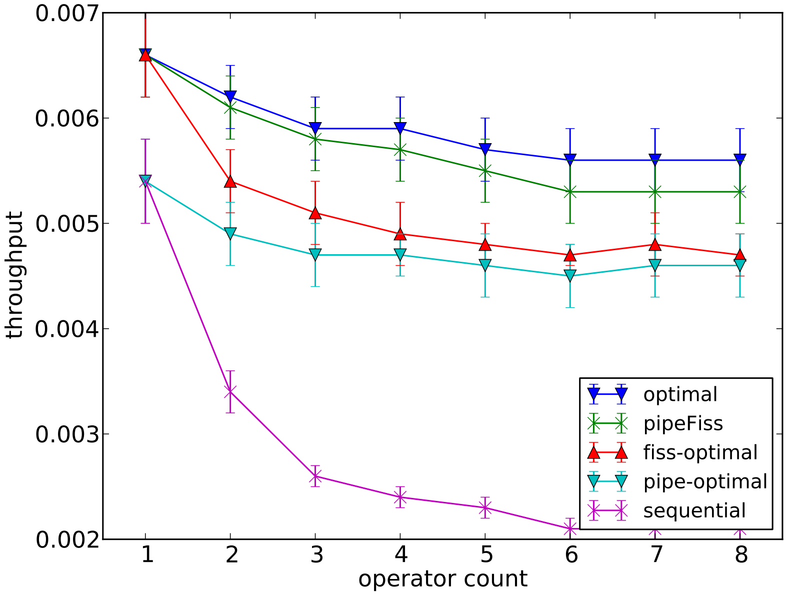


Figure 5.7: The impact of the number of operators.

5.2.7 Number of Operators

The impact of the number of operators on the performance of our solution is shown in Figure 5.7. The figure plots the throughput (y-axis) as a function of the number of operators (x-axis) for different approaches. We observe that as the number of operators increases, the performance of the pipelining-only solution relative to the fission-only solution increases. The reason is that the pipeline parallelism cannot help a single operator, so it is not as effective for small number of operators. Our pipelined fission solution provides up to 18% higher throughput compared to the closest alternative. While the gap between the optimal solution and ours increases with increasing number of operators, eventually throughput flattens due to the fixed number of cores available. Importantly, our approach stays within 5% of the optimal solution.

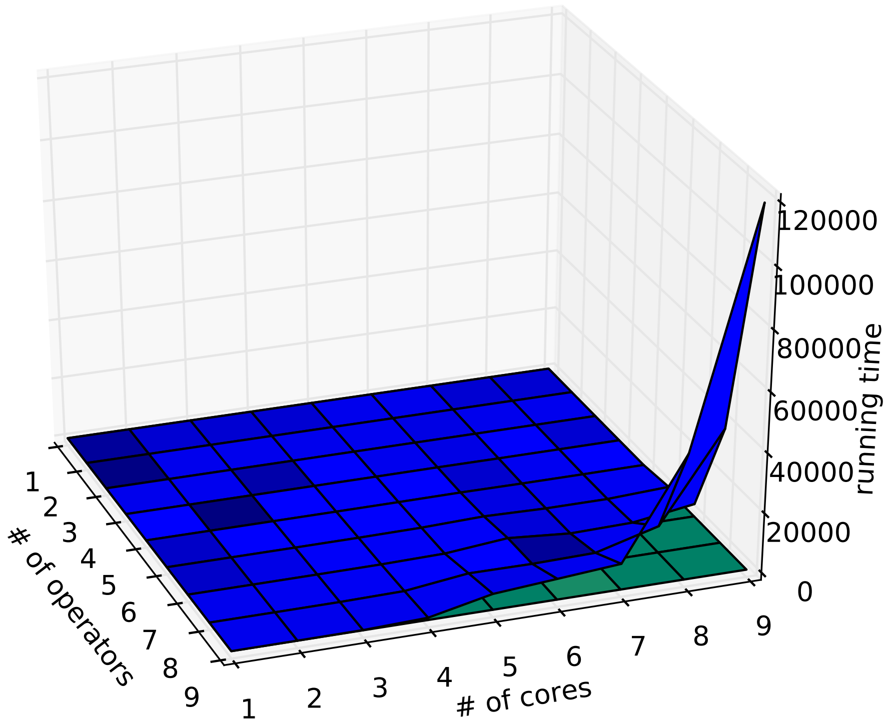


Figure 5.8: Running time.

5.2.8 Running Time

We also evaluate the running time of our pipelined fission algorithm. Figure 5.8 plots running time in terms of milliseconds, for our pipelined fission solution and the exhaustive optimal approach. Unfortunately, the running time of the optimal solution dramatically increases with increasing number of operators and threads. However, even if the number of operators and threads are high, our pipelined fission algorithm completes in a small amount of time.

5.3 SPL Experiments

In our second set of experiments, we use IBMs SPL language and its InfoSphere Streams runtime to evaluate the effectiveness of our solution. In order to perform this experiment, we need to determine the value of the replication cost factor and the thread switching overhead for the InfoSphere Streams runtime.

5.3.1 Thread Switching Overhead

For determining the thread switching overhead, we use a simple pipeline of two operators. We run this topology twice, once with a single thread and again with two threads. Let c be the cost of the operator. For the case of two threads, the throughput achieved, denoted as T_p, is given by:

$$T_p \;=\; 1/(c+\delta) \tag{5.1}$$

On the other hand, for the case of a single thread, the throughout achieved, denoted as T_s, is given by:

$$T_s \;=\; 1/(2\cdot c) \tag{5.2}$$

By using T_s and T_p, we can compute the thread switching overhead, δ, without needing to know the operator cost c. We have:

$$\delta \;=\; \frac{1}{T_p} - \frac{1}{2\cdot T_s} \tag{5.3}$$

In order to calculate the thread switching overhead for our SPL experiments, we measure the throughput of the topology with and without pipeline parallelism for varying tuple sizes, and use Equation 5.3 to compute the thread switching overhead. The use of different tuple sizes is due to the implementation of thread switching within the SPL runtime, which requires a tuple copy (the cost of which depends on the tuple size).

5.3.2 Replication Cost Factor

For determining the replication cost factor, we use a simple pipeline of three operators, where the first and the last operators are the source and the sink operators with no work performed and the middle operator has cost c. We then run this topology with different number of parallel channels used for the middle operator. Let n denote the number of channels used. We can formulate the throughput as:

$$T_p(n) \quad = \quad \left(\frac{2 \cdot \delta + c}{n} + \log_2 n \cdot c_p\right)^{-1} \tag{5.4}$$

If we know the throughput for two different number of channels, say $T(n_1)$ and $T(n_2)$, then we can compute the replication cost factor, c_p, independent of other factors, such as the cost c, as follows:

$$c_p \quad = \quad \frac{\frac{1}{n_2 \cdot T_p(n_1)} - \frac{1}{n_1 \cdot T_p(n_2)}}{\frac{\log_2 n_1}{n_2} - \frac{\log_2 n_2}{n_1}} \tag{5.5}$$

In order to calculate the replication cost factor for our SPL experiments, we measure the throughput of our sample topology with different number of replicas for varying tuple sizes, and use Equation 5.5 to compute the replication cost factor.

By using the calculated thread switching overhead and replication cost factor values, we perform SPL experiments to evaluate our solution for varying operator count, selectivity, cost, and kind. Throughput is again our main metric of evaluation.

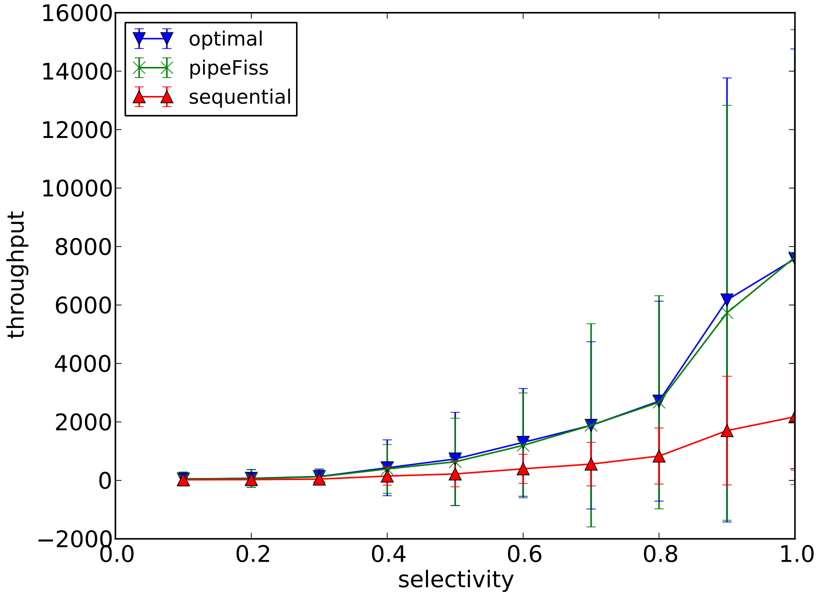


Figure 5.9: Impact of selectivity – SPL.

5.3.3 Operator Selectivity

Figure 5.9 plots throughput (y-axis) as a function of the mean operator selectivity (x-axis) for the optimal, pipelined fission, and sequential solutions using SPL. We see that all approaches achieve higher throughput as the operator selectivity increases. Pipelined fission solution provides practically the same performance as the optimal solution. This shows that our solution is even more effective in the context of a real-world stream processing system.

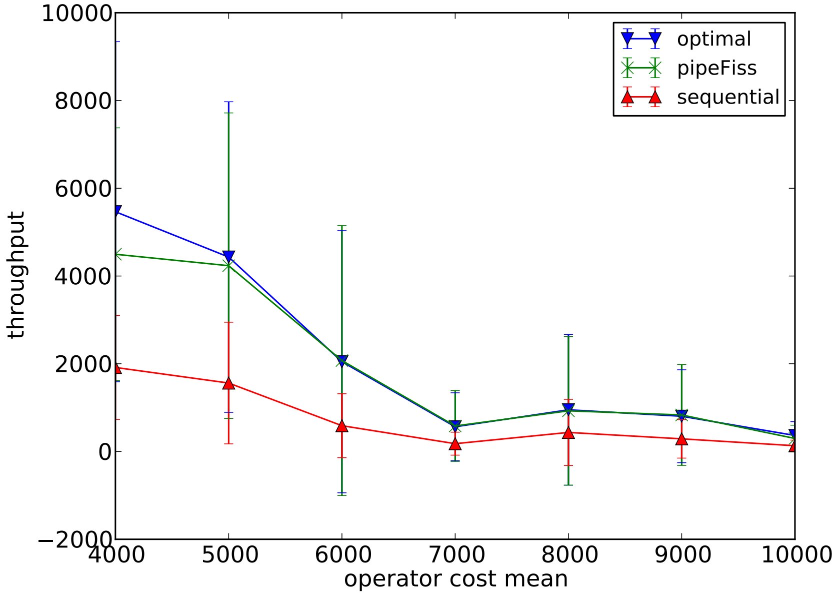


Figure 5.10: The impact of operator cost – SPL.

5.3.4 Operator Cost Mean

Figure 5.10 plots throughput (y-axis) as a function of the mean operator cost (x-axis) for the optimal, pipelined fission, and sequential solutions using SPL. It is not surprising that the throughput decreases as the mean operator cost increases, for all approaches. More interestingly, our approach again performs as good as the optimal approach throughout the entire cost range, except for the lowest cost point, for which we are still within 15% of the optimal.

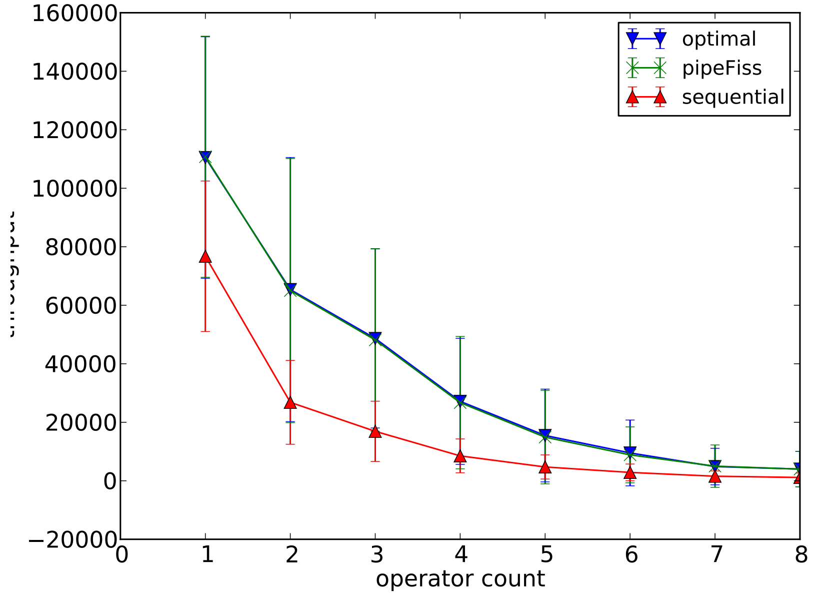


Figure 5.11: The impact of the number of operators – SPL.

5.3.5 Number Of Operators

Figure 5.11 plots throughput (y-axis) as a function of the number of operators (x-axis) for the optimal, pipelined fission, and sequential solutions using SPL. As expected, as the number of operators in a topology increases, throughput of a topology decreases for all approaches. Even for high number of operators, the throughput achieved by pipelined fission solution is as good as the optimal one.

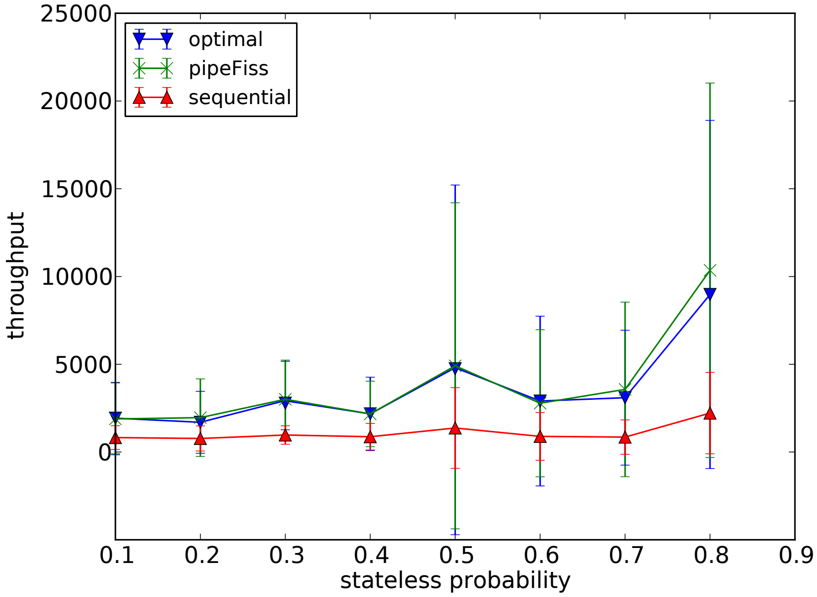


Figure 5.12: The impact of the operator kind – SPL.

5.3.6 Operator Kind

Figure 5.12 plots throughput (y-axis) as a function of the fraction of stateless operators. The change in operator kind does not affect the sequential solution as in Figure 5.3 from earlier. On the other hand, as the percentage of stateless operator increases, the throughput achieved increases. The reason is that stateless operators can benefit from both data and pipeline parallelism. As it can be see from the figure, our pipelined fission solution again performs practically same as the optimal solution, except for the highest stateless fraction (0.8, as 0.2 of the operators are fixed as partitioned stateful), at which point we are still within 5% of the optimal.

Chapter 6

Related Work

Our work belongs to the general area of auto-parallelization. We first overview prior work in this area, and then focus on work related to the core subject of this thesis: auto-parallelization in streaming systems.

6.1 Multi-threaded concurrency platforms

Determining parallelizable code regions and appropriately assigning those regions to computing units for execution are the two major issues that must be addressed by any automatic parallelization systems.

Multi-threaded concurrency platforms, such as Cilk++ [21], OpenMP [22], and x10 [23], decouple expressing a program's innate parallelism from its execution configuration. OpenMP and Cilk++ are widely used language extensions for shared memory programs, which help express parallel execution in a program at development-time and take advantage of it at run-time.

Various platforms are proposed in the literature for automatically finding parallelizable program regions. One example is Kremlin [24], which is an auto-parallelization framework that complements OpenMP [22]. Kremlin recommends to programmers a list of regions for parallelization, which is ordered by achievable program speedup. The speedup is calculated based on an improved critical path analysis.

Cilkview [25] is a Cilk++ analyzer of program scalability in terms of number of cores. Cilkview performs system-level modeling of scheduling overheads (e.g., the bookkeeping costs to set up context and the overhead of cache misses), and predicts program speedup. Bounds on the speedup are presented to programmers for further analysis.

Autopin [26] is an auto-configuration framework for finding the best mapping between system cores and threads. Using profile runs, Autopin exhaustively probes all possible mappings and finds the best pinning configuration in terms of performance. Then, threads are re-pinned using the best mapping found.

Alchemist [27] is a dependence profiling technique based on post-dominance analysis and is used to detect candidate regions for parallel execution. It is based on the observation that a procedure with few dependencies with its continuation benefits more from parallelization.

There has been extensive research in the literature on compiler support for instruction-level or fine-grained pipelined parallelism [28]. In our work, we look at coarse-grained pipelining techniques that address the problem of decomposing an application into higher-level pieces that can execute in pipeline as well as data parallel. Relevant to our study is the work in [29], which provides compiler support for coarse-grained pipelined parallelism. To automate pipelining, it selects a set of candidate filter boundaries (a middleware interface exposed by DataCutter [30]), determines the communication volume for these boundaries, and performs decomposition and code generation in order to minimize the execution time. To select the best filters, communication costs across each filter boundary are estimated by static program analysis and a dynamic programming algorithm is used to find the optimal decomposition. In comparison, our work performs combined pipelining and fission and has support for partitioned stateful operators.

6.2 Pipelining/Fusion in Streaming Systems

In most streaming systems, operators that are fused together use the same thread, whereas nonfused operators can be run in parallel. The key problem is to divide a program into fused pieces that can be run in parallel, typically in a pipelined configuration. For instance, StreamIt [31], which is a language for creating streaming applications, uses fusion to coarsen the granularity of the graph to the target number of cores, based on cost estimates [32]. This is somewhat similar to our region configuration step, but is limited to stateless operators or operators that only have read-only sliding window state. Aurora data stream management system uses fusion to minimize scheduling overhead [33]. SPADE [34] uses the COLA [16] fusion optimizer to combine operators as much as possible, until a single processing element fills the entire capacity of a core. A different approach is taken by Tang and Gedik [10], where the stream program initially runs as completely fused, and an auto-pipeliner is used to detect bottlenecks and inject new threads into the runtime system to improve throughput. With the exception of StreamIt, which we further cover shortly, these systems are limited to pipelined parallelism, and do not perform combined pipelining and fission.

6.3 Fission in Streaming Systems

StreamIt [35] performs both pipelining and fission. It addresses the safety question of fission by only replicating operators that are either stateless or whose operator state is a read-only sliding window. As opposed to StreamIt, which targets synchronous dataflow systems, our work targets data stream management systems that typically contain operators that are partitioned stateful and exhibit dynamic selectivity. Thus, rather than having a static schedule based execution model, we adopt a backpressure based runtime system. We model its throughput in order to formulate a pipelined fission configuration that can provide optimal throughput. Work on elastic operators [19] also generalizes fission beyond the StreamIt setting to work on stateful operators with dynamic data rates. However, the work is limited to fission only and does not support pipelining.

A related problem is to perform fission dynamically, that is to adjust the width of the parallel region based on the changing runtime and workload conditions. SEDA achieves this via a thread-pool controller, which can adjust the number of threads to increase parallelism, while preserving locality [36]. MapReduce systems dynamically adjust the number of workers assigned to the map tasks [37]. Elastic operators [19] adjust the number of threads assigned to an operator by using a control loop. An extension of it [12] applies similar kind of control in a distributed setup, where replication is not limited to a single operator and replicas can run across different hosts. While our thesis does not particularly deal with the adaptation aspect, its model based approach and efficient heuristic solver makes it perfectly suitable for runtime optimization based on feedback from a performance profiler.

Overall, our work is distinguished by earlier work on streaming systems, as it is the only work that combines pipelining and fission in the context of partitioned stateful operators with dynamic selectivity.

Chapter 7

Conclusion

We proposed a pipelined fission solution that can quickly locate a parallelization configuration for accelerating data stream processing applications, and can provide throughput close to the optimal that can be achieved. In order to achieve this aim, our algorithm incorporates stages that greedily perform data and pipeline parallelism with a varying fraction of resources dedicated to the two different parallelization approaches. Our model based experimental evaluation shows that our proposed algorithm is effective both in terms of running time and throughput under varying operator, system, and workload properties. Our evaluation using an industrial-strength stream processing engine showcases even stronger results, where our pipelined fission solution provides throughput that is almost indistinguishable from the optimal.

Bibliography

[1] X. J. Zhang, H. Andrade, B. Gedik, R. King, J. F. Morar, S. Nathan, Y. Park, R. Pavuluri, E. Pring, R. Schnier, P. Selo, M. Spicer, V. Uhlig, and C. Venkatramani, "Implementing a high-volume, low-latency market data processing system on commodity hardware using IBM middleware," in *Workshop on High Performance Computational Finance (SC-WHPCF)*, 2009.

[2] E. Bouillet, R. Kothari, V. Kumar, L. Mignet, S. Nathan, A. Ranganathan, D. S. Turaga, O. Udrea, and O. Verscheure, "Processing 6 billion CDRs/day: from research to production (experience report)," in *International Conference on Distributed Event Based Systems (DEBS)*, 2012.

[3] H. Andrade, B. Gedik, and D. Turaga, *Fundamentals of Stream Processing: Application Design, Systems, Analytics*, 1st ed. Cambridge, UK: Cambridge Press, 2014, ch. 12.4 – The Semiconductor Process Control application.

[4] D. M. Sow, A. Biem, M. Blount, M. Ebling, and O. Verscheure, "Body sensor data processing using stream computing," 2010.

[5] "StreamBase Systems," http://www.streambase.com, retrieved Nov, 2013.

[6] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soul, and K.-L. Wu, "Streams processing language: Analyzing big data in motion," *IBM Journal of Research and Development*, vol. 57, pp. 7:1–7:11, 2013.

[7] "Storm project," http://storm-project.net/, retrieved Nov, 2013.

[8] D. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik, "The design of the Borealis stream processing engine," in *Innovative Data Systems Research Conference (CIDR)*, 2005.

[9] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom, "STREAM: The Stanford stream data manager," *IEEE Data Engineering Bulletin*, vol. 26, no. 1, 2003.

[10] Y. Tang and B. Gedik, "Auto-pipelining for data stream processing," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 24, no. 12, pp. 2344–2354, 2013.

[11] S. Schneider, M. Hirzel, B. Gedik, and K.-L. Wu, "Safe data parallelism for general streaming," 2013.

[12] B. Gedik, S. Schneider, and K.-L. W. M. Hirzel, "Elastic scaling for data stream processing," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2013.

[13] M. Hirzel and B. Gedik, "Streams that compose using macros that oblige," in *ACM Workshop on Partial Evaluation and Program Manipulation (PEPM)*, 2012.

[14] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin, "Flux: An adaptive partitioning operator for continuous query systems," in *IEEE International Conference on Data Engineering (ICDE)*, 2003.

[15] H. Andrade, B. Gedik, K.-L. Wu, and P. S. Yu, "Processing high data rate streams in system s," in *Journal of Parallel and Distributed Computing (JPDC)*, vol. 71, no. 2, 2011, pp. 145–156, special Issue on Data Intensive Computing.

[16] R. Khandekar, K. Hildrum, S. Parekh, D. Rajan, J. Wolf, K.-L. Wu, H. Andrade, and B. Gedik, "COLA: Optimizing stream processing applications via graph partitioning," in *ACM/IFIP/USENIX Middleware Conference (Middleware)*, 2009.

[17] B. Gedik, H. Andrade, and K.-L. Wu, "A code generation approach to optimizing high-performance distributed data stream processing," in *ACM International Conference on Information and Knowledge Management (CIKM)*, 2009.

[18] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soulé, and K.-L. Wu, "Streams processing language: Analyzing big data in motion," *IBM Journal of Research and Development*, vol. 57, no. 3/4, pp. 7:1–7:11, 2013.

[19] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu, "Elastic scaling of data parallel operators in stream processing," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2009.

[20] B. Gedik and H. Andrade, "A model-based framework for building extensible, high performance stream processing middleware and programming language for IBM InfoSphere Streams," *Software: Practice and Experience*, 2012.

[21] "Intel cilk++," http://software.intel.com/en-us/articles/intel-cilk-plus/, retrieved October, 2014.

[22] "Openmp," http://www.openmp.org, retrieved October, 2014.

[23] P. Charles, C. Grothoff, V. A. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," in *International Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 2005.

[24] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor, "Kremlin: Rethinking and rebooting gprof for the multicore age," in *International Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[25] Y. He, C. E. Leiserson, and W. M. Leiserson, "The cilkview scalability analyzer," in *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2010.

[26] T. Klug, M. Ott, J. Weidendorfer, and C. Trinitis, "Autopin: Automated optimization of thread-to-core pinning on multicore systems," *Transactions on High-Performance Embedded Architectures and Compilers (HiPEAC)*, vol. 3, pp. 219–235, 2011.

[27] X. Zhang, A. Navabi, and S. Jagannathan, "Alchemist: A transparent dependence distance profiling infrastructure," in *International Symposium on Code Generation and Optmizaiton (CGO)*, 2009, pp. 47–58.

[28] S. M. Krishnamurthy, "A brief survey of papers on scheduling for pipelined processors," *ACM SIGPLAN Notices*, vol. 25, no. 7, pp. 97–106, 1990.

[29] W. Du, R. Ferreira, and G. Agrawal, "Compiler support for exploiting coarse-grained pipelined parallelism," in *Supercomputing Conference (SC)*, 2003, p. 8.

[30] M. D. Beynon, T. M. Kurç, Ü. V. Çatalyürek, C. Chang, A. Sussman, and J. H. Saltz, "Distributed processing of very large datasets with DataCutter," *Parallel Computing Journal*, vol. 27, no. 11, pp. 1457–1478, 2001.

[31] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[32] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, , and S. Amarasinghe, "A stream compiler for communication-exposed architectures," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002, pp. 291–303.

[33] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: A new model and architecture for data stream management," *VLDB Journal*, vol. 12, no. 2, pp. 120–139, 2003.

[34] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo, "SPADE: The system s declarative stream processing engine," in *ACM International Conference on Management of Data (SIGMOD)*, 2008, pp. 1123–1134.

[35] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006, pp. 151–162.

[36] M. Welsh, D. Culler, and E. Brewer, "SEDA: An architecture for well-conditioned, scalable internet services," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2001, pp. 230–243.

[37] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," 2004, pp. 137–150.