Auto-pipelining for Data Stream Processing

Yuzhe Tang, *Student Member, IEEE,* Buğra Gedik, *Member, IEEE*

Abstract—Stream processing applications use online analytics to ingest high-rate data sources, process them on-the-fly, and generate live results in a timely manner. The *data flow graph* representation of these applications facilitates the specification of stream computing tasks with ease, and also lends itself to possible run-time exploitation of parallelization on multi-core processors. While the data flow graphs naturally contain a rich set of parallelization opportunities, exploiting them is challenging due to the combinatorial number of possible configurations. Furthermore, the best configuration is dynamic in nature; it can differ across multiple runs of the application, and even during different phases of the same run. In this paper, we propose an *auto-pipelining* solution that can take advantage of multi-core processors to improve throughput of streaming applications, in an *effective* and *transparent* way. The solution is effective in the sense that it provides good utilization of resources by dynamically finding and exploiting sources of pipeline parallelism in streaming applications. It is transparent in the sense that it does not require any hints from the application developers. As part of our solution, we describe a light-weight runtime profiling scheme to learn resource usage of operators comprising the application, an optimization algorithm to locate best places in the data flow graph to explore additional parallelism, and an adaptive control scheme to find the right level of parallelism. We have implemented our solution in an industrial-strength stream processing system. Our experimental evaluation based on micro-benchmarks, synthetic workloads, as well as real-world applications confirms that our design is effective in optimizing the throughput of stream processing applications without requiring any changes to the application code.

Index Terms—stream processing; parallelization; auto-pipelining

1 Introduction

With the recent explosion in the amount of data available as live feeds, *stream computing* has found wide application in areas ranging from telecommunications to healthcare to cyber-security. Stream processing applications implement data-in-motion analytics to ingest high-rate data sources, process them on-the-fly, and generate live results in a timely manner. Stream computing middleware provides an execution substrate and runtime system for stream processing applications. In recent years, many such systems have been developed in academia [1], [2], [3], as well as in industry [4], [5], [6].

For the last decade, we have witnessed the proliferation of multi-core processors, fueled by diminishing gains in processor performance from increasing operating frequencies. Multi-core processors pose a major challenge to software development, as taking advantage of them often requires fundamental changes to how application code is structured. Examples include employing thread-level primitives or relying on higher-level abstractions that have been the focus of much research and development [7], [8], [9], [10], [11], [12]. The high-throughput processing requirement of stream processing applications makes them ideal for taking advantage of multi-core processors. However, it is a challenge to keep the simple and elegant data flow programming model of stream computing, while best utilizing the multiple cores available in today's processors.

Stream processing applications are represented as data flow graphs, consisting of reusable operators connected to each other via stream connections attached to operator *ports*. This is a programming model that is declarative at the flow manipulation level and imperative at the flow composition level [13]. The data flow graph representation of stream processing applications contains a rich set of parallelization opportunities. For instance, *pipeline parallelism* is abundant in stream processing applications. While one operator is processing a tuple, an upstream operator can process the next tuple concurrently. Many data flow graphs contain bushy segments that process the same set of tuples, and which can be executed in parallel. This is an example of *task parallelism*. It is noteworthy that both forms of parallelism have advantages in terms of preserving the semantics of a parallel program. On the other hand, exploiting *data parallelism* has additional complexity due to the need for morphing the graph to create multiple copies of an operator and to re-establish the order between tuples. Pipeline and task parallelism do not require morphing the graph and preserve the order without additional effort. These two forms of parallelism can be exploited by inserting the right number of threads into the data flow graph at the right locations. It is desirable to perform this kind of parallelization in a transparent manner, such that the applications are developed without explicit knowledge of the amount of parallelism available on the platform. We call this process *auto-pipelining*.

There are several challenges to performing effective and transparent auto-pipelining in the context of stream processing applications.

• Y. Tang is a Ph.D. student at the College of Computing, Georgia Institute of Technology, 801 Atlantic Drive, Atlanta, GA 30332. E-mail: yztang@gatech.edu. *The work was done while the author was at the IBM T.J. Watson Research Center.*

• B. Gedik is an Asst. Professor at the Computer Engineering Deparment, Bilkent University, 06800, Ankara, Turkey. E-mail: bgedik@cs.bilkent.edu. *Part of the work was done while the author was at the IBM T.J. Watson Research Center.*

First, optimizing the parallelization of stream processing applications requires determining the relative costs of operators. The prevalence of user-defined operators in real-world streaming applications [5] means that cost modeling, commonly applied in database systems [14], is not applicable in this setting. On the other hand, profile-driven optimization that requires one or more profile runs based on compiler-generated instrumentation [15], [16], while effective, suffers from usability problems and lack of runtime adaptation. On the usability side, requiring profile runs and specification of additional compilation options has proven to be unpopular among users in our own experience (see Appendix J). In terms of runtime adaptation, the profile run may not be representative of the final execution. In summary, a light-weight dynamic profiling of operators is needed in order to provide effective and transparent auto-pipelining.

Second, and more fundamentally, it is a challenge to efficiently (time-wise) find an effective (throughput-wise) configuration that best utilizes available resources and harnesses the inherent parallelism present in the streaming application. Given N operator ports and up to T threads, there are combinatorial possibilities, $\sum_{k=0}^{T} \binom{N}{k}$ to be precise. In the absence of auto-pipelining, we have observed application developers struggling to insert threads manually[1] to improve throughput. This is no surprise, as for a medium size application with 50 operators on an 8-core system, the number of possibilities reach multiple billions. Thus, a practical optimization solution needs to quickly and automatically locate an effective configuration at runtime.

Finally, deciding the right level of parallelism is a challenge. The behavior of the system is difficult to predict for various reasons. User-defined operators can contain locks that inhibit effective parallelization. The overhead imposed by adding an additional thread in the execution path is a function of the size of the tuples flowing through the port. The behavior of the operating system scheduler can not be easily modeled and predicted. The impact of these and other system artifacts are observable only at runtime and treated as a blackbox. While the optimization step can come up with threading configuration changes that are expected to improve performance, such decisions need to be tried out and dynamically evaluated to verify their effectiveness. As such, we need a control algorithm that can backtrack from bad decisions.

In this paper we describe an auto-pipelining solution that addresses all of these challenges. It consists of:

- A light-weight run-time profiling scheme that uses a novel metric called *per-port utilization* to determine the amount of time each thread spends downstream of a given operator input port.
- A greedy optimization algorithm that finds locations in the data flow graph where inserting additional threads helps eliminate bottlenecks and improve throughput.
- A control algorithm that decides when to stop inserting additional threads and also backtracks from decisions that turn out to be ineffective.
- Runtime mechanics to insert/remove threads while maintaining lock correctness and continuous operation.

We implemented our auto-pipelining solution on IBM's System S [3] — an industrial strength stream processing middleware. We evaluate its effectiveness using micro-benchmarks, synthetic workloads, and real-world applications. Our results show that auto-pipelining provides better throughput compared to hand-optimized applications at no cost to application developers.

1. SPL language [5] provides a configuration called 'threaded port' that can be used to manually insert threads into a data flow graph.

2 Background

We provide a brief overview of the basic concepts associated with stream processing applications, using SPL [5] as the language of illustration. We also describe the fundamentals of runtime execution in System S.

2.1 Basic concepts

Listing 1 in Appendix A gives the source code for a very simple stream processing application in SPL, with its visual representation depicted in Figure 1 below.

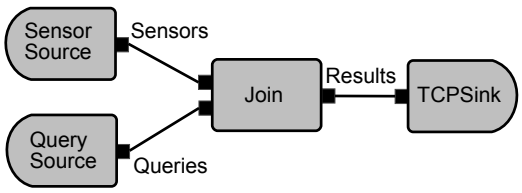


Fig. 1: Data flow graph for the `SensorQuery` app.

The application is composed of *operator instances* connected to each other via *stream connections*. An operator instance is a vertex in the application graph. An operator instance is always associated with an *operator*. For instance, the operator instance shown in the middle of the graph in Figure 1 is an instance of a `Join` operator. In general, operators can have many different instantiations, each using different stream types, parameters, or other configurations such as windows. Operator instances can have zero or more input and output *ports*. Each output port generates a uniquely named *stream*, which is a sequence of tuples. Connecting an output port to the input of an operator establishes a *stream connection*. Operators are often implemented in general purpose languages, using an event driven interface, by reacting to tuples arriving on operator input ports. Tuple processing generally involves updating some operator-local state and producing result tuples that are sent out on the output ports.

There are two important aspects of real-world applications that are highly relevant for our work:

- Real-world applications are usually much larger in terms of the number of operators they contain, reaching hundreds or even thousands.

• Real-world applications contain many user-defined reusable operators to implement cross-domain or domain-specific manipulations.

The former point motivates the need for automatic parallelization, whereas the latter motivates the need for dynamic profiling.

2.2 Execution model

A distributed stream processing middleware, such as System S, 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 paper we focus on a single multi-core host executing the entire graph. Our auto-pipelining technique can be applied independently on each host when the whole application consists of multiple, distributed processing elements.

There are two main sources of threading in our streaming runtime system, which contribute to the execution of the data flow graphs. The first one is *operator threads*. Source operators, which do not have any input ports, are driven by a separate thread. 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 data flow 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. We refer to this optimization as *operator fusion* [16], [15]. Non-source operators can also create operator threads, but this is rare. In general, the number and location of operator threads are not flexible because they are dictated by the application and the operator implementations.

The second source of threading is *threaded ports*. Threaded ports can be inserted at any operator input port. When a tuple reaches a threaded port, the currently executing thread will insert the tuple into the threaded port buffer, and go back to executing upstream logic. A separate thread, dedicated to the threaded port, will pick up the queued tuples and execute the downstream operators. Threaded port buffers are implemented as cache-optimized concurrent lock-free queues [17].

The goal of our auto-pipelining solution is to automatically place threaded ports at operator input ports during run-time, so as to maximize throughput.

3 System Overview

In this section we give an overview of our auto-pipelining solution. Figure 2 depicts the functional components and the overall control flow of the solution. It consists of five main stages that run in a continuous loop until a termination condition is reached.

The first stage is the *profiling stage*. In this stage a light-weight profiler determines how much time each of the existing threads spend on executing the operators in the graph. This profiling information, termed *per-port utilization*, is used as input to the *optimization* stage. An optimization algorithm that uses a greedy heuristic determines what the next action should be. The next action could either be to halt, as it could find nothing but an empty set of threaded ports at this time, or it could be to add additional threads at specific input ports. If the optimizer decides to add new threads, then the *thread insertion* component applies this decision. This is followed by the *evaluation* component, which evaluates the performance of the system after the thread insertions. The performance results from the evaluation are put into the *controller* component as a feedback, which takes one of two possible actions. It could vet all the thread insertions and go to the next iteration of the process. Alternatively, it could remove some or all of the inserted threads, reverting back the decisions taken by the optimizer. This could be followed by moving to the next iteration of the process or halting the process. In the former case, it applies a *blacklisting* algorithm to avoid coming up with the same ineffective configuration in the next iteration.

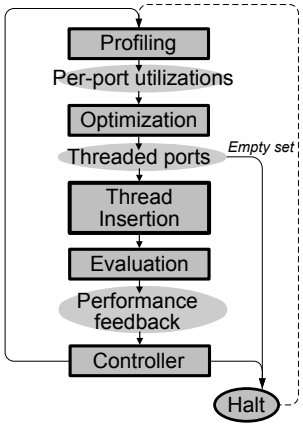


Fig. 2: Overview of the auto-pipelining system.

The system can be taken out of the halting state in case a shift in the workload conditions is detected. However, the focus of this work is on finding an effective operating point right after the application launch.

3.1 An Example Scenario

Throughout the paper we use an example application to illustrate various components of our solution. The compile-time and run-time data flow graphs for this application are given in Figures 3 and 4, respectively. For simplicity of exposition, we assume that all operators have a single input port and a single output port. However, our solution trivially extends to the general case and has been implemented and evaluated for the multi-port scenario (see Section 8).

The sample application consists of an 11-operator graph as shown in Figure 3. There are four source operators (namely, o_0, o_2, o_5 and o_7) which generate tuples. At runtime, there are four threads initially, $t_0, \ldots, t_3$, that execute the program, assuming no threaded ports have been inserted. Figure 4 shows the execution path of different threads in different colors and shapes. Note that some operators are present in the execution path of multiple threads. For instance, threads t_0 and t_1 share operator o_3 in their execution paths.

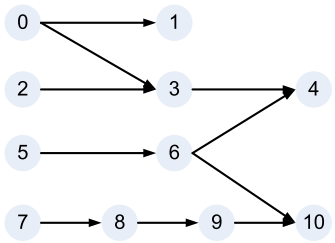


Fig. 3: Operator graph

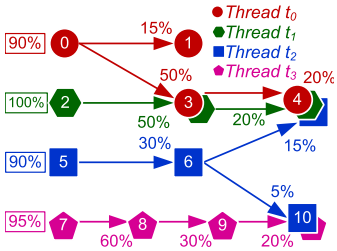


Fig. 4: Runtime op. graph

The runtime graph cannot be derived solely from the compile-time graph. The paths threads take can depend on tuple runtime values, as well as operator runtime behavior, such as selectivity or tuple submission decisions. Hence, the compile-time graph restricts each thread in terms of what operators it can traverse, but does not exactly define its path. We derive the runtime graph based on runtime profiling (see Section 6).

We now look at the metrics that will help us formulate the auto-pipelining problem.

3.1.1 Profiling Metrics

The main profiling metric collected by our auto-pipelining solution is called the *per-port utilization*, which we denote with $\mu(o,t)$. The variable o represents any arbitrary operator, and t represents any thread that can execute that operator. We define the utilization, $\mu(o,t)$, to be the amount of CPU utilized by thread t when executing *all* downstream operators starting from the input port of operator o. During program execution, the profiler maintains $\mu(o,t)$ for every operator/thread pair for which the thread t executes the operator o. In Figure 4, for example, the input port of operator o_6 is associated with utilization 30%, meaning that thread t_2 spends 30% of the CPU time on executing o_6 and its downstream operators, which are operators o_4 and o_{10}. Thus $\mu(o_6,t_2)=0.3$.

For each thread, we also define *per-thread utilization*, denoted as $\mu(t)$, which is the overall CPU utilization of thread t. For example, in Figure 4, thread t_2 has a utilization of 90%, thus $\mu(t_2)=0.9$.

The reason we pick per-port CPU utilization, $\mu(o,t)$, as our main profiling metric is that it simplifies predicting the relative work distribution between threads after inserting a new thread on an input port. For instance, if a threaded port is being added in front of operator o_6 in Figure 4, the newly created thread will take 30% CPU utilization from the existing thread t_2.

Predicting the relative work distribution for a potential thread insertion is performed in the following way. Assume that $T(o)=\{t\,|\,\mu(o,t)>0\}$ denotes the list of threads that contain a given operator o in their execution path. Adding a threaded port at operator o will have two consequences. First, all of the threads in $T(o)$ will execute only up to the input port of operator o. Second, a new thread, t', will execute the rest of the executions paths for all threads in $T(o)$. The prediction of the work distribution for the newly created thread t' is $\mu'(t')=\sum_{t\in T(o)}\mu(o,t)$. For an existing thread $t\in T(o)$, the prediction is $\mu'(t)=\mu(t)-\mu(o,t)$. For instance, in Figure 4, when a threaded port is added to operator o_3, we predict $\mu'(t_0)=0.4$, $\mu'(t_1)=0.5$, and $\mu'(t')=1$.

It is important to note that μ' is a relative metric of how the work is partitioned between the existing threads and the newly created thread. It is *not* an accurate prediction of what the CPU utilizations will be after the thread insertion. The expectation is that, given enough processing resources and enough work present in the application, the actual utilizations (μ) will be higher than the relative predictions (μ'). For instance, consider a simple chain of operators executed by a single thread that has $\mu(t_0)=1$. Adding a threaded port in the middle of this chain will result in $\mu'(t_0)=0.5$ and $\mu'(t_1)=0.5$. We use these relative utilization values to assess whether or not inserting a new thread in this location will improve performance. After the insertion, the optimistic expectation is that $\mu(t_0)=\mu(t_1)>0.5$, because $u'(t_0)<1$ and $u'(t_1)<1$, which leaves room for improvement in throughput. The evaluation and control stages of our solution deal with cases where this expectation does not hold.

3.1.2 Utility Function

The predicted relative utilizations are used to define a utility function that measures a threaded port insertion's goodness. Given an insertion at operator o, causing the creation of thread t', we define its *utility* as

$$U(o,t')=max(\mu'(t)\,|\,t\in T(o)\cup\{t'\}).$$

The utility function for a given operator and its new thread is the *largest* predicted relative work distribution across all of the threads with that operator in its path. Our goal is to *minimize* this utility function. The intuition behind the utility function is simple: the thread that has the highest predicted work (μ') will become the bottleneck of the system.

Suppose $T(o)=\{t_0\}$ and our predictions after insertion of a new thread t' at operator o are $\mu'(t_0)=0.3$ and $\mu'(t')=0.6$. The utility of this insertion is $U(o,t')=max(0.6,0.3)=0.6$. A better insertion at a different operator o', where $T(o')=\{t_0\}$, that would give a lower utility value is: $\mu'(t_0)=0.5$ and $\mu'(t')=0.5$, leading to $U(o',t')=0.5$. However, it may not always be possible to find such an insertion based on the per-port utilizations of the operators reported by profiling.

For a set of thread insertions, say $C=\{\langle o,t'\rangle\}$, we define an *aggregate utility function* $U(C)$ as:

$$U(C)=max(U(o,t')\,|\,\langle o,t'\rangle\in C).$$

Here, we pick the maximum of the individual utilities. We will further discuss and illustrate the aggregate utility function shortly.

3.2 The Optimization Problem

Recall that the goal of the optimization stage is to find one or more threaded ports that will improve the

throughput of the system. We propose the following heuristic for the optimization stage:

> *Minimize the aggregate utility function while making sure that one and only one threaded port is inserted in the execution path of each heavily utilized thread.*

This formulation is based on three core principles:

1) Help the needy: At each step, we only insert threads in the execution path of heavily utilized threads. A heavily utilized thread is a bottleneck, which implies that if it has more resources, overall throughput will improve.

2) Be greedy but generous: By definition our solution is greedy, as at each step it comes up with incremental insertions that will improve performance. However, inserting a single thread at a time does not work, which is why we make sure that a thread is inserted in the execution path of each heavily utilized thread. To see this point, consider the scenario where two threads execute a simple chain of four equal sized operators. The first thread executes the first two operators, and the second thread executes the remaining two. As an incremental step, if we only help the first thread, we will end up having three threads, where the last thread still executes two operators. This imbalance will become the bottleneck and thus the throughput will not increase. But, if we help both of the two original threads, we expect the throughput to increase.

A more subtle, but critical, point is the requirement that one and only one thread is added to the execution path of each heavily utilized thread. This is strongly related to the greedy nature of the algorithm. If we are to insert more than one thread in the execution path of a given thread, then the prediction of a thread's μ' requires significantly more profiling information (such as the amount of CPU time a thread spends downstream of a port when it reaches that port by passing through a given set of upstream input ports). We want to maintain a light-weight profiling stage that will not disturb application performance during profiling. Hence, we make our algorithm *greedy* by inserting at most one thread in the execution path of an existing thread, but for each one of the heavily utilized threads (thus *generous*).

3) Be fair: We minimize the utility function U, which means that new threads are inserted such that the newly created and the existing threads have balanced load.

4 Optimization Algorithm

We now describe a base optimization algorithm and a set of enhancements that improve its running time. A cost analysis is provided in Appendix B.

4.1 The Algorithm

For a simple chain of operators, designing an algorithm that meets the criteria given in Section 3.2 is straightforward. However, operators that are shared across threads complicate the design in the general case. We need to make sure that one and only one thread is inserted in the execution path of each existing thread, even though the same thread can be inserted in the execution path of multiple existing threads. The main idea behind the algorithm is to reduce the search space via selection and removal of shared operators from the set of possible solutions, and then explore each sub-space separately.

Before describing the algorithm in detail, we first introduce a simple matrix form that represents a sub-space of possible solutions.

Matrix representation: For each thread, we initially have all the operators in the execution path of it as a possible choice for inserting a threaded port. As the algorithm progresses, we gradually remove some of the operators from the list to reduce the search space. For instance, the runtime operator graph from Figure 4 can be converted into the following matrix representation:

$$\begin{array}{c} \\ t_0 \\ t_1 \\ t_2 \\ t_3 \end{array} \left(\begin{array}{c|ccc} o_0, 90\% & o_1, 15\% & o_3, 50\% & o_4, 20\% \\ o_2, 100\% & o_3, 50\% & o_4, 20\% & \\ o_5, 90\% & o_6, 30\% & o_4, 15\% & o_{10}, 5\% \\ o_7, 95\% & o_8, 60\% & o_9, 30\% & o_{10}, 20\% \end{array} \right)$$

The matrix contains one row for each thread in the unmodified application. For each row, it lists the set of operators that are in the execution path of the thread with their associated CPU utilization metrics, which is $\mu(o, t)$. Note that the source operators are placed on the first column and are separated from the rest. They are not considered as potential places to add threaded ports as they have no input ports. We exclude them from the matrix representation for the rest of the paper. The remaining operators are in no particular order, but we sort them by their index for ease of exposition.

The algorithm is composed of four major phases, namely *bottleneck selection*, *solution reduction*, *candidate formation*, and *solution selection*.

Bottleneck Selection: The first phase is the bottleneck selection, which identifies highly utilized threads. A threshold $\beta \in [0, 1]$ is used to eliminate threads whose CPU utilizations are below it. For instance, if $\beta = 0.92$, threads t_0 and t_2 are eliminated since their utilizations are smaller than the threshold and thus are not deemed bottlenecks. For the rest of this section, we assume $\beta = 0.8$ for the running example, which means all of the four threads are considered as bottlenecks.

Solution Reduction: The second phase is the solution reduction, which performs a tree search to reduce the solution space. At the root of the tree is the initial matrix. At each step, we choose one of the leaf matrices that still contains shared operators based on the runtime data flow graph. We pick one of these shared operators for that leaf matrix and perform *selection* and *removal* to yield two sub-matrices in the tree.

Selection means that we select the shared operator as part of the solution, and thus remove all other operators from the rows that contain the shared operator. Furthermore, we remove all operators that originally appeared together with the shared operator in the same row, from other rows, since they cannot be selected in a

valid solution. Figure 5 shows an example. Consider the edge labeled $S3$, which represents the case of selecting shared operator 3. After the selection, the first two rows now have operator 3 as the only choice. Furthermore, operators 1 and 4—which previously appeared in the same row as 3—are removed from all rows, as picking them would result in inserting more than one thread on the execution paths of the first two threads.

Removal means we exclude the shared operator from the solution, and thus we remove it from all rows where it appears. Figure 5 shows an example. Consider the edge labeled $R3$, which represents the case of removing the shared operator 3.

The solution reduction phase continues until the leaf matrices have all of their shared operators removed.

Candidate Formation: After the solution reduction phase, all leaves of the tree contain *pre-candidate* solutions. The goal of the candidate formation phase is to create candidate solutions out of the pre-candidate ones. As part of candidate formation, first we apply a filtering step. If we encounter a leaf matrix where a thread is left without an operator in its row, yet there is another dependent thread that has a non-empty row, then we eliminate this leaf matrix. We consider threads that share operators in their execution paths as dependent.

As an example, the rightmost two leaves in Figure 5 are removed in the filtering step. After the filtering step, we convert each remaining pre-candidate solution into a candidate solution by making sure that each non-empty row contains a single operator, i.e., we convert each matrix into a column vector. When there are multiple operators in a row, we compute the utility function $U(o,t)$ for each, and pick the one that gives the lowest value. As an example, the pre-candidate solution pointed at by arrow $S4$ in Figure 5 is converted into a candidate solution by picking operator 8 as opposed to operator 9. Operator 8 has a lower utility value, $U(o_8,t_3) = 0.6$, compared to operator 9's utility, $U(o_9,t_3) = 0.65$.

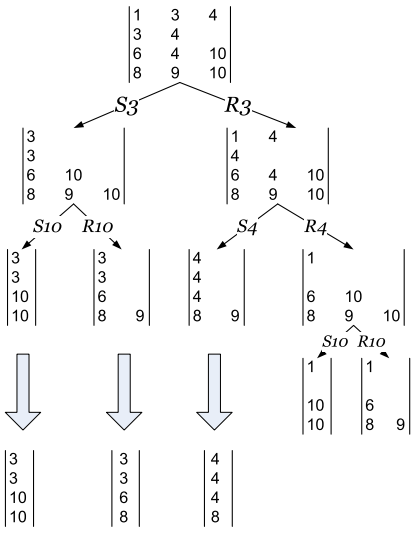


Fig. 5: Solution reduction and candidate formation.

Solution Selection: In the solution selection phase, we pick the best candidate among the ones produced by the candidate formation phase. Recall that our utility function $U(o,t)$ was defined on a per-thread basis. To pick the best candidate, we use the aggregate utility function $U(C)$, where $C = \{\langle o,t\rangle\}$ represents a candidate solution. Recall that we pick the maximum of the individual utilities[2], thus $U(C) = max(U(o,t) \,|\, \langle o,t\rangle \in C)$. We pick the candidate solution with the minimum aggregate utility as the final solution. In the running example, this corresponds to picking $C = \{\langle o_4,t_0\rangle, \langle o_4,t_1\rangle, \langle o_4,t_2\rangle, \langle o_8,t_3\rangle\}$ with aggregate utility of 0.8.

4.2 Algorithm Enhancements

We further propose and employ two enhancements to our basic algorithm.

Pruning: Our enhanced algorithm stops branching when it finds that the utility function value for some of the rows in the current matrix is already equal to or larger than 100%. For example, in Fig. 5, there is no point to continue branching after $R4$, since thread t_1 has no potential threaded port to add and will remain bottlenecked after inserting other threads.

Sorting: In the solution reduction phase, we use the degree of operator sharing as our guideline for picking the next solution to further reduce. We sort the shared operators based on the number of rows they appear in. This way, if a shared operator shows up in the execution path of many threads, it is considered earlier in the exploration as it will result in more effective reduction in the search space, especially when used with pruning. When selected, shared operators have a higher chance of causing the utility function value to go over 100% due to contribution from multiple threads.

5 Evaluation and Control

The thread insertions proposed by the optimization stage are put into effect by the runtime. After inserting the new threads, the evaluation stage measures the throughput on input ports which received a threaded port. The throughput is defined as the number of tuples processed per second. If the throughput increased for all of the input ports that has received a threaded port, then the controller stage moves on to the next iteration.

If the throughput has not increased for some of the input ports, then the control stage performs *blacklisting*. The ports for which the throughput has not improved are blacklisted. Furthermore, the thread insertions are reverted by removing these threaded ports from the flow graph. Blacklisted input ports are excluded from consideration in future optimization stages. If the percentage of blacklisted input ports exceeds a pre-defined threshold $\alpha \in [0,1]$, then the process halts. Otherwise, we move on to the next iteration. It is possible that the process halts even before the threshold α is reached, as a feasable solution may not be found during the optmization stage.

Alternative blacklisting policies can be applied to reduce the change of getting stuck at a local minima. For instance, the blacklisted ports can be maintained on a

2. When there are more than one dependent thread groups, utility U is computed independently for each group and the maximum is taken as the final aggregate utility.

per-pipelining configuration basis rather than globally, at the cost of keeping more state around.

6 Profiler

We describe the basic design of the profiler component. The implementation details can be found in Appendix C.

Our profiler follows the design principle of *gprof* [18], that is, to use both instrumentation and periodic sampling for profiling. However, the instrumentation is not part of the generated code. Instead, the SPL runtime has lightweight instrumentation which records thread activity with respect to operator execution. More specifically, the instrumented SPL runtime monitors the point at which a thread enters or exits an input port, so that it can track which ports are currently active. It uses a special per-thread stack, called the E-stack, for this purpose.

In order to collect the amount of CPU time a thread spends downstream of an input port, our system periodically samples the thread status and traverses the E-stacks. We call the period between two consecutive samplings the *sampling period*, denoted by p_s. If there are N occurrences during the last p_o seconds where thread t was found to be active doing work downstream of operator o's input port, then the per-port thread utilization $\mu(o,t)$ is given by $\frac{N}{p_o/p_s}$. The intuition for this calculation is that it is the number of observations (N) divided by how many times we sample during a given time period (p_o/p_s).

Periodic sampling is inherently subject to statistical inaccuracy, thus enough samples should be collected for accurate results. This could be achieved by either increasing the duration of profiling (p_o) or decreasing the sampling period (p_s). Given the long running nature of streaming applications, we favor the former approach.

7 Dynamic Thread Insertion/Removal

Thread insertion and removal is implemented by dynamically adding and removing threaded ports. Both activities require suspending the current flow of data for a very brief amount of time, during which the circular buffer associated with the threaded port is added/removed to/from the data flow graph. Finally, the suspended flow is resumed. Suspending the flow, however, is not the only step necessary to preserve safety. In the presence of *stateful* operators, dynamic lock insertion and removal is required to ensure mutually exclusive access to shared state. This is further discussed in Appendix D. Our implementation does make use of thread pools, since the additional work that is performed during thread injection and removal dominates the overall cost.

8 Experimental Results

We evaluate the effectiveness of our solution based on experimental results. We perform three kinds of experiments. First, we use micro-benchmarks to evaluate the components of our solution and verify the assumptions that underlie our techniques. Second, we evaluate the running time efficiency of our optimization algorithm under varying topologies and application sizes, using synthetic applications. Third, using three real-world applications, we compare the throughput our auto-pipelining scheme achieves to that of manual optimization as well as no optimization. The second set of experiments, based on synthetic applications, can be found in Appendix F.

8.1 Experimental Setup

We have implemented our auto-pipelining scheme in C++, as part of the SPL runtime within System S [3].

All of our experiments were performed on a host with 2 Intel Xeon processors. Each processor has 4 cores, and each core is a 2-way SMT, exposing 16 hardware threads per node, but only 8 independent cores. When running the experiments, we turn off hyperthreading so that the number of virtual cores equals the number of physical cores (which is 8)[3].

8.2 Micro-benchmarks

For the micro-benchmarks, we use a simple application topology that consists of a chain of 8 operators. All operators have the same cost and perform the same operation (a series of multiplications). The cost of an operator is configurable. Plots for cost-throughput tradeoff are given in Appendix E.

8.2.1 Pipelining benefit

Pipelining is beneficial under two conditions. First, enough hardware resources should exist to take advantage of an additional thread. Second, the overhead of copying a tuple to a buffer and a thread switch-over should be small enough to benefit from the additional parallelism. When these conditions do not hold, the evaluation and control stages of our auto-pipelining solution will detect this and adjust the adaptation process.

We evaluate the pipelining benefit and show how it relates to the overhead associated with threaded ports by measuring the speedup obtained when executing our application with two threads instead of one. Figure 6 plots the speedup as a function of the per-tuple processing cost, for different tuple sizes. When the per-tuple processing cost is small, it is expected that using an additional thread will introduce significant overhead. In fact, we observe that the additional thread reduces the performance (speedup less than 1). As the per-tuple processing cost gets higher, we see that perfect speedup of $2\times$ is achieved. The tuple sizes also have an impact on the benefit of pipelining. For large tuple sizes, the additional copying required to go through a buffer creates overhead. Thus, the crossover point for achieving $> 1\times$ speedup happens at a lower per-tuple cost for smaller sized tuples. For small tuples, custom allocators [19] can be used to further improve the performance. For large tuples, the copying of the data contents dominates the cost. While copy-on-write (COW) techniques can be

3. This is done to avoid impacting the scalability micro-benchmarks.

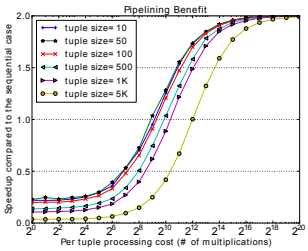


Fig. 6: Speedup vs. processing cost.

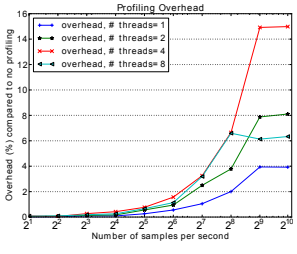


Fig. 7: Profiling overhead vs. sampling rate.

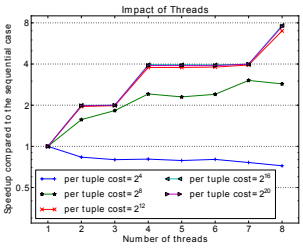


Fig. 8: Speedup for different # of threads

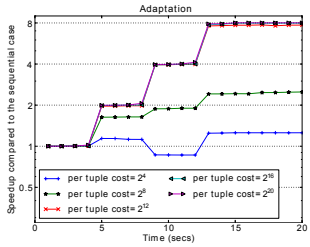


Fig. 9: Adaptation with auto-pipelining

used to avoid this cost, it is well accepted that COW optimizations are not effective in the presence of multi-threading.

8.2.2 Profiling overhead

Light-weight profiling that does not disturb application performance is essential for performing auto-pipelining at run-time. In Figure 7 we study the profiling overhead. The overhead is defined as the percent reduction in the throughput compared to the non-profiling case. The figure plots the overhead as a function of the number of samples taken per second, for different number of threads. The operators are evenly distributed across threads. We observe that, as a general trend, the profiling overhead increases as the profiling rate grows. For the remainder of the experiments in this paper, we use a profiling sampling rate of $p_s = 100$, which corresponds to a 3% reduction in performance. Note that the profiler is only run for a specific period (for p_o seconds) during one iteration of the adaptation phase. Once the adaptation is complete, no overhead is incurred due to profiling.

We further observe that increasing the profiling rate beyond a threshold does not increase the overhead anymore. This is because the system starts to skip profiling signals when the sampling period p_s is shorter than the time needed to run the logic associated with the profiling signal. Interestingly, the profiling overhead does not monotonically increase with the number of threads. At first glimpse, this may be unexpected since more threads means more execution stacks to go through during profiling. However, with more threads, each execution stack has less entries, which decreases the overhead.

For most operator graphs, it is the depth of the operator graph that impacts the worst case profiling cost, rather than the number of threads used. For instance, for a linear chain, the number of stack entries to be scanned only depends on the depth of the graph. For bushy graphs this number can also depend on the number of threads, even though it is rarely linear in the number of threads in practice (a reverse tree is the worst case).

8.2.3 Impact of threads

Recall that one of the principles of our optimization is to insert a threaded port in the execution path of each bottleneck thread. We do this because the speedup from adding threads one-at-a-time will result in a series of non-improvements, followed by a jump in performance when all bottleneck threads finally get help. In Figure 8, we verify this effect. The figure plots the speedup as a function of the number of threads, for different tuple costs. The threads are inserted in a balanced way, by picking the thread that executes the highest number of operators and partitioning it into two threads.

We observe that, for sufficiently high per-tuple processing costs, the speedup is a piece-wise function which jumps at certain number of threads, like 2, 4, and 8. Each such jump point corresponds to a partitioning where all threads execute the same number of operators. This result justifies our algorithm design which inserts multiple threaded ports in one round. For low per-tuple processing costs (such as 2^8) the speedup is not ideal, and for very low per-tuple processing costs (such as 2^4), the performance degrades.

8.2.4 Adaptation

We evaluate the adaptation capability of our solution by turning on auto-pipelining in an application whose topology is a simple chain of `Functor` operators. For this experiment, we measure the throughput of the application as a function of time. The adaptation period is set to 5 seconds. We report the throughput relative to the sequential case. Figure 9 reports these results for different per-tuple processing costs.

We observe that our algorithm intelligently achieves optimal speedup for different per-tuple costs. For instance, when the per-tuple cost is 2^4, our algorithm finds out that its second optimization decision does not improve overall throughput, and thus it rolls back to the previous state. For higher per-tuple costs, such as 2^{20}, the algorithm does not stop adding threaded ports until it reaches the unpartitionable state, that is 1 operator per thread. Comparing Figures 8 and 9, we see that auto-pipelining lands on the globally optimal configuration in terms of the throughput.

The total adaptation time of the system depends on two major components: (i) the number of steps taken, and (ii) the adaptation period. Since our algorithm helps all bottlenecked threads at each step, its behavior with respect to the number steps taken is favorable. For instance it takes $\log_2(8) = 3$ steps to reach 8 threads in Figure 9. For more dynamic scenarios, we can reduce the adaptation period to reduce the overall adaptation time. The only downside is that, reducing the adaptation period without decreasing the accuracy of the profiling

data requires increasing the profile sampling rate, which can increase the profiling cost.

8.3 Application Benchmarks

The application benchmarks consist of three real-world stream processing applications with their associated workloads. These applications are named Lois, Vwap, and LinearRoad. The LinearRoad application (the smallest of the three) is depicted in Figure 10, whereas other applications are depicted in Appendix G.

The Lois [20] dataset is collected from a Scandinavian radio-telescope under construction in northwestern Europe. The goal of the Lois application is to detect cosmic ray showers by processing the live data received from the radio-telescope.

The Vwap [21] dataset contains financial market data in the form of a stream of real-time bids and quotes. The goal of the Vwap application is to detect bargains and trading opportunities based on the processing of the live financial feed.

LinearRoad [22] dataset contains speed, direction, and position data for vehicles traveling on road segments. The goal of the application is to compute tolls for vehicles traveling on the hypothetical "Linear Road" highway.

The breakdown of the operators constituting the applications and summaries of the application characteristics are given in Appendix G. It is important to note that the Lois and LinearRoad applications have few bush segments in their topology, whereas Vwap has many. The LinearRoad application makes heavy use of custom operators, whereas the other applications are composted of mostly built-in operators.

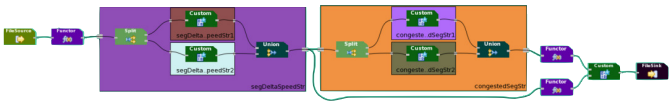


Fig. 10: LinearRoad – A vehicle toll computation app.

We run three versions of these programs: unoptimized, hand-optimized, and auto-pipelined. The hand-optimized versions are created by explicitly inserting threaded ports in the SPL code of the application. This was carried out by the application developers, independent of our work. For all cases, we measure the total execution time for the entire data set. For the auto-pipelined version, the adaptation period is also included as part of the total execution time.

Figure 11 gives the results. For the Lois application, we see around $1.5\times$ speedup compared to the unoptimized version, for Vwap we see around $3\times$ speedup, and for LinearRoad we see $2.56\times$ speedup. Note that these are real-world applications, where sequential portions and I/O bound pieces (sources and sinks) make it difficult to attain perfect speedup. It is impressive that our auto-pipelining solution matches the hand-optimized performance in the case of Lois, and improves upon it by around $2\times$ for both Vwap and LinearRoad. It is also worth noting that in the case of Lois, the programmer has statically added threaded ports based on her experience and the suggestion from a fusion optimization tool called COLA [16]. Considering that the auto-pipeliner takes around 20 seconds to adapt in this particular case, the throughout attained for the auto-pipelining solution is in fact higher than the hand-optimized case.

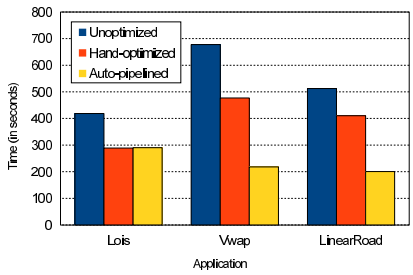


Fig. 11: Running time for Lois and Vwap

Overall, auto-pipelining provides equal or significantly better performance compared to hand optimization, at no additional cost to the application developers.

9 Related Work

Our work belongs to the area of auto-parallelization and we survey the related topics accordingly. Coverage of related work on profiling is given in the Appendix H.

Dynamic multi-threaded concurrency platforms, such as Cilk++ [8], OpenMP [7], and x10 [12], 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.

Kremlin [23] is an auto-parallelization framework that complements OpenMP [7]. Kremlin recommends to programmers a list of regions for parallelization, which is ordered by achievable program speedup.

Cilkview [24] is a Cilk++ analyzer of program scalability in terms of number of cores. Cilkview performs system-level modeling of scheduling overheads and predicts program speedup. Bounds on the speedup are presented to programmers for further analysis.

Autopin [25] 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.

StreamIt [26] is a language for creating streaming applications and can take advantage of parallelism present in data flow graph representation of applications, including task, pipeline, and data parallelism. However, StreamIt is mostly a synchronous streaming system, where static scheduling is performed based on compile-time analysis of filters written in the StreamIt language.

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.

Task assignment in distributed computing has been an active research problem for decades. General task assignment is intractable. In [28], several programs with special structures are considered and the optimal assignment is found by using a graph theoretic approach.

There has been extensive research in the literature on compiler support for instruction-level or fine-grained pipelined parallelism [29]. In this 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 parallel.

Relevant to our study is the work in [30], 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 [31]), 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.

A more detailed analysis of the differences of our work from others is given in Appendix I.

10 Conclusion

In this paper, we described an auto-pipelining solution for data stream processing applications. It automatically discovers pipeline and task parallelism opportunities in stream processing applications, and applies dynamic profiling and controlling to adjust the level of parallelism needed to achieve the best throughput. Our solution is transparent in the sense that no changes are required on the application source code. Our experimental evaluation shows that our solution is also effective, matching or exceeding the speedup that can be achieved via expert tuning. Our solution has been implemented on a commercial-grade data stream processing system. We provide directions for future work in Appendix K.

References

[1] 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.

[2] 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 *CIDR*, 2005.

[3] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani, "Design, implementation, and evaluation of the linear road benchmark on the Stream Processing Core," in *ACM SIGMOD*, 2006.

[4] "StreamBase Systems," http://www.streambase.com, retrieved October, 2011.

[5] 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.

[6] "S4 distributed stream computing platform," http://www.s4.io/, retrieved October, 2011.

[7] "Openmp. http://www.openmp.org," retrieved October, 2011.

[8] "Cilk++. http://software.intel.com/en-us/articles/intel-cilk-plus/," retrieved October, 2011.

[9] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007.

[10] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the Chapel language," *IJHPCA*, vol. 21, pp. 291–312, 2007.

[11] G. L. S. Jr., "Parallel programming and code selection in fortress," in *ACM PPoPP*, 2006.

[12] 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 *OOPSLA*, 2005.

[13] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo, "Spade: The System S declarative stream processing engine," in *ACM SIGMOD*, 2008.

[14] M. M. A. et al, "System R: A relational approach to data management," *ACM TODS*, vol. 1, no. 2, pp. 97–137, 1976.

[15] B. Gedik, H. Andrade, and K.-L. Wu, "A code generation approach to optimizing high-performance distributed data stream processing," in *ACM CIKM*, 2009.

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

[17] J. Giacomoni, T. Moseley, and M. Vachharajani, "FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue," in *ACM PPoPP*, 2008.

[18] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: A call graph execution profiler (with retrospective)," in *Best of PLDI*, 1982, pp. 49–57.

[19] "TCMalloc: Thread-caching malloc," http://goog-perftools.sourceforge.net/doc/tcmalloc.html, retrieved August, 2012.

[20] "Lois. http://www.lois-space.net/," retrieved October, 2011.

[21] H. Andrade, B. Gedik, K.-L. Wu, and P. S. Yu, "Processing high data rate streams in System S," *JPDC*, vol. 71, no. 2, pp. 145–156, 2011.

[22] A. Arasu, S. Babu, and J. Widom, "The cql continuous query language: Semantic foundations and query execution," *The VLDB Journal*, vol. 15, no. 2, pp. 121–142, 2006.

[23] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor, "Kremlin: Rethinking and rebooting gprof for the multicore age," in *PLDI*, 2011.

[24] Y. He, C. E. Leiserson, and W. M. Leiserson, "The cilkview scalability analyzer," in *ACM SPAA*, 2010.

[25] T. Klug, M. Ott, J. Weidendorfer, and C. Trinitis, "Autopin: Automated optimization of thread-to-core pinning on multicore systems," *HiPEAC*, vol. 3, pp. 219–235, 2011.

[26] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *ASPLOS*, 2006.

[27] X. Zhang, A. Navabi, and S. Jagannathan, "Alchemist: A transparent dependence distance profiling infrastructure," in *CGO*, 2009, pp. 47–58.

[28] S. H. Bokhari, "Assignment problems in parallel and distributed computing," in *Kluwer Academic Publishing*, 1987.

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

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

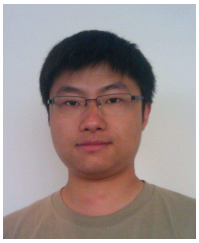
[31] M. D. Beynon, T. M. Kurç, Ü. V. Çatalyürek, C. Chang, A. Sussman, and J. H. Saltz, "Distributed processing of very large datasets with DataCutter," *Parallel Computing Journal*, vol. 27, no. 11, pp. 1457–1478, 2001.

[32] E. Jeřábek, "Dual weak pigeonhole principle, Boolean complexity, and derandomization," *Annals of Pure and Applied Logic*, vol. 129, pp. 1–37, 2004.

[33] S. Liang and D. Viswanathan, "Comprehensive profiling support in the Java virtual machine," in *COOTS*, 1999, pp. 229–242.

[34] "Oprofile. http://oprofile.sourceforge.net/about/," retrieved October, 2011.

[35] J.-A. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl, "Continuous Profiling: Where have all the cycles gone?" in *SOSP*, 1997, pp. 1–14.



Yuzhe Tang received the BSc degree, and MSc degree in Computer Science and Engineering from Fudan University, Shanghai, China, in 2006 and 2009, respectively. At the time of this writing he was an intern at the IBM T. J. Watson Research Center working with Dr. Gedik on high performance streaming systems. He is currently a Ph.D student at the Data Intensive Distributed Systems Lab, in College of Computing, at the Georgia Institute of Technology. His research interests include distributed systems and cloud computing, databases, system security and privacy. He has worked on HBase and Hadoop ecosystem, profiling and system optimizations, anonymity protocols and data management over DHT networks.

Buğra Gedik is currently an Assistant Professor at the Computer Engineering Department, Bilkent University, Turkey. Prior to that he worked as a Research Staff Member at the IBM T. J. Watson Research Center. His research interests are in distributed data-intensive systems with a particular focus on stream computing. In the past, he served as the Chief Architect for IBM's InfoSphere Streams product. He is the co-inventor of the SPL and the SPADE stream processing languages. He is the co-recipient of the IEEE ICDCS 2003, IEEE DSN 2011, ACM DEBS 2011 and 2012 best paper awards. He served as the co-PC chair for the ACM DEBS 2009 and IEEE CollaborateCom 2007 conferences. He is an associate editor for the IEEE Transactions on Services Computing journal. He served on the program committees of numerous conferences, including IEEE ICDCS, VLDB, ACM SIGMOD, IEEE ICDE, and EDBT. He has published over 60 peer-reviewed articles in the areas of distributed computing and data management. He has applied for over 30 patents, most of them related to his work on streaming technologies. He was named an IBM master inventor and is the recipient of an IBM Corporate Award for his work in the System S project. He has obtained his Ph.D. degree in Computer Science from Georgia Institute of Technology, USA and prior to that, his B.S. degree in Computer Engineering and Information Science from Bilkent University, Turkey.

APPENDIX A
SENSORQUERY SPL APPLICATION

Listing 1: `SensorQuery`: A simple application in SPL.

```
composite SensorQuery {
 type
   Location = tuple<float32 x, float32 y>;
   Sensor = tuple<uint64 sid, float64 value, Location sloc>;
   Query = tuple<uint64 qid, Location qloc, float64 radius>;
   Result = Sensor, Query, tuple<float32 distance>;
 graph
   stream<Sensor> Sensors = SensorSource() {}
   stream<Query> Queries = QuerySource() {}
   stream<Result> Results = Join(Sensors as S; Queries as Q) {
    window Sensors: sliding, time(10.0);
           Queries: sliding, count(0);
    param match: distance(S.sloc, Q.qloc) <= Q.radius;
    output Results: distance = distance(S.sloc, Q.qloc);
   }
   () as Sink = TCPSink(Results) {
    param
      role : client;
      address : "192.168.0.10";
      port : 40000;
   }
}
```

APPENDIX B
COMPLEXITY ANALYSIS

Here, we provide a cost analysis of the base algorithm.

Suppose there are initially r threads, which is equal to the number of rows in the initial solution matrix. Further assume that there are s shared operators in the runtime graph. To analyze the complexity of our algorithm, we start by calculating the number of leaf candidates, denoted as $cn(s,r)$, in the solution reduction phase. $cn(s,r) = \sum_x cn(s,r,x)$, where $cn(s,r,x)$ denotes the number of candidates with x shared operators in them. $cn(s,r,x)$ must be smaller than the cardinality of x choose s, that is $cn(s,r,x) \leq \binom{s}{x}$. Note that a single candidate can have up to $\frac{r}{2}$ shared operators in it as each shared operator fixes the assignments for at least two rows of the solution matrix. Therefore,

$$\begin{aligned} cn(s,r) &= \sum_{x\in[0,\min(s,\frac{r}{2})]} cn(s,r,x) \\ &\leq \sum_{x\in[0,\min(s,\frac{r}{2})]} \binom{s}{x} \\ &= \begin{cases} \Theta(2^s) & \text{if } \frac{r}{2} \geq \frac{s}{2} - \sqrt{s}, \\ \Theta\left[\frac{\binom{s}{r/2}}{\left(1-r/s\right)}\right] & \text{otherwise.} \end{cases} \end{aligned} \quad (1)$$

The last step is due to a closed form for partial sum of binomial coefficients [32]. $cn(s,r)$ can be used as a rough bound for algorithm complexity, assuming each candidate costs $O(1)$ computation units. The complexity of the algorithm can be further bounded by considering that the solution reduction operates independently for sets of initial threads that are disjoint in terms of their operators. Thus we can represent the number of candidates as $\sum_{\langle s_i,r_i\rangle} cn(s_i,r_i)$ where $\sum_i r_i = r$ and s_i represents the number of shared operators in the ith thread group. For large graphs, $max(\{r_i\})$ is often smaller that r. However, this does not change the worst case complexity.

We can provide a finer-grained complexity analysis by breaking down the per-candidate cost. First, we use m to denote the number of operators that appear in the initial solution matrix. The algorithm finds the set of shared operators in a two-level nested loop, in which case each loop is a scan of the whole matrix, costing $O(m^2)$ units. During the solution reduction phase, $cn(s,r)$ leaf candidates in the search tree implies $cn(s,r) - 1 = O(cn(s,r))$ internal nodes. The split operation at each internal node needs a full scan of the matrix, resulting in a cost of at most m. The candidate formation phase needs to scan the matrix of each candidate, thus leading to per-candidate cost of at most m. The last phase, solution selection, costs r units per candidate since each candidate is a column vector. In total, the complexity is bounded by $O(m^2 + cn(s,r)\cdot(2m+r))$, which further simplifies to $O(m^2 + cn(s,r)\cdot m)$.

As we will see later in Section 8, the algorithm runs quite fast in practice for large graphs, especially when the the pruning optimization is applied.

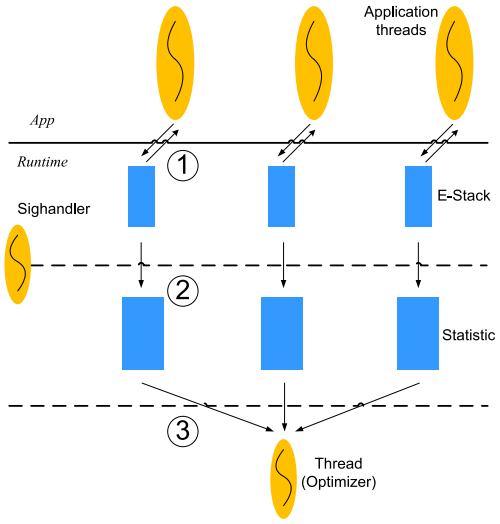


Fig. 12: Profiler implementation

APPENDIX C
PROFILER IMPLEMENTATION

Our runtime profiler implementation consists of three components. Figure 12 illustrates each component, which we also describe below.

E-stack

For each application thread, the profiling system maintains a simple execution stack, called an *E-stack*. The system pushes/pops entries into/from the E-stack every time the thread associated with the stack enters/exits an operator. Figure 13 shows a snapshot of the E-stack of a thread executing operator o_{10} after going through operators o_5 and o_6.

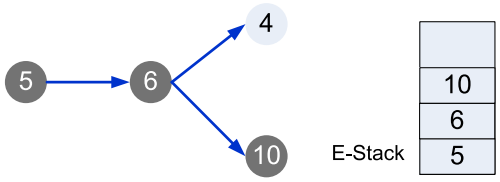


Fig. 13: Example of an E-stack

Signal handler

For each sampling period p_s, the profiling system checks the execution stacks of all actively running threads. This is achieved by registering a timer for the signal `SIGPROF` with the timer interval set to p_s. The operating system then sends signal `SIGPROF` every p_s seconds, which one of the application threads receives inside of a signal handler. Upon each receipt of the signal, the signal handler takes snapshots of the E-stack for all currently active[4] threads. For each stack entry in an E-stack snapshot, it increments a counter for the thread and operator pair associated with the entry. Note that it scans the entire stack because our goal is to compute the amount of CPU time a thread spends *downstream* of a given operator input port. For instance, in Figure 13, it increments the counters for operators o_5, o_6, and o_{10}, as the active thread is doing work downstream of all of these three operators at this time.

Summarization

Every optimization period p_o, the counters maintained for each thread and operator pair are summarized into the per-port thread utilization numbers. These are the final set of statistics that will be used by the optimization stage.

A problem associated with the simple counting scheme used for profiling is that, in a multi-threaded environment, the more active threads there are, the more frequently (in wall clock time) signal `SIGPROF` is delivered. This skews the statistics. We use a mechanism called *frequency autoscaling* to correct this skew. Rather than incrementing each counter by 1 at each sampling step, we increment it by $1/A$ where A is the number of currently active threads.

To handle profiling in a multi-threading environment, the consistency problem arises when E-stacks are written by SPL runtime while being read by the signal handler. HProf [33] tackles this problem by suspending threads at every sampling, which however interrupts program execution and increases overhead. We avoid thread suspension or any locking mechanism by writing E-stack entries in an atomic yet efficient way.

Each stack entry is stored as a 64-bit volatile value in a 64-bit system — a 32-bit operator identifier and a 32-bit port index. We rely on the details of the Intel architecture to ensure that reading the stack entries is atomic. The current size of the stack is also kept as a volatile counter. Since the updating of the stack entries and the stack size are not transactional, sometimes the profiler can scan an entry that is not active. However, since this happens with very low frequency it barely impacts the aggregate values computed by the profiler.

4. The `proc` system in Linux is used for this purpose in the current prototype.

Appendix D
Locking & Thread Insertion/Removal

There is a subtle concern regarding thread safety in the presence of thread insertion/removal and multi-threaded execution of *stateful* operators. Before we describe the potential problems that may result from thread insertions, we first give a brief overview of the relevant aspects of the programming model used to develop operators in SPL.

Operators are implemented through an event-driven interface as described in Section 2.1. An operator that contains state which is modified as a result of processing tuples delivered to one of its input ports is said to be stateful. Such operators need to ensure that the state is protected against concurrent modification. Recall that operators can be executed concurrently by multiple threads. In SPL, stateful operators use an *auto port mutex* object to protect their state from concurrent modification. An auto port mutex is a scoped mutex that either creates a critical section around a block of code, or simply reduces to a no-op at the cost of an untaken branch.

The SPL runtime decides which one of these behaviors is to be exhibited depending on safety analysis. Operator developers always protect their state from concurrent access using auto port mutexes, yet the runtime can decide to effectively remove these mutexes when it is safe to do so. The safety analysis is performed by a simple process of *thread propagation* to decide if an operator can potentially be called by multiple threads.

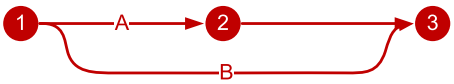


Fig. 14: Thread propagation: Initial state

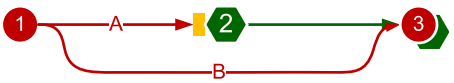


Fig. 15: Thread propagation: After adding port 2

The thread propagation analysis is performed every time a thread is inserted or removed at runtime. When a thread is inserted, the analysis is needed to *turn on* some of the auto port mutexes to ensure safety. For a thread removal, it needs to be performed to *turn off* some of the auto port mutexes to ensure good performance.

Figure 14 shows an example SPL application. In this example, we want to add a threaded port to operator o_2. As a result of this change, there will be two threads executing the downstream operator o_3 as shown in Figure 15. If o_3 is a stateful operator, then the auto port mutexes used by the operator are turned on by

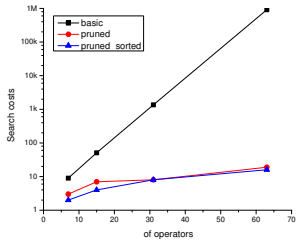
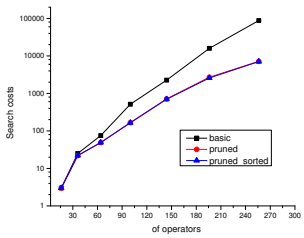
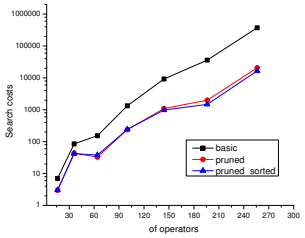


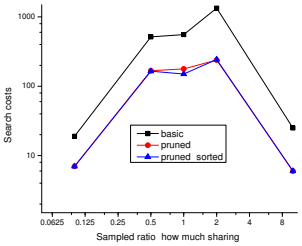
Fig. 16: Performance of optimization algorithm with the reverse tree topology



(a) Less shared threads ($sr = 0.5$)



(b) More shared threads ($sr = 2$)



(c) As a function of sampled ratio

Fig. 17: Performance of optimization algorithm with random topology

the runtime, before the flow is resumed following the insertion.

APPENDIX E
OPERATOR COST VS. THROUGHPUT

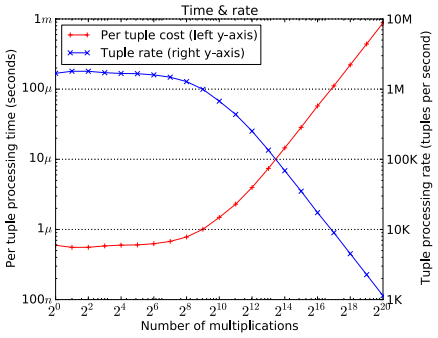


Fig. 18: Per-tuple processing time and max. rate as a function of processing cost.

Figure 18 plots the time it takes to process a single tuple (left y-axis), as well as the maximum rate of processing that can be achieved with a single operator (right y-axis) as a function of the per-tuple processing cost represented as the number of multiplications.

APPENDIX F
SYNTHETIC BENCHMARKS

This set of experiments evaluate the running-time efficiency of our optimization algorithm, as well as the effectiveness of the two enhancements, *pruning* and *sorting*.

In these experiment, we use two different kinds of synthetic topologies. These are the *reverse tree* and the *random graph* topologies. For a reverse tree topology, each leaf-level operator serves as a starting point for a different thread and each thread executes the set of operators that forms a path from the leaf-level operator to the root. We pick this topology as it represents an extreme scenario where there is massive amount of sharing across threads. In our experience, this kind of topology is not seen often in practice.

For a random topology, operator IDs are randomly picked from a finite domain of integers $[0, d)$. Here, a parameter called *sampled ratio*, denoted by sr, is used to measure the degree of overlapping between threads, that is, how many operators in one thread can be shared by another thread. Suppose there are n threads and each thread executes m operators, then $sr = \frac{n \cdot m}{d}$. Utilization values are uniformly distributed among operators for each thread.

We run each experiment 5 times and report the average performance numbers. The reported performance is the search cost, which is quantified by the number of tree nodes traversed during the execution of the optimization algorithm.

Figure 16 presents the results for the reverse tree topology, in which different tree sizes are tested from 2^3 to 2^6. As expected, search costs of our basic algorithm (i.e., without any enhancements) grow exponentially with the number of shared operators [5]. We observe that compared to the basic algorithm, pruning is very effective. It achieves an order of magnitude saving in search cost (the y-axis is in log scale). Applying sorting on top of pruning provides modest additional improvement, only for small number of operators.

Our results are further corroborated by the random graph based experiments. For these experiments, we varied two parameters: the number of operators n and the sampled ratio sr. These results are shown in Figure 17.

Figure 17a plots the search cost as a function of number of operators with fewer sharing (the sampled ratio sr is as small as 0.5), while Figure 17b plots the search cost as a function of number of operators with more sharing (sr is set to 2). We observe that although the search costs with more sharing between threads tend to be higher, the overall trend with increasing number of operators is similar. Search costs largely grow linear with the number of operators.

Figure 17c plots the search cost as a function of the sampled ratio. We observe that the search costs first grow with increasing sampled ratios and then drop. While increased sharing raises the cost of the algorithm initially, excessive sharing results in shrinking the search space, resulting in this bi-modal behavior.

Overall, Figure 17 shows that pruning is a very effective optimization strategy for our algorithm, under

5. In a reverse tree topology, the number of operators equals that of shared operators.

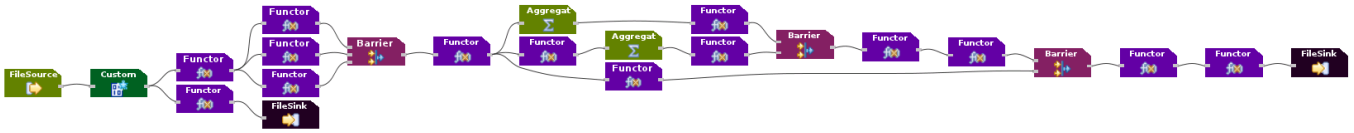


Fig. 19: Lois – Cosmic ray shower detection application

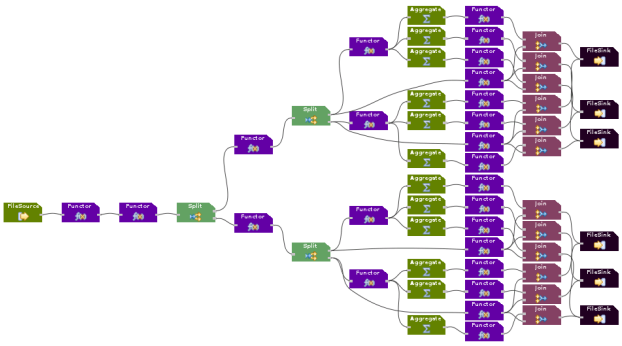


Fig. 20: Vwap – Bargain detection application

various circumstances, often providing an order of magnitude improvement in running time.

Appendix G
Application Graphs

The Vwap and Lois applications are depicted in Figures 20 and 19, respectively. LinearRoad application is depicted in Figure 10.

Vwap is a medium-scale application (58 operators) that contains large number of Functor, Join, and Aggregate operators. The heavy joins and aggregations create opportunities for pipeline parallelism.

Lois is a small-scale application (22 operators) that contains large number of Functor operators. It contains 3 Barriers, which is indicative of task parallelism being present in the flow graph. LinearRoad is a small-scale application (15 operators) that contains large number of Custom operators. It contains 2 Unions, which is indicative of task parallelism also being present in the flow graph.

The Lois and LinearRoad applications have few branches in their topology, whereas Vwap has many. The LinearRoad application makes heavy use of custom operators, whereas the other applications are composted of mostly built-in operators.

Table 1 gives a breakdown of the operators constituting the three applications.

Appendix H
Related Work on Profilers

There are generally two ways to implement a program profiler; statistical sampling and code instrumentation. While sampling is less disruptive to the base program, instrumentation-based profiling can obtain more accurate results.

OProfile [34] and DCPI [35] are representative of sampling-based profiling schemes. They use hardware

	Instance Count		
Operator Kind	Lois	Vwap	LinearRoad
Functor	13	24	3
Join	0	12	0
Split	0	3	2
Barrier	3	0	0
Union	0	0	2
Aggregator	2	12	0
Custom	1	0	5
FileSource	1	1	1
FileSink	2	6	1

TABLE 1: Breakdown of operators used in the Lois and Vwap applications

performance counters to attain high frequency with fairly low overhead.

Code instrumentation for profiling can be applied statically (i.e., before program execution) or dynamically (i.e., during execution). In particular, static instrumentation code can be added to source code manually, automatically through compiler assist, or to the compiled binary via binary translation.

Gprof [18] is a hybrid profiler in the sense that both instrumentation and sampling are used. Static instrumentation keeps track of caller graph and the execution time is obtained by statistical sampling. Our profiler used in auto-pipelining is similar in the sense that it uses both sampling and instrumentation, but the instrumentation is dynamically turned on/off within the SPL runtime, rather then being injected at compile-time.

Appendix I
Comparison to Related Systems

Compared to [30] our auto-pipelining scheme is executed during runtime, since stream processing applications can contain arbitrary user-defined operators and dynamic runtime behavior, making static analysis impractical. This leads to various different design choices: First, rather than relying on static program analysis, we use online profiling to estimate the system overheads and to discover system bottlenecks. Second, while offline decomposition can afford to use dynamic programming for global optimal filter/threaded port selection, our online approach relies on a greedy algorithm for exploring local optimality. Last, computing units in our pipelines do not need to be linearly chained.

Compared to StreamIt, SPL applications follow the asynchronous streaming model, where operator selectivity, cost, and behavior are not known at compile-time. The approach we presented in this paper is thus completely dynamic, both in terms of profiling and optimization.

Compared to other existing work, our auto-pipelining

solution is similar to Autopin in terms of run-time auto-configuration. However, our solution avoids exhaustive search by employing a novel optimization algorithm which is based on profiling statistics. In terms of automatically finding parallelism opportunities, our system is similar to Kremlin and Cilkview. However, the threaded ports exposed in the SPL runtime provide a flexible mechanism for optimization, only requiring operator-level profiling information. Thus we avoid the heavy profiling needs of systems like Kremlin and Cilkview. Finally, auto-pipelining requires no programmer intervention and is designed for data stream processing systems.

Appendix J
Offline versus online profiling

Online profiling has the advantage that it can be performed dynamically at runtime, without user intervention. On the downside, it may adversely impact the runtime performance. In our context, we have shown that the online profiling can be done in a lightweight manner (see Appendix C).

Offline profiling often requires the application developers to perform profile runs and manage additional compilation options for this purpose. While this has the advantage that the actual deployment will not be impacted by the profiling overheads, it requires that the workload and resource setup used for the profile and deployment runs are the same. Furthermore, it cannot adapt to runtime changes in workload and resource availability and application behavior.

In our own experience with the IBM InfoSphere Streams product and its research prototype precursor System S, application developers avoid the use of offline profiling, except for the difficult cases where manual performance optimization runs into difficulties. The work proposed in this paper addresses this problem by removing the additional burden put on the developers with respect to profiling, at a very small cost at runtime.

Appendix K
Future Work

Here, we list a number of future research directions that can build upon this work.

- Placement of the threads created by our algorithm to cores in the system is an additional dimension that can improve the performance. The placement can be performed with the goal of minimizing the overhead of accessing shared resources. This problem can be extended to the case of NUMA platforms, where the access to memory is non-uniform.
- While we looked at pipelining in this paper, another important kind of parallelism is data parallelism. Some operators in data stream processing systems can be replicated and data partitioning can be used to improve throughput. However, data parallelism requires safety analysis that involves understanding some properties of the operators (such as selectivity and partitioned state). An interesting research direction is to combine data parallelism and pipelining for auto-parallelizing streaming applications.
- Finally, the auto-pipelining problem can be extended to the case of distributed stream processing applications that can cross host boundaries.

Acknowledgements

We would like to thank Scott Schneider and Kun-Lung Wu from IBM T. J. Watson Research Center, and Ling Liu from Georgia Institute of Technology for their contributions that has greatly helped in improving this work.