Tools and strategies for debugging distributed stream processing applications‡

Buğra Gedik¹, *, †, Henrique Andrade¹, Andy Frenkiel¹, Wim De Pauw¹, Michael Pfeifer², Paul Allen², Norman Cohen¹ and Kun-Lung Wu¹

¹IBM Research, 19 Skyline Dr, Hawthorne, NY 10532, U.S.A.
²IBM Software Group, 3605 Highway 52 N, Rochester, MN 55901, U.S.A.

SUMMARY

Distributed data stream processing applications are often characterized by data flow graphs consisting of a large number of built-in and user-defined operators connected via streams. These flow graphs are typically deployed on a large set of nodes. The data processing is carried out on-the-fly, as tuples arrive at possibly very high rates, with minimum latency. It is well known that developing and debugging distributed, multi-threaded, and asynchronous applications, such as stream processing applications, can be challenging. Thus, without domain-specific debugging support, developers struggle when debugging distributed applications.

In this paper, we describe tools and language support to support debugging distributed stream processing applications. Our key insight is to view debugging of stream processing applications from four different, but related, perspectives. First, debugging the semantics of the application involves verifying the operator-level composition and inspecting the flows at the logical level. Second, debugging the user-defined operators involves traditional source-code debugging, but strongly tied to the stream-level interactions. Third, debugging the deployment details of the application require understanding the runtime physical layout and configuration of the application. Fourth, debugging the performance of the application requires inspecting various performance metrics (such as communication rates, CPU utilization, etc.) associated with streams, operators, and nodes in the system. In light of this characterization, we developed several tools such as a debugger-aware compiler and an associated stream debugger, composition and deployment visualizers, and performance visualizers, as well as language support, such as configuration knobs for logging and tracing, deployment configurations such as operator-to-process and process-to-node mappings, monitoring directives to inspect streams, and special sink adapters to intercept and dump streaming data to files and sockets, to name a few. We describe these tools in the context of Spade—a language for creating distributed stream processing applications, and System S—a distributed stream processing middleware under development at the IBM Watson Research Center. Published in 2009 by John Wiley & Sons, Ltd.

Received 5 March 2009; Revised 26 May 2009; Accepted 7 August 2009

*Correspondence to: Buğra Gedik, IBM Research, 19 Skyline Dr, Hawthorne, NY 10532, U.S.A.
†E-mail: bgedik@us.ibm.com
‡This article is a U.S. Government work and is in the public domain in the U.S.A.

Contract/grant sponsor: Publishing Arts Research Council; contract/grant number: 98-1846389

Published in 2009 by John Wiley & Sons, Ltd.
1. INTRODUCTION

The ever increasing rate of digital information available from online sources, such as live stock and option trading feeds in financial services, physical link statistics in networking and telecommunications, sensor readings in environmental monitoring and emergency response, and satellite and live experimental data in scientific computing to name a few, has resulted in the development of several Data Stream Processing Systems, DSPSs in short, both in the academia \cite{1-6} and in the industry \cite{7-10}. DSPSs, unlike traditional databases, process data on-the-fly, as it arrives, and produce live results, continuously. In other words, DSPSs execute continual processing \cite{11-13} on streaming data sources and produce streaming results. To achieve high throughput and low latency, DSPS applications are often structured as data-flow graphs that contain large number of operators\footnote{An operator is the building block of a stream processing application. An operator carries out fundamental domain-specific operations such as a join in stream-relational algebra or a fast Fourier transform in signal processing applications.} connected via streams and are deployed on a large set of nodes, in a distributed fashion.

Verifying the correctness of applications, as well as asserting that they exhibit good performance characteristics is an important part of the software development cycle. Debugging stream processing applications can be particularly challenging, as it combines the difficulties of dealing with large distributed applications with asynchronous data flows that are dependent on external sources such as the arrival of a trade transaction in a stock exchange or the violation of a correctness threshold in a manufacturing plant. On the upside, stream processing applications exhibit considerable amount of structure, captured in the form of data-flow graphs and well-structured message formats. The inter-operator interactions can get arbitrarily complex, for example, in applications with many user-defined operators. However, in practice, distributed stream processing applications are not as free form as applications adopting other flavours of distributed computation, such as Parallel Virtual Machine (PVM) \cite{14} or Message Passing Interface (MPI) \cite{15} applications. Taking advantage of this data-flow graph structure, we describe domain-specific tools as well as language and compiler support targeted at easing the debugging of distributed stream processing applications.

We view debugging of distributed stream processing applications from four different, but related, perspectives. Accordingly, we provide tools and language support to address the following perspectives.

1.1. Semantic debugging

Semantic debugging involves making sure that (i) the operators in the data-flow graph are properly configured and parametrized; (ii) the composition of the application at the operator graph-level is correct; and (iii) the output streams do carry the expected contents based on the operator configurations and the contents of the input streams.
We address the first two problems by providing an IDE environment, called StreamsStudio, which includes a tightly coupled Spade [16,17] source editor and application visualizer combination, used to inspect operator configurations (through the source editor) as well as stream connections and overall operator composition (through the visualizer) in a coordinated manner.

To address the second problem we provide several solutions. First and foremost we developed a stream debugger called System S Debugger, sdb for short. By enabling the inspection of an operator’s inputs and outputs, sdb provides a means of debugging from a stream-level and operator-centric point of view. Through user-defined probe points, the stream debugger is capable of intercepting tuples at any port, modifying their contents, and capturing them in memory or in log files. The Spade compiler also includes the capability to compile a distributed application into a stand-alone executable that does not require the full System S runtime and provides a convenient debugging environment when paired with sdb. For applications whose semantics are sensitive to the deployment layout, the sdb can also be run independently on all or some of the application processes as selected by the user, running under the System S runtime.

Second, we provide a monitor directive in the Spade language and associated tooling to view the contents of the streams at runtime, without the need for running the application in debugging mode. Third, we provide custom sink operators to write the stream contents to files, sockets, etc. The latter functionality mostly addresses the ‘light-debugging’ needs of users, analogous to print statements that programmers often employ to debug simple issues that do not require bringing up a full-scale debugger.

1.2. User-defined operator debugging

User-defined operator debugging is accomplished with traditional source-code debugging, in our case through the GNU Debugger (gdb) [18] for C++, for understanding the low-level implementation details of operator logic. This includes debugging the internals of user-defined operators, as well as third-party or user-defined type-generic operator toolkits (see Section 2). We address the need for debugging user-defined code by extending sdb to support mixed-mode sdb/gdb sessions. The mixed-mode sessions allow users to take advantage of sdb’s stream-level debugging support to set conditional probe points on select ports of operators that make up the application flow graph and then launch gdb to inspect the user-defined aspects of the operator logic.

In addition to sdb’s mixed-mode, we also provide the usual light-debugging capabilities for user-written code, such as logging at different levels of verbosity, fine-tuning the logging level via filters on content labels, providing on-screen logging on a terminal, as well as language constructs to specify which pieces of the application are to be run within a stand-alone terminal (i.e. an xterm) or under the debugger. Spade also provides language facilities to conveniently test user-defined functions and complex expressions (such as regular expressions) at compile-time.

1.3. Deployment debugging

Spade can automatically map operators to runtime execution containers, named processing elements (see Section 2), and processing elements to nodes in the system. Therefore, it is important that the user can understand and verify the resulting runtime layout and configuration of the application. Even though the semantic correctness of the applications may not depend on the runtime layout in many cases, the specifics of the layout are often relevant to common tasks such as application scheduling.
monitoring as well as load balancing. As a result, it is important to ensure that the deployment layout of the application is correct and is inline with the compile-time constraints specified at the language-level. We provide two mechanisms for the purpose of deployment debugging. First, the StreamsStudio IDE provides a deployment view that presents a hierarchical decomposition in terms of operators, processing elements, and computing nodes. Second, a runtime visualization tool called StreamSight [19] provides the live view of the deployment layout as well as the state of the application components.

1.4. Performance debugging

Performance debugging typically requires inspecting various middleware and application-specific performance metrics (such as communication rates, CPU utilization, etc.) associated with streams, operators, and nodes in the system. We provide the user with mechanisms to analyse each of these items. First, we provide a tool called perfviz for visualizing performance metrics, as well as associated runtime support APIs for creating custom performance metrics in user-defined operators. All operators in Spade have certain predefined performance metrics by default. Second, we provide special sink operators that can convert numerical attributes within streams into performance metrics. Third, the aforementioned StreamSight tool has various visualization modes for presenting live performance data, creating visualization metaphors that directly help developers and application analysts alike. Finally, the Spade compiler provides a profiling mode to capture operator-level processing time and port-level rate metrics through sampling for off-line processing.

This paper makes the following contributions: (1) it introduces a framework for debugging and understanding the behaviour of distributed data stream processing applications, including their semantics, runtime layout, and performance characteristics; (2) it proposes a novel debugging approach that integrates high-level, stream-centric and operator-based debugging with low-level, traditional source-code debugging; and (3) it describes the tools used to address the various challenges we have enumerated in this section that collectively form a basis for building novel debugging interfaces and environments.

Note that this paper focuses on fundamental debugging and application understanding concepts and infrastructure support for implementing them in the context of stream computing and not on cutting edge user-interfaces (UIs) or integrated development environments. We observe that many of the UI aspects will evolve with the maturity of System S as a stream processing platform.

The remainder of this paper is organized as follows. In Section 2 we give a brief overview of System S and Spade. We present a small-scale Spade application in Section 3. Semantic debugging, user-defined operator debugging, deployment debugging, and performance debugging are described in Sections 4, 5, 6, and 7, respectively. Section 8 discusses our experiences with real-world applications. We discuss the related work in Section 9 and conclude the paper in Section 10.

2. BACKGROUND

System S applications are composed of jobs that take the form of data-flow graphs. A data-flow graph is a set of operators inter-connected via streams. Each operator can have zero or more input ports, as well as zero or more output ports. An operator that lacks input ports is called a source and, similarly, an operator that lacks output ports is called a sink. Each output port creates a stream, which
carries tuples flowing towards the input ports that subscribe to that output port. An output port can publish to multiple input ports. Similarly, an input port can subscribe to multiple output ports, as long as the streams generated by these output ports have matching schemas. Cycles are allowed in data-flow graphs as long as the cycle forming edges are connected to control ports. Data-flow graphs can be deployed across the compute nodes of a System S cluster. The placement, scheduling, and other resource allocation decisions with respect to data-flow graphs are handled autonomically by the System S runtime, whereas they can also be influenced by the users through the parameters exposed at the Spade language level.

Spade is a rapid application development front-end for System S. It consists of a language, a compiler, and auxiliary support for building distributed stream processing applications to be deployed on System S. It provides three key functionalities:

- A language to compose parameterizable distributed stream processing applications, in the form of data-flow graphs. The Spade language provides a stream-centric, operator-level programming model. The operator logic can optionally be implemented in a lower-level language, like C++, whereas the Spade language is used to compose these operators into logical data-flow graphs. The Spade compiler is able to coalesce logical data-flow graphs into physical ones that are more appropriate for deployment on a given hardware configuration, from a performance point of view. This is achieved by fusing several operators and creating bigger ones that ‘fit’ in available compute nodes.

- A type-generic streaming operator model, which captures the fundamental concepts associated with streaming applications, such as windows on input streams, aggregation functions on windows, output attribute assignments, punctuation markers in streams, etc. Spade comes with a default toolkit of operators, called the relational toolkit, which provides common operators that implement relational algebra-like operations in a streaming context.

- Support for extending the language with new type-generic, highly configurable, and reusable operators. This enables third parties to create application or domain-specific toolkits of reusable operators.

3. A SAMPLE APPLICATION

In this section we describe a small Spade application that will be used throughout the paper for illustrating different debugging strategies.

Figure 1 shows an operator-view depiction of this sample application in StreamsStudio. The application, named Auction Processor, consists of four basic operators, which in concert provide
a simple emulation of an auction management platform. The application consists of two source operators, one join operator and one sink operator. One source operator (in the top left corner of the middle view of Figure 1), which has a single output port, creates a stream called AuctionBids. This stream is created by reading tuples from a file in comma-separated format. This source operator serves as an edge adapter, bringing in external data into the Spade application. In a deployment setting, the source operator could be configured to read from other streaming sources, such as sockets, RSS feeds, or message queues. The other source operator (in the top right corner of the middle view of Figure 1) is configured similarly, except that it reads from a different file and creates the stream ProductsAuctioned. AuctionBids and ProductsAuctioned streams have different schemas. AuctionBids contains tuples that represent bids and ProductsAuctioned contains tuples that represent products that are being auctioned. These two streams are fed into a third operator, a stream-relational join, matching the auctioned products and bids on these products. The join operator defines a window of size 30 tuples on its first input port and an empty window on its second input port. Effectively, for each auction tuple received, the join operator looks at the last 30 bids and outputs the ones that satisfy the join condition. The join condition is specified as matching product names and bid price being greater than or equal to the offer price in the auction. The join operator has a single output port, which generates the stream MatchingProducts. Finally, this stream is fed into a sink operator, which has a single input port. This sink operator is also an edge adapter. It is configured to write the incoming tuples representing the actual completed sale transactions to a file.

The Auction Processor is an artificially contrived application. Our aim is to have just about enough complexity to illustrate the debugging and application understanding techniques introduced in this paper. It contains one operator that has application logic embedded in it, which is the join operator.
Figure 2. A sample Spade application illustrating operators with multiple input/output ports as well as fan-in/fan-out.

In this case, debugging any problems that may exist in the join operator requires inspecting the tuples that enter and leave the operator through its input and output ports.

From a debugging perspective, two important aspects of the data-flow graphs in Spade are (i) the configurations of the operator instances and (ii) the relationships between the streams attached to the input and output ports of operators. Verifying the correctness of an operator often involves inspecting the contents and interactions of the tuples arriving and leaving its input and output ports, and making sure that they match the behaviour expected from the operator instance based on its configuration.

In the general case, Spade operators can have multiple input ports with arbitrary fan-in and multiple output ports with arbitrary fan-out. When an input port subscribes to more than one stream with the same schema (i.e. a fan-in topology), tuples from these streams are merged in arrival order and from the operator’s perspective the input streams are indistinguishable. To differentiate between them, one often needs to locate the output ports of the operators that generate these streams.

In Figure 2, we depict an operator with multiple input and output ports, as well as input ports with fan-ins and output ports with fan-outs. For demonstration purposes, we replaced the built-in join operator from the Auction Processor application with a user-defined join operator that has multiple input and output ports. We also inserted additional sinks and connections to illustrate fan-ins and fan-outs. We will now illustrate how a user could debug this application.

4. SEMANTIC DEBUGGING

A common paradigm used in testing the semantic correctness of a stream processing application is to replace the streaming versions of source and sink operators with file-based sources, rather than
the actual external sources (e.g. an RSS feed or a direct connection to a messaging platform, such as IBM WebSphere MQ). A small-scale sample workload is then stored in the source files, for which either an expected output is known or certain properties of the output file can be verified against the expected values. A file-based source then emulates a stream-based configuration by creating and forwarding stream tuples from line entries in the input file. In the case of a non-conforming result, there are several strategies to debug the application, with differing levels of detail. We look at these strategies in increasing order of complexity from the perspective of an application developer.

A common first step in problem determination is to assess the correctness of the data-flow graph. To help with this task, StreamsStudio offers a stream operator view (see Figure 2). This perspective provides a visualization of the data-flow graph, which includes operators, ports, and streams, as well as flyovers for additional information, such as subscriber information for output ports, publisher information for input ports, and schema information for streams. In order to scale up the process of composition verification to large graphs, we adjust the level of visualization detail based on the zoom level. As one zooms out, the level of detail is lowered to improve visibility, providing a global application perspective. Figure 3 shows the low, medium, and high detail views at high, medium, and low zoom levels, respectively. To further help with large-scale applications, StreamsStudio offers a stream search view (see Figure 1), which facilitates finding streams and operators using regular expression-based search patterns.

Once the composition of the flow graph is verified, the next step usually involves making sure that the operators are configured properly. For this purpose StreamsStudio offers a Spade source editor that is synchronized with the stream operator view. By selecting operators or streams in the stream operator view, one can automatically switch to the definitions of the respective operators in the source editor. One common inconvenience in working with large-scale programs is to view several related streams and operators at once, such as a pair of consumer and producer operators. In the source editor, this can be problematic due to operators being defined in distant locations in the source code. In the stream operator view, similar difficulties arise, due to operators being located far away from each other in the overall topology of the data-flow graph. To solve this problem, StreamsStudio offers a stream detail view (see Figure 4), which copresents the currently selected operator and the schemas of all the streams that are attached to its input or output ports. It can also optionally show consumer and producer operators around the currently selected operator, alleviating the need for manually scrolling around in the stream operator view.

Once the composition and configuration of operators in a data-flow graph are verified, one can look into the contents of the streams, inspecting the actual tuples they transport. This process usually starts from looking at the raw data ingested by the source edge adapter and progresses towards the sinks until an unexpected or incorrect intermediate result is located. Spade provides two ways to achieve this. First, one can employ temporary sink operators to store intermediate results in files, data bases, or other external sinks. The intermediate result can then be checked for correctness off-line. The other alternative is to mark the streams that one needs to examine as monitored in the Spade application source code, in which case the contents of the stream can be viewed live at runtime, using a command-line monitoring tool. By pausing the flow of data using the monitoring tool, one can create back-pressure that propagates all the way up to the source operators.\footnote{The back-pressure may eventually result in dropping tuples at the sources. Alternatively, the application designers can insert load shedding operators at any stage in the graph, by specifying a maximum buffer size. This could be taken one step further by employing load-aware operators\cite{24,25}.}
Although useful for small-scale debugging tasks, employing sinks and monitors become cumbersome for applications that require a higher level of interactivity during the debugging process. For such scenarios, Spade provides a stream debugger, which we discuss next.

4.1. Stand-alone applications and SDB

For detailed debugging of port-level interactions of an operator and its associated streams, Spade provides a stream application debugger, called sdb, which is often used in conjunction with
stand-alone applications. The Spade compiler is able to compile stream processing applications into stand-alone executables without any changes to the application source code. Employing this approach significantly improves the debugging process in two ways: (1) The turn-around time in a debugging session is significantly reduced due to the avoidance of bringing up a full-scale runtime middleware; and (2) It simplifies the semantic debugging of an application by shielding the developer from the intricacies of distributed execution.

The Spade compiler relies on code generation to instrument an application, in either stand-alone or distributed mode, so that it supports registration of debug hooks during execution. This enables the application to interact with and be controlled by a debugger. However, unless explicitly requested, such instrumentation points do not exist in the generated code, which considerably lowers the runtime overhead in the production version. In an application compiled with debugging support, a tuple hitting an input or output port can be intercepted using port signal objects that are a part of the generated instrumentation code. Such port signals are used for debugging as well as profiling purposes [23]. For instance, the debugger can request the installation of a debug hook associated with events on an operator port, in order to service a breakpoint request from the end user.

The basic functionality provided by sdb lies in providing an interface to the end user for employing probe points and, in this way, controlling the application’s flow of execution. This involves adding, removing, enabling, disabling, previewing, and listing probe points, in addition to saving and restoring them to and from configuration files. A probe point provides the following debugging

---

1Stand-alone applications are constrained to running on a single compute node and do not make use of any System S middleware services.
Figure 5. Probe points under fusion.

capabilities:

- **Breakpointing**: A breakpoint is defined on a port and is activated at any time a tuple hits the port. When a breakpoint gets activated, sdb provides an interface to examine and optionally modify the contents of the tuple, before letting it continue on its way. sdb also provides a special type of breakpoint, called *conditional* breakpoint. A conditional breakpoint is defined on a port and is configured via specifying a match condition on properties of the tuples flowing through the port. It is activated when a tuple that satisfies the match condition hits the port. Once activated, it behaves like a breakpoint.

- **Tracepointing**: A tracepoint is defined on a port and is configured via specifying a window size, which defines how much trace information should be kept around. Specifically, the tracepoint stores the most recent tuples that hit the port in its window and can save the contents of the window to a file or present them to the user upon request. Similar to breakpoints, tracepoints also have conditional variants. Conditional tracepoints only store tuples that satisfy the matching condition and ignore tuples that do not.

When compiled in stand-alone mode, all operators in an application are *fused* into a single process, with function calls implementing the stream connections between operators. Figure 5 illustrates the inner structure of a stand-alone application with debug hooks installed. As operators in Spade can potentially be multithreaded, a port can be called concurrently by more than one thread. Spade’s debugging framework ensures that only one probe point is active on a port at any time in order to preserve the original order of incoming requests. However, more than one probe point can be active as long as they are defined on different ports. As expected, sdb allows the users to list all the active breakpoints and continue the execution from that point on as desired.

### 4.2. A debugging session with SDB

We now look at a sample debugging session with sdb. For this purpose we employ the *Auction Processor* application from Section 3, with a small modification. We assume that the join operator...
is implemented as a user-defined operator in a general purpose programming language such as C++ as is the case in our example.\textsuperscript{11}

Assuming we have a sample workload for which we know what the expected results are, we demonstrate how to launch sdb to verify the correctness of these results. Figure 6 shows the steps involved, using the actual commands supported by sdb. First we set a tracepoint on the input port of the sink operator (t BIOP\_MatchingProductSink1 i 0). Then we let the application run (g). After the source file is depleted, we list the probe points (l) and show the list of tuples in our tracepoint window (s 0 t). Assume that we are expecting a result item whose productName attribute has the value ‘Laptop’, for which the bidPrice is equal to the matchingPrice. We note that this tuple is missing from the results. We save our probe point configuration (savecfg) and quit (q).

At this point, we have established that there is a problem in our application (i.e. incomplete results), and we have a lead that would help us to further debug this problem (i.e. a specific tuple that is missing). The next set of steps, shown in Figure 7, involves rerunning the application with additional breakpoints to further investigate the problem. We first load the probe point configuration from the previous run (l oadcfg). Then we list the set of operators (o) and define two breakpoints, one for each input port of the join operator that matches bids and auctions (b UDOP\_JoinBidAndAsk i *). Note that each probe point is assigned a unique id. For instance, the tracepoint that was loaded from the configuration file has id 0, whereas the two breakpoints we have just set are assigned ids 1 and 2. Finally, we let the application run (g).

As tuples hit the join operator’s input ports, our breakpoints are activated. We continue both breakpoints (c 1 and c 2) until we see the tuples that refer to the ‘Laptop’ item. Once we encounter both tuples, we copy them as shown in Figure 8 (using s 1 for showing breakpoint 1).

\textsuperscript{11}Note that the Spade language supports the creation of user-defined operators as a means for extending the language or for integrating legacy code as a part of new stream processing applications.
Now that we located the two tuples that were supposed to match, but did not in the previous run, we can modify the tuple values to test how the join operator reacts to various boundary conditions. For instance, we can check to see if the two tuples are not being matched because their bid and offer prices are equal, in which case there might be a bug in the match condition implementation of the user-defined join operator. Thus, we increment the bidPrice in the tuple from the first port by 1 (u1 bidPrice 901). Then we list our probe points (l), disable our breakpoints (d1 and d1), and continue all active breakpoints (c*) (see Figure 9).

Finally, after the source file is processed, we list our probe points (l) and show the tuples stored in our tracepoint’s window (s0t). We verify that the tuple representing a sale with productName value of ‘Laptop’ is in the results, as shown in Figure 10. This confirms the initial suspicion that there was a bug in the join operator’s matching condition implementation when the bidPrice and the offerPrice are exactly equal. In the next section, we continue debugging this example and show how source-code debugging can be employed to further pin down the exact location of this particular bug.
5. USER-DEFINED OPERATOR DEBUGGING

For user-defined operators written in a general programming language (C++ in our example***), semantic debugging via sdb is often not sufficient by itself, as it does not allow us to debug the internals of an operator. For debugging user-defined operator code, Spade provides two main methods: logging and tight gdb/sdb integration.

***Spade also supports user-defined operators written in Java.
5.1. Logging

Operator developers are encouraged to include logging directives in their source code and observe the logging output during runtime. Spade provides two main mechanisms to control the logging output. First, the developers can annotate their log messages with logging levels and use a logging verbosity knob to decide at which levels the log messages should be output during runtime. Moreover, the log messages can be further annotated with labels. A label defines a debugging aspect and multiple labels can be associated with a log message. A debugging aspect can be defined in terms of high-level concepts such as tuple transport operations or memory management operations. In this way, content-based filters can be defined on labels associated with particular debugging aspects and specific log messages can be filtered to the log output. For instance, filters can be defined to collect messages from only certain components, assuming that the log messages are annotated with labels based on the source-code component they originate from. Note that both Spade applications as well as System S components implement a common remote logging and tracing API, enabling a multiplicity of client tools to instantaneously inspect runtime log and trace messages.

System S provides three main ways in which one can examine the logs during the execution of an application. First, one can fetch a snapshot of the logging output into the local file system (the original logs are usually remote and collected in the node where an operator is running). Second, an application developer can selectively mark operators in the Spade language to be brought up in x11 terminals (as well as in gdb or sdb terminals), in which case the logging output will be redirected to the terminal. Finally, the System S runtime provides a monitoring tool (see Figure 11) that shows all the processing elements as well as middleware daemons that are currently running. This tool enables the user to examine live streaming log messages at the desired verbosity level as well as inspect middleware performance metrics associated with them.

For testing and debugging expressions without the need for actually running them in the context of a full-scale application, Spade allows users to define a function debug section in their programs. Expressions specified in the function debug section are evaluated at compile-time. This is often useful to test built-in and user-defined functions. This is similar to on-the-fly evaluation of expressions in the Java debugging in the Eclipse IDE. One common use case is testing regular expressions. Figure 12 provides a few examples.

5.2. GDB integration in SDB

As we have seen in Section 4, one sometimes needs to debug an operator’s source code. However, this need is not limited to user-defined operators, but also applies to operators that belong to third-party toolkits and the built-in relational toolkit. We address this need by extending sdb to support mixed-mode sdb/gdb sessions.

In an sdb session, breakpoints and conditional breakpoints can be used to intercept tuples of interest right before they are consumed by an input port or leave an output port. When the port of interest is an input port, the next step for a tuple is to go through the operator logic. When an user-defined operators are implemented by following a well-defined API. They inherit and extend a system-provided operator class that has virtual methods that must be implemented by the new operator subclass. This is how gdb is able to automatically locate the methods on which to define gdb breakpoints.
sdb breakpoint defined on an input port is activated, the user can decide to launch gdb and closely inspect the processing logic as it is carried out on that tuple. This action brings up a gdb window and will automatically insert a breakpoint into the function that handles the tuples from the port in question. From this point both gdb and sdb are used in concert. For instance, continuing the breakpoint in sdb will result in the corresponding breakpoint in gdb being activated. Similarly, gdb
can be launched with more than one defined breakpoint. Source operators can also be debugged with \texttt{gdb} despite their lacking of input ports. In this case the \texttt{gdb} breakpoint will be inserted into the function that handles the main loop of the source operator, which is in charge of interacting with external sources, transforming external raw data into stream tuples.

With \texttt{sdb/gdb} integration at our disposal, let us continue our debugging session from Section 4.2. Recall that using \texttt{sdb} alone, we came to suspect that the problem was in the user-defined join operator source code. Once we intercept the input tuples, which were unexpectedly not being matched, at the join operator's input ports, we can launch \texttt{gdb} to debug the operator code. Figure 13 shows the state after launching \texttt{gdb} on the second join port (\texttt{gdb 2}), assuming we already let the tuple from the first port get into the join operator. This means that the tuple is staying in the join window. Once \texttt{gdb} is brought up, we unblock the execution by issuing a \texttt{continue} command in \texttt{sdb} (\texttt{c 2}). This results in \texttt{gdb} activating the breakpoint defined on function <\texttt{oper-name}>::\texttt{processInput<port-index>}. From Figure 13 one can get the gist of the operator code that handles an input tuple. At this point all the functionalities in \texttt{gdb} are available to the developer. In this example, it is easy to see that the join condition has a small bug: instead of using the \texttt{greater than} (\texttt{>} operation, it should instead have been using the \texttt{greater than or equal to} (\texttt{>=}) operation in the join expression \texttt{bid.get_bidPrice() > ask.get_offerPrice()}.  

We are currently working on debugging of processing windows in user-defined operators. Recall that a window defines a group of tuples to be processed together. This involves developing a common
windowing library framework that will enable us to insert debugging hooks at well-known locations in the windowing code, as opposed to trying to reason about operator-specific logic.

6. DEPLOYMENT DEBUGGING

Various problems may arise if the Spade applications are not properly laid out on the System S runtime. Some of these issues are related to application correctness such as unintended or missing stream connections. Others are related to suboptimal resource utilization and allocation such as violated resource constraints, degraded performance, and wide fluctuations in application throughput.

The Spade language provides two major abstractions that help to manage the deployment of stream processing applications, namely partitions and nodes. Note that these same abstractions form the basis for our automatic optimization framework [23]. A partition defines a subset of the data-flow graph that is to be executed in the context of the same processing element (PE). In other words, a partition is a set of operators that get compiled into a PE. The PE is the basic execution container in System S. PEs can be assigned to nodes at compile-time, using the Spade language’s placement directives, or at runtime using the System S optimizing scheduler [22] (other such optimizers exist in the literature [26]). An important distinction between operator-to-PE assignment and PE-to-node assignment is that the latter can change at runtime, whereas the former is fixed at compile-time. The runtime PE relocations are usually triggered by resource re-allocations or node failures.

As operator partitions are created at compile-time and do not change during runtime, the stream operator view provides a layout to examine the operator-to-PE assignments and spot any errors that might exist as early as possible. Figure 14 shows a layout that groups operators based on their partitions. Even though there is no requirement that the operators within the same partition should be connected, in most practical scenarios partitions represent a set of connected operators. Being able to see the operator partitions is especially important for developers who chose to hand-optimize the operator graph partitioning. As we discussed earlier, the Spade compiler can optionally create operator partitions automatically, using a profiling-driven optimization scheme.

StreamSight’s runtime view of stream processing applications complements the compile-time views of StreamsStudio. This includes assignment of PEs to nodes, which cannot be completely resolved at compile-time in the most general case as developers might elect to leave such decisions to the runtime optimizer. This can be the case in large configurations where the runtime optimizer is aiming at globally optimizing the resource allocation across multiple applications sharing the runtime environment. For instance, Figures 15 and 16 show PE and node layouts of an application in StreamSight, respectively.

StreamSight views are live. One can observe applications being brought up and torn down. Failures of PEs as well as nodes are reflected in these live views. Note that a failure of a PE indicates failures of all the operators fused within that PE, as they are running in the same process space.

‡‡‡Please see the Spade language reference for further details on windowing semantics [17].
Figure 14. A top-to-bottom StreamsStudio PE/operator view.
7. PERFORMANCE DEBUGGING

System S targets high-throughput and low-latency distributed stream processing applications, where performance plays a critical role in testing and debugging. Reaching the desired performance level not only depends on the efficiency of the operator-level analytic implementations, but also on the operator composition, flow-graph partitioning, and PE deployment among other dynamic factors such as runtime resource adaptation, variability in incoming workloads, and global optimization directives issued by the runtime optimizer. Debugging performance problems involves locating performance bottlenecks in the flow graph by looking at various performance indicators, such as throughput and latency metrics.
System S provides a common performance metric API to define metrics and associated tools to view them during runtime. A performance metric is simply a name–value pair, where the value has a numeric type. A set of related performance metrics is called a performance metric collection. Such collections can be associated with any entity in the underlying middleware (e.g. performance metrics of the System S job management and routing component) as well as with different application components, such as an operator, PE, operator port, PE port, etc. Each operator and PE generated by Spade has predefined performance metric collections. These include common metrics such as number and size of tuples received and transmitted at the port, operator, and PE level. Developers of user-defined operators are free to create new performance metrics of their own that capture performance critical aspects in these operators.

The Spade language offers additional ways to create performance metrics, without the need for going into user-defined operator code. This is achieved by special type of edge adapters, called perfmetric sinks. These sink operators create a performance metric collection out of the stream they receive and convert the numerical attributes in the stream into performance metric readings. Each time a tuple is received, these performance metrics are updated with the latest value of their associated stream attribute. This enables developers to use the Spade language for manipulating streams carrying performance-related data and export the results through the performance metric interface, which makes it possible to use the visualization tools provided by System S. A popular use case is latency measurements. Time-stamping mechanisms can be used to annotate tuples with latency information, which might be exported through perfmetric sinks and visualized accordingly.

System S provides a command-line text-based tool called perfclient as well as a visualization tool called perfviz for viewing performance metrics during runtime. Both tools allow the performance metric values to be collected in the form of continuous time-series data. perfviz presents such time-series data as sliding line charts with time on the x-axis and the actual metric readings on the y-axis. Figure 17 shows an example where tuples received and transmitted by a PE, both in terms of number (tuples/sec) and size (bytes s\(^{-1}\)), are visualized as live charts.

Third, the StreamSight tool has various visualization cues for presenting live performance data. This includes dynamic colouring of PEs and ports based on stream rates and CPU utilization, tooltips summarizing performance data for PEs and ports, and dynamic sizing of stream connections based on the data-flow rates [19,27] by controlling the thickness of lines representing these connections. Figure 18 illustrates some of these features.

Colouring of PEs based on their CPU utilization is often useful in locating the performance bottlenecks. A PE turning red might indicate a CPU bottleneck. This can be due to several reasons. For instance, it can be due to too many PEs placed on the same node, in which case looking at the node placement view may help to locate the problem. Alternatively, it can be due to assigning too many operators to a single PE, and switching to compile-time partitioning view will reveal the operator topology within the PE, helping to locate the issue. Yet another reason might be that one or more CPU-heavy operators that implement complex streaming analytics are competing for the same limited CPU resources. Debugging the latter requires accessing operator-level CPU utilization metrics. However, operators can be running in the context of one process and in certain cases in the context of the same thread, and thus it is often difficult to extract CPU utilization information at the operator granularity, with little overhead. For this purpose the Spade compiler provides a profiling mode.

Spade’s profiling framework [23] consists of three main components, namely code instrumentation, statistics collection, and statistics refinement. Code instrumentation is used at compile-time...
Figure 17. perfviz showing PE-level metrics.

Figure 18. Performance-related visual cues in StreamSight.
to inject profiling instructions into the generated processing elements. Statistics collection is the runtime process of executing the instrumented operator logic to collect raw statistics regarding the communication and computation characteristics of operators. Statistics refinement involves post-processing these raw statistics to create refined statistics that are suitable for consumption by the end user. These include CPU usage information on a per-operator basis, average tuple rate and average tuple size on a per-operator port basis.

8. EXPERIENCES

In this section we describe some of our experiences in using the tools described so far in multiple real-world stream processing applications. We look at applications of different scales, namely: small-scale, medium-scale, and large-scale applications.

8.1. Small-scale applications

We characterize small-scale applications as flow graphs with 1–10 operators that can be efficiently run on a single node. For instance, an application that detects weather alerts based on raw readings from a weather station, or an application that computes summary statistics over a stream of transactions qualifies as a small-scale application. Furthermore, many streaming operators can be tested in isolation by augmenting them with simple source and sink operators. Such operator unit tests can also be considered as small-scale applications and are quite common in a developer’s workflow.

Semantic and source-code debugging are more prevalent in small-scale applications, whereas deployment and performance debugging are often not needed. In terms of semantic debugging, small-scale applications usually employ sinks and monitors to verify their results, sometimes based on auto-generated workloads (using built-in operators for workload generation or compiler-driven workload generation, which is also supported by the Spade compiler). One important requirement for small-scale applications is quick turn-around time for the debug cycle. For this reason, compiling stand-alone applications is very popular for small-scale applications. Owing to the simple nature of the flow graphs in such applications, usually gdb alone on the stand-alone executable is sufficient for source-code debugging.

Note that, despite the apparent simplicity of these small-scale applications, their debugging flows are extremely important as a stepping stone. Often developers make use of dynamic application composition as a means to develop complex applications out of smaller applications that are used as building blocks [27].

8.2. Medium-scale applications

We characterize medium-scale applications as flow graphs with up to 100 operators that can be effectively run on up to 10 nodes. An example of this class of applications is depicted in Figure 19.

\[\text{§§§ Dynamic application composition is the capability provided by System S to integrate several applications at runtime based on the support available to applications for importing and exporting streams.}\]
This is a radio-astronomy application, being developed as part of the Australian Square Kilometre Array Pathfinder Project (ASKAP) [28,29]. The application performs synthesis radio imaging—the process of transforming radio data collected from an array of antennas into a visible image.

Semantic debugging is often performed with the use of the StreamsStudio to verify application composition and sdb in distributed mode to perform stream-level verification. An example of the latter is to set a trace point to catch a condition under which a convolution operator receives an incorrect image block index from the channel-blocking operator.

In medium-scale applications, user-defined operator debugging is often performed using the logging and tracing features, supplemented by running sdb on operators that are identified as potential sources of defects. Being able to fetch and filter logs from multiple nodes helps to identifying erroneous conditions. An example is monitoring parts of all the logs that relate to the convolution functionality (using debug aspects) and are tagged as being at the error level. Once particular problems are identified, sdb’s support for launching gdb is typically used to perform further source-code debugging as reported by several application developers.

StreamSight serves as an excellent tool for deployment debugging as well as performance debugging for medium-scale applications, being able to see placement of PEs on hosts, and operators in PEs work well at this scale. For example, StreamSight’s host view can be used to verify that the data-parallel convolution operators are located on different hosts. The PE colouring and connection size hints, as well as tooltips used for metric reporting are very effective in identifying bottlenecks in the graph. These system-supplied metrics are usually complemented by application-specific performance metrics. perfviz is then used to track both kinds of metrics live. An example of the latter is the average time it takes to perform convolution on an image block, which can be used as an application-level metric to detect imbalance across data-parallel sections of the convolution computation.
Next, we look at how these techniques scale, and our ongoing work towards improving support for debugging and monitoring large-scale applications.

8.3. Large-scale applications

We characterize large-scale applications as flow graphs with 100–1000s of operators that can be effectively run on 10–100s or more nodes. One particular example that we have experience with is an automated trading application running on an IBM Blue Gene supercomputer, employing 512 nodes. A PE view of this application is depicted in Figure 20.

Verifying the flow composition becomes a really challenging task for large-scale applications, as the global view of the flow graph is often hard to work with, as shown in Figure 20. Zooming in provides only limited help, as the layout of the graph may not exactly cluster functionally similar or logically related parts of the application together. There are various techniques we use to cope with this kind of complexity. Some of these techniques are already in place, whereas some others are our ongoing work. We first discuss the existing techniques.

First, Spade provides flow composition constructs such as for loops to make it easier to replicate structured flows in a programmatic way. Furthermore, StreamsStudio provides an outline view of a Spade application, where the control flow constructs are not fully expanded, unlike the StreamSight view. For instance, by selecting a for loop in the outline view, one can highlight all the quoters in the automated trading application.

Second, as mentioned earlier, the System S runtime and the Spade language provide a dynamic composition feature, through which large applications can be partitioned into smaller applications and deployed incrementally\footnote{There is more into dynamic composition than just incremental deployment. In particular, it supports dynamic composition of applications through runtime discovery of cross-application streams and application-controlled rerouting of stream connections.}. For example, the automated trading application can be partitioned...
into subapplications such as data ingestion, data routing, order book management, quoters, market actuators, etc.

Finally, we are currently working on the second version of the Spade language [17] and its associated tooling. One of the most important language features is the composite operator. A composite operator can contain a flow graph comprising multiple operators used in conjunction to implement a more complex processing logic. When a composite operator is instantiated, it looks like a regular primitive operator. This construct will naturally help to create a hierarchic organization for large-scale applications. For instance, at the outermost layer, the automated trading application may look as simple as a flow graph with five operators, yet one can drill down further to discover additional structure.

Our experience with the automated trading application also identified challenges in the systems aspects of monitoring and debugging large-scale streaming applications. In particular, the performance of the visualization tool as well as the application server component of the middleware it interacted with had scalability issues. This resulted in architectural changes in the implementation, where the topology was incrementally computed and notification interfaces were implemented to ship changes to the visualizer only for the parts of the graph it is currently visualizing.

Large-scale applications also create challenges in performance debugging. In such situations, incremental application deployment come to the rescue. By nature, streaming applications are composed of stages, where after each stage the rate of the data is reduced by performing progressively more complex processing [30]. A common performance debugging strategy is to measure the performance by incrementally adding more stages to the computation. Another interesting observation we made is that developers of large-scale applications not only rely on custom performance metrics for performance measurement, but also develop their own subapplications for deriving detailed performance measurements. For instance, the automated trading application has a latency measurement subapplication that contains several custom operators used to produce refined latency statistics, which are eventually converted to application-specific performance metrics.

The ability to create user-defined partitions as well as explicit placement of partitions (PEs) on hosts has been critical features for achieving good performance in the automated trading application we have deployed. This is quite contrary to system-supported query optimization techniques that are at the heart of relational database systems. Query optimization techniques work reasonably well given the predominant use of relational operators in queries, for which the database system has detailed knowledge about. In our experience, stream processing applications contain large number of user-defined operators with complex nature (multithreaded operators, operators that interact with the system, etc.). Furthermore, there are complex requirements that are difficult to convey to a monolithic optimizer. For instance, on an IBM Blue Gene supercomputer each compute node has limited processing capabilities and reducing all the overheads to a minimum is key (thus we employed a single processing element with multiple operators fused together), the fan-in/fan-out is limited due to the buffer size requirements of the low-latency BGSM transport, certain operators need to be placed on I/O nodes that have connection to the outside sources (as opposed to on compute nodes that do not have such access), certain operator threads need to be pinned down to certain cores to avoid placement of operators on cores that are handling hardware interrupts,

---

The visualizer was run outside of the IBM Blue Gene supercomputer on a host with more processing capability than a Blue Gene compute node.
etc. We have not yet developed an optimization framework that is flexible enough to address such constraints. In the automated trading application, manual optimization was performed through a user-defined partitioning and placement module, which uses statistics from the earlier day of trading and relies on Spade’s placement and partitioning constructs to recompile the application overnight for the next day. The user-defined optimizer takes all the architecture and application-specific constraints into account. Nevertheless, we are not ignoring the automatic optimization problem. In fact, we provide a profile-driven operator partitioning framework at compile-time [23] and a host placement optimizer at runtime [22]. We have also worked on auto-parallelization techniques for fine-grain data-parallel operators [29].

9. RELATED WORK

There has been much work on data stream processing systems, both in academia (Borealis [1], STREAM [2], Aurora [3], TelegraphCQ [4], XStream [5], StreamIt [6], to name a few) and in industry (Coral8 [7], Gigascope [8], StreamBase [10], and many others). For a comparison of such systems with System S and Spade, we refer the reader to our earlier work [16,20]. The literature on these systems does not address the important issue of debugging distributed stream processing applications. StreamBase, a commercial tool that is an offshoot of the Aurora and Borealis projects, provides an Eclipse plug-in where application development, deployment, and debugging can be undertaken. Their approach is to provide developers with a canvas where operators can be dragged and dropped and configured, and connections between operators can be established. Support for controllable data playback is also provided. A similar approach is also employed by iFlow [31] and StreamIt’s StreamIt Graph Editor (SGE) [32]. However, these systems do not provide tools for debugging distributed stream processing applications. To our knowledge, none of the existing DSPSs provide comprehensive debugging support across the four dimensions we have identified in this paper.

In the distributed computing middleware area, the research community has created many visualization tools for locating performance problems. For applications built using the MPI framework, two examples are VAMPIR [33] and PARvis [34]. For applications built using the PVM paradigm, an early effort was Poet [35]. In the areas of performance and application understanding, tools such as TAU [36], PVaniM [37], and Projections [38] aim at providing performance data gathering and visualization infrastructure for distributed and parallel programs (TAU for MPI and PVM, PVaniM for PVM, and Projections for Charm++ [39]). None of these tools are geared towards streaming applications directly. Indeed, stream processing applications represent a new data processing paradigm and better metaphors and tools can be employed for the purpose of performance debugging and application understanding as we have demonstrated in this paper.

The literature contains strategies for low-cost debugging of distributed applications [40] as well as several full-featured interactive debuggers for parallel systems, such as Mantis [41], TotalView [42], and p2d2 [43]. All support the isolation of specific processes, as well as attaching a sequential debugger to remote application instances, which enables breakpoints to be set in individual processes. The literature also includes several tools for debugging, monitoring, and visualizing MPI [44–46], PVM [46–48], and Charm++ [49] applications. Tools that support stream-level debugging of applications via conditional breakpoints (such as LabView [50]), as well as source code-level debugging of flows via gdb (as in [49,51]) exist in the literature. These are similar in nature to our
sdb debugger. However, none of these works have provided a complete framework for debugging and monitoring distributed stream processing applications.

10. CONCLUSIONS

The proliferation of fast-changing online information sources, coupled with the availability of large-scale affordable computational infrastructure, is making it possible to develop large, real-world streaming processing applications. Developing these applications and verifying their correctness as well as having the tooling for understanding their performance characteristics are important parts of the development cycle. In this paper, we have introduced tools and language support for debugging distributed stream processing applications. We identified four different areas that require unique tools and strategies for locating problems, namely semantic debugging, user-defined operator debugging, deployment debugging, and performance debugging. For each area we developed associated tools and language support for easing the burden on application developers.

We believe that we have made significant contributions in the domain of tooling and debugging support for large-scale streaming applications by employing a concerted and integrated effort along the main dimensions that affect the programmer’s productivity. To the best of our knowledge, most efforts in this arena are still in a very early stage. On the other hand, the tooling we have described in this paper has been used in developing large, complex applications in domains spanning manufacturing [52], scientific data processing [29], financial applications [30], fraud detection [53], among others. These applications go from medium size running on a handful of compute nodes to thousands of operators running on hundreds of compute nodes, including nodes in regular cluster of workstations to large collections of compute nodes on the IBM Blue Gene supercomputer.

REFERENCES


Published in 2009 by John Wiley & Sons, Ltd.

todo


