Model-driven Performance Analysis and Deployment Planning for Real-time Stream Processing

Kyoungho An and Aniruddha Gokhale
ISIS, Dept. of EECS, Vanderbilt University, Nashville, TN 37235, USA
Email: {kyoungho.an, a.gokhale}@vanderbilt.edu

***Abstract*—Real-time stream processing in the cloud is gaining significant attention for its ability to mine massive amounts of data for a variety of applications, such as in reconnaissance missions or search-and-rescue operations. In cloud-based real-time streaming applications, dynamic resource management mechanisms are needed to support the real-time requirements of these applications. However, for any dynamic resource management technique to work, there is first a need to understand the controllable properties (or parameters) of the stream processing applications. Pinpointing these properties and separating them from the application-specific properties that cannot be controlled is hard and requires a scientific approach to obtain these insights. This paper presents a model-driven performance analysis approach for real-time streaming applications to pinpoint their controllable properties. The same modeling framework then makes deployment planning decisions, which is one dimension of dynamic resource management. The presented research is part of our larger effort towards a holistic framework to support real-time and dependable cloud-based applications.**

***Index Terms*—model-based performance analysis and deployment, real-time data processing, cloud computing.**

I. Introduction

Recent trends indicate an increased demand for real-time stream processing in the cloud involving massive amounts of continuous streams of data. For example, in military-based reconnaissance missions or in search-and-rescue operations, there is a need for real-time processing of massive amounts of continuous, incoming streams of data to identify specific enemy targets or survivors, respectively. A number of stream processing platforms have been developed for distributed, real-time stream processing, such as Storm [1], S4 [2], and Flume [3]. The design of these platforms is inspired by Hadoop to process large-scale data sets, however, they are developed to accomplish real-time stream processing unlike batch processing in Hadoop.

Meeting the real-time requirements of the stream processing tasks requires an assurance of predictable, end-to-end execution times from the infrastructure that hosts the different tasks comprising the distributed stream processing activity, which in turn requires effective dynamic resource management. Despite the availability of sophisticated stream processing frameworks, such as Storm, which aim to process data streams in real-time by parallelizing the processing, assuring such bounds through effective dynamic resource management is hard for a variety of reasons. First, the task execution times depend on the input data size, capacity of physical machines that host these tasks, and the number of threads used in concurrent processing. Additionally, queuing delays in the network that hosts the distributed, communicating tasks are not predictable causing further difficulty in bounding the end-to-end execution times. Finally, a lack of effective mapping (*i.e.*, deployment or placement) for allocating the stream processing framework components to the compute resources (*e.g.*, virtual machines in the cloud) gives rise to additional unpredictable performance bottlenecks in processing the streams. All these factors degrade latencies and throughput of the systems in unpredictable ways, and makes it hard to design effective dynamic resource management mechanisms.

More often than ever these problems are handled by a trial-and-error approach. However, such an approach is inefficient, non-scientific, not reusable, and does not provide a dependable way to meeting the end-to-end real-time properties of the applications. For effective dynamic resource management, it is important to understand in what way can the application be dynamically controlled so that the real-time properties of the application can be met. This elicits the need to pinpoint and separate the dynamically controllable properties of these applications from the non-controllable properties. Non controllable properties are those that are imposed by the application logic, which in turn dictates a specific structure to the way stream processing blocks are composed. However, many other factors, such as deployment decisions allocating stream processing blocks to physical hosts and in turn the virtual machines, tuning the infrastructure that hosts these processing blocks, and configuring the network paths are all controllable properties.

An ad hoc approach to pinpointing the controllable properties is not a dependable solution. We surmise that a scientific approach based on model-driven performance analysis and deployment planning (shown in Figure 1) for distributed real-time stream processing may provide the desired solution to better understand these performance-related issues so that subsequently effective dynamic resource management mechanisms can be designed based on the insights gained. Model-based performance analysis has hitherto been applied in many domains. In our prior work [4], we applied model-driven engineering (MDE) [5] for performance analysis of reconfigurable conveyor systems in the context of variability in the physical layouts. Moreno, et al. [6] describe a general approach for

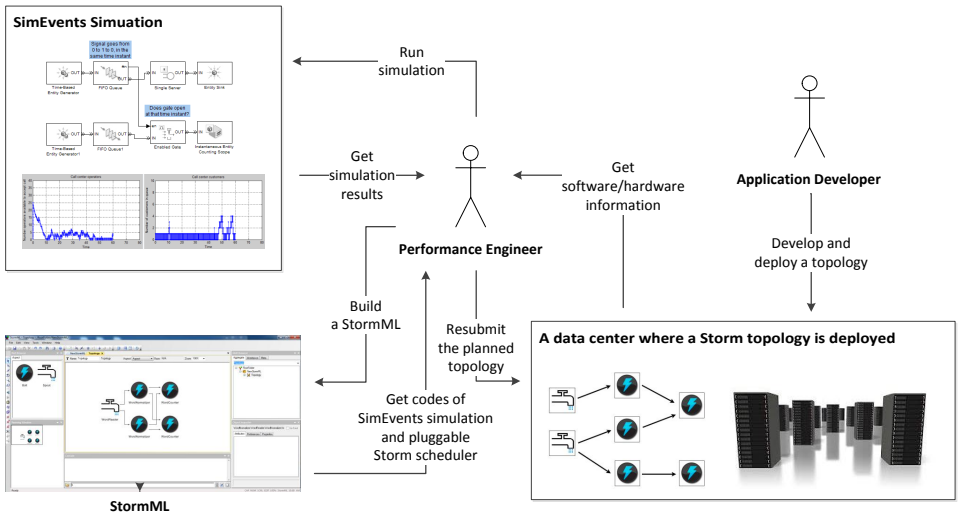

Fig. 1. Model-based Process of Performance Analysis and Deployment Planning

exploiting MDE for software performance analysis. MDE is not only applied in software analysis, but also deployment and configuration. In our prior work [7], [8], we showed how model-driven engineering is used to deploy and configure distributed real-time and embedded (DRE) applications.

For this paper we have focused on the Storm [1] stream processing framework. Our approach is based on profiling several different Storm-based stream applications with different topologies, where a topology in Storm parlance is the structural composition of Storm processing blocks each of which performs some stream processing task. In this context, our MDE-based solution comprises the following artifacts: first, a domain-specific modeling language (DSML) [9] defined for Storm provides intuitive abstractions to performance engineers to run simulations and to deploy their proposed topologies. Second, using the models defined in the DSML, performance details such as bottleneck points, throughput, and latency of each software component, and end-to-end latency of streams are analyzed by automating the execution of a discrete event simulation and obtaining the insights. Third, generative capabilities of the DSML are used to automate the overall analysis and the deployment process.

The rest of the paper is organized as follows: Section II describes our model-based performance analysis approach; and Section III offers concluding remarks alluding to future work.

II. Model-based Analysis for Storm

This section describes our model-based process for analyzing the performance bottlenecks in real-time stream processing applications implemented in Storm.

A. Background of Storm

For our research we have used the Storm stream processing framework. Applications in Storm use two specific building blocks or Executors called Spouts – which are the source of data streams, and Bolts – which process the data streams, may perform operations such as filtering and join, and may produce other streams. Spouts and Bolts can be composed in various configurations to form Storm application topologies. The connections between Spouts and Bolts can be defined based on grouping strategies. A topology can be arbitrarily complex. These logical abstractions must be deployed on hardware resources called a Storm cluster, which is made up of two kinds of hardware nodes: Nimbus (master node) and Supervisor (worker node). Nimbus is responsible for distributing code as well as assigning tasks to Supervisors. Supervisors execute software components of a topology.

Because performance of a Topology can be affected by hardware specifications, each Supervisor contains its hardware specifications as attributes such as the number of CPUs, memory size, and network bandwidth. Each Supervisor includes Slots where worker processes execute. Each Slot is differentiated by its port number. A worker process executes a subset of a specific topology and runs one or more executors.

B. StormML: MDE Framework for Storm

Figure 1 shows the overall process of performance bottleneck analysis and deployment planning for Storm applications using our MDE framework called StormML. First, an application developer develops a Storm-based application, which is then deployed in a Storm cluster by the Storm's default scheduler. The Storm's default scheduler uniformly uses resources by ordering supervisors in terms of available slots. Note that this default deployment may not necessarily provide the best performance. Next, to conduct performance analysis, a Storm model is built using our DSML using data from performance profiling of the test execution. Once the Storm model is built, a Simulink SimEvents model from the

original model is generated by the GME interpreter to run discrete event simulations. Once the SimEvents simulation completes, performance engineers can identify the software components that are bottlenecks, and overall application's throughput and end-to-end latency. Based on the simulation result, the performance engineers can suggest a new deployment plan to improve performance and run another simulation. Through an iterative process, if an optimal deployment plan is determined, a Storm topology is resubmitted to a cluster to run.

The Generic Modeling Environment (GME) [10] is used to develop the DSML named StormML and generative capabilities for StormML. Figure 2 illustrates the StormML meta-model, which is at the heart of the DSML. The StormML meta-model consists of the first class concepts of Storm cluster for hardware components and Storm Topology for software components. For analyzing a topology, we have defined profiler performance metrics such as the number of tasks, average execution time, average input size, average input rate are defined as attributes in the Executor model. In the Connection models, a grouping is defined as an attribute because Storm provides several grouping mechanisms for routing output streams differently.

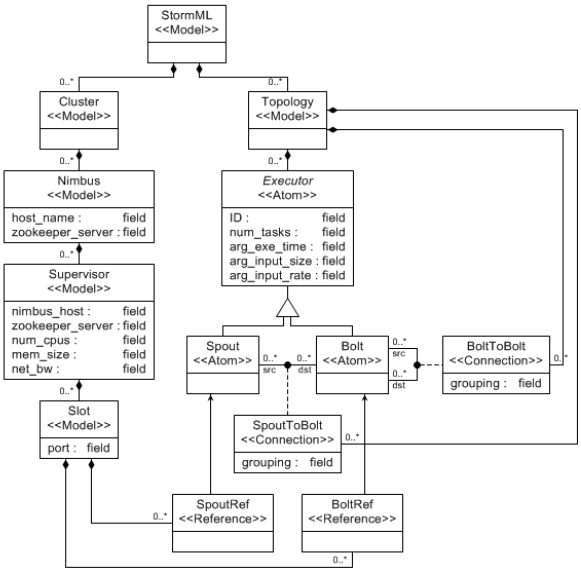

Fig. 2. Meta-model of StormML

To find an optimal deployment, the StormML provides modeling components describing how software is mapped to hardware. We used the GME Reference feature to refer entities defined in one model to be referred in another. Spout and Bolt-based software topologies have reference components, and the reference components are contained in Slots under Supervisors as it is referred as working slots. Our StormML runs a Storm topology which is deployed by Storm's default scheduler and retrieves information defined in the meta-model.

Figure 3 shows a model of a topology defined using StormML for a canonical example of word counting that executes in a Storm cluster. In the example, input streams flow into the topology via a Spout named WordReader. Next, output streams of the Spout flow in two Bolts named WordNormalizers, where sentence streams are split into words. A Shuffle grouping (*i.e.* which is a routing strategy) is used for the stream to balance the stream into multiple Bolts. After the word normalizing process, output streams of each WordNormalizer are sent to the next Bolts named WordCounters, which finally count the words.

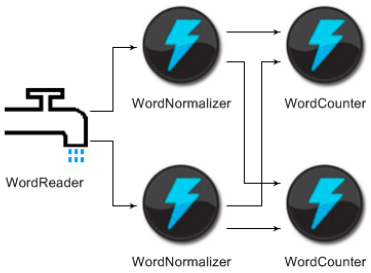

Fig. 3. StormML for Word Count

Figure 4 illustrates the generated SimEvents model for the Word Count example. In the generated model, the WordReader generates a stream periodically by a defined interval. OutputSwitch distributes a stream into two input queues of WordNormalizers in a round robin fashion. Each Bolt has its own FIFO queue. WordNormalizers contain OutputSwitch because processed tuples should be sent to multiple Bolts. In our model, OutputSwitch also distributes tuples in a round robin fashion. Such a distribution may be changed in the SimEvents model depending on what is the grouping strategy used (*e.g.*, Shuffle or Field are one of many groupings provided by Storm that defined how streams are routed in a topology). The total throughput and end-to-end latency can be computed using timer components: StartTimer and ReadTimer. Moreover, to find bottleneck Bolts, server and queue components offer statistical data.

III. Concluding Remarks

The paper described a model-driven tool for analyzing and deploying real-time stream processing applications so that performance bottlenecks can be pinpointed, and subsequently dynamic resource management solutions can be defined. In the current state, the overall process and meta-model of the tool has been developed. Our ongoing work is implementing GME interpreters to generate SimEvents models from StormML and implementations of Storm's pluggable schedulers from StormML. Currently, StormML and SimEvents models are manually created. If interpreters are completed, hardware and software models for StormML are automatically generated by the interpreters from results of test operations. Moreover, the interpreters will transform the StormML into a SimEvents

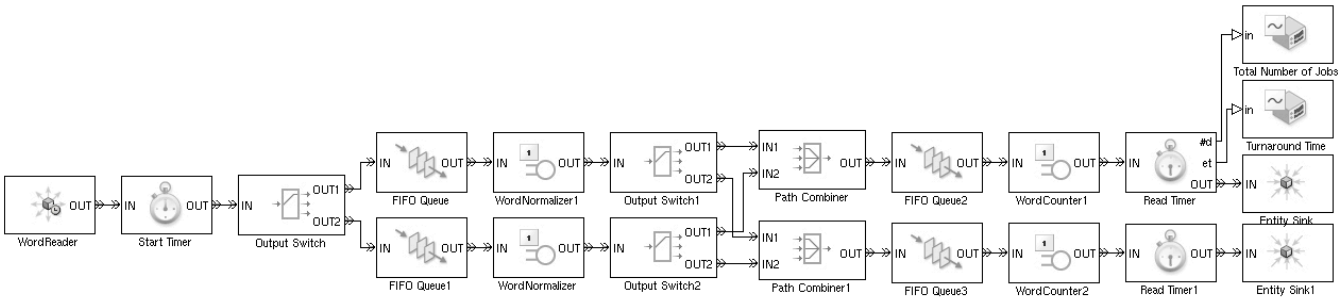

Fig. 4. SimEvents Model for Word Count

model to run discrete event simulation. As a result, through performance evaluation by simulations, an optimal deployment plan is decided by performance engineers and a Java implementation of a pluggable scheduler is made based on the determined deployment plan.

Once the GME interpreters are implemented, we need to refine generated SimEvents models to make it more realistic to actual running Storm applications. In Storm, there are various groupings in Storm such as All grouping, Global grouping, Fields grouping, and Shuffle grouping. In the word count example, only Fields grouping and Shuffle grouping are used, and our current model does not consider that the number of tuples sent to multiple executors is different because of different word frequency. Statistical data of input sizes and input arrival rate for each Bolts should be collected from test operations and applied as parameters in simulation models to refine simulation results.

Moreover, we would like to improve our modeling application to automatically find out an optimal deployment plan for users instead of manual analysis. There have been various research conducted in deployment optimization problem with diverse techniques like genetic algorithms and constraint satisfaction problems (CSP) [11], [12]. In our prior work, we applied a hybrid algorithm that combines worst-fit bin packing with evolutionary algorithms (genetic and particle swarm optimization) for maximizing service uptime of smartphone-based DRE systems [13]. In the work, we extended a framework for spatial deployment algorithm called ScatterD [14]. Likewise, we can extend and apply existing solving techniques and frameworks to find an optimal hardware/software mapping for real-time stream processing.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation NSF SHF/CNS Award CNS 0915976 and NSF CAREER CNS 0845789. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] "Storm," https://github.com/nathanmarz/storm/wiki.
[2] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*. IEEE, 2010, pp. 170–177.
[3] "Apache Flume," http://flume.apache.org.
[4] K. An, A. Trewyn, A. Gokhale, and S. Sastry, "Model-driven Performance Analysis of Reconfigurable Conveyor Systems used in Material Handling Applications," in *Second IEEE/ACM International Conference on Cyber Physical Systems (ICCPS 2011)*. Chicago, IL, USA: IEEE, Apr. 2011, pp. 141–150.
[5] D. C. Schmidt, "Model-Driven Engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006.
[6] G. Moreno and P. Merson, "Model-driven performance analysis," *Quality of Software Architectures. Models and Architectures*, pp. 135–151, 2008.
[7] A. Gokhale, B. Natarajan, D. C. Schmidt, A. Nechypurenko, J. Gray, N. Wang, S. Neema, T. Bapty, and J. Parsons, "CoSMIC: An MDA Generative Tool for Distributed Real-time and Embedded Component Middleware and Applications," in *Proceedings of the OOPSLA 2002 Workshop on Generative Techniques in the Context of Model Driven Architecture*. Seattle, WA: ACM, Nov. 2002.
[8] T. Lu, E. Turkay, A. Gokhale, and D. C. Schmidt, "CoSMIC: An MDA Tool suite for Application Deployment and Configuration," in *Proceedings of the OOPSLA 2003 Workshop on Generative Techniques in the Context of Model Driven Architecture*. Anaheim, CA: ACM, Oct. 2003.
[9] M. Mernik, J. Heering, and A. M. Sloane, "When and How to Develop Domain-specific Languages," *ACM Computing Surveys*, vol. 37, no. 4, pp. 316–344, 2005.
[10] Á. Lédeczi, Á. Bakay, M. Maróti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai, "Composing Domain-Specific Design Environments," *Computer*, vol. 34, no. 11, pp. 44–51, 2001.
[11] D. Saha, R. Mitra, and A. Basu, "Hardware software partitioning using genetic algorithm," in *VLSI Design, 1997. Proceedings., Tenth International Conference on*. IEEE, 1997, pp. 155–160.
[12] Y. Vanrompay, P. Rigole, and Y. Berbers, "Genetic algorithm-based optimization of service composition and deployment," in *Proceedings of the 3rd international workshop on Services integration in pervasive environments*. ACM, 2008, pp. 13–18.
[13] A. Shah, K. An, A. Gokhale, and J. White, "Maximizing service uptime of smartphone-based distributed real-time and embedded systems," in *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2011 14th IEEE International Symposium on*. IEEE, 2011, pp. 3–10.
[14] J. White, B. Dougherty, C. Thompson, and D. C. Schmidt, "Scatterd: Spatial deployment optimization with hybrid heuristic/evolutionary algorithms," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 6, no. 3, p. 18, 2011.