Twitter Heron: Towards Extensible Streaming Engines

Maosong Fu[t], Ashvin Agrawal[m], Avrilia Floratou[m], Bill Graham[t], Andrew Jorgensen[t]
Mark Li[t], Neng Lu[t], Karthik Ramasamy[t], Sriram Rao[m], Cong Wang[t]
[t]*Twitter, Inc.* {@Louis_Fumaosong, @billgraham, @ajorgensen, @objclone, @luneng90, @karthikz, @c0ngwang}
[m]*Microsoft* {asagr, avflor, sriramra}@microsoft.com

***Abstract*—Twitter's data centers process billions of events per day the instant the data is generated. To achieve real-time performance, Twitter has developed Heron, a streaming engine that provides unparalleled performance at large scale. Heron has been recently open-sourced and thus is now accessible to various other organizations. In this paper, we discuss the challenges we faced when transforming Heron from a system tailored for Twitter's applications and software stack to a system that efficiently handles applications with diverse characteristics on top of various Big Data platforms. Overcoming these challenges required a careful design of the system using an extensible, modular architecture which provides flexibility to adapt to various environments and applications. Further, we describe the various optimizations that allow us to gain this flexibility without sacrificing performance. Finally, we experimentally show the benefits of Heron's modular architecture.**

I. Introduction

Twitter's daily operations rely heavily on real-time processing of billions of events per day. To efficiently process such large data volumes, Twitter has developed and deployed Heron [1], a streaming engine tailored for large scale environments that is able to meet Twitter's strict performance requirements. Heron has been shown to outperform Storm [2], the first generation streaming engine used at Twitter and at the same time provides better manageability. Heron is now the de facto stream data processing system in Twitter and is used to support various types of applications such as spam detection, real time machine learning and real time analytics, among others.

Twitter has recently open sourced Heron. As many organizations rely on stream processing for various applications such as IOT and finance among others, Heron has already attracted contributors from multiple institutions, including Microsoft. However, making Heron publicly available poses significant challenges. Heron must now be able to operate on potentially different environments (private/public cloud), various software stacks, and at the same time support applications with diverse requirements without sacrificing performance. For example, Twitter deploys Heron on top of the Aurora scheduler [3] in its private cloud. However, many organizations already manage a solutions stack based on the YARN scheduler [4] since it efficiently supports batch processing systems such as Apache Hive [5]. Co-locating Heron along with these batch processing systems on top of YARN, can reduce the total cost of ownership and also improves maintenance and manageability of the overall software stack. Similarly, depending on the application requirements, one might want to optimize for performance or for deployment cost especially in the context of a cloud environment which employs a pay-as-you-go model. Along the same lines, a data scientist might want to use Python to implement a machine learning model in a streaming system, whereas a financial analyst might prefer to use C++ in order to fully optimize the performance of her application.

To address these challenges, we carefully designed Heron using a *modular architecture*. In this approach, each component of the system provides the minimum functionality needed to perform a particular operation. The various modules of the system communicate and provide services to each other through well-specified, communication protocols. An important aspect of Heron is that it allows extensibility of the modules. More specifically, Heron allows the application developer or the system administrator to create a new implementation for a specific Heron module (such as the scheduler, resource manager, etc) and plug it in the system without disrupting the remaining modules or the communication mechanisms between them. The benefit of this approach is that the developer needs to understand and implement only the basic functionality of a particular module, using well-specified APIs, and does not need to interact with other system components. Another important aspect of this architecture is that different Heron applications can seamlessly operate on the same resources using different module implementations. For example, one application might manage resources optimizing for load balancing while another application might optimize for total cost. Providing such functionality is critical for the adoption of Heron across different organizations.

To the best of our knowledge, Heron is the first streaming system that adopts and implements a modular architecture. Note that although other systems such as Storm [2] face similar challenges, they have taken the approach of implementing specialized versions of the system for different software stacks. For example, there are separate repositories for Storm on Mesos [6], Storm on YARN [7], and Storm on Slider [8]. Heron, on the other hand, has a more flexible and scalable architecture that allows it to easily integrate with different Big Data platforms, as long as the appropriate module implementations are available. Despite the benefits of general-purpose architectures, such as Heron's modular architecture, a common belief is that specialized solutions tend to outperform general-purpose ones because they are optimized for particular environments and applications. In this paper, we show that by carefully optimizing core components of the system, Heron's general-purpose architecture can actually provide better performance than specialized solutions such as Storm.

The contributions of this paper are the following:

- We describe Heron's modular architecture and present in detail various important modules of the system.

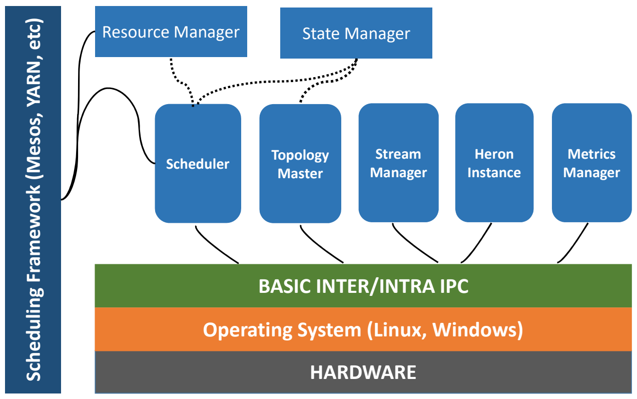

Fig. 1. Heron's Modular Architecture

- We compare and contrast Heron's general-purpose architecture to specialized approaches such as Storm [2] and Spark Streaming [9].
- We experimentally highlight the performance advantages of Heron's general-purpose architecture.

II. Heron's General-Purpose Architecture

In this section, we provide a high-level overview of Heron's modular and extensible architecture. Heron's architecture is inspired by that of microkernel-based operating systems. Unlike monolithic kernel-based operating systems, microkernel-based operating systems provide a core set of services each one implementing one basic mechanism such as IPC and scheduling among others. As pointed in [10], microkernel-based operating systems have been developed to facilitate the adoption of new device drivers, protocol stacks, and file systems. These components were normally located in the monolithic kernel and as result their adoption required a considerable amount of work. In microkernel-based systems, these components run in user space, outside the kernel. Along the same lines, due to the heterogeneity of today's cloud environments and Big Data platforms, we decided to design Heron using extensible, self-contained modules that operate on top of a kernel which provides the basic functionality needed to build a streaming engine. This architecture facilitates the adoption of Heron to run on different underlying software stacks. Furthermore, it simplifies the detection of performance problems as well as tuning and maintainance of the overall software stack.

Figure 1 shows the various Heron modules and the interactions between them. As opposed to a monolithic architecture, Heron consists of several modules (in blue) that communicate through basic inter/intra process communication mechanisms (IPC). The IPC mechanisms essentially constitute the kernel of the system. Every other Heron module is extensible and easily pluggable to the system, similar to microkernel-based operating systems.

As described in [1], a Heron topology is a directed graph of spouts and bolts. The spouts are sources of input data such as a stream of Tweets, whereas the bolts perform computations on the streams they receive from spouts or other bolts. When a topology is submitted to Heron, the `Resource Manager` first determines how many containers should be allocated for the topology. The first container runs the `Topology Master` which is the process responsible for managing the topology throughout its existence. The remaining containers each run a `Stream Manager`, a `Metrics Manager` and a set of `Heron Instances` which are essentially spouts or bolts that run on their own JVM. The `Stream Manager` is the process responsible for routing tuples among `Heron Instances`. The `Metrics Manager` collects several metrics about the status of the processes in a container. As we will discuss later, the `Resource Manager` determines the allocation of `Heron Instances` to containers based on some resource management policy. It then passes this information to the `Scheduler` which is responsible for allocating the required resources from the underlying scheduling framework such as YARN or Aurora. The `Scheduler` is also responsible for starting all the Heron processes assigned to the container. In the following sections we discuss how the various modules behave when a failure is detected or when a user updates the topology configuration.

Heron's architecture provides flexibility to the application developers. For example, a `Heron Instance` can execute user code that can be written in Java or Python. Along the same lines, an application developer can use the YARN implementation of the `Scheduler`, if she operates on top of a YARN cluster. If her environment changes, she can easily switch to another `Scheduler` implementation (e.g., Aurora) without having to change her topology or her Heron setup. Similarly, the application developer can pick the `Resource Manager` implementation that better suits her needs or even implement new policies. The user can configure the Heron modules either at topology submission time through the command line or using special configuration files.

III. General-Purpose vs. Specialized Streaming Architectures

Heron's modular architecture provides flexibility to incorporate various cloud environments and Big Data platforms. In that sense, Heron's architecture is a general-purpose architecture as opposed to specialized streaming engines which have tight dependencies between various parts of the system. In this section, we compare Heron's general-purpose architecture with the architectures of Storm [2] and Spark Streaming [9]. More specifically, we highlight the major differences between Heron and these two systems focusing on their resource isolation mechanisms in shared environments as well as their data processing and communication layers.

A. *Comparison with Apache Storm*

Storm is also a large-scale stream processing system but has a significantly different architecture than Heron. Although it provides some extensibility points, like support for different topology specification languages, Storm has a much more rigid architecture than Heron.

Scheduling and Resource Sharing. Heron is designed with the goal of operating in a cloud environment on top of a scheduling framework such as Aurora or YARN (although it can also run on local mode). As a result, it leverages the resource isolation mechanisms implemented by these frameworks. Storm, on the other hand implements parts of the functionality of the Heron `Resource Manager`, the Heron `Scheduler` and the underlying scheduling framework in the same abstraction. More specifically, Storm is responsible for deciding how resources will be allocated to a given topology and how resources will be allocated across different topologies without taking into account the underlying scheduling framework. Apart from the extensibility issues, this approach has several drawbacks. First, it is not clear what is the relationship between Storm's scheduling decisions and that of the underlying scheduling framework if one is part of the software stack. Second, in the absence of a scheduling framework, it becomes very hard to provide resource isolation across different topologies in an environment where resources are shared. Finally, unlike Heron where resources are acquired when the topologies are deployed, the resources for a Storm cluster must be acquired before any topology can be submitted. As a result, the system administrator needs to know a priori how many topologies will run on the cluster and their expected resource requirements in order to preallocate resources accordingly.

Data Processing Layer. Heron's modular architecture provides resource isolation not only across different topologies (through the underlying scheduling framework) but also among the processes of the same topology. This is because every spout and bolt run as separate `Heron Instances` and thus do not share the same JVM. This architecture helps isolating performance problems and also recover from failures. Storm, on the other hand, packs multiple spout and bolt tasks into a single executor. Each executor shares the same JVM with other executors. As a result, it is very difficult to provide resource isolation in such an environment.

Communication Layer. In Heron's modular architecture, a dedicated process named `Stream Manager` is responsible for all the data transfers among the `Heron Instances`. Separating this component from the processing units (`Heron Instances`) makes the system scalable, allows various optimizations at the data transfer level and simplifies the manageability of the system. In Storm, on the other hand, the threads that perform the communication operations and the actual processing tasks share the same JVM [2]. As a result, it is much harder to isolate performance bottlenecks and thus optimize the overall performance.

B. *Comparison with Spark Streaming*

Spark Streaming [9] is another popular open-source streaming engine. Its architecture differs from that of Heron and Storm as it uses a separate processing framework (Spark [11]) to process data streams. Because of its architecture, it operates on small batches of input data and thus it is not suitable for applications with latency needs below a few hundred milliseconds [12].

Scheduling and Resource Sharing. Spark Streaming depends on Spark for extensibility. For example, since Spark can support two different types of schedulers (YARN and Mesos), Spark Streaming is able to operate on top of these frameworks. Apart from streaming applications, Spark is typically also used for batch analytics, machine learning and graph analytics on the same resources. As a result, it is not easy to customize it particularly for streaming applications. Spark runs `executor` processes for each application submitted. Each executor process can run multiple tasks in different threads. Unlike Heron's `Resource Manager`, Spark Streaming does not provide a way to specify new policies for assigning tasks to executors.

Data Processing Layer. It is worth noting that Spark (and, as a result, Spark Streaming) has a similar architecture with Storm that limits the resource isolation guarantees it can provide. As noted above, each executor process can run multiple tasks in different threads. As opposed to Heron, this model does not provide resource isolation among the tasks that are assigned to the same executor.

Communication Layer. Finally, unlike Heron that implements an extensible `Stream Manager`, all the Spark Streaming communication mechanisms rely on Spark and are not customizable.

Overall, Heron's modular architecture makes it much easier for the application developer to configure, manage, and extend Heron. Although Storm and Spark Streaming provide some extensibility, their respective architectures contain more tightly-coupled modules and are less flexible than that of Heron.

IV. Heron Modules

In this section, we present several important Heron modules and describe in detail their functionality. To further highlight the simplicity and extensibility of the Heron modules, we also provide the corresponding APIs. The source code of all the implemented modules can be found in the Heron code repository [13].

A. *Resource Manager*

The `Resource Manager` is the module that determines how resources (CPU, memory, disk) are allocated for a particular topology. Note that the `Resource Manager` is not a long-running Heron process but is invoked on-demand to manage resources when needed. More specifically, it is the component responsible for assigning `Heron Instances` to containers, namely generating a *packing plan*. The packing plan is essentially a mapping from containers to a set of `Heron Instances` and their corresponding resource requirements. The `Resource Manager` produces the packing plan using a *packing algorithm*. After the packing plan is generated, it is provided to the `Scheduler` which is the component responsible for requesting the appropriate resources from an underlying scheduling framework such as YARN, Mesos or Aurora.

When a topology is submitted for the first time, the `Resource Manager` generates an initial packing plan which is used to distribute the `Heron Instances` to a set of available containers. Heron provides the flexibility to the user to adjust the parallelism of the components of a running Heron topology. For example, a user may decide to increase the number of instances of a particular spout or bolt. This functionality is useful when load variations are observed. In such case, adjusting the parallelism of one or more topology components can help meet a performance requirement or better utilize the available resources. When the user invokes a topology scaling command, the `Resource Manager` adjusts the existing packing plan given the specific user requests.

The `Resource Manager` implements the basic functionality of assigning `Heron Instances` to containers through the following simple APIs:

```
public interface ResourceManager {
 void initialize(Configuration conf, Topology
    topology)
 PackingPlan pack()
 PackingPlan repack(PackingPlan currentPlan, Map
    parallelismChanges)
 void close()
}
```

The `pack` method is the core of the `Resource Manager`. It implements a packing algorithm which generates a packing plan for the particular topology and it is invoked the first time a topology is submitted. The `Resource Manager` can implement various policies through the `pack method`. For example, a user who wants to optimize for load balancing can use a simple `Round Robin` algorithm to assign `Heron Instances` to containers. A user who wants to reduce the total cost of running a topology in a pay-as-you go environment can choose a `Bin Packing` algorithm that produces a packing plan with the minimum number of containers for the given topology [14]. Note that Heron's architecture is flexible enough to incorporate user-defined resource management policies which may require different inputs than the existing algorithms.

The `repack` method is invoked during topology scaling operations. More specifically, it adjusts the existing packing plan by taking into account the parallelism changes that the user requested. Various algorithms can be implemented to specify the logic of the repack operation. Heron currently attempts to minimize disruptions to the existing packing plan while still providing load balancing for the newly added instances. It also tries to exploit the available free space of the already provisioned containers. However, the users can implement their own policies depending on the requirements of their particular application.

It is worth noting that the `Resource Manager` allows the user to specify different resource management policies for different topologies running on the same cluster. This is an important feature of the general-purpose architecture which differentiates it from specialized solutions.

B. *Scheduler*

The `Scheduler` is the module responsible for interacting with the underlying scheduling framework such as YARN or Aurora and allocate the necessary resources based on the packing plan produced by the `Resource Manager`.

The `Scheduler` can be either *stateful* or *stateless* depending on the capabilities of the underlying scheduling framework. A *stateful* `Scheduler` regularly communicates with the underlying scheduling framework to monitor the state of the containers of the topology. In case a container has failed, the stateful `Scheduler` takes the necessary actions to recover from the failure. For example, when Heron operates on top of YARN, the Heron `Scheduler` monitors the state of the containers by communicating with YARN through the appropriate YARN APIs. When a container failure is detected, the `Scheduler` invokes the appropriate commands to restart the container and its associated tasks. A *stateless* `Scheduler`, on the other hand, is not aware of the state of the containers while the topology is running. More specifically, it relies on the underlying scheduling framework to detect container failures and take the necessary actions to resolve them. For example, the Heron `Scheduler` is stateless when Aurora is the underlying scheduling framework. In case of a container failure, Aurora invokes the appropriate command to restart the container and its corresponding tasks.

The `Scheduler` implements the following APIs:

```
public interface Scheduler {
 void initialize(Configuration conf)
 void onSchedule(PackingPlan initialPlan);
 void onKill(KillTopologyRequest request);
 void onRestart(RestartTopologyRequest request);
 void onUpdate(UpdateTopologyRequest request);
 void close()
}
```

The `onSchedule` method is invoked when the initial packing plan for the topology is received from the `Resource Manager`. The method interacts with the underlying scheduling framework to allocate the resources specified in the packing plan. The `onKill` and `onRestart` methods are invoked when the `Scheduler` receives a request to kill or restart a topology, respectively. Finally, the `onUpdate` method is invoked when a request to update a running topology has been submitted. For example, during topology scaling the `Scheduler` might need to update the resources allocated to the topology given a new packing plan. In this case, the `Scheduler` might remove existing containers or request new containers from the underlying scheduling framework.

It is worth noting that the various underlying scheduling frameworks have different strengths and limitations. For example, YARN can allocate heterogeneous containers whereas Aurora can only allocate homogeneous containers for a given packing plan. Depending on the framework used, the Heron `Scheduler` determines whether homogeneous or heterogeneous containers should be allocated for the packing plan uncder consideration. This architecture abstracts all the low level details from the `Resource Manager` which generates a packing plan irrespective of the underlying scheduling framework.

Heron currently supports several scheduling frameworks. More particularly, it has been tested with Aurora and YARN. The Heron community is currently extending the `Scheduler` component by implementing the above APIs for various other frameworks such as Mesos [15], [16], Slurm [17] and Marathon [18]. Note that since Heron's architecture is able to seamlessly incorporate scheduling frameworks with different capabilities, there is no need to create separate specialized versions of Heron for each new scheduling framework.

C. *State Manager*

Heron uses the `State Manager` module for distributed coordination and for storing topology metadata. More specifically, the `Topology Master` advertises its location through the `State Manager` to the `Stream Manager` processes of all the containers. As a result, in case the `Topology Master` dies, all the `Stream Managers` become immediately aware of the event. Heron stores several metadata in

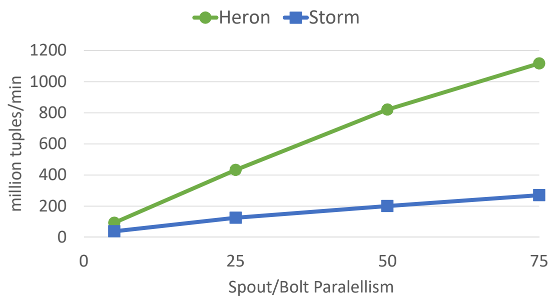

Fig. 2. Throughput with acks

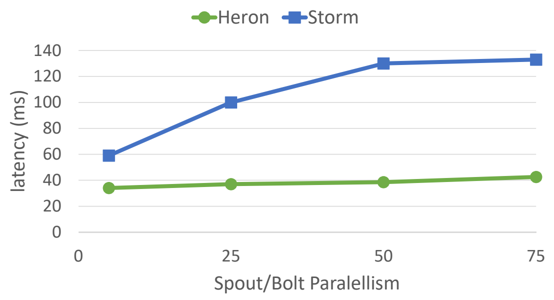

Fig. 3. End-to-end latency with acks

the State Manager including the topology definition, the packing plan generated by the Resource Manager, the host and port information of all the containers, and the URL of the underlying scheduling framework among others.

The State Manager abstraction allows easy integration with various filesystems and coordination frameworks. Heron provides a State Manager implementation using Apache Zookeeper [19] for distributed coordination in a cluster environment and also an implementation on the local file system for running locally in a single server. Both implementations currently operate on tree-structured storage where the root of the tree is supplied by the Heron administrator. However, Heron provides the flexibility to extend the State Manager in order to incorporate other state management mechanisms.

V. Optimizing the Communication Layer

One of the concerns with general-purpose architectures, such as Heron's modular architecture, is the potential overhead of data transfer between the different modules of the system. As opposed to tighter architectures, such as Storm, where the data transfer and the processing threads share the same JVM, the Stream Manager is a separate process in Heron and thus incurs inter-process instead of intra-process communication.

In this section, we first discuss how we enable fast data transfers between Heron Instances by applying various optimizations to the Stream Manager module, and as a result, outperform other tightly coupled architectures (such as Storm). We then provide more insights about the Stream Manager by discussing two important configuration parameters that affect the overall performance.

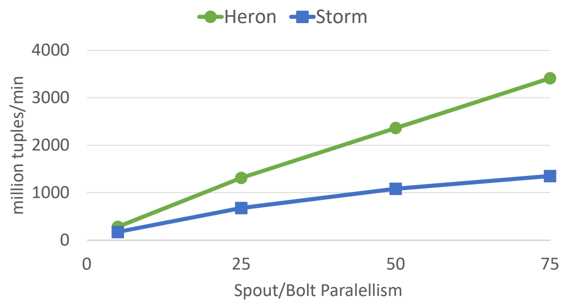

Fig. 4. Throughput without acks

A. Stream Manager Optimizations

The Stream Manager is an extensible and pluggable module. It can potentially support different inter-process communication mechanisms (sockets, shared memory, etc) and different data serialization formats (e.g., binary encoding, Protocol Buffers [20]). The module is implemented in C++ because it provides tighter control on the memory and cpu footprint and to avoid the overhead of copying data between the native heap and the JVM heap.

The Stream Manager implementation relies on two major techniques to achieve high performance. First, it avoids data copies as much as possible. Currently, the Stream Manager implementation uses Protocol Buffers to serialize and exchange data between different processes. Our implementation allows reusability of the Protocol Buffer objects by using memory pools to store dedicated objects and thus avoid the expensive new/delete operations. Second, the Stream Manager performs in-place updates of Protocol Buffer objects and uses lazy deserialization whenever possible. For example, when one Stream Manager process receives a message from another Stream Manager process, it parses only the destination field that determines the particular Heron Instance that must receive the tuple. The tuple is not deserialized but is forwarded as a serialized byte array to the appropriate Heron Instance. In Section VI, we experimentally show the impact of these two optimizations in Heron's overall performance both in terms of throughput and latency.

B. Performance Tuning

Although the current Stream Manager implementation can provide very good performance out-of-the-box due to the optimizations that we described above, Heron provides the flexibility to the user to further tune the communication layer. We now discuss two important configuration parameters that can be tuned by the user to actively control the data transfer rates in case this might be beneficial for a particular workload. As part of future work, we plan to automate the process of configuring the values for these parameters based on real-time observations of the workload performance.

The first parameter called *max spout pending*, represents the maximum number of tuples that can be pending on a spout task at any given time. A pending tuple is one that has been emitted from a spout but has not been acknowledged or failed yet. By configuring this parameter the user can determine how many tuples can be in flight at any point of time and thus

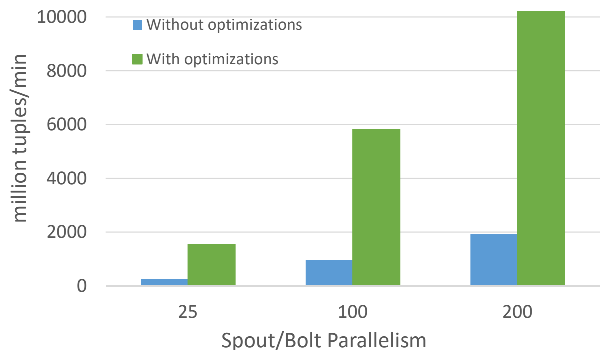

Fig. 5. Throughput without acks

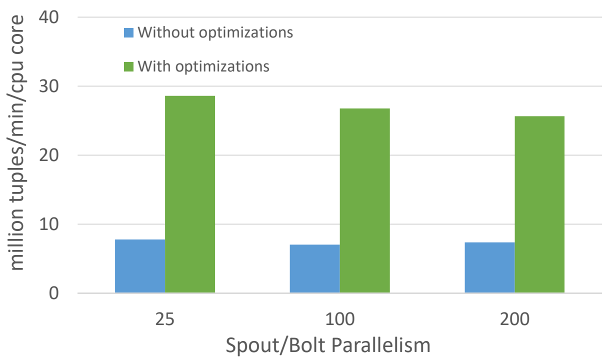

Fig. 6. Throughput per CPU Core without acks

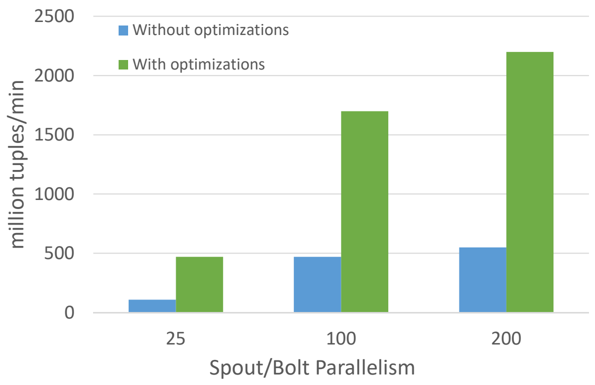

Fig. 7. Throughput with acks

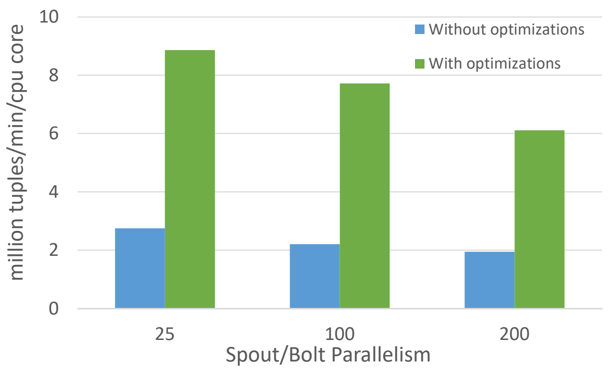

Fig. 8. Throughput per CPU Core with acks

affect both the throughput and the latency of the workload. Section VI provides an analysis of the performance observed as the value of this parameter varies.

The second parameter called *cache drain frequency*, is reated to the `Stream Manager` tuple cache. The tuple cache is a cache that temporarily stores the incoming and outcoming data tuples before routing them to the appropriate `Heron Instances`. The cache stores tuples in batches along with the `Heron Instance` id that is the recipient of the batch. The *cache drain frequency* determines the frequency in milliseconds to drain the tuple cache. As we show in Section VI, the value of this parameter can affect the end-to-end performance both in terms of latency and throughput.

VI. Experimental Evaluation

In this section, we present experiments that demonstrate the benefits of Heron's modular architecture. In all our experiments we use Heron version 0.14.4 and Storm version 1.0.2.

A. Comparing Heron with Storm

In this experiment, we compare Heron's general-purpose architecture with Storm on top of the YARN scheduler. As in previous work [1], we use the `Word Count` topology since it is a good measure of the overhead introduced by either Storm or Heron. This is because its spouts and bolts do not perform significant work. In this topology, the spout picks a word at random from a set of 450K Engish words and emits it. Hence spouts are extremely fast, if left unrestricted. The spouts use hash partitioning to distribute the words to the bolts which in turn count the number of times each word was encountered.

The experiments were performed on Microsoft HDInsight [21]. Each machine has one 8-core Intel Xeon E5-2673 CPU@2.40GHz and 28GB of RAM. In each experiment we use the same degree of parallelism for spouts and bolts and we perform four experiments with different degrees of parallelism. Figures 2 and 3 present the overall throughput and latency results when acknowledgements are enabled. As shown in the figures, Heron outperforms Storm by approximately 3-5X in terms of throughput and at the same time has 2-4X lower latency. Figure 4 presents the total throughput when acknowledgements are disabled. As shown in the figure, the throughput of Heron is 2-3X higher than that of Storm.

Our experiment demonstrates that although Heron employs an extensible and general-purpose architecture, it can significantly outperform Storm's more specialized architecture.

B. Impact of Stream Manager Optimizations

In this section, we quantify the impact of the optimizations in the `Stream Manager` module presented in Section V. We use the `Word Count` topology and vary the parallelism of the spouts and bolts. All experiments were run on machines with dual Intel Xeon E5645@2.4GHZ CPUs, each consisting of 12 physical cores with hyper-threading enabled and 72GB of RAM.

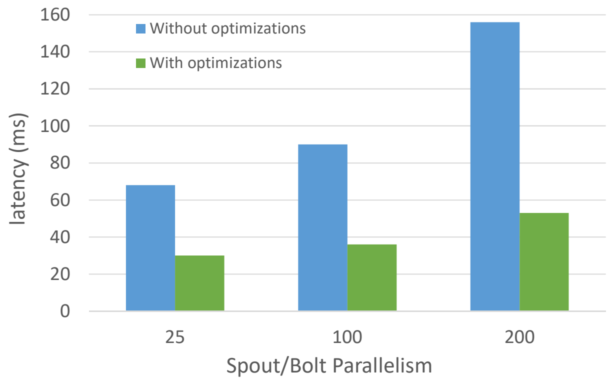

Fig. 9. End-to-end latency with acks

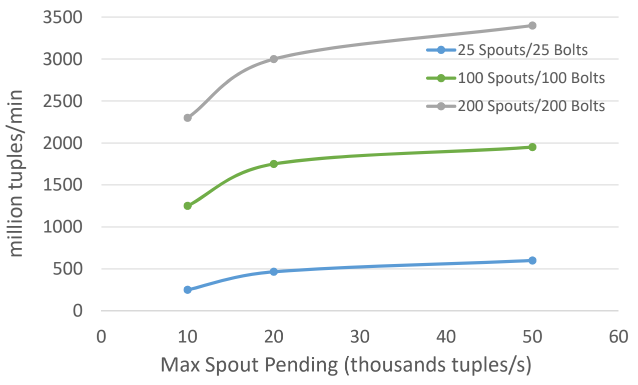

Fig. 10. Throughput vs max spout pending

Figures 5 and 6 present our results when acknowledgements are not enabled. As shown in Figure 5, our `Stream Manager` optimizations provide 5-6X performance improvement in throughput. Figure 6 shows the throughput per cpu core provisioned. When the `Stream Manager` optimizations are enabled, there is approximately a 4-5X performance improvement per CPU core which shows that Heron is then able to utilize resources more efficiently.

The throughput results when acknowledgments are enabled are presented in Figures 7 and 8. As shown in Figure 7, the `Stream Manager` optimizations provide a 3.5-4.5X performance improvement. At the same time, as shown in Figure 8 there is a substantial performance improvement per CPU core. Figure 9 shows that the `Stream Manager` optimizations can also provide a 2-3X reduction in end-to-end latency.

C. Performance Tuning

In this section, we present experiments that highlight the impact of the configuration parameters described in Section V-B on the overall performance. We use the same hardware setup as in the previous experiment and run the `WordCount` topology with acknowledgements enabled.

In this experiment, we quantify the performance impact of varying the maximum number of tuples that can be pending on a spout task (*max spout pending*). Figures 10 and 11 show how the throughput and end-to-end latency vary for different degrees of parallelism. As shown in Figure 10, as the value of the parameter increases the overall throughput also increases until the topology cannot handle more in-flight tuples. This behavior is expected since a small number of pending tuples can result in underutilization of the provisioned resources. Figure 11 shows that as the number of maximum pending tuples increases, the end-to-end latency also increases. This is because the system has to process a larger number of tuples and thus additional queuing delays are incurred. For the particular topology examined, a value of 20 Ktuples/s seems to provide the best tradeoff between latency and throughput.

We now explore the impact of the *cache drain frequency* on the overall performance. As mentioned in Section V-B, this parameter determines how often the `Stream Manager` tuple cache is flushed. Figures 12 and 13 show how the throughput and end-to-end latency vary for different degrees of parallelism. As shown in Figure 12, as the time threshold to drain the cache increases the overall throughput gradually increases until it reaches a peak point. After that point, the throughput starts decreasing. This behavior is consistent across all the configurations tested and can be explained as follows: When the time threshold to drain the cache is low, the system pays a significant overhead in flushing the cache state. This overhead gets amortized as the time threshold increases since the flush operations become less frequent. However, further increasing the time threshold might actually reduce the throughput observed. As noted previously, the maximum in-flight tuples that can be handled by the system is fixed (see *max spout pending* parameter). At the same time, increasing the time threshold to drain the cache will increase the end-to-end latency. As a result, fewer tuples will be processed during a given time interval since the system has a bounded number of pending tuples and is not able to process more in-flight tuples.

Figure 13 shows that if the time threshold to drain the cache is low, the system pays a large overhead to flush the cache. Similar to the throughput curves, as the time threshold increases, the latency improves until the system reaches a point where the additional queuing delays hurt performance.

D. Heron Resource Usage Breakdown

In this experiment, we present resource usage statistics for Heron that highlight that Heron's general-purpose architecture manages resources efficiently. We used a real topology that

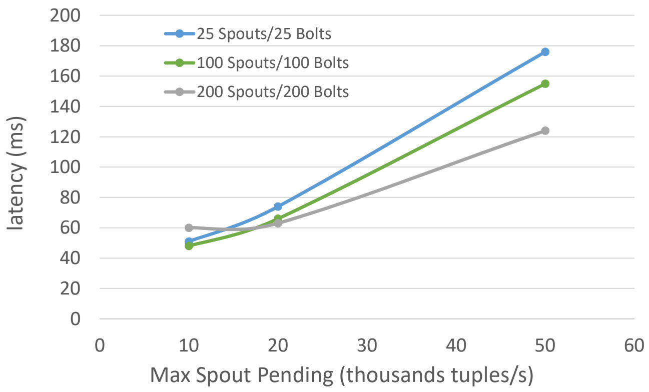

Fig. 11. Latency vs max spout pending

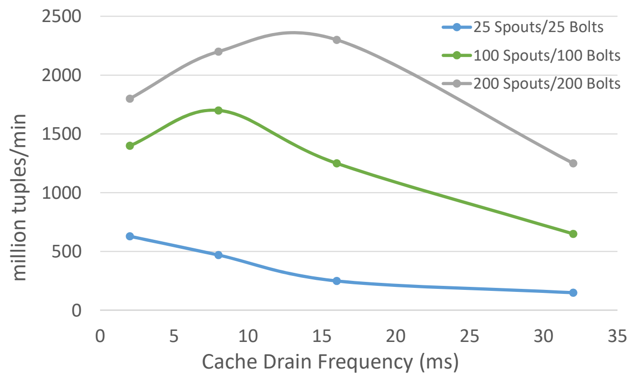

Fig. 12. Throughput vs cache drain frequency

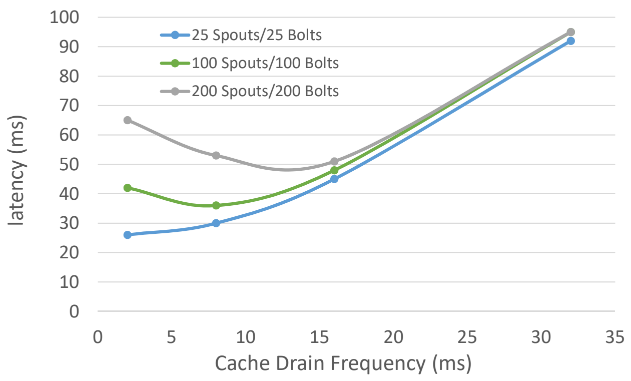

Fig. 13. Latency vs cache drain frequency

reads events from Apache Kafka [22] at a rate of 60-100 million events/min. It then filters the tuples before sending them to an aggregator bolt, which after performing aggregation, stores the data in Redis [23]. We have provisioned 120 and 24 CPU cores for Heron and Redis, respectively. The memory provisioned is 200 GB and 48 GB for Heron and Redis, respectively.

We profiled the topology in stable state and tracked the resources used from each system component. Figure 14 shows the overall resource consumption breakdown. As shown in the figure, Heron consumes only 11% of the resources. This number includes the overheads introduced due to data transfers, internal data serialization and deserialization, and metrics reporting. The remaining resources are used to fetch data from Kafka (60%), execute the user logic (21%) and write data to Redis (8%).

Overall, this experiment highlights that Heron can efficiently manage the available resources and shows that most of the processing overheads are related to data reading and writing to external services.

VII. Conclusions

In this paper, we presented Heron's modular and extensible architecture that allows it to accommodate various environments and applications. We compared Heron's general-purpose architecture with other specialized approaches and showed that with careful system design especially at the communication layer, general-purpose architectures can outperform their specialized counterparts without sacrificing manageability.

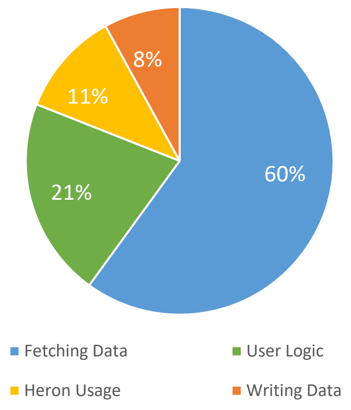

Fig. 14. Resource Consumption

References

[1] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter Heron: Stream Processing at Scale," in *2015 ACM SIGMOD*.

[2] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm@Twitter," in *2014 ACM SIGMOD*.

[3] "Apache Aurora," http://aurora.apache.org/.

[4] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *4th Annual Symposium on Cloud Computing*, 2013.

[5] "Apache Hive," http://hive.apache.org/.

[6] "Storm on Mesos," https://github.com/mesos/storm.

[7] "Storm on YARN," https://github.com/yahoo/storm-yarn.

[8] "Storm on Apache Slider," http://hortonworks.com/apache/storm/.

[9] "Spark Streaming," http://spark.apache.org/streaming/.

[10] "Microkernel," https://en.wikipedia.org/wiki/Microkernel.

[11] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, 2010.

[12] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *In SOSP*, 2013, pp. 423–438.

[13] "Heron Code Repository," https://github.com/twitter/heron.

[14] "The Bin Packing Problem," http://mathworld.wolfram.com/Bin-PackingProblem.html.

[15] "Apache Mesos," http://mesos.apache.org/.

[16] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11, 2011, pp. 295–308.

[17] "The Slurm Workload Manager," http://slurm.schedmd.com/.

[18] "Marathon," https://mesosphere.github.io/marathon/.

[19] "Apache Zookeeper," http://zookeeper.apache.org/.

[20] "Protocol Buffers," https://developers.google.com/protocol-buffers/.

[21] "Microsoft HDInsight," https://azure.microsoft.com/en-us/services/hdinsight/.

[22] "Apache Kafka," http://kafka.apache.org/.

[23] "Redis," http://redis.io/.