Twitter Heron: Towards Extensible Streaming Engines

Karthik Ramasamy[t], Ashvin Agrawal[m], Avrilia Floratou[m], Maosung Fu[t], Bill Graham[t], Andrew Jorgensen[t]
Mark Li[t], Neng Lu[t], Sriram Rao[m], Cong Wang[t]
[t]*Twitter, Inc.* {@karthikz, @Louis_Fumaosong, @billgraham, @andrew, @objclone, @luneng90, @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 billion 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. 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 Hadoop. 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've taken the approach of implementing specialized versions of the system for different software stacks. For example, there are separate repositories for Storm on Mesos [5], Storm on YARN [6], and Storm on Slider [7]. 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.

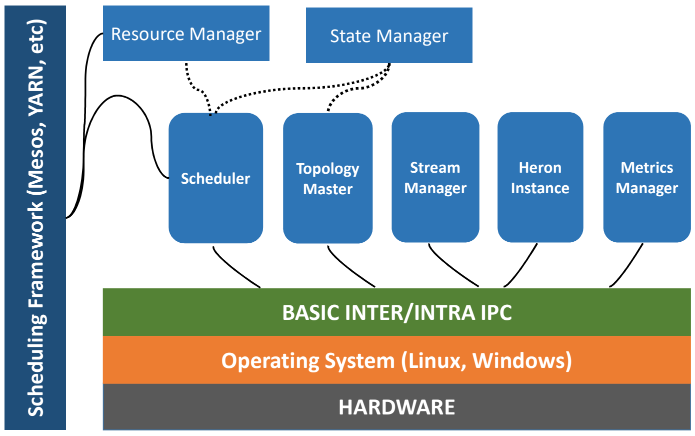

Fig. 1. Heron's Modular Architecture

- We compare and contrast Heron's general-purpose architecture to specialized approaches such as Storm [2] and Spark Streaming [8].
- We experimentally highlight that the performance adavantages 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 [9], 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 [8].

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.

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, Storm needs to collect information about the resources available on each machine and rack of the cluster in order to efficiently allocate resources. Collecting such information is very difficult in a cloud environment where resources are shared among different applications and are not dedicated to Storm.

Another major difference is that 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.

In Heron's modular architecture, there is a dedicated process (`Stream Manager`) that 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 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 optimize the system's performance.

B. Comparison with Spark Streaming

Spark Streaming [8] is another popular open-source streaming engine. Its architecture differs from that of Heron and Storm as it uses a separate processing framework (Spark [10]) 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 [11].

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. For example, it is not possible for an application developer to specify a particular resource management policy for her streaming application. This is because Spark has its own resource allocation policy which is used for all types of applications and is not customizable per application. Similarly, unlike Heron that implements an extensible `Stream Manager`, all the Spark Streaming communication mechanisms rely on Spark and are not customizable.

It is also 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. More specifically, Spark runs `executor` processes for each application submitted. 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.

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 architecture is much more tight and less flexible than that of Heron.

IV. Heron Modules

In this section, we present several important Heron modules and describe in detail their functionality.

A. Resource Manager

The `Resource Manager` is the module that determines how resources (CPU, memory, disk) are allocated for a particular topology. 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. 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 [12]. 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. Depending on the framework used, the Heron `Scheduler` determines whether homogeneous or heterogeneous containers should be allocated for a given packing plan. 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 [13], Slurm [14] and Marathon [15]. Note that unlike other systems such as Storm, Heron's architecture is able to seamlessly incorporate scheduling frameworks with different capabilities and as a result there is no need to create separate specialized versions of Heron for each new scheduling framework.

C. *State Manager*

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

From Heron's perspective, the `State Manager` is a tree structured storage where the root of the tree is supplied by the Heron administrator. This abstraction allows easy integration with various filesystems and coordination frameworks. Heron provides a `State Manager` implementation using Zookeeper for distributed coordination in a cluster environment and also an implementation on the local file system for running locally in a single server or laptop. Heron provides the flexibility to extend the `State Manager` in order to incorporate other state management mechanisms.

V. SYSTEM OPTIMIZATIONS

One of the concerns with general-purpose architectures, such as Heron's modular architecture, is the potential over-

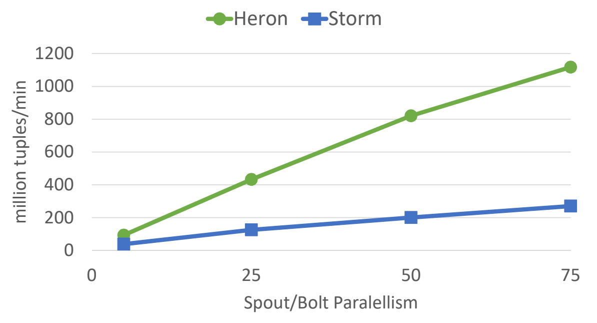

Fig. 2. Throughput with acks

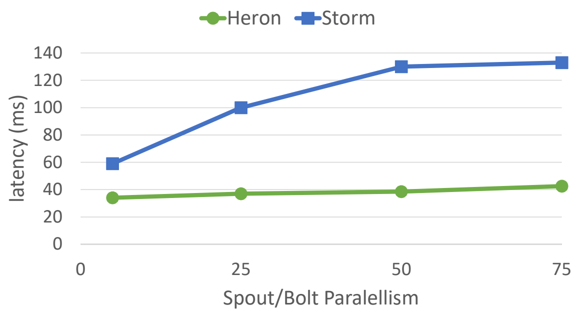

Fig. 3. End-to-end latency with acks

heads 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. However, by applying various optimizations to this module, we can enable fast data transfers between `Heron Instances` and as a result outperform other tightly coupled architectures (such as Storm).

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 (protobufs) [16]). 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 achieves high performance by avoiding data copies as much as possible. Currently, the `Stream Manager` implementation is using protobufs to exchange data between different processes. Our implementation, allows reusability of protobuf objects by using memory pools to store dedicated protobuf objects and thus avoid the expensive new/delete operations. Moreover, whenever is possible, the `Stream Manager` performs in-place updates of protobuf objects and uses lazy deserialization. For example, when one `Stream Manager` process receives a message from another `Stream Manager`, 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 optimizations in Heron's overall performance.

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[1]. 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.

[1]In the final version of the paper, we will also include experiments with Spark Streaming.

The experiments were performed on Microsoft's HDInsight cloud [17]. Each machine has 8 Intel Xeon E5-2673 CPUs@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 without acknowledgements 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.

Figures 5 and 6 present our results. As shown in Figure 5, our `Stream Manager` optimizations provide 5-6X

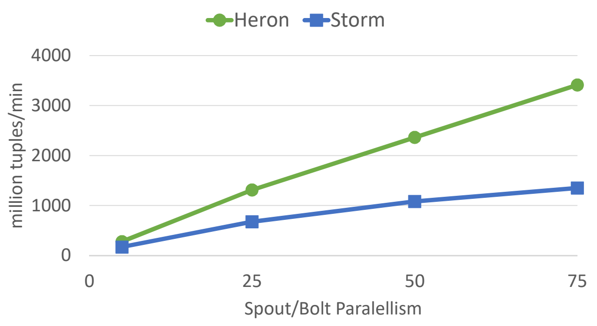

Fig. 4. Throughput w/o acks

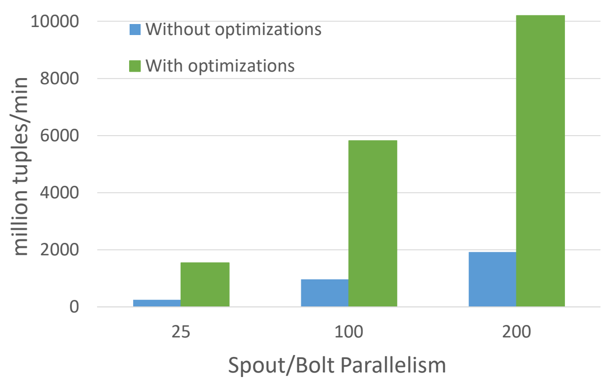

Fig. 5. Total Throughput

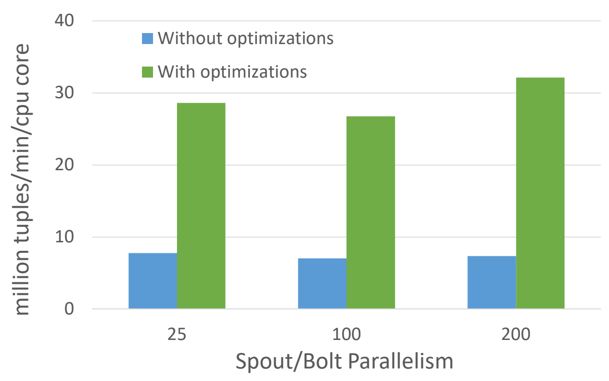

Fig. 6. Throughput per CPU Core

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.

C. 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 reads events from Kafka [18] 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 [19]. We have provisioned 120 and 24 CPU cores for Heron and Redis, respectively. The memory provisioned is $200GB$ and $48GB$ for Heron and Redis, respectively.

We profiled the topology in stable state and tracked the resources used from each system and component. Figure 7 shows the overall resource consumption breakdown. As shown in the Figure, Heron consumes only 11% of the resources. This result shows that Heron's general-purpose architecture can efficiently manage the available resources. The remaining resources are used mostly to fetch data in the spouts and execute the user logic. After carefully profiling the spouts, we noticed that they spend 23% of their time fetching data from Kafka and 63% of their time deserializing them. The bolts spend 19% of their time serializing the data they received and 68% of their time writing to Redis.

Overall, this experiment highlights Heron's efficiency and shows that most of the processing overheads are related to data reading and writing to external services.

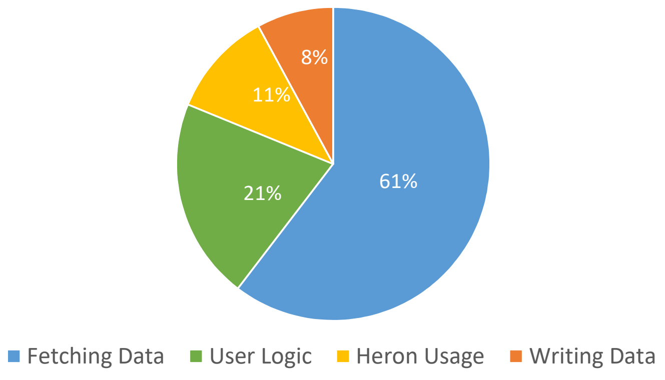

Fig. 7. Resource Consumption

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, general-purpose architectures can outperform their specialized counterparts without sacrificing manageability.

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

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

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

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

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

[10] 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.

[11] 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.

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

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

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

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

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

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

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

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