MIPS: Instance Placement for Stream Processing Systems based on Monte Carlo Tree Search

Xi Huang, Ziyu Shao, Yang Yang

School of Information Science and Technology, ShanghaiTech University
Email: {huangxi, shaozy, yangyang}@shanghaitech.edu.cn

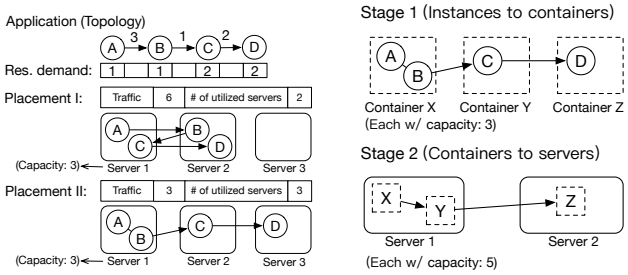
***Abstract*—For up-to-date data stream processing systems, *e.g.*, Apache Heron, the distribution of processing units, *a.k.a. instance placement*, is determined in two stages, *i.e.*, first mapping instances to containers and then mapping containers to servers. The placement, if improperly decided, can induce considerable traffic across servers and inefficient resource allocation. However, it is an open problem to decide the placement effectively, due to the complex interaction among instances, dependency between the decision making in two stages, and the trade-off between traffic reduction and resource utilization improvement. In this paper, we formulate such a problem as two sequential decision making problems. By adopting Monte Carlo Tree Search (MCTS) methods, we propose *MIPS*, *i.e.*, a MCTS-based Instance Placement Scheme that decides the two-stage placement in a unified manner, achieving a well balance between computational efficiency and optimality. Results from simulations show that, with mild-value of samples, MIPS surpasses baseline schemes with significant improvement in both traffic reduction and utilization. To our best knowledge, this paper is the first to study and solve the two-staged mapping problem in such systems based on Heron.**

I. Introduction

Stream processing engines enable modern systems to conduct large-scale and real-time analytics over incessant data streams from a wide assortment of applications. Among the engines [1]–[6] proposed in recent years, Apache Heron [5] is typical for its particular design targeting at high scalability and responsiveness, with a wide adoption by numerous companies.

In Heron, an applications is constructed as a direct-acyclic graph (DAG), *a.k.a. topology*, where data streams are processed through pipelined processing units, *i.e.*, *components*. To launch an application, users should submit a request that specifies the application's topology, *parallelism* of each component, *i.e.*, the number of its instances, and their resource demands. With such information, Heron instantiates the components and decides the distribution of their instances, *a.k.a. instance placement*, in a cluster of heterogeneous servers. The instance placement is decided in two stages, with instances run as an independent processes [7]. In the first stage, all instances are packed into containers for better performance isolation, often known as *instance-to-container mapping*. In the second stage, these containers are assigned to different servers, *a.k.a. container-to-server mapping*. Decision-making in this stage is often delegated to established resource managers, *e.g.*, YARN [8] and Nomad [9], to achieve efficient resource allocation.

To Heron, the instance placement, if wrongly decided, can impose unexpected impacts on the system performance. On the one hand, placing successive instances on distant servers can induce considerable traffic across servers, leading to prolonged response time and hence degrading the quality of service [10]. On the other hand, disregarding the heterogeneity in servers' resource availability and instances resource demand may result in inefficient resource utilization [11]. Yet, it is challenging to decide the optimal placement for either stage, in front of the complex interaction between instances, resource contention among applications, the intrinsic complexity of the problem, and the trade-off between traffic reduction and efficient resource utilization, as illustrated by Figure 1 (a).



(a) A potential trade-off between traffic reduction and high resource utilization (b) Instance placement in two stages with discrepant goals

Fig. 1. *Basic settings:* A topology with four components, each having one instance. *Description:* (a) Placement I incurs 6 units of traffic between Server 1 and 2; Placement II reduces the traffic by 3 units but utilizes all 3 servers. (b) Stage 1: instances are packed into containers with minimum traffic. Stage 2: containers are deployed to servers in the order of (X, Y, Z), each assigned to the server with minimum but sufficient resources (best fit).

The stage-by-stage form of placement makes the decision making even more challenging. For example, in Figure 1 (b), if one attempts to minimize traffic in the first stage and to maximize resource utilization in the second stage, the resulting placement can be far from optimum. Particularly, instances are mapped to containers with minimum traffic across containers in Stage 1, and containers are spread onto as few servers as possible in Stage 2. Compared to the resulting placement, the cross-server traffic can be further reduced by 50%, with container Y and Z on Server 1 and container X in Server 2. This is due to the discrepancy between the optimization objectives in two stages. The placement can be improved by more effective decision-making with carefully chosen objectives. It is still an open problem to solve the above challenges.

In this paper, we study the two-staged instance placement problem. By formulating it as two integer programming problems, we transform each of them into a sequential decision problem. Regarding the instance-to-container mapping problem, it is equivalent to finding such a decision sequence to place instances onto containers successively that leads to the

placement with least containers and minimum traffic between them. Regarding the container-to-server mapping problem, it seeks a decision sequence that assigns containers to servers with minimum cross-server traffic. Instead of designing algorithms to solve the two problems separately, we adopt Monte Carlo Search Tree (MCTS) methods and propose *MIPS*, a unified scheme that decides two-stage placement in an efficient and timely manner. Particularly, the instance placement is obtained by applying MIPS firstly to find an effective instance-to-container mapping, and with this mapping as input, find the container-to-server mapping in the second stage. By constructing a decision tree for each problem incrementally through a series of random sampling, MIPS effectively reduces its search space and yields an effective decision sequence, achieving a well balance between computational efficiency and optimality.

Our main contributions are summarized as follows.

- ◇ **Modeling and Formulation:** By modeling the stream processing systems based on Apache Heron, we formulate the two-staged instance placement problem as two integer programming problems, with carefully chosen objectives in both traffic reduction and high resource utilization.
- ◇ **Algorithm Design:** By transforming each stage of placement into a sequential decision process and adopting MCTS methods with Upper Confidence bound for Trees (UCT), we propose MIPS, a randomized scheme that decides instance placement with a tunable trade-off between computational efficiency and optimality.
- ◇ **Experimental Verification and Investigation:** Most placement schemes are designed for Apache Storm [4] and hence not directly applicable to Heron. To make them comparable with MIPS, we propose their variants under Heron as the baselines. We conduct extensive simulations to evaluate MIPS, discuss how it performs with different settings, and analyze how it excels the baselines.

To the best of our knowledge, this paper is the first to study the two-staged placement problem for the systems based on Apache Heron. Our solution is also applicable to other stream processing systems. The rest of the paper is organized as follows. Section II describes the system model and problem formulation. Section III presents the design of MIPS, Section IV shares the results and analysis from simulations, and Section V concludes the paper. More details are referred to our technical report [to be added].

II. Related Work

So far, data stream processing engines have evolved over four generations [12]. Most up-to-date engines are particularly designed for emerging applications in both cloud and edge computing [1]–[6], [13]. In general, they follow two processing models. One is the operator-graph model [1]–[6]. As in Apache Heron, this model treats each application as a DAG of components, *a.k.a.*, operators, that process data streams in fine-grained units. The other is micro-batch model [13]. In this model, data streams are processed in batches over short time windows. Regarding instance placement problem, among engines that adopt the operator-graph model, previous works mainly focus on Apache Storm [4], the processing engine prior to Heron in Twitter.

In Storm, components are instantiated as tasks, which are directly packed into threads within worker processes on different servers. For task placement, Storm only provides naive schemes that can induce considerable cross-server traffic and inefficient resource allocation. To mitigate such issues, two lines of efforts have proceeded in parallel. On the one hand, a body of works have proposed various schemes to improve the traffic-awareness or resource-awareness for task placement in Storm [10], [11], [14], [15]. On the other hand, Twitter launched Heron to replace Storm [5] in 2015, with a significant re-design in system architecture. Notably, Heron decouples the logical placement from resource management by deciding the instance placement in two stages [7]. However, most previous schemes designed for Storm are not applicable to solve the two-stage placement problem in Heron. Notably, Eskandari *et al.* [15] proposed a hierarchical placement scheme that determines task placement by directly applying graph partitioning algorithm in two stages. Though their solution can be tailored to Heron with proper refinement, we note that it is one-off and deterministic, and without resource-awareness. Instead, MIPS is particularly designed for solving such a problem for Heron. First, it determines the placement by jointly reducing traffic and promoting resource efficiency. Second, with random sampling, it can trade off extra computation for further improvement in the resulting placement. Third, MIPS can be easily adapted to solve such placement problem in other stream processing systems.

III. Model and Problem Formulation

We develop a model for stream processing systems based on Heron and formulate the problem of two-staged instance placement. We present a sample of the model in Figure 2.

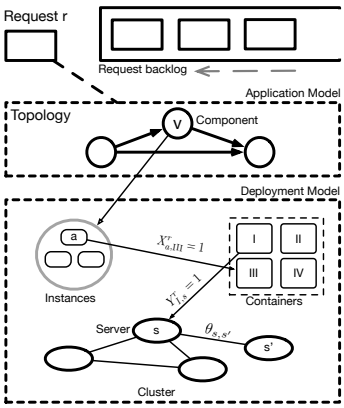


Fig. 2. A sample of the system model.

A. System Model

We consider a Heron-based data stream processing system that is deployed in a cluster with some resource manager such as YARN [8] or Nomad [9]. Servers in the cluster, denoted by set $\mathcal{S}$, are connected by some network topology [16] [17]. Each server s has a resource capacity of $r(s)$, and between server s and s', there is a communication cost $\theta_{s,s'}$ of transferring

a traffic unit, *e.g.*, the number of hops on the shortest path or the estimated RTT.

Requests are submitted to Heron's scheduler successively. For request k, the scheduler builds its application as a DAG of components, denoted by $\mathcal{G}^k \triangleq \left(\mathcal{V}^k, \mathcal{E}^k\right)$, where $\mathcal{V}^k$ is the set of components, and $\mathcal{E}^k$ is the set of directed data streams between components. In practice, the diameter of $\mathcal{G}^k$ is often not too large, mostly less than four [4]. For component $v \in \mathcal{V}^k$, we denote its parallelism by $p(v)$, set of instances by $\mathcal{I}_v$, and the resource demand of its instance by $r(v)$. Besides, the request also specifies the containers, denoted by set $\mathcal{C}^k$, to host the application. Each container c has a resource capacity $r(c)$.

In the first stage, the scheduler maps the set of its instances, $\mathcal{I}^k \triangleq \bigcup_{v \in \mathcal{V}} \mathcal{I}_v$, to containers $\mathcal{C}^k$. The mapping decision is then submitted to the resource manager. In the second stage, the resource manager decides how to distribute containers $\mathcal{C}^k$ to servers $\mathcal{S}$. Based on the two-stage decisions, the resource manager initiates containers and their hosting instances. Then the application is set ready to run. We'll specify the notations for the above decisions in Section III-B.

B. Placement Decisions

Instance-to-container mapping: For request k, we denote the mapping from $\mathcal{I}^k$ to $\mathcal{C}^k$ by $\boldsymbol{X}^k$. Particularly, for $i \in \mathcal{I}^k$ and $c \in \mathcal{C}^k$, $X^k_{i,c}$ is a binary variable such that: $X^k_{i,c} = 1$ indicates instance i is mapped to container c and 0 otherwise. Such decisions should ensure that each instance is only mapped to one container and subject to the resource constraints, *i.e.*,

$$\begin{aligned} &\sum_{c \in \mathcal{C}^k} X^k_{i,c} = 1, \ \forall i \in \mathcal{I}^k, \\ &\sum_{i \in \mathcal{I}^k} d(i) \leq d(c), \ \forall c \in \mathcal{C}^k. \end{aligned} \tag{1}$$

Container-to-server mapping: We denote the decision of mapping from $\mathcal{C}^k$ to $\mathcal{S}$ by $\boldsymbol{Y}^k$. For $c \in \mathcal{C}^k$ and $s \in \mathcal{S}$, $Y^k_{c,s} \in \{0, 1\}$ indicates whether container c is mapped to server s. Likewise, the decisions should map each container to exactly one server and subject to the resource constraints, *i.e.*,

$$\begin{aligned} &\sum_{s \in \mathcal{S}} Y^k_{c,s} = 1, \ \forall c \in \mathcal{C}^k, \\ &\sum_{c \in \mathcal{C}^{-k}_s} d(c) + \sum_{c \in \mathcal{C}^k} Y^k_{c,s} d(c) \leq d(s), \ \forall s \in \mathcal{S}, \end{aligned} \tag{2}$$

where $\mathcal{C}^{-k}_s$ denotes the set of containers of other applications that are deployed before the arrival of request k.

C. Optimization Objectives

Regarding instance-container mapping, it is highly desirable to place successive instances with data streams in between into the same containers in order to minimize cross-container traffic, reducing considerable communication overheads and shortening response time [10]. Formally, given decision $\boldsymbol{X}^k$ for request k, the total traffic from container c to c' is

$$T_{c,c'}(\boldsymbol{X}^k) \triangleq \sum_{i,i' \in \mathcal{I}^k} X^k_{i,c} X^k_{i',c'} w(i, i'), \tag{3}$$

TABLE I
MAIN NOTATIONS

Notation	Description
$\mathcal{S}$	The set of servers in the cluster
$\theta_{s,s'}$	The communication cost per traffic unit from server s to s'
$r(s)$	The resource capacity of server s
$\mathcal{G}^k$	Graph that corresponds to the topology (specified by request k)
$\mathcal{V}^k$	The set of components in topology k
$\mathcal{E}^k$	The set of data streams in topology k
$\mathcal{I}_v$	The set of instances of component v
$\mathcal{I}^k$	Set of all instances in topology of request k
$r(v)$	The resource demand of each instance of component v
$w(i, i')$	The traffic rate from instance i to i'
$\mathcal{C}^k$	The set of containers of topology of request k
$r(c)$	The resource capacity of container c
$\mathcal{C}^{-k}_s$	The set of containers on server s before deploying the topology of request k
$X^k_{i,c}$	Decision that places instance i in container c
$Y^k_{c,s}$	Decision that places container c on server s

where $w(i, i')$ is defined as the traffic volume from instance i to i'. In practice, it can be estimated by running the application in testing environment or traces from history logs. Thereby we can write the total cross-container traffic as

$$T(\boldsymbol{X}^k) \triangleq \sum_{c,c' \in \mathcal{C}^k} T_{c,c'}(\boldsymbol{X}^k). \tag{4}$$

On the other hand, deploying containers also incurs additional resource overheads [18]. For efficient resource allocation, instances should be mapped to as few containers as possible. Formally, given decision $\boldsymbol{X}^k$, we define the number of utilized containers as

$$U(\boldsymbol{X}^k) \triangleq \sum_{c \in \mathcal{C}^k} U_c(\boldsymbol{X}^k), \tag{5}$$

where $U_c(\boldsymbol{X}^k) \triangleq \min\left\{1, \sum_{i \in \mathcal{I}^k} X^k_{i,c}\right\}$, such that $U_c(\boldsymbol{X}^k) = 1$ if container c hosts any instances and 0 otherwise.

Regarding container-server mapping, containers with intensive traffic in between should be placed closely to minimize the inter-server communication cost. Fixed $\boldsymbol{X}^k$ and given $\boldsymbol{Y}^k$, the communication cost for request k from server s to s' is

$$W_{s,s'}(\boldsymbol{Y}^k) \triangleq \sum_{c,c' \in \mathcal{C}^r} Y^k_{c,s} Y^k_{c',s'} \theta_{s,s'} T_{c,c'}(\boldsymbol{X}^k). \tag{6}$$

Thereby we obtain the total communication cost as

$$W(\boldsymbol{Y}^k) \triangleq \sum_{s,s' \in \mathcal{S}} W_{s,s'}(\boldsymbol{Y}^k). \tag{7}$$

D. Problem Formulation

For request k, we define the two-stage placement problem as follows. First, we define the instance-container mapping

problem (**ICMP**) as

$$\begin{aligned} \underset{\boldsymbol{X}^k}{\text{Minimize}} \quad & \alpha T(\boldsymbol{X}^k) + (1-\alpha)U(\boldsymbol{X}^k) \\ \text{Subject to} \quad & X_{i,c}^k \in \{0,1\} \text{ and } (1). \end{aligned} \tag{8}$$

where $\alpha \in [0,1]$ is a parameter that weights the importance of traffic reduction compared to minimizing the number of utilized containers. And we define the container-server mapping problem (**CSMP**) as

$$\begin{aligned} \underset{\boldsymbol{Y}^r}{\text{Minimize}} \quad & W(\boldsymbol{Y}^k) \\ \text{Subject to} \quad & Y_{c,s}^k \in \{0,1\} \text{ and } (2). \end{aligned} \tag{9}$$

IV. Algorithm Design

ICMP and **CSMP** are both non-linear combinatorial optimization problems. Such problems are generally $\mathcal{NP}$-hard with a huge search space size, *i.e.* $O(2^{|\mathcal{I}^k|\times|\mathcal{C}^k|})$ for **ICMP** and $O(2^{|\mathcal{C}^k|\times|\mathcal{S}|})$ for **CSMP**. The coupled resource constraints add even more complexity. We adopt a new perspective by viewing each stage of mapping as a sequential decision process. Instead of developing algorithms to solve the two problems separately, we adopt MCTS methods and propose an efficient scheme to solve them in an unified manner, while achieving a well balance between computational efficiency and optimality.

Algorithm 1 MIPS for mapping problem (**ICMP** or **CSMP**)

Input: For request k, given set $\mathcal{M}^k$, $\mathcal{N}^k$, and objective function f. For **ICMP**, $(\mathcal{M}^k, \mathcal{N}^k) = (\mathcal{I}^k, \mathcal{C}^k)$ with f defined in (8). For **CSMP**, $(\mathcal{M}^k, \mathcal{N}^k) = (\mathcal{C}^k, \mathcal{S})$ with f defined in (9).

Output: An action sequence $\mathcal{A}$ that maps $\mathcal{M}^k$ to $\mathcal{N}^k$.

Initialize count $\leftarrow 0$, action sequence $\mathcal{A} \leftarrow \emptyset$
Initialize the root node n_{root}
while count $< |\mathcal{M}|$ **do**
  $(a,\ n) \leftarrow$ NEXT_ACTION$(n_{root}, \mathcal{A})$
  Update $\mathcal{A} \leftarrow \mathcal{A} \bigcup \{a\}$ and $n_{root} \leftarrow n$
  count $\leftarrow$ count $+ 1$
return $\mathcal{A}$

A. Modeling Decision Trees

To leverage MCTS, we need to fit each mapping stage into a sequential decision process, respectively.

Decision Tree for ICMP: First, we transform the decision process for instance-container mapping into a decision tree. For request k, we construct a tree for its instance-container mapping, with each node n denoting the state associated with a given mapping. The root node n_{root} corresponds to the state where no instances are mapped to containers, *i.e.*, $X_{i,c}^k = 0$ for all i and c. Each outgoing edge of n_{root} indicates an action of mapping some unmapped instance to some container, subject to the resource constraint in (1). For example, the action of mapping instance i_0 to container c_0 changes the mapping state from n_{root} to n with only $X_{i_0,c_0}^k = 1$, denoted by $a(n_{root}, n)$. Similarly, its child nodes then point to their descendants. Recursively defined in this way, the tree eventually reaches leaf nodes. The reward Δ for each leaf node is set as the associated objective value defined in (8) given its mapping.

Algorithm 2 Sub-functions for MIPS

function NEXT_ACTION(n_0, $\mathcal{A}$)
  Initialize $t \leftarrow 0$ and MAX_SAMPLE_NUM.
  while $t <$ MAX_SAMPLE_NUM **do**
    $\mathcal{A}' \leftarrow \mathcal{A}$
    $(n, \mathcal{A}') \leftarrow$ TRAVERSE$(n_0, \mathcal{A}')$
    $(\Delta, \mathcal{A}') \leftarrow$ SIMULATE$(n, \mathcal{A}')$
    if $\Delta > 0$ **then**
      BACK_PROP(n, Δ) and $t \leftarrow t + 1$
  best_child $\leftarrow$ BEST_CHILD$(n_0, 0)$
  return action$(n_0,$ best_child$)$, best_child

function TRAVERSE(n, $\mathcal{A}'$)
  Initialize $n' \leftarrow n$
  while n' is not a leaf **do**
    if $\mathcal{A}_{\text{untried}}(n') \neq \emptyset$ **then**
      return EXPAND(n', $\mathcal{A}'$)
    else $n' \leftarrow$ BEST_CHILD$(n', \omega = \sqrt{2})$
  return n'

function EXPAND(n, $\mathcal{A}'$)
  Choose $a \in \mathcal{A}_{\text{untried}}(n)$ by policy in Sec.IV-B-ii
  $\mathcal{A}' \leftarrow \mathcal{A}' \bigcup \{a\}$
  return next_node(n, a), $\mathcal{A}'$

function BEST_CHILD(n, ω)
  return $\underset{n' \in V(n)}{\arg\min} \frac{Q(n')}{N(n')+1} - \omega\sqrt{\frac{2\ln N(n)}{N(n')}}$

function SIMULATE(n, $\mathcal{A}'$)
  while n is not a leaf **do**
    Choose $a \in \mathcal{A}_{\text{untried}}(n)$ by policy in Sec.IV-B-iii
    $\mathcal{A}' \leftarrow \mathcal{A}' \bigcup \{a\}$ and $n \leftarrow$ next_node(n', a)
  if all constraints are satisfied **then**
    return $(f(\mathcal{A}'), \mathcal{A}')$ %% Reward at leaf node n.
  else return $(-1, \mathcal{A}')$ %% The mapping is invalid.

function BACK_PROP(n, Δ)
  while n is not n_{root} **do**
    $N(n) \leftarrow N(n) + 1$ and $Q(n) \leftarrow Q(n) + \Delta$
    $n \leftarrow$ parent(n)

Decision Tree for CSMP: Regarding CSMP, for request r, we construct another tree for container-server mapping. The root corresponds to the state where no given containers of request k are mapped to servers, while each of its outgoing edge denotes the action of mapping some unmapped container to some server subject to resource constraint in (2). Edges then point to its child nodes with resultant mapping, and in turn point to more descendants. Leaf nodes are either valid mappings from containers to servers, or mappings interrupted due to limited resource. The reward for each leaf node is the corresponding objective value defined in (9) given its mapping.

B. Algorithm Design

Every node n in the decision trees maintains four kinds of states. The first is $N(n)$, denoting the times of node n being visited. The second is $Q(n)$, denoting the total accumulated rewards that node n has received so far. In this way, $\frac{Q(n)}{N(n)}$ is the empirical average reward induced by the sampled decision sequences through node n. The third is $\mathcal{A}_{\text{untried}}(n)$, denoting the set of all unvisited child nodes. The last is $V(n)$, denoting the set of n's visited child nodes.

To find the optimal decision sequence for each decision tree, we propose *MIPS*, MTCS-based Instance Placement Scheme, to decide mapping in each stage. We show MIPS' pseudocode in Algorithm 1 and its sub-functions in Algorithm 2. MIPS solves each mapping problem (**ICMP** and **CSMP**) by deciding the action sequence on a round basis. Basically, each round consists of four steps.

i) Traversal policy: To find a path that leads to the best possible mapping, MIPS has to decide: for its present node, either to exploit historical information by choosing from the visited ones with the minimum estimated reward, or to explore an unvisited node with some unknown reward, *a.k.a.* the *exploitation-and-exploration* tradeoff [19]. MIPS must well balance the tradeoff since 1) if over-dependent on the historical information, MIPS may miss out unvisited nodes that lead to better placement, and 2) radical exploration might waste resources on nodes with far worse rewards. To this end, MIPS adopts the widely adopted Upper Confidence bound for Trees (UCT) [20], by choosing the best child node with the minimum upper confidence bound 1 (UCB1) value (line 26 of Alg.2) [19]. It consists of two terms. The first is the empirical average reward by choosing the node (to ensure the term to be finite, we add one to the denominator) and the second one reflects the visited frequency. If a node has never been visited, the term goes to infinity and its UCB value is $-\infty$, thus the node must be chosen with precedence. If a node is rarely visited but its parent node has been visited a great number of times, the node will have a higher chance to be chosen. In this way, MIPS attains different levels of balance with different weight w. The greater value of w, the more explorative search. Nonetheless, MIPS must explore all child nodes before choosing the visited ones again. However, some actions can lead to obviously high objective values. We bias such node n by initializing $N(n) = 1$ and $Q(n)$ with a large positive value, pretending that the node has been visited once *a priori* with an unfavorably high objective value.

ii) Expansion policy: For **ICMP**, given node n with unvisited children (untried actions), MIPS favors the action that places an instance to such a container that hosts any of its successive instances, to reduce cross-container traffic. To this end, MIPS assigns each of such actions with a positive score, and zero score otherwise, then selects an action randomly from those with high scores. Regarding **CSMP**, MIPS prefers actions that put containers on such servers that 1) host containers with intensive traffic in between to reduce cross-server traffic, and 2) possess least free resources to ensure efficient resource utilization. Likewise, MIPS assigns high scores for such actions and chooses them with precedence. Once expanded, the chosen node is marked visited.

3) Simulation: Starting from an expanded node, the canonical MTCS simulates the next action uniformly at random. Such an aimless policy can lead to mappings with considerable traffic and inefficient resource allocation. Instead, MIPS conducts the simulation with preferences, by rolling out the next node with the same policy as in Section IV-B-ii.

4) Back-propagation: After a simulation, each visited node in the explored tree is updated in terms of the visited times $N(\cdot)$ and the cumulative reward $Q(\cdot)$. The update process is repeated along the way backwards to the root node.

C. System Workflow

Upon request r's arrival, the system first parses its topology, resource demand, and parallelism requirement. The system scheduler first applies MIPS to obtain the instance-container mapping. Next, it eliminates the containers with no instances assigned and submits each container as a job to the underlying resource manager [9]. Then the cluster scheduler applies MIPS to decide the mapping from containers to servers and enforces the deployment. In practice, MIPS can be implemented as a custom module through APIs provided by Heron and cluster schedulers [5] [8] [9].

V. Simulation

A. Basic Settings

Cluster Topology: We prototype a stream processing system based on Heron [5] and cluster resource manager Nomad [9]. We implement two custom schedulers based on MIPS in the system and Nomad for instance-container and container-server mapping, respectively. The system is deployed in clusters that are constructed using two widely adopted topologies, Jellyfish [17] and Fat-Tree [16], respectively. In each cluster are 24 homogeneous switches and 16 heterogeneous servers. For any two servers, their unit communication cost is set as the number of hops of the shortest path between them.

Deployment Resources: Regarding resource allocation, we consider CPU cores and memory on servers [5]. Every server has a number of CPU cores ranging from 16 to 64 and memory from 8G to 32G. For each stream processing application, all of its containers have identical resource capacities.

Stream Processing Applications: We progressively submit requests to the system scheduler to deploy applications with common topologies [7] [10] [11]. Each request specifies a topology with a depth varying from 3 to 5, and a number of components ranging from 3 to 6. Besides, the parallelism for each component ranges from 2 to 6. Instances of the same component have identical functionalities. Instances' resource demand varies from 2 to 6 CPU cores and 4 to 8GB memory. Applications' traffic follows the pattern from the measurements that are drawn from real-world network systems [21].

Compared Schemes: Besides Heron's first-fit-decreasing (FFD) scheme, most existing schemes are designed for Storm [11] [10]. To make them comparable with MIPS, we propose their variants for instance-container mapping under Heron – *R-Heron* and *T-Heron*. *R-Heron*: Given an application, initialize all its containers. Enumerate its components by a breadth first traversal on its topology. If the topology has more than one sink node, then add a virtual root node that precedes all sink nodes and apply the traversal. Next, for each component,

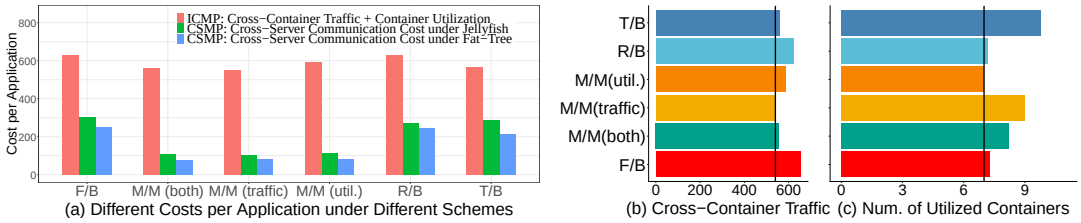


Fig. 3. Comparisons between MIPS/MIPS (M/M) and other schemes: R-Heron/Best-fit (means R-Heron in the first stage and Best-fit in the second stage), T-Heron/Best-fit, and FFD/Best-fit, denoted by R/B, T/B, F/B, respectively. We run M/M under different values of α, including $\alpha = 1$ (*M/M traffic*: minimizing traffic only), $\alpha = 0$ (*M/M util.*: minimizing container utilization only), and $\alpha = 0.5$ (*M/M both*: targeting both objectives), according to (8).

enumerate its instances and repeat the following process. For each instance, assign it to the container with minimum resource distance, where the distance is defined as the traffic rate between the container and other containers, adding the euclidean distance between the instance's resource demand vector and the container's available resource vector. If no containers have enough resources to host an instance, then an error will be raised.

T-Heron: Given an application, initialize all its containers. Sort all instances by their descending order of (incoming and outgoing) traffic rate. Then assign each instance to one of its application's containers with minimum incremental traffic and without exceeding the resource capacity of the container.

FFD [5]: Given an application, initialize all its containers, an empty active container list, and a list of unmapped instances. While there still exists unmapped instances, repeat the process: 1) Choose the next unmapped instance from the list; 2) sort the active containers by descending order of their available resources; 3) pick the first active container with sufficient resource; if no active container can host the instance, add a new container to the list and assign the instance to it.

Meanwhile, we adopt the best-fit scheme in Nomad [9] as the underlying container-server mapping scheme for baselines, which assigns each container to the server with free resources that best match its resource demands.

B. Results and Analysis

We show and analyze the results from our extensive simulations of MIPS. Since MIPS is a randomized algorithm, we repeat each simulation for 100 times and take the average of the results to eliminate the impact of randomness. For ease of description, we denote the two-staged MIPS by M/M, R-Heron/Best-fit (R-Heron in the first stage and Best-fit in the second stage) by R/B, T-Heron/Best-fit by T/B, and FFD/Best-fit by F/B. We also vary the names of M/M with different values of α: 1) *M/M (traffic)*: minimizing traffic only with $\alpha = 1$, 2) *M/M (util.)*: minimizing container utilization only with $\alpha = 0$, and 3) *M/M (both)* with $\alpha = 0.5$.

Performance against Other Schemes: Figure 3 compares two-staged MIPS (M/M) with other three schemes in terms of costs in two stages. The number of samples per round is fixed as 500 for MIPS. Figure 3 (a) makes a comparison of total costs in each of the two stages induced by different schemes, respectively. Note that for any scheme, the total cost of ICMP remains the same while only the cost of CSMP differs under Fat-Tree and Jellyfish, since the decision making for ICMP does not involve communication costs that vary in topologies. We make the following observations.

In the first stage, MIPS (M/M) with different values of α effectively reduces the total cost of cross-container traffic and container utilization compared to other schemes. For example, M/M (traffic) with α=1 leads to the minimum cost, which is 13% lower than F/B, 4% than T/B, and 12% than R/B. M/M (util) and M/M (both) also lead to cost reduction but slightly less than inferior to M/M (traffic).

Figure 3 (b) shows that MIPS (traffic) incurs the minimum cross-container traffic. Figure 3 (c) shows that M/M (util.) achieves the minimum container utilization at 7. Meanwhile, M/M (util.) also outperforms heuristics FFD and R-Heron that focus on utilization by 6% less traffic. On the other hand, along with the traffic reduction, extra container utilization comes as a price to M/M (traffic) for its traffic reduction. Nonetheless, M/M (traffic) still outperforms T-Heron in container utilization. M/M (both) achieves a better balance with both traffic increase and little extra utilization to the optimum. All such advantages are conduced by MIPS's well exploitation of random sampling.

In the second stage, Figure 3 (a) shows that M/M significantly outpaces other schemes by an up to 60% reduction in cross-server traffic under both topologies. This ascribes to not only the advantage taken from the mapping in the first stage, but also the effectiveness of MIPS in the second stage.

Performance under Different Values of α: In Figure 4, we verify the potential tradeoff between traffic reduction and resource utilization, by showing the costs by MIPS under Fat-Tree with α growing from 0 to 1: container utilization gently increases while cross-container traffic notably decreases, and cross-server traffic decreases as well. Compared to other schemes, MIPS advertently places instances to minimize the impact of dependence between placement decisions.

Performance with Different Sampling Numbers: A natural question is that how many samples are sufficient to decide placements with low traffic and resource consumption. Figure 4 investigates the relationship between sampling number and MIPS's performance, with $\alpha = 0.5$. As the sampling number grows from 0 to 500, there is a significant reduction in the container utilization and cross-container traffic. However, as the sampling number continues to grow, the improvement gradually fades. Finally, both the utilization and cross-container/server traffic converge at around 1000 samples. This implies that in practice, compared to its enormous sample space size, MIPS requires only mild-value of sampling number

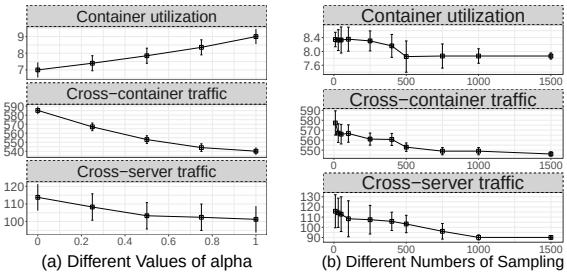


Fig. 4. MIPS's Performance sunder various choices of α and sample numbers.

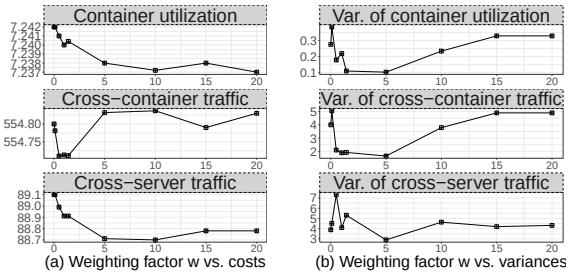


Fig. 5. MIPS's Performance sunder various choices of α and sample numbers.

to make efficient and timely decisions with effective placement with both low traffic and few container resources.

Performance under Different Exploration-Exploitation Tradeoffs: Figure 5 investigates the impact of weighting parameter w on the system performance and the variance among repeated simulations, with $\alpha = 0.5$ and sample number as 500 per round under Fat-Tree topology. Figure 5 (a) shows that as parameter w varies from 0 to 20, MIPS incurs costs roughly at the same level. This is reasonable since with fixed settings those costs are supposed to remain constant on the long-term average. However, we can still see a slight fluctuation among the results under different choices of w. The reason lies in the sampling quality induced by different exploration-exploitation tradeoffs being made. With a smaller value of w, MIPS tends to exploit those decision sequences with known estimates, making the resultant decision largely dependent on a limited set of sequences while missing those with unknown but potentially better rewards. On the other hand, a greater value of w leads to a more explorative search. Due to the randomness of sampling, either undue exploitative or explorative search can have a large variance among different simulations. This is further verified by Figure 5 (b).

VI. Conclusion

In this paper, we studied the two-staged instance placement problem for stream processing engines. By modeling each stage as a sequential decision-making process and leveraging MCTS to the problem, we proposed MIPS, a randomized scheme that decides the instance placement in two stages in an efficient and timely manner. To promote the sampling quality, we refined MCTS from various aspects and discussed practical issues. To evaluate MIPS against existing schemes, we proposed variants of the schemes in Heron-like systems. Results from extensive simulations show that, with only mild-value of sampling, MIPS outperforms existing schemes with both low traffic and high resource utilization.

References

[1] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Samza: stateful scalable stream processing at linkedin," in *Proceedings of the VLDB Endowment*, 2017.

[2] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," 2015.

[3] D. Le-Phuoc, M. Dao-Tran, M.-D. Pham, P. Boncz, T. Eiter, and M. Fink, "Linked stream data processing engines: Facts and figures," in *Proceedings of International Semantic Web Conference*, 2012.

[4] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, "Storm@twitter," in *Proceedings of ACM SIGMOD*, 2014.

[5] "Heron documentation," https://apache.github.io/incubator-heron/docs.

[6] G. J. Chen, J. L. Wiener, S. Iyer, A. Jaiswal, R. Lei, N. Simha, W. Wang, K. Wilfong, T. Williamson, and S. Yilmaz, "Realtime data processing at facebook," in *Proceedings of ACM SIGMOD*, 2016.

[7] 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 *Proceedings of ACM SIGMOD*, 2015.

[8] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of ACM SoCC*, 2013.

[9] "Nomad," https://www.nomadproject.io/.

[10] J. Xu, Z. Chen, J. Tang, and S. Su, "T-storm: Traffic-aware online scheduling in storm," in *Proceedings of IEEE ICDCS*, 2014.

[11] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-storm: Resource-aware scheduling in storm," in *Proceedings of AMC*, 2015.

[12] M. D. de Assuncao, A. da Silva Veith, and R. Buyya, "Distributed data stream processing and edge computing: A survey on resource elasticity and future directions," *Journal of Network and Computer Applications*, vol. 103, pp. 1–17, 2018.

[13] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of ACM SOSP*, 2013.

[14] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive online scheduling in storm," in *Proceedings of ACM SIGMOD*, 2013.

[15] L. Eskandari, Z. Huang, and D. Eyers, "P-scheduler: adaptive hierarchical scheduling in apache storm," in *Proceedings of ACSW*, 2016.

[16] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proceedings of ACM SIGCOMM*, 2008.

[17] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey, "Jellyfish: Networking data centers, randomly." in *Proceedings of USENIX NSDI*, 2012.

[18] "Docker document," https://docs.docker.com/network/.

[19] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE T-CIAIG*, vol. 4, no. 1, pp. 1–43, 2012.

[20] L. Kocsis, C. Szepesvári, and J. Willemson, "Improved monte-carlo search," *Univ. Tartu, Estonia, Tech. Rep*, vol. 1, 2006.

[21] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of ACM SIGCOMM*, 2010.