Reactive Resource Provisioning Heuristics for Dynamic Dataflows on Cloud Infrastructure

Article *in* IEEE Transactions on Cloud Computing · January 2015

DOI: 10.1109/TCC.2015.2394316

4 authors:

Alok Gautam Kumbhare
University of Southern California
19 PUBLICATIONS **606** CITATIONS
SEE PROFILE

Yogesh Simmhan
Indian Institute of Science
168 PUBLICATIONS **4,970** CITATIONS
SEE PROFILE

Marc Frincu
West University of Timisoara
111 PUBLICATIONS **661** CITATIONS
SEE PROFILE

V. Prasanna
University of Southern California
762 PUBLICATIONS **14,460** CITATIONS
SEE PROFILE

Reactive Resource Provisioning Heuristics for Dynamic Dataflows on Cloud Infrastructure

Alok Gautam Kumbhare, *Student Member, IEEE,* Yogesh Simmhan, *Senior Member, IEEE,* Marc Frincu, *Member, IEEE,* and Viktor K. Prasanna, *Fellow, IEEE*

Abstract—The need for low latency analysis over high-velocity data streams motivates the need for distributed continuous dataflow systems. Contemporary stream processing systems use simple techniques to scale on elastic cloud resources to handle variable data rates. However, application QoS is also impacted by variability in resource performance exhibited by clouds and hence necessitates autonomic methods of provisioning elastic resources to support such applications on cloud infrastructure. We develop the concept of **"dynamic dataflows"** which utilize *alternate tasks* as additional control over the dataflow's cost and QoS. Further, we formalize an optimization problem to represent deployment and runtime resource provisioning that allows us to balance the application's *QoS*, *value*, and the resource *cost*. We propose two greedy heuristics, *centralized* and *sharded*, based on the variable-sized bin packing algorithm and compare against a *Genetic Algorithm (GA)* based heuristic that gives a near-optimal solution. A large-scale simulation study, using the Linear Road Benchmark and VM performance traces from the AWS public cloud, shows that while GA-based heuristic provides a better quality schedule, the greedy heuristics are more practical, and can intelligently utilize cloud elasticity to mitigate the effect of variability, both in input data rates and cloud resource performance, to meet the QoS of fast data applications.

Index Terms—Dataflows, stream processing, cloud, resource management, scheduling, high velocity data, runtime adaptation

1 Introduction

The expansion of ubiquitous virtual and physical sensors, leading up to the Internet of Things, has accelerated the rate and quantity of data being generated continuously. As a result, the need to manage and analyze such high *velocity* data in real-time forms one of the three dimensions of "Big Data", besides *volume* and *variety* [1]. While the past decade has seen sophisticated platforms for scalable offline analytics on large data volumes [2], Big Data systems for continuous analytics that adapt to the number, rate and variability of streams are relatively less well-studied.

There is a growing class of streaming applications in diverse domains: trend analysis and social network modeling for *online advertising* [3], real-time event processing to detect abnormal behavior in *complex systems* [4], and mission-critical use-cases such as *smart traffic* signaling [5] and demand-response in *smart power grids* [6]. These applications are characterized by the variety of input data streams, each with variable data rates. Further, data arrives at high velocity and needs to be analyzed with guaranteed low-latency even in the presence of data rate fluctuations. Hence, such applications lie at the intersection of the velocity and variety dimensions of the Big Data landscape.

While run-time scalability and seamless fault-tolerance together are the key requirements for handling high velocity variable-rate data streams, in this paper, we emphasize on *scalability* of stream processing applications with increasing data velocity and their *adaptation* to fluctuating data rates. Existing stream processing (or continuous dataflow) systems (SPS) [7], [8], [5], [9] allow users to compose applications as task graphs that consume and process continuous data, and execute on distributed commodity clusters and clouds. These systems support scalability with respect to high input data rates over static resource deployments, assuming the input rates are stable. When the input rates change, their static resource allocation causes over- or under-provisioning, resulting in wasted resources during low data rate periods and high processing latency during high data rate periods. Storm's *rebalance* [8] function allows users to monitor the incoming data rates and redeploy the application on-demand across a different set of resources, but requires the application to be paused. This can cause message loss or processing delays during the redeployment. As a result, such systems offer limited *self-manageability* to changing data rates, which we address in this article.

Recent SPSs such as Esc [10] and StreamCloud [11] have harnessed the cloud's elasticity to dynamically acquire and release resources based on application load. However, several works [12], [13], [14] have shown that the performance of public and private clouds themselves vary: for different resources, across the data center, and over time. Such variability impacts low latency streaming applications, and causes adaptation algorithms that assume reliable resource performance to fail. Hence, another gap we address here is autonomic *self-optimization* to respond to cloud performance variability for streaming applications. In addition, the pay-per-use cost model of commercial clouds requires intelligent resource management to minimize the real cost while satisfying the streaming application's quality of service (QoS) needs.

- *A. Kumbhare is with the Department of Computer Science, and V. K. Prasanna and M. Frincu are with Department of Electrical Engineering at University of Southern California, Los Angeles, USA. Email: {kumbhare, prasanna, frincu}@usc.edu*
- *Y. Simmhan is with Supercomputer, Education and Research Centre (SERC) at Indian Institute of Science, Bangalore, India. Email: simmhan@serc.iisc.in*

In this article we push towards autonomic provisioning of continuous dataflow applications to enable scalable execution on clouds by leveraging cloud elasticity and addressing the following issues:

1) Autonomous runtime adaptions in response to fluctuations in both input data rates and cloud resource performance.
2) Offering flexible trade-offs to balance monetary cost of cloud resources against the users' perceived application value.

This article extends our previous work [15], which introduced the notion of "dynamic dataflows" and proposed greedy reactive resource provisioning heuristics (§ 8) to exploit cloud elasticity and the flexibility offered by dynamic dataflows. Our contributions in this article are:

- **Application Model and Optimization Problem:** We develop the application model for **dynamic dataflows** (§ 2) as well as the infrastructure model to represent IaaS cloud characteristics (§ 3), and propose an optimization problem (§ 6) for resource provisioning that balances the resource cost, application throughput and the domain value based on user-defined constraints.
- **Algorithms:** We present a *Genetic Algorithm (GA)*-based heuristic for deployment and runtime adaptation of continuous dataflows (§ 7) to solve the optimization problem. We also propose efficient greedy heuristics (*centralized* and *sharded* variants) that sacrifice optimality over efficiency, which is critical for low latency streaming applications (§ 8).
- **Evaluation:** We extend the *Linear Road Benchmark (LRB)* [16] as a **dynamic dataflow** application, which incorporates dynamic processing elements, to evaluate the reactive heuristics through large-scale simulations of LRB, scaling up to 8,000 msgs/sec, using VM and network performance traces from Amazon AWS cloud service provider. Finally, we offer a *comparative analysis* of the greedy heuristics against the GA in terms of scalability, profit, and QoS (§ 9.2).

2 Dynamic Dataflow Application Model

We leverage the familiar Directed Acyclic Graph (DAG) model to define *Continuous Dataflows* (Def. 1). This allows users to compose loosely coupled applications from individual tasks with data dependencies between them defined as streaming dataflow edges. While, in practice, this model can be extended to include more complex constructs like back flows/cycles, we limit our discussion to DAGs to keep the application model simple and the optimization problem tractable.

Def. 1: A continuous dataflow G is a quadruple $G = \langle P, E, I, O \rangle$, where $P = \{P_1, P_2, ..., P_n\}$ is the set of *Processing Elements (PE)* and $E = \{\langle P_i, P_j \rangle \mid P_i, P_j \in P\}$ is a set of directed *dataflow* edges without cycles such that data *messages* flow from P_i to P_j. $I \neq \varnothing \subset P$ is a set of *input PEs* which receive messages only from external data streams, and $O \neq \varnothing \subset P$ is a set of *output PEs* that emit output messages only to external entities.

Each PE represents a long-running, user-defined task which executes continuously, accepting and consuming messages from its incoming ports and producing messages on the outgoing ports. A directed edge between two PEs connects an output port from the source PE to an input port of the sink PE, and represents a flow of messages between the two. Without loss of generality, we assume *and-split semantics* for edges originating from the same output port of a PE (i.e., output messages on a port are duplicated on all outgoing edges) and *multi-merge semantics* [17] for edges terminating at an input port of another PE (i.e., input messages from all incoming edges on a port are interleaved).

We define *Dynamic Dataflows* (Def. 2) as an extension to continuous dataflows by incorporating the concept of dynamic PEs [15]. Dynamic PEs consists of one or more user-defined alternative implementations *(alternates)* for the given PE, any one of which may be selected as an active alternate at run-time. Each alternate may possess different *performance characteristics*, *resource requirements* and *domain perceived functional quality (value)*. Heterogeneous computing [18] and (batch processing) workflows [19] incorporate a similar notion where the active alternates are *decided once at deployment time* but thereafter remain fixed during execution. We extend this to continuous dataflows where alternate selection is an *on-going process at runtime*. This allows the execution framework to perform autonomic adaptations by dynamically altering the active alternates for an application to meet its QoS needs based on current conditions.

Def. 2 (Dynamic Dataflow): A Dynamic Dataflow $D = \langle \mathcal{P}, E, \mathcal{I}, \mathcal{O} \rangle$ is a continuous dataflow where each PE $\mathcal{P}_i \in \mathcal{P}$ has a set of alternates $\mathcal{P}_i = \{p_i^1, p_i^2, ...p_i^j \mid j \geq 1\}$ where $p_i^j = \langle \gamma_i^j, c_i^j, s_i^j \rangle$. γ_i^j, c_i^j, and s_i^j denote the *relative value*, the *processing cost* per message, and the *selectivity* for the alternate p_i^j of PE $\mathcal{P}_i$ respectively.

Selectivity, s_i^j, is the ratio of the number of output messages produced to the number of input messages consumed by the alternate p_i^j to complete a logical unit of operation. It helps determine the outgoing data rate of a PE relative to its input data rate, and thereby its cascading impact on downstream PEs in the dataflow.

Each alternate has associated cost and value functions to assist with alternate selection and resource provisioning decisions. The *relative value*, $0 < \gamma_i^j \leq 1$, for an alternate p_i^j is:

$$\gamma_i^j = \frac{f(p_i^j)}{\text{MAX}_j\{f(p_i^j)\}} \tag{1}$$

where $f : \mathcal{P}_i \to \mathbb{R}$ is a user-defined *value function* for the alternates. It quantifies the relative domain benefit to the user of picking that alternate. For e.g., a Classification PE that classifies its input tuples into different classes may use the F_1 score[1] as the quality of that algorithm to the domain, and F_1 can be used to calculate the *relative value* for alternates of the PE.

1. $F_1 = 2 \times \frac{precision \times recall}{precision + recall}$ is a measure of the classifier's labeling accuracy.

```
<dataflow>
 <PE id="parser">
  <in id="tweet" type="string" />
  <out id="cleaned" type="string" />
  <alternate impl="Parser.TweetParser" />
 </PE>
 <PE id="classifier">
  <in id="cleaned_twt" type="string" />
  <out id="tpc_twt" type="string" />
  <alternate impl="Classifier.Bayes" value="0.65" />
  <alternate impl="Classifier.LDA" value="0.70" />
  <alternate impl="Classifier.MWE" value="1.00" />
 </PE>

 <edge source="parser:cleaned"
       sink="classifier:cleaned_twt" />
</dataflow>
```

Fig. 1: Sample declarative representation of Dynamic Dataflow using XML. Equivalent visual representation is in Fig. 2(a).

Finally, the *processing cost* per message, c_i^j, is the time (in seconds) required to process one message on a "reference" CPU core (§ 3) for the alternate p_i^j. The processing needs of an alternate determines the resources required to processes incoming data streams at a desired rate.

The concept of dynamic PEs and *alternates* provides a powerful abstraction and an additional point of control to the user. A sample dynamic dataflow is shown in Fig. 1 using a generic XML representation, with a visual equivalent shown in Fig. 2(a). Any existing dataflow representation, such as declarative [20], functional [8] or imperative [9] may also be used. The sample dataflow continuously parses incoming tweets, and classifies them into different topics. It consists of two PEs: *parser*, and *classifier*, connected using a dataflow edge. While the *parser* PE consists of only one implementation, the *classifier* PE consists of three alternates, using the Bayes, Latent Dirichlet Allocation (LDA) and Multi-Word enhancement (MWE) to LDA algorithms, respectively. Each alternate varies in classification accuracy and hence has different *value* to the domain; these are normalized relative to the best among the three. The three alternates are available for dynamic selection at runtime. For brevity, we omit a more detailed discussion of the dynamic dataflow programming abstraction.

The execution and scalability of a dynamic dataflow application depends on the capabilities of the underlying infrastructure. Hence, we develop an infrastructure model to abstract the characteristics that impacts the application execution and use that to define the resource provisioning optimization problem (§ 6).

3 Cloud Infrastructure Model

We assume an *Infrastructure as a Service (IaaS)* cloud that provides access to virtual machines (VMs) and a shared network. In IaaS clouds, a user has no control over the VM placement on physical hardware, the multi-tenancy, or the network behavior between VMs. The cloud environment provides a set of *VM resource classes* $\mathcal{C} = \{C_1, C_2, ..., C_n\}$ that differ in the number of available virtual CPU cores N, their *rated core speed* π, and their *rated network bandwidth* β. In this article, we focus on CPU bound PEs that operate on incoming data streams from the network. As a result, we ignore memory and disk characteristics and only use the VM's CPU and network behavior in the infrastructure performance model used by our adaptation heuristics.

As CPU core speeds may vary across VM classes, we define the *normalized processing power* π_i of a resource class C_i's CPU core as the ratio of its processing power to that of a *reference* VM core. Naïvely, this may be the ratio of their clock speeds, but could also be obtained by running application benchmarks on different sets of VMs and comparing them against a defined reference VM, or use Cloud-providers' "ratings" such as Amazon's Elastic Compute Units (ECUs).

The set of *VM resources acquired* till time t is denoted by $R(t) = \{r_1, r_2, ..., r_n\}$. Each VM is described by $r_i = \langle C_j^i, t_{start}^i, t_{stop}^i \rangle$ where C_j^i is the resource class to which the VM belongs, and t_{start}^i and t_{stop}^i are the times at which the VM was acquired and released, respectively. $t_{stop} = \infty$ for an active VM.

The peer-to-peer network characteristic between pairs of VMs, r_i and r_j, is given by $\lambda_{i\times j}$ and $\beta_{i\times j}$, where $\lambda_{i\times j}$ is the *network latency* between VM r_i and r_j and $\beta_{i\times j}$ is their *available bandwidth*.

VMs are typically charged at whole VM-hours by current cloud providers. The user is billed for the entire hour even if a VM is released before an hour boundary. The *total accumulated monetary cost* for the VM r_i at time t is then calculated as:

$$\mu_i(t) = \lceil \frac{\min(t_{stop}, t) \; - \; t_{start}}{60} \rceil \times cost \; per \; VM \; hour \quad (2)$$

where $\min(t_{stop}, t) - t_{start}$ is the duration in minutes for which the VM has been active.

We gauge the on-going performance of virtualized cloud resources, and the variability relative to their rated capability, using a presumed monitoring framework. This periodically probes the compute and network performance of VMs using standard benchmarks. The *normalized processing power* of a VM r_i observed at time t is given by $\pi_i(t)$, and the *network latency and bandwidth* between pairs of active VM r_i and r_j are $\lambda_{i\times j}(t)$ and $\beta_{i\times j}(t)$, respectively. To minimize overhead, we only monitor the network characteristics between VMs that host neighboring PEs in the DAG to assess their impact on dataflow throughput. We assume that rated network performance as defined by the provider is maintained for other VM pairs. Two PEs collocated in the same VM are assumed to transfer messages in-memory, i.e., $\lambda_{i\times i} \to 0$ and $\beta_{i\times i} \to \infty$.

4 Deployment and Adaptation Approach

Based on the dynamic dataflow and cloud infrastructure models, we propose a deployment and autonomic runtime adaptation approach that attempts to balance *simplicity*, realistic *cloud characteristics* (e.g., billing model, elasticity), and *user flexibility* (e.g., dynamic PEs). Later, we formally define a meaningful yet tractable optimization problem for the deployment and runtime adaptation strategies (§ 6).

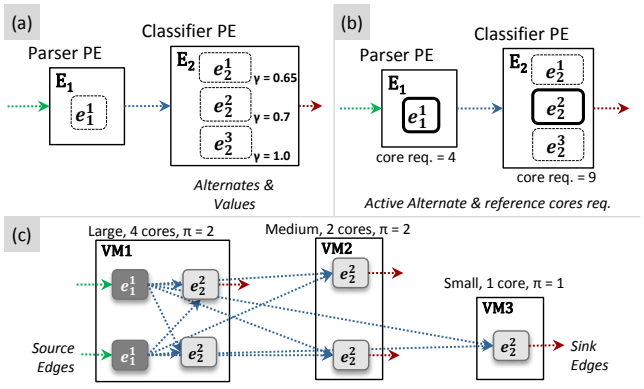


Fig. 2: (a) A sample dynamic dataflow. (b) Dataflow with selected alternates (e_1^1, e_2^2) and their initial core requirements. (c) A deployment of the dataflow onto VMs.

We make several practical assumptions on the continuous dataflow processing framework, reflecting features available in existing systems [21], [8]:

1) The dataflow application is deployed on distributed machines, with a *set of instances of each PE* ($\varphi(t)$ at time t) running in parallel across them. Incoming messages are actively load-balanced across the different PE instances based on the processing power of the CPU cores they run on and the length of the pending input queue. This allows us to easily scale the application by increasing the number of PE instances if the incoming load increases.
2) Within a single multi-core machine, multiple instances of different PEs run in parallel, isolated on separate cores. A core is exclusively allocated to one PE instance. We assume that there is minimal interference from system processes.
3) Running n data-parallel instances of a PE on n CPU cores, each with *processing power* $\pi = 1$, is equivalent to running 1 instance of the PE on 1 CPU core with $\pi = n$.
4) The active alternate for a PE is not dependent on the active alternate of any other PE, since each alternate for a given PE follows the same input/output format. This allows us to independently switch the active alternate for different PEs during runtime.
5) The framework can spin up or shutdown cloud VMs on-demand (with an associated startup/shutdown latency) and can periodically monitor the VM characteristics, such as CPU performance and network bandwidth.

Given these assumptions we define the following deployment model. When a dynamic dataflow, such as Fig. 2(a), is submitted for execution, the *scheduler* for the stream processing framework needs to make several decisions: alternate selection for each dynamic PE, acquisition of VMs, mapping of these PEs to the acquired VMs, and deciding the number of data parallel instances per PE. These activities are divided into two phases: *deployment time* and *runtime* strategies.

Deployment time strategies select the initial active alternate for each PE, and determine their CPU core requirements (relative to the "reference" core) based on *estimated* initial message data rates and rated VM performance. Fig. 2(b) shows the outcome of selecting alternates, picking e_1^1 and e_2^2 for PEs E_1 and E_2 with their respective core requirements. Further, it determines the VMs of particular resource classes that are instantiated, and the mapping $\mathcal{M}$ from the data-parallel instances of the each PE $(\varphi(t))$ to the active VMs $(R(t))$, following which the dataflow execution starts. Fig. 2(c) shows multiple data-parallel instances of these PEs deployed on a set VMs of different types. Note that the number of PE instances in Fig. 2(c) – $\varphi(t)$ is 2 and 5 for e_1^1 and e_2^2 – is not equal to the core requirements in Fig. 2(b) – 4 and 9 cores – since some instances are run on faster CPU cores ($\pi > 1$).

Runtime strategies, on the other hand, are responsible for periodic adaptations to the application deployment in response to the variability in the input data rates and the VM performance obtained from the monitoring framework. These active decisions are determined by the runtime heuristics that can decide to switch the active alternate for a PE, or change the resources allocated to a PE within or across VMs. The acquisition and release of VMs are also tied to these decisions as they determine the actual cost paid to the cloud service provider. A formal definition of the optimization problem and these control strategies will be presented in § 6.

5 Metrics for Quality of Service

In § 2, we captured the value, and processing requirements for individual PEs and their alternates using the metrics: *relative value* (γ_i^j), *alternate processing cost* (c_i^j), and *selectivity* (s_i^j). In this section, we expand these QoS metrics to the entire dataflow application.

We define an *optimization period* T for which the dataflow is executed. This optimization period is divided into *time intervals* $T = \{t_0, t_1, ..., t_n\}$. We assume these *interval lengths* are constant, $\triangle t = t_{i+1} - t_i$. For brevity we omit the suffix i while referring to the time interval t_i, unless necessary for disambiguation.

A dataflow is initially deployed with a particular configuration of alternates which can later be switched during runtime to meet the application's QoS. To keep the problem tractable and avoid repetitive switches, these changes are only made at the start of each interval t_i. This means that during a time interval t, only a specific alternate for a PE $\mathcal{P}_i$ is active. The *value* of the PE $\mathcal{P}_i$ during the time interval t is thus:

$$\Gamma_i(t) = \sum_{p_i^j \in P_i} \left(A_i^j(t) \cdot \gamma_i^j \right)$$

$$A_i^j(t) = \begin{cases} 1, & \text{if alternate } p_i^j \text{ is active at time } t \\ 0, & \text{otherwise} \end{cases}$$

Since value can be perceived as an additive property [22] over the dataflow DAG, we aggregate the individual values of active alternates to obtain the value for the entire dynamic dataflow during the time interval t.

Def. 3 (Normalized Application Value): The *normalized application value*, $0 < \Gamma(t) \leq 1$, for a dynamic dataflow D during the time interval t is:

$$\Gamma(t) = \frac{\sum_{\mathcal{P}_i \in \mathcal{P}} \Gamma_i(t)}{|\mathcal{P}|} \tag{3}$$

where $|\mathcal{P}|$ is the number of PEs in the dataflow.

The application's value thus obtained gives an indication of its overall quality from the domain's perspective and can be considered as one QoS dimension.

Another QoS criterion, particularly in the context of continuous dataflows, is the observed application throughput. However, raw application throughput is not meaningful because it is a function of the input data rates during that time interval. Instead we define the *relative application throughput*, Ω, built up from the *relative throughput* $0 < \Omega_i(t) \leq 1$ of individual PEs $\mathcal{P}_i$ during the interval t. These are defined as the ratio of the PEs' current output data rate (absolute throughput) $o_i(t)$ to the maximum achievable output data rate $o_i^{max}(t)$: $\Omega_i(t) = \frac{o_i(t)}{o_i^{max}(t)}$

The output data rate for a PE depends on the selectivity of the active alternate, and is bound by the total resources available to the PE to process the inputs given the processing cost of the active alternate. The actual *output data rate* during the interval t is:

$$o_i(t) = \frac{\min\left(q_i(t) + i_i(t) \cdot \Delta t, \quad \frac{\phi_i \cdot \Delta t}{c_i^j}\right) \times s_i^j}{\Delta t} \tag{4}$$

where $q_i(t)$ is the *number of pending input messages* in the queue for PE $\mathcal{P}_i$, $i_i(t)$ is the *input data rate* in the time interval t for the PE , ϕ_i is its *total core allocation* for the PE ($\phi_i = \sum_k \pi_k$), and c_i^j is the processing cost per message and s_i^j is the selectivity for the active alternate p_i^j.

The *maximum output throughput* is achieved when there are enough resources for $\mathcal{P}_i$ to process all incoming data messages including messages pending in the queue at the start of the interval t. This is given by $o_i^{max}(t) = \frac{(q_i(t)+i_i(t)\times\Delta t)\times s_i^j}{\Delta t}$.

While the input data rate for the source PE is determined externally, the input rate for other PEs can be characterized as follows. The flow of messages between consecutive PEs is limited by the bandwidth, during the interval t, between the VMs on which the PEs are deployed. We define the *flow* $f_{i,j}(t)$ from $\mathcal{P}_i$ to $\mathcal{P}_j$ as:

$$f_{i,j}(t) = \begin{cases} \min\left(o_i, \quad \frac{\beta_{i,j}(t)\cdot\triangle t}{m}\right), & \langle \mathcal{P}_i, \mathcal{P}_j \rangle \in E \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

where $\beta_{i,j}(t)$ is the available cumulative bandwidth between all instances of $\mathcal{P}_i$ and $\mathcal{P}_j$ at time t and m is the *average output message size*.

Given the *multi-merge semantics* for incoming edges, the input data rate for $\mathcal{P}_k$ during time t is:

$$i_k(t) = \sum f_{j,k}(t) \tag{6}$$

Unlike the application's value, its relative throughput is not additive as it depends on the critical processing path of the PEs during that interval. The relative throughput for the entire application is the ratio of observed cumulative outgoing data rate from the output PEs, $\mathcal{O} = \{\mathcal{O}_i\}$, to the maximum achievable output rate for those output PEs, for the current input data rate.

Def. 4 (Relative Application Throughput): The *relative application throughput*, $0 < \Omega(t) \leq 1$, for dynamic dataflow $D = \langle \mathcal{P}, E, \mathcal{I}, \mathcal{O} \rangle$ during the time interval t is:

$$\Omega(t) = \frac{\sum_{\mathcal{P}_i \in \mathcal{O}} \Omega_i(t)}{|\mathcal{O}|} \tag{7}$$

The output data rate for the output PEs is obtained by calculating the output data rate of individual PEs (Eqn. 4) followed by the flow $f_{i,j}$ between consecutive PEs (Eqn. 5), repeatedly in a breadth-first scan of the DAG starting from its input PEs, till the input and output data rates for the output PEs is obtained.

Normalized Application Value, $\Gamma(t)$, and *Relative Application Throughput*, $\Omega(t)$, together provide complimentary QoS metrics to assess overall application execution, which we use to define an optimization problem that balances these QoS metrics based on user-defined constraints in the next section.

6 Problem Formulation

We formulate the optimization problem as a *constrained utility maximization* problem during the period T for which the dataflow is executed. The *constraint* ensures that the expected relative application throughput meets a threshold, $\Omega \geq \widehat{\Omega}$; the *utility* to be *maximized* is a function of the normalized application value, Γ; and the cost for cloud resources, μ, during the optimization period T.

For a dynamic dataflow $D = \langle \mathcal{P}, E, \mathcal{I}, \mathcal{O} \rangle$, the *estimated* input data rates, $I(t_0) = \{i_j(t_0)\}$, at each input PE, $\mathcal{P}_j \in \mathcal{I}$, at initial time, t_0, is given. During each subsequent time interval, t_i, based on the observations of the monitoring framework during t_{i-1}, we have the following: the *observed* input data rates, $I(t) = \{i_j(t)\}$; the set of active VMs, $R(t) = \{r_1, r_2, ..., r_m\}$; the normalized processing power per core for each VM r_j, $\pi(t) = \{\pi_j(t)\}$; the network latency and the bandwidth between pairs of VMs $r_i, r_j \in R(t)$ hosting neighboring PEs are $\lambda(t) = \{\lambda_{i\times j}(t)\}$ and $\beta(t) = \{\beta_{i\times j}(t)\}$, respectively.

At any time interval t, we can calculate the *relative application throughput* $\Omega(t)$ (Eqn. 7), the *normalized application value* $\Gamma(t)$ (Eqn. 3), and the cumulative monetary cost $\mu(t)$ till time t (Eqn. 2).

The *average* relative application throughput (Ω), the *average* relative application value (Γ), and the *total* resource cost (μ) for the entire optimization period $T = \{t_0, t_1, \ldots, t_n\}$ are:

$$\Omega = \frac{\sum_{t\in T} \Omega(t)}{|T|} \qquad \Gamma = \frac{\sum_{t\in T} \Gamma(t)}{|T|} \qquad \mu = \mu(t_n)$$

We define the combined *utility* as a function of both the total resource cost (μ) and the average application value (Γ). To help the users to trade-off between cost and value, we allow them to define the expected *maximum cost* at which they break-even for the two extremes of application value, i.e., the values obtained by selecting the *best alternates* for all PEs, on one end, and by selecting the *worst alternates*, on the other. For simplicity, we assume a linear function to derive the expected maximum resource cost at an intermediate application value, as

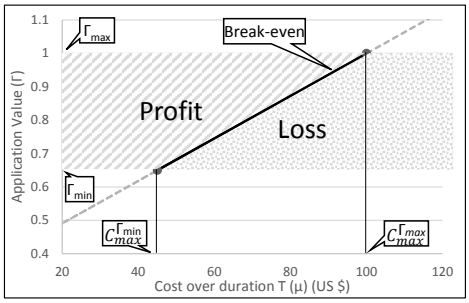


Fig. 3: A sample linear function for trade-off between cost (C) and value (Γ). The line denotes break-even, with a slope = σ.

shown in Fig. 3. If the actual resource cost lies below this break-even we consider it as profit, while if it lies above, we consider it as loss. This can be captured using the following *objective function*, Θ , which is to be maximized over the optimization period under the constraint $\Omega \geq \widehat{\Omega}$:

$$\Theta = \Gamma - \sigma \cdot (\mu \ - C_{max}^{\Gamma_{max}}) \ + \ 1 \tag{8}$$

where σ is a *equivalence coefficient* between cost and value given by the slope:

$$\sigma = \frac{\Gamma_{max} - \Gamma_{min}}{C_{max}^{\Gamma_{max}} - C_{max}^{\Gamma_{min}}} \tag{9}$$

Γ_{max} and Γ_{min} are the maximum and minimum possible relative application values when picking the alternates with the best and worst values for each PE, respectively, while $C_{max}^{\Gamma_{max}}$ and $C_{max}^{\Gamma_{min}}$ are the user-defined break-even resource cost at Γ_{max} and Γ_{min}.

Given the deployment approach (§ 4), the above objective function Θ can be maximized by choosing appropriate values for the following *control parameters* at the start of each interval t_i during the optimization period:

- $A_i^j(t)$, the active alternate j for the PE P_i
- $R(t) = \{r_j(t)\}$, the set of VMs in $R(t)$
- $\varphi(t) = \{\varphi_j(t)\}$, the set of data-parallel instances for each PE P_j; and
- $\mathcal{M}(t) = \{\varphi_j(t) \rightarrow \mathcal{M}_{j \times k}(t)\}$, the mapping of a data-parallel instances φ_j for PE P_j to the actual VM r_k

Optimally solving the objective function Θ with the Ω constraint is *NP-Hard*. The proof is outside the scope of this article and a sketch is presented in our earlier work [15]. While techniques like integer programming and branch-and-bound have been used to optimally solve some NP-hard problems [23], these do not adequately translate to low-latency solutions for continuous adaptation decisions. The dynamic nature of the application and the infrastructure, as well as the tightly-bound decision making interval means that fast heuristics performed repeatedly are better than slow optimal solutions. We thus propose simplified heuristics to provide an approximate solution to the objective function.

Other approximate procedures such as gradient descent are not directly applicable to the problem at hand since the optimization problem presents a non-differentiable, non-continuous function. However, nature-inspired search algorithms such as Genetic Algorithms (GAs), ant-colony optimization, and particle-swarm optimization, which follow a guided randomized search, are sometimes more effective than traditional heuristics in solving similar single and multi-objective workflow scheduling problems [24], [25], [26], [27]. In this article, we also explore GAs for finding an approximate solution to the optimization problem. While GAs are usually slow and depend on the population size and complexity of the operators used, they provide a good baseline to compare against our greedy heuristics and are discussed in the next section, followed by the greedy deployment and adaptation heuristics in section 8.

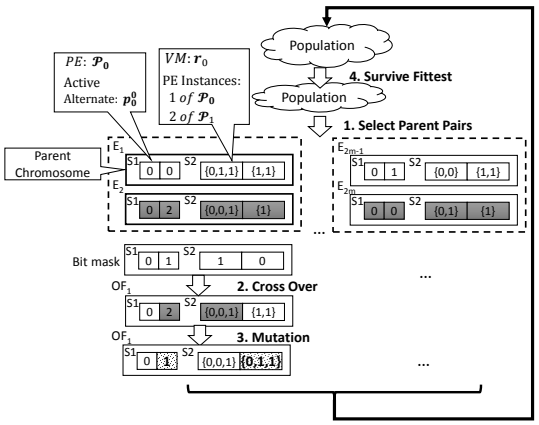


Fig. 4: Sample iteration for the GA heuristic.

7 Genetic Algorithm-based Scheduling

A GA [28] is a meta-heuristic used in optimization and combinatorial problems. It facilitates the exploration of a large search space by iteratively evolving a number of candidate solutions towards the global optimum. The GA meta-heuristic abstracts out the structure of the solution for a given problem in terms of a *chromosome* made up of several *genes*. It then explores the solution space by evolving a set of chromosomes (potential solutions) over a number of generations.

The GA initially generates a random population of chromosomes which act as the seed for the search. The algorithm then performs genetic operations such as *crossover* and *mutations* to iteratively obtain successive generations of these chromosomes. The *crossover* operator takes a pair of parent chromosomes and generates an *offspring* chromosome by crossing over individual genes from each parent. This helps potentially combine partial solutions from the parents into a single offspring. Further, the *mutation* operator is used to randomly alter some parts of a given chromosomes and advance the search by possibly avoiding getting stuck in a local optimum. The algorithm then applies a *selection* operator which picks the best chromosomes from the entire population based on their *fitness values* and eliminates the rest. This process is repeated until a *stopping criterion*, such as a certain number of iterations or the convergence of a fitness value, is met.

We adapt GA to our optimization problem by defining these domain-specific data structures and operators.

Chromosome: The solution to the optimization problem (Eqn. 8) requires (1) determining the best alternate to activate for each PE, and (2) the type and number of

VMs, and the mapping of data-parallel PE instances to these VMs. We capture both of these aspects of the deployment configuration using a double stranded *chromosome* (Fig. 4). Each chromosome represents one possible deployment configuration and the goal is to find the chromosome with the optimal configuration. The first strand (S1) represents the PEs and their active alternates, with each gene in the strand representing a PE $\mathcal{P}_i$ in the dataflow The value of the i^{th} gene holds the index j of the active alternate p_i^j for the corresponding PE. The second strand (S2) contains genes that represent the list of available VMs with the list of PE instances running on them. The value of the i^{th} gene in strand S2 holds the list of index values j for PEs $\mathcal{P}_j$ mapped to the VM r_i. The chromosome size (i.e. number of genes) is fixed based on the available resource budget.

For example, in Fig 4, the chromosome E_1 represents a deployment where strand S1 has two PEs $\mathcal{P}_0$ and $\mathcal{P}_1$ with active alternates p_0^0 and p_1^0, respectively. Strand S2 identifies two VMs, r_0 and r_1, with r_0 running one instance of PE $\mathcal{P}_0$ and two instances of PE $\mathcal{P}_1$ on it, while r_1 has two instances of PE $\mathcal{P}_1$ running on it.

Fitness Function: The objective function Θ acts as the fitness function for the chromosome, and depends on both the strands of the chromosome. The throughput constraint $\Omega \geq \widehat{\Omega}$ is incorporated using a weighted *penalty function* that penalizes the fitness value if a constraint is violated. This ensures that while the chromosome is penalized, it is not disregarded immediately and given a chance to recover through mutations. In addition, during runtime, the penalty function also considers the number of changes induced on the deployment to reduce the overhead of frequent changes in the system.

$$penalty = \begin{cases} -\alpha \cdot \mid \Omega - \widehat{\Omega} \mid \cdot it & \text{, deployment} \\ -\alpha \cdot \mid \Omega - \widehat{\Omega} \mid \cdot it + \alpha' \cdot v \cdot it & \text{, runtime} \end{cases}$$

where, α and α' are constants, v is the number of deployment changes observed in the chromosome as compared to the previous deployment and it is the iteration count for GA. it ensures that the penalty is increased as the chromosome survives over multiple iterations and hence allows removal of unfit chromosomes.

Crossover: Each parent chromosome is first selected from the population using a probabilistic ranked model (similar to the Roulette wheel) [29] while also retaining the top 5% of the chromosomes with best fitness values. Next the parent chromosomes are paired randomly and a random bit mask is generated to choose the a gene from either parent to produce the offspring chromosome. For e.g., in Fig. 4, parents E_1 (white) and E_2 (gray) are selected for crossover, and a random bit mask is used to decide if the gene from the first parent (bit is 0) or the second parent (bit is 1) is retained in the offspring OF_1.

Mutation: We allow independent mutation for the two chromosome strands. For the first strand with PE alternates the mutation involves randomly switching the active alternate. For the second strand of VM instances and mapping, we probabilistically decide whether to remove or add a PE instance for each VM. For example, in Fig. 4, the offspring OF_1 undergoes mutation by switching the active alternate for $\mathcal{P}_1$ from $p_1^2 \rightarrow p_1^1$, and by adding an instance of $\mathcal{P}_0$ to the second VM. Mutated genes are shown with a dotted pattern.

While GAs explore a wide range of solutions and tend to give near-optimal solutions, their convergence performance becomes a bottleneck during runtime adaptation. Hence we design sub-optimal greedy heuristics that trade optimality for speed, making them better suited for streaming applications.

8 Greedy Deployment & Adaptation Heuristics

In this section, we propose greedy heuristics to find an approximate solution to the optimization problem. As before, the algorithm is divided into the initial deployment phase and the runtime adaptation phase.

For the proposed heuristic, we provide *sharded* (SH) and *centralized* (CE) variants that differ in the quanta of information needed and the execution pattern of the scheduler. The *sharded* version uses one scheduler per PE, and all data-parallel instances of a PE communicate with their scheduler, potentially across VMs. However, schedulers for different PEs do not communicate. Hence each scheduler only has access to its PE instances. In the *centralized* version, a single scheduler gathers information about the entire dataflow and hence has a global view of the execution of all PEs. As we show (§ 9.2), while the SH scheduler is inherently decentralized and reduces the transfer of monitoring data during execution, the CE variant, due to its global view, is more responsive to changes in the execution environment.

8.1 Initial Deployment Heuristic

The initial deployment algorithm (Alg. 1) is divided into two stages: Alternate selection (lines 2–11) and Resource allocation (lines 12–25). These algorithms are identical for both SH and CE schedulers; however, their costing functions differ (Table 1).

The alternate selection stage ranks each PE alternate based on the ratio of its value to estimated cost (line 4), and chooses the one with the highest ratio. Since we do not know the actual cost for the alternates until resource allocation, the heuristic uses the estimated processing requirements (c_P^A) as an approximation. The GetCostOfAlternate function varies between the SH and CE versions. The SH strategy calculates an alternate's cost based on only its processing requirements, while the CE strategy calculates the cost of the alternate as the sum of both its own processing needs and that of its downstream PEs – intuitively, if an upstream PE has more resources allocated, its output message rate increases and this has a cascading impact on the input rate of the succeeding PEs. Also, a higher selectivity upstream PE will further impact the successors' cost since they will have to process more messages. This cost is calculated using a dynamic programming algorithm by traversing

Algorithm 1 Initial Deployment Heuristic Algorithm

```
1: procedure InitialDeployment(Dataflow D)
                                          ▷ Alternate Selection Stage
2:     for PE P ∈ D do
3:         for Alternate A ∈ P do
4:             c_P^A ← GetCostOfAlternate(A)
5:             γ ← A.Value
6:             if γ/c_P^A ≥ best then
7:                 best ← γ/c_P^A
8:                 selected ← A
9:             end if
10:        end for
11:    end for
                                          ▷ Resource Allocation Stage
12:    while Ω ≤ Ω̂ do
13:        if (VM.isAvailable = false) then
14:            VM ← InitializeVM(LargestVMClass)
15:        end if
16:        P ← GetNextPE
17:        CurrentAlloc ← AllocateNewCore(P, VM)
18:        Ω ← GetEstimatedThruput(D,CurrentAlloc)
19:    end while
20:    for PE P ∈ D do
21:        if IsOverProvisioned(P) then
22:            RepackPEInstances(P)  ▷ Move PE instances to a VM
   with lower core capacity
23:        end if
24:    end for
25:    RepackFreeVMs  ▷ Repack PEs in VMs with free cores to VMs
   with less number of cores
26: end procedure
```

the dataflow graph in reverse BFS order rooted at the output PEs.

This is followed by a resource selection stage (lines 12–25) which operates similar to the *variable-sized bin packing (VBP)* problem [30]. For the initial deployment, in the absence of running VMs, we assume that each VM from a resource class behaves ideally as per its rated performance. The algorithm picks PEs (objects) in an order given by GetNextPE, and puts an instance of each PE in the largest VM (bin) (line 17), creating a new VM (bin) if required. It then calculates the estimated relative throughput for the application given the current allocation (line 18) using Eqn. 7 and repeats the procedure if the application constraint is not met. It should be noted that the GetEstimatedThroughput function considers both the allocated cores and the available bandwidth to calculate the relative throughput and hence scales out when either becomes a bottleneck.

The intuition behind GetNextPE is to choose PEs in an order that not only increases VM utilization but also limits the message transfer latency between the PEs by *collocating* neighboring PEs in the dataflow within the same VM. We order the PEs using a forward DFS traversal, rooted at the input PEs, and allocate resources to them in that order so as to increase the probability of collocating neighboring PEs. It should be noted that the CPU cores required for the individual PEs are not known in advance as the resource requirements depend on the current load which in turn depends on the resource requirements of the preceding PE. Hence, after assigning at least one CPU core to each PE (IncrementAllocation), the deployment algorithm chooses PEs in the order of largest bottlenecks in the dataflow, i.e., lowest relative PE throughput (Ω_i). This ensures that PEs needing more resources are chosen first for allocation. This in turn may increase the input rate (and processing load) on the successive PEs, making them the bottlenecks. As a result, we end up with an iterative approach to incrementally allocate CPU cores to PEs using the *VBP* heuristic until the throughput constraint is met. Since the resource allocation only impacts downstream PEs, this algorithm is bound to converge. We leave a theoretical proof to future work.

TABLE 1: Functions used in Initial Deployment Strategies

Function	Sharded (SH)	Centralized (CE)
GetCostOf-Alternate	$A.cost$	$A.cost + S_i \times \sum successor.cost$
GetNextPE	**if** All PEs assigned **then** return $\underset{P_j \in \mathcal{P}}{\operatorname{argmin}}(\Omega_t^j)$ **else** **return** Next PE in DFS **end if**	
RepackPEInstances	N/A	Move PE instance to smallest VM big enough for required core-secs
RepackFreeVMs	N/A	Iterative Repacking [30]

At the end, the algorithm performs two levels of repacking. After a solution is obtained using VMs from just the largest resource class, we first move one instance for all the over-provisioned PEs to the smallest resource class large enough to accommodate that PE instance (best fit, using RepackPEInstances). This may free up capacity on the VMs, and hence, we again use iterative repacking [30] (RepackVMs) to repack all the VMs with spare capacity to minimize wasted cores. During this process, we may sacrifice instance collocation in favor of reduced resource cost. Our evaluation however shows that this is an acceptable trade-off toward maximizing the objective function. Note that these algorithms are all performed off-line, and the actual deployment is carried out only after these decision are finalized.

Both, the order in which PEs are chosen and the repacking strategy affects the quality of the heuristic. While the sharded strategy SH uses a local approach and does not perform any repacking, the centralized strategy CE repacks individual PEs and VMs, as shown in Table 1.

8.2 Runtime Adaptation Heuristic

The runtime adaptation kicks in periodically over the lifetime of the application execution. Alg. 2 considers the current state of the dataflow and cloud resources – available through monitoring – in adapting the alternate and resource selection. The monitoring gives a more accurate estimate of data rates, and hence the resource requirements and its cost.

As before, the algorithm is divided into two stages: Alternate selection and Resource allocation. However, unlike the deployment heuristic, we do not run both the stages at the same time interval. Instead, the alternates are selected every m intervals and the resources reallocated every n intervals. The former tries to switch alternates to achieve the throughput constraint given

the existing allocated resources, while the latter tries to balance the resources (i.e. provision new VMs or shutdown existing ones) given the alternates that are active at that time. Separating these stages serves two goals. First, it makes the algorithm for each stage more deterministic and faster since one of the parameters is fixed. Second, it reduces the number of retractions of deployment decisions occurring in the system. For e.g., if a decision to add a new VM leads to over-provisioning at a later time (but before the hourly boundary), instead of shutting down the VM, the alternate selection stage can potentially switch to an alternate with higher value, thus utilizing the extra resources available and in the process increase the application's value.

During the alternate selection stage, given the current data rate and resource performance, we first calculate the resources needed for each PE alternate (line 6). We then create a list of "feasible" alternates for a given PE, based on whether the current relative throughput is lesser or greater than the expected throughput $\widehat{\Omega}$. Finally, we sort the feasible alternates in decreasing order of the ratio between value to cost, and select the first alternate which can be accommodated using the existing resource allocation. After this phase the overall value either increases or decreases depending on whether the application was over-provisioned or under-provisioned to begin with, respectively.

The RESOURCEREDEPLOY procedure is used to allocate or de-allocate resources to maintain the required relative throughput while minimizing the overall cost. If the $\Omega \leq \widehat{\Omega} - \epsilon$, the algorithm proceeds similar to the initial deployment algorithm. It incrementally allocates additional resources to the bottlenecks observed in the system and repacks the VMs. However, if $\Omega > \widehat{\Omega} + \epsilon$, the system must scale in to avoid resource wastage and has two decisions to make: first, which PE needs to be scaled in and second, which instance of the PE is to be removed, thus freeing the CPU cores. The over-provisioned PE selected for scale in is the one with the maximum relative throughput (Ω). Once the over-provisioned PE is determined, to determine which instance of that PE should be terminated, we get the list of VMs on which these instances are running and then weigh these VMs using the following "weight" function (eqn 10). Finally, a PE instance which is executing on the least weighted VM is selected for removal.

$$VM\ Weight(r_i) = \begin{array}{l} T_c(r_i) \times \left(\frac{FreeCores(r_i)}{TotalCores(r_i)}\right) \\ \times(1 - \frac{\varphi_{r_i}}{\varphi}) \times \left(\frac{TotalCores(r_i) \times \pi}{Cost\ Per\ VM\ Hour}\right) \end{array} \quad (10)$$

where T_c is the time remaining in the current cost cycle (i.e. time till the next hourly boundary), φ_{r_i} is the number of PE instances for the over-provisioned PE on the VM r_i, and φ is the total number of instances for that PE across all VMs. The VM Weight is lower for VMs with less time left in their hourly cycle, and thus preferred for removal. This increases temporal utilization. Similarly, VMs with fewer cores used are prioritized for removal. Further, VMs with higher cost per normalized core have a lower weight so that they are selected first for shutdown. Hence the VM Weight metric helps us pick the PE instances in a manner that can help free up costlier, under-utilized VMs that can be shutdown at the end of their cost cycle to effectively reduce the resource cost.

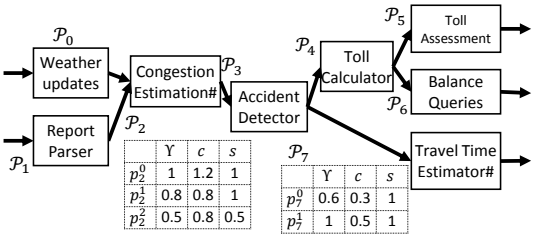
Algorithm 2 Runtime Adaptation Heuristic Algorithm

```
1: procedure AlternateReDeploy(DataflowD, Ω_t)   ▷ Ω_t is the observed relative throughput
2:     for PE P ∈ D do                            ▷ Alternate selection phase
3:         alloc ← CurrAllocatedRes(P)            ▷ Gets the current allocated resources (accounting for Infra. variability)
4:         requiredC ← RequiredRes(P.activeAlternate)   ▷ Gets the required resources for the selected alternate
5:         for AlternateA ∈ P do
6:             requiredA ← ActualResRequirements(A)
7:             if Ω_t ≤ Ω̂ − ε then
8:                 if requiredA ≤ requiredC then   ▷ Select alternate with lower requirements
9:                     feasible.Add(A)
10:                end if
11:            else if Ω_t ≥ Ω̂ + ε then
12:                if requiredA ≥ requiredC then   ▷ Select alternate with higher requirements
13:                    feasible.Add(A)
14:                end if
15:            end if
16:        end for
17:        Sort(feasible)                          ▷ Decreasing order of value/cost
18:        for feasible alternate A do
19:            if requiredA < alloc then
20:                SwitchAlternate(A)
21:                done
22:            end if
23:        end for
24:    end for
25: end procedure
1: procedure ResourceReDeploy(DataflowD, Ω_t)
2:     if Ω_t ≤ Ω̂ − ε then
3:         Same procedure as initial deployment
4:     else if Ω_t ≥ Ω̂ + ε then
5:         while Ω ≥ Ω̂ do
6:             PE ← overProvisionedPE
7:             instance ← SelectInstanceToKill(PE)
8:             newAlloc ← RemovePEInstance(instance)
9:             Ω ← GetEstimatedThruput(D,newAlloc)
10:        end while
11:        repackFreeVMs()   ▷ Repack PEs in the VMs with free cores onto smaller VMs with collocation
12:    end if
13: end procedure
```

9 EVALUATION

We evaluate the proposed heuristics through a simulation study. To emulate real-world cloud characteristics, we extend CloudSim [31] simulator to *IaaSSim*, that incorporates temporal and spatial performance variability using VM and network performance traces collected from IaaS Cloud VMs [2]. Further, *FloeSim* simulates the execution of the dynamic dataflows [21], on top of IaaSSim, with support for dataflows, alternates, distributed deployment, runtime scaling, and plugins for different schedulers. To simulate data rate variations, the given data rate is considered as an average value and the instantaneous data rate is obtained using a random walk between $\pm 50\%$ of that value. However, to enable comparisons between different simulation runs, we generate this data trace once and use the same across all simulation runs.

2. IaaSSim and performance traces are available at http://github.com/usc-cloud/IaaSSimulator



	Υ	c	s
p_2^0	1	1.2	1
p_2^1	0.8	0.8	1
p_2^2	0.5	0.8	0.5

	Υ	c	s
p_7^0	0.6	0.3	1
p_7^1	1	0.5	1

Fig. 5: Dynamic Continuous Dataflow for LRB. Alternates for $\mathcal{P}_2$ and $\mathcal{P}_7$ have value (γ), cost (c) and selectivity (s).

For each experiment, we deploy the Linear Road Benchmark (LRB) [16] as a dynamic dataflow using FloeSim, run it for 12 simulated hours ($T = 12\ hrs$) on simulated VMs whose performance traces were obtained from real Amazon EC2 VMs. Each experiment is repeated at least three times and average values are reported. We use a timestep duration of $t = 5\ mins$ at the start of which adaptation decisions are made.

9.1 Linear Road Benchmark (LRB)

The Linear Road Benchmark [16] is used to compare the performance of data stream management systems and has been adopted to general purpose stream processing systems [32]. Using this as a base, we develop a dynamic continuous dataflow application to evaluate the proposed scheduling heuristics. LRB models a road toll network within a confined area (e.g., 100 sq. miles), in which the toll depends on several factors including time of the day, current traffic congestion levels and proximity to accidents. It continuously ingests "position reports" from different vehicles on the road and is responsible for (i) detecting average speed and traffic congestion for a section, (ii) detecting accidents, (iii) providing toll notifications to vehicles whenever they enter a new section, (iv) answering account balance queries and toll assessments, and (v) estimating travel times between two sections. The goal is to support the highest number of expressways while satisfying the desired latency constraints. To simulate realistic road conditions, the data rate varies from around 10 msgs/sec to around 2,000 msgs/sec per expressway.

Fig. 5 shows the LRB benchmark implemented as a dynamic continuous dataflow. The *Weather Updates* ($\mathcal{P}_0$) and *Report Parse* ($\mathcal{P}_1$) PEs act as the input PEs for the dataflow. While the former receives low frequency weather updates, the latter receives extremely high frequency position reports from individual vehicles (each car sends a position report every 30 secs) and exhibits variable data rates based on the current traffic conditions. The *Congestion Estimation* PE ($\mathcal{P}_2$) estimates current as well as near-future traffic conditions for different sections of all the monitored expressways. This PE may have several alternates using different machine learning techniques that predict traffic with different accuracy and future horizons. We simulate three alternates with different value (γ), cost (c) and selectivity (s) values as shown in the tables in Fig. 5. The *Accident Detector* PE ($\mathcal{P}_3$) detects accidents based on the position reports, which is forwarded to *Toll Calculator* ($\mathcal{P}_4$) and *Travel Time Estimator* ($\mathcal{P}_7$) PEs. The former notifies toll values ($\mathcal{P}_5$) and account balances ($\mathcal{P}_6$) to the vehicles periodically. The latter ($\mathcal{P}_7$) provides travel time estimates, and has several alternates based on different forecasting models. For simulations, we use two alternates, e.g., (1) decision/regression tree model which takes several historical factors into account, and (2) time series models which predict using only the recent past traffic conditions.

9.2 Results

We compare the proposed centralized and sharded heuristics (CE and SH) and the GA algorithm with a brute force approach (BR) that explores the search tree but uses intelligent pruning to avoid searching sub-optimal or redundant sub-trees. We evaluate their overall profit achieved, overall relative throughput and the monetary cost of execution over the optimization period of $T = 12\ hrs$. An algorithm is better than another if it meets the necessary relative application throughput constraint, $\Omega \geq \widehat{\Omega} - \epsilon$, and has a higher value for the objective function Θ (Eqn. 8) Note that the necessary constraint for Ω *must* be met but higher values beyond the constraint do not indicate a better algorithm. Similarly, an algorithm with a higher Θ value is not better *unless* it also meets the Ω constraint.

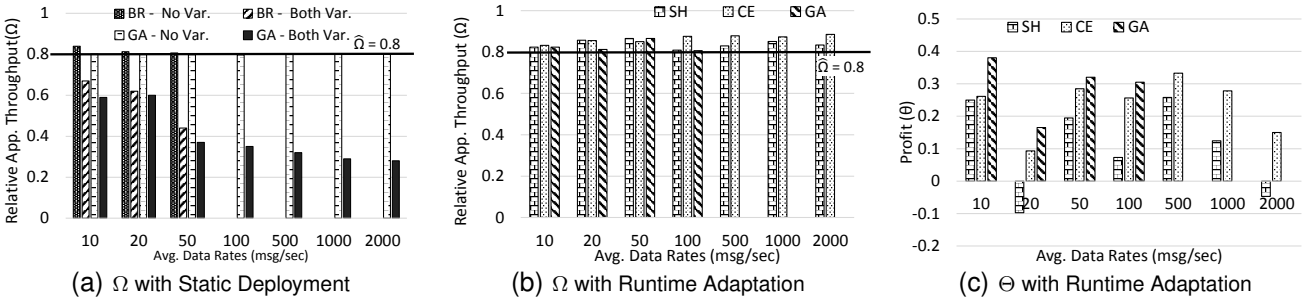
For all the experiments, we define the relative throughput threshold as $\widehat{\Omega} = 0.8$ with a tolerance of $\epsilon = 0.05$. We calculate σ for the LRB dataflow using Eqn. 9 by setting $C_{max}^{\Gamma_{min}} = \frac{0.5 \times T \times DataRate}{10}$ and $C_{max}^{\Gamma_{max}} = \frac{1.0 \times T \times DataRate}{10}$. We empirically arrive at these bounds by observing the actual break-even cost for executing the workflow using a brute force static deployment model.

1) Effect of Variability: Table 2 shows the overall profit and the relative throughput for a static deployment of LRB using different scheduling algorithms with an average input data rate of 50 msgs/sec. The overall profit which is a function of application value and resource cost remains constant due to a static deployment (without runtime adaptation). However, the relative throughput varies as we introduce infrastructure and data variability. In the absence of any variability, the brute force (BR) approach gives the best overall profit and also meets the throughput constraint ($\Omega \geq 0.8$). Further, GA approaches a near optimal solution with $\Theta_{GA} \rightarrow \Theta_{BR}$ when neither infrastructure nor input data rates vary, and is within the tolerance limit ($\widehat{\Omega} - \epsilon < \Omega = 0.79 < \widehat{\Omega} + \epsilon$). The SH and CE heuristics meet the throughput constraint but give a lower profit when there is no variability.

However, when running simulations with infrastructure and/or data variability, none of the approaches

TABLE 2: Effect of variability of infrastructure performance and input data rate on relative output throughput using different scheduling algorithms. Static LRB deployment with average of 50 msgs/sec input rate, $\widehat{\Omega} = 0.8$.

		Relative Application Throughput (Ω)			
Algo.	**Profit** (Θ)	Neither	Infra. Perf.	Data Rate	Both
BR	0.67	0.80	0.68	0.59	0.44
GA	0.65	0.79	0.67	0.48	0.37
SH	0.45	0.81	0.60	0.40	0.29
CE	0.58	0.81	0.66	0.42	0.31



(a) Ω with Static Deployment (b) Ω with Runtime Adaptation (c) Θ with Runtime Adaptation

Fig. 6: Effect of infrastructure and data rate variability on Static Deployment and Runtime Adaptation, as input data rate rises.

meet the desired throughput constraints with Ω values ranging between 0.29 to 0.68, which is much less than the goal of $\widehat{\Omega} = 0.8$. Since these experiments do not change the deployment at runtime, the initial deployment is based on assuming constant data rates and infrastructure performance. Even the two best static deployment strategies in the absence of variability, BR and GA, rapidly degrade in output throughput when both infrastructure and data variability are present. This is more so as the average input data rate increases from 10 msgs/sec to 2,000 msgs/sec in Fig. 6(a). Due to explosion in state space, we could not run the BR algorithm beyond 50 msgs/sec input rate. This analysis motivates the need for autonomic adaptations to application deployment.

2) Improvements with Runtime Adaptation: Figs. 6(b) and 6(c) show the improvements in relative throughput and overall application profit by utilizing different runtime adaptation techniques in the presence of both infrastructure as well as data variability. We use GA as the upper bound (instead of BR) since BR is prohibitively slow for runtime adaptations and, as shown in Table 2, GA approaches the optimal in many scenarios. However, as discussed later (Fig. 7(a)), even GA becomes prohibitive for data rates ≥ 500 msg/sec, and hence the missing entries in Figs. 6(b) and 6(c).

We observe that with dynamic adaptation both GA and the greedy heuristics (SH and CE) achieve the desired throughput constraint ($\Omega \geq \widehat{\Omega} = 0.8$) for all the input data rates tested. This allows us to compare their achieved application profit (Fig. 6(c)) and make several key observations. First, profit from GA is consistently more than the SH and CE greedy heuristics. Second, CE achieves a better profit than SH and reaches between 60% to 80% of GA's profit. In fact, SH gives negative profit (loss) in some cases. Understandably, the CE scheduler having a global view performs significantly better than SH that performs local optimizations. However, CE has a higher overhead due to centralized collection of monitoring data (the exact overhead is not available from simulations). Lastly, comparing the static and dynamic deployment from Table 2 and Fig. 6(c), for a data rate of 50 msg/sec under variable conditions, we do see a drop in profit using runtime adaptation as it tries to achieve the Ω constraint. For GA and CE, the profits for adaptive deployment drop from $0.65 \rightarrow 0.34$ and $0.58 \rightarrow 0.29$, respectively, but are still well above the break-even point of 0. But the static deployments violate the throughput constraint by a large margin which makes their higher profits meaningless.

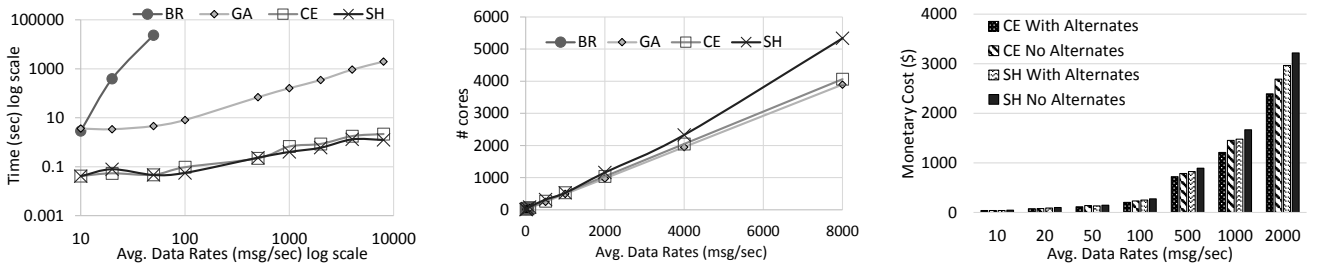
3) Scalability of Algorithms: Figs. 7(a) and 7(b) show algorithm scalability with respect to algorithm runtime and the number of cores for the initial deployment algorithm with increase in the incoming data rates. While the BR and the GA algorithms provide (near) optimal solutions for smaller data rates, their overhead is prohibitively large for higher data rates (fig. 7(a)) ($> 10,000$ secs for BR at 50 msg/sec, and $> 1,000$ secs for GA at 8,000 msg/sec). This is due to the search space explosion with the increase in the number of required VMs as shown in fig 7(b). On the other hand, both CE and SH greedy heuristics take just ~ 2.5 secs to compute at 8,000 msg/sec, and scale linearly ($O(|\varphi| + |R|)$) with the number of PE instances ($|\varphi|$) and number of virtual machines($|R|$). Further we see that SH algorithm leads to higher resource wastage (more cores) with increase in the data rates, while CE and GA show a linear relation to the data rates in Fig. 7(b). Similar results are seen for the adaptation stage for SH and CE algorithms but the plots are omitted due to space constraints.

4) Benefit of using Alternates: We study the reduction in monetary cost to run the continuous dataflow due to the use of alternates, as opposed to a dataflow deployed with only a single implementation for the PEs; we choose the implementation with the highest value ($\Gamma = 1$). Fig. 7(c) shows the cost (US$) of resources required to execute the LRB dataflow for the optimization interval $T = 12\ hr$ using the greedy heuristics with runtime adaptation, in the presence of both infrastructure and data variability. We use AWS's EC2 prices using m1.* generation of VMsfor calculating the monetary cost.

We see that the use of alternates by runtime adaptation leads to a reduction in total monetary cost by 6.9% to 27.5%, relative to the non-alternate dataflow; alternates with different processing requirements provide an extra dimension of control to the scheduler. In addition, the benefit of alternates increases with high data rate – as the input data rate and hence the resource requirement increases, even small fluctuations in data rate or infrastructure performance causes new VMs to be acquired to meet the Ω constraint, and acquiring VMs has a higher overhead (e.g., the hourly cost cycle and startup overheads) than switching between alternates.

10 Related Work

Scientific workflows [33], continuous dataflow systems [7], [8], [5], [9] and similar large-scale distributed



(a) Algorithm time as a function of datarate. (b) # Cores as a function of datarate (c) Monetary cost benefit of using alternates

Fig. 7: Algorithm scalability (a,b) and advantage of alternates (c)

programming frameworks [34], [35] have garnered a renewed research focus due to the recent explosion in the amount of data, both archived and real time, and the need for large-scale data analysis on this "Big Data". Our work is based upon the stream processing and continuous dataflow systems that allow a task-graph based programming model to execute long running continuous applications which process incoming data streams in near-real time. Other related work includes flexible workflows and heterogeneous computing, service oriented architecture (SOA) and autonomic provisioning and resource management in clouds. We discuss the state-of-the-art in each of these research areas.

Continuous dataflow systems have their root in Data Stream Management Systems, which process continuous queries over tuple streams composed of of well-defined operators [36]. This allows operator-specific query optimizations such as operator split and merge to be performed [37]. General-purpose continuous dataflow systems such as S4 [7], Storm [8], and Spark [34], on the other hand, allow user-defined processing elements, making it necessary to find generic auto-scaling solutions such as operator scaling and data-parallel operations [38]. Several solutions, including one leveraging cloud elasticity to support auto-scaling, have been proposed [11]. However, these systems [10], [39], only consider data variability as a factor for auto-scaling decisions and assume that the underlying infrastructure offers the same performance over time. Our work shows that this assumption does not hold in virtualized clouds.

Autonomic provisioning for workload resource management on clouds have been proposed. These use performance monitoring and model-based approach [40], [41]. We use a similar approach and propose heuristics for dynamic continuous dataflows that handle not only data rate variations but also changes in the underlying infrastructure performance. Recent work [32] integrates elastic scale out and fault tolerance for stateful stream processing but adopts a local only policy based on CPU utilization for scaling. In this article, we assume stateless PEs, and fault tolerance is beyond the scope of this work. Our results do show that using local scale-out strategies that ignore the dataflow structure under-perform, and hence motivates heuristics with a global view.

Flexible workflows [42], [19] and service selection in SOA [43] allow workflow compositions to be transformed at runtime. This provides a powerful compositional tool to the developer to define business-rule based generic workflows that can be specialized at runtime depending on the environmental characteristics. The notion of "alternates" we propose is similar in that it offers flexibility to the developer and a choice of execution implementations at runtime. However, unlike flexible workflows where the decision about task specialization is made exactly once based on certain deterministic parameters, in continuous dataflows, this decision has to be re-evaluated regularly due to their continuous execution model and dynamic nature of the data streams.

To exploit a heterogeneous computing environment, an application task may be composed of several subtasks that have different requirements and performance characteristics. Various dynamic and static task matching and scheduling techniques have been proposed for such scenarios [18], [44], [45]. The concept of alternates in dynamic dataflow is similar to these. However, currently, we do not allow heterogeneous computing requirements for these alternates, though they may vary in processing requirements. Even with this restriction, the concept of alternates provides a powerful programming abstraction that allows us to switch between them at runtime to maximize the overall utility of the system in response to changing data rates or infrastructure performance.

Several studies have compared the performance of the virtualized environment against the barebones hardware to show their average performances are within an acceptable tolerance limit of each other. However these studies focused on the average performance characteristics and not on the variations in performance. Recent analysis of public cloud infrastructure [12], [46], [47], [14], [48] demonstrate high fluctuations in various cloud services, including cloud storage, VM startup and shutdown time as well as virtual machines core performance and virtual networking. However, the degree of performance fluctuations vary across private and different public cloud providers [13]. Reasons for this include multi-tenancy of VMs on the same physical host, use of commodity hardware, collocation of faults, and roll out of software patches to the cloud fabric. Our own studies confirm these. On this basis, we develop an abstraction of the IaaS cloud that incorporates infrastructure variability and also include it in our IaaS Simulator.

Meta-heuristics have been widely used to address the task scheduling problem [24], [25], [26], [27]. Most of the approaches are nature inspired and rely on GA [28], ant colony optimization [49], particle swarm optimization [50] or simulated annealing [51] techniques to search

for sub-optimal solutions. A number of studies to analyze the efficiency of meta-heuristics [24] show that in certain scenarios GAs can over perform greedy heuristics. More generally, Zamfirache et. al. [27] show that population based GA meta-heuristics of classical greedy approaches provide, through mutations, solutions that are better compared to their classic versions. Recently, a comparison of ant colony optimization and particle swarm optimization with a GA for scheduling DAGs on clouds was proposed [26]. None of these task scheduling algorithms or meta-heuristics based solutions take into account a dynamic list of task instances. Our own prior work [25] uses a GA to elastically adapt the number of task instances in a workflow to incoming web traffic but does not consider alternates or performance variability.

11 Conclusion

In this article, we have motivated the need for online monitoring and adaptation of continuous dataflow applications to meet their QoS constraints in the presence of data and infrastructure variability. To this end we introduce the notion of **dynamic dataflows**, with support for alternate implementations for dataflow tasks. This not only gives users the flexibility in terms of application composition, but also provides an additional dimension of control for the scheduler to meet the application constraints while maximizing its value.

Our experimental results show that the continuous adaptation heuristics which makes use of application dynamism can reduce the execution cost by up to 27.5% on clouds while also meeting the QoS constraints. We have also studied the feasibility of GA based approach for optimizing execution of dynamic dataflows and show that although the GA based approach gives near-optimal solutions its time complexity is proportional to the input data rate, making it unsuitable for high velocity applications. A hybrid approach which uses GA for initial deployment and the CE greedy heuristic for runtime adaptation may be more suitable. This is to be investigated as future work.

In addition, we plan to extend the concept of dynamic tasks which will further allow for alternate implementations at coarser granularity such as "alternate paths", and provide end users with more sophisticated controls. Further, we plan to extend the resource mapping heuristics for an ensemble of dataflows with a shared budget and address issues such as fairness in addition to throughput constraint and application value.

Acknowledgments

This material is based on research sponsored by DARPA, and the Air Force Research Laboratory under agreement number FA8750-12-2-0319. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, and the Air Force Research Laboratory or the U.S. Government.

References

[1] L. Douglas, "3d data management: Controlling data volume, velocity, and variety," Gartner, Tech. Rep., 2001.

[2] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *Communications*. ACM, 2008, vol. 51, no. 1, pp. 107–113.

[3] M. Gatti, P. Cavalin, S. B. Neto, C. Pinhanez, C. dos Santos, D. Gribel, and A. P. Appel, "Large-scale multi-agent-based modeling and simulation of microblogging-based online social network," in *Multi-Agent-Based Simulation XIV*. Springer, 2014, pp. 17–33.

[4] I. Fronza, A. Sillitti, G. Succi, M. Terho, and J. Vlasenko, "Failure prediction based on log files using random indexing and support vector machines," in *Journal of Systems and Software*. Elsevier, 2013, vol. 86, no. 1, pp. 2–11.

[5] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, and C. Moran, "Ibm infosphere streams for scalable, real-time, intelligent transportation services," in *International Conference on Management of data*. ACM SIGMOD, 2010, pp. 1093–1104.

[6] Y. Simmhan, S. Aman, A. Kumbhare, R. Liu, S. Stevens, Q. Zhou, and V. Prasanna, "Cloud-based software platform for big data analytics in smart grids," in *Computing in Science Engineering*, July 2013, vol. 15, no. 4, pp. 38–47.

[7] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *IEEE International Conference on Data Mining Workshops (ICDMW)*, 2010.

[8] "Storm: Distributed and fault-tolerant realtime computation," http://storm.incubator.apache.org/, accessed: 2014-06-30.

[9] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *USENIX conference on Hot Topics in Cloud Ccomputing*, 2012, pp. 10–10.

[10] B. Satzger, W. Hummer, P. Leitner, and S. Dustdar, "Esc: Towards an elastic stream computing platform for the cloud," in *IEEE International Conference on Cloud Computing (CLOUD)*, July 2011, pp. 348–355.

[11] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez, "Streamcloud: An elastic and scalable data streaming system," in *Transactions on Parallel and Distributed Systems*. IEEE, 2012, vol. 23, no. 12, pp. 2351–2365.

[12] A. Iosup, N. Yigitbasi, and D. Epema, "On the performance variability of production cloud services," in *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2011.

[13] A. Li, X. Yang, S. Kandula, and M. Zhang, "Cloudcmp: comparing public cloud providers," in *conference on Internet measurement*. ACM SIGCOMM, 2010, pp. 1–14.

[14] I. Moreno, P. Garraghan, P. Townend, and J. Xu, "Analysis, modeling and simulation of workload patterns in a large-scale utility cloud," in *Transactions on Cloud Computing*. IEEE, April 2014, vol. 2, no. 2, pp. 208–221.

[15] A. Kumbhare, Y. Simmhan, and V. K. Prasanna, "Exploiting application dynamism and cloud elasticity for continuous dataflows," in *International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 57.

[16] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts, "Linear road: a stream data management benchmark," in *international conference on Very Large Databases*. VLDB Endowment, 2004, pp. 480–491.

[17] W. M. van Der Aalst, A. H. Ter Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow patterns," in *Distributed and parallel databases*. Springer, 2003, vol. 14, no. 1, pp. 5–51.

[18] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, "Dynamic mapping of a class of independent tasks onto heterogeneous computing systems," in *Journal of Parallel and distributed Computing*. Elsevier, 1999, vol. 59, no. 2, pp. 107–131.

[19] J. Wainer and F. de Lima Bezerra, "Constraint-based flexible workflows," in *Groupware: Design, Implementation, and Use*. Springer, 2003, pp. 151–158.

[20] C. Ouyang, E. Verbeek, W. M. Van Der Aalst, S. Breutel, M. Dumas, and A. H. Ter Hofstede, "Formal semantics and analysis of control flow in ws-bpel," in *Science of Computer Programming*. Elsevier, 2007, vol. 67, no. 2, pp. 162–198.

[21] Y. Simmhan and A. G. Kumbhare, "Floe: A continuous dataflow framework for dynamic cloud applications," *CoRR*, vol. abs/1406.5977, 2014.

[22] M. C. Jaeger, G. Rojec-Goldmann, and G. Muhl, "Qos aggregation for web service composition using workflow patterns," in *IEEE*

International conference on Enterprise distributed object computing., 2004, pp. 149–159.
[23] G. J. Woeginger, "Exact algorithms for np-hard problems: A survey," in *Combinatorial Optimization–Eureka, You Shrink!* Springer, 2003, pp. 185–207.
[24] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," in *Journal of Parallel and Distributed Computing*. Elsevier, Jun. 2001, vol. 61, no. 6, pp. 810–837.
[25] M. E. Frincu, "Scheduling highly available applications on cloud environments," in *Future Generation Computer Systems*. Elsevier, 2014, vol. 32, pp. 138–153.
[26] Z. Wu, X. Liu, Z. Ni, D. Yuan, and Y. Yang, "A market-oriented hierarchical scheduling strategy in cloud workflow systems," in *The Journal of Supercomputing*. Springer, 2013, vol. 63, no. 1, pp. 256–293.
[27] F. Zamfirache, M. Frincu, and D. Zaharie, "Population-based metaheuristics for tasks scheduling in heterogeneous distributed systems," in *Numerical Methods and Applications*. Springer, 2011, vol. 6046, pp. 321–328.
[28] J. H. Holland, *Adaptation in Natural and Artificial Systems*. Cambridge, MA, USA: MIT Press, 1992.
[29] D. E. Goldberg and K. Deb, "A Comparative Analysis of Selection Schemes Used in Genetic Algorithms," in *Foundations of Genetic Algorithms*, 1990, pp. 69–93.
[30] J. Kang and S. Park, "Algorithms for the variable sized bin packing problem," in *European Journal of Operational Research*. Elsevier, 2003, vol. 147, no. 2, pp. 365–372.
[31] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya, "Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," in *Software: Practice and Experience*. Wiley Online Library, 2011, vol. 41, no. 1, pp. 23–50.
[32] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *International conference on Management of data*. ACM SIGMOD, 2013, pp. 725–736.
[33] J. Yu and R. Buyya, "A taxonomy of scientific workflow systems for grid computing," in *Sigmod Record*. ACM, 2005, vol. 34, no. 3, p. 44.
[34] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *USENIX conference on Hot topics in cloud computing*, 2010.
[35] S. Pallickara, J. Ekanayake, and G. Fox, "Granules: A lightweight, streaming runtime for cloud computing with support, for map-reduce," in *IEEE International Conference on Cluster Computing*. IEEE, 2009.
[36] S. Babu and J. Widom, "Continuous queries over data streams," in *Sigmod Record*. ACM, 2001, vol. 30, no. 3, pp. 109–120.
[37] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin, "Flux: An adaptive partitioning operator for continuous query systems," in *International Conference on Data Engineering*. IEEE, 2003, pp. 25–36.
[38] Y. Zhou, B. C. Ooi, K.-L. Tan, and J. Wu, "Efficient dynamic operator placement in a locally distributed continuous query system," in *On the Move to Meaningful Internet Systems: CoopIS, DOA, GADA, and ODBASE*. Springer, 2006, pp. 54–71.
[39] R. Tolosana-Calasanz, J. Angel Bañares, C. Pham, and O. Rana, "End-to-end qos on shared clouds for highly dynamic, large-scale sensing data streams," in *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2012, pp. 904–911.
[40] A. Quiroz, H. Kim, M. Parashar, N. Gnanasambandam, and N. Sharma, "Towards autonomic workload provisioning for enterprise grids and clouds," in *IEEE/ACM International Conference on Grid Computing*, Oct 2009, pp. 50–57.
[41] Y. Hu, J. Wong, G. Iszlai, and M. Litoiu, "Resource provisioning for cloud computing," in *Conference of the Center for Advanced Studies on Collaborative Research*. Riverton, NJ, USA: IBM Corp., 2009, pp. 101–111.
[42] G. J. Nutt, "The evolution towards flexible workflow systems," in *Distributed Systems Engineering*. IOP Publishing, 1996, vol. 3, no. 4.
[43] D. Guinard, V. Trifa, S. Karnouskos, P. Spiess, and D. Savio, "Interacting with the soa-based internet of things: Discovery, query, selection, and on-demand provisioning of web services," in *Transactions on Services Computing*. IEEE, 2010, vol. 3, no. 3, pp. 223–235.
[44] L. Wang, H. J. Siegel, V. P. Roychowdhury, and A. A. Maciejewski, "Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach," in *Journal of Parallel and Distributed Computing*. Elsevier, 1997, vol. 47, no. 1, pp. 8–22.
[45] H. Topcuoglu, S. Hariri, and M.-y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," in *Transactions on Parallel and Distributed Systems*. IEEE, 2002, vol. 13, no. 3, pp. 260–274.
[46] A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. H. Epema, "Performance analysis of cloud computing services for many-tasks scientific computing," in *Transactions on Parallel and Distributed Systems*. IEEE, 2011, vol. 22, no. 6, pp. 931–945.
[47] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, and N. J. Wright, "Performance analysis of high performance computing applications on the amazon web services cloud," in *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2010, pp. 159–168.
[48] Z. Ou, H. Zhuang, A. Lukyanenko, J. Nurminen, P. Hui, V. Mazalov, and A. Yla-Jaaski, "Is the same instance type created equal? exploiting heterogeneity of public clouds," in *Transactions on Cloud Computing*. IEEE, July 2013, vol. 1, no. 2, pp. 201–214.
[49] A. Colorni, M. Dorigo, and V. Maniezzo, "Distributed Optimization by Ant Colonies," in *European Conference on Artificial Life*, 1991, pp. 134–142.
[50] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *IEEE International Conference on Neural Networks*, vol. 4, 1995, pp. 1942–1948 vol.4.
[51] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of State Calculations by Fast Computing Machines," in *The Journal of Chemical Physics*. AIP, 1953, vol. 21, no. 6, pp. 1087–1092.

Alok Gautam Kumbhare is a PhD candidate in Computer Science at the Univ. of Southern California and a Research Assistant in the DARPA XDATA project. His research interests are in the areas of resource management and fault tolerance for workflow and dataflows on distributed environment such as Clouds.

Yogesh Simmhan is an Assistant Professor at the Indian Institute of Science. Previously, he was a Research Assistant Professor in Electrical Engineering at the University of Southern California and a postdoc at Microsoft Research. His research explores scalable abstractions, algorithms and applications for distributed platforms, and helps advance fundamental knowledge while offering a practitioner's insight. He has a Ph.D. in Computer Science from Indiana University. Senior member of ACM and IEEE.

Marc Frincu received his Ph.D. from West University of Timisoara Romania in 2011. During his Ph.D. he also worked as a junior researcher at the e-Austria Research Institute Romania. In 2012 he joined University of Strasbourg France as a postdoctoral researcher focusing on cloud scheduling. In 2013 he took a postdoctoral research associate position at Univ of Southern California dealing with cloud computing and smart grid systems.

Viktor K. Prasanna is the Powell Chair in Engineering, Professor of Electrical Engineering and Computer Science, and Director of the Center for Energy Informatics at the Univ of Southern California. His research interests include HPC, Reconfigurable Computing, and Embedded Systems. He received his MS from the School of Automation, Indian Institute of Science and PhD in Computer Science from Penn State. He is a Fellow of IEEE, ACM and AAAS.