

Accurate Latency Estimation in a Distributed Event Processing System

Badrish Chandramouli[†], Jonathan Goldstein[‡], Roger Barga[†], Mirek Riedewald*, Ivo Santos[†]

[†]*Microsoft Research*

{badrishi, barga, ivosan}@microsoft.com

[‡]*Microsoft Corporation*

jongold@microsoft.com

^{*}*Northeastern University*

mirek@ccs.neu.edu

Abstract—A distributed event processing system consists of one or more nodes (machines), and can execute a directed acyclic graph (DAG) of operators called a *dataflow* (or *query*), over long-running high-event-rate data sources. An important component of such a system is *cost estimation*, which predicts or estimates the “goodness” of a given input, i.e., operator graph and/or assignment of individual operators to nodes. Cost estimation is the foundation for solving many problems: optimization (plan selection and distributed operator placement), provisioning, admission control, and user reporting of system misbehavior.

Latency is a significant user metric in many commercial real-time applications. Users are usually interested in quantiles of latency, such as worst-case or 99th percentile. However, existing cost estimation techniques for event-based dataflows use metrics that, while they may have the side-effect of being correlated with latency, do not directly or provably estimate latency. In this paper, we propose a new cost estimation technique using a metric called **Mace** (*Maximum cumulative excess*). Mace is provably equivalent to *maximum system latency* in a (potentially complex, multi-node) distributed event-based system. The close relationship to latency makes Mace ideal for addressing the problems described earlier. Experiments with real-world datasets on Microsoft StreamInsight deployed over 1–13 nodes in a data center validate our ability to closely estimate latency (within 4%), and the use of Mace for plan selection and distributed operator placement.

I. INTRODUCTION

Many established and emerging applications can be naturally modeled using event-based dataflows; examples include network monitoring [10, 14] and real-time delivery of Web advertisements [5]. Users register dataflows in the form of *continuous queries* (*CQs*) with the *event processing system* (*EPS*). CQs typically run on an EPS for long periods (weeks or months) and continuously produce incremental output in real time for newly arriving input events.

A CQ is usually specified declaratively using a higher-level language such as LINQ, Esper, or StreamSQL. The CQ is converted into a *dataflow plan* which consists of a *directed acyclic graph* (*DAG*) of operators connected by queues of events. There may be many equivalent plans for a CQ, with different performance characteristics. Furthermore, in a distributed system, the operators may themselves be distributed amongst the available *nodes* (machines) in different ways.

With the advent of many commercial event-based systems (e.g., StreamInsight, Oracle CEP, Streambase), the EPS is faced with a need for solutions to several related problems:

- **Query Optimization**: For a given set of CQs, we seek to find the best dataflow plans and/or assignment of operators to nodes. A closely related problem is *re-optimization*, which is the periodic adjustment of the CQs on the basis of detected changes [21] in input behavior.
- **Admission Control**: When we try to add or remove a CQ from the system, we need to quickly and accurately predict the corresponding *impact* on the system.
- **System Provisioning**: The system administrator must determine the effect of making more or fewer CPU cycles or nodes available to the system under its current CQ load.
- **User Reporting**: Users need a meaningful estimate of the behavior of their CQs. Such an estimate should be based on a metric that is relevant to users. This could also be used as a basis for performance guarantees from the system.

A common requirement of each of the above problems is a good *cost metric*, and a corresponding *estimation* (i.e., *prediction*) technique. Our experience with commercial event-based applications indicates that *latency* — the time taken for an input event to produce a result — is one of the most important cost metrics to end users in a real-time system. Users are interested in quantiles such as worst-case, average, and 99.9th percentile of latency [16]. Thus, latency is an ideal starting point as a metric to solve the above problems.

Challenges Consider the following commercial event-based application in the context of real-time Web advertising.

Example 1 (Real-Time Targeted Advertising [5]). Consider an event processing system that processes complex CQs over URL clickstreams. Here, each event may be a user click that navigates the browser from one page to another. Associated with each event is user-specific demographic data. Such a system could answer multiple real-time CQs whose results could be used to display user- or URL-tailored targeted Web advertisements, or report interesting real-time statistics. Figure 1 depicts the input event rates seen in such an event

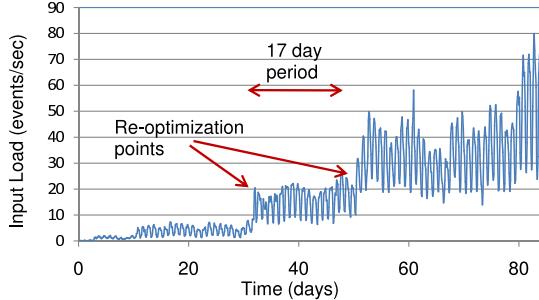


Fig. 1. Input load distribution for click-stream data.

click-stream, that we derived using actual data collected on an advertisement delivery system over a period of 84 days.

There are several points worth noting from this example.

1) Latency is important: A fast (low latency) response to incoming events is important to avoid stale decisions such as the choice of targeted ads (or stock trades). Low worst-case latency is crucial for applications that monitor critical infrastructure, as well as for fraud, intrusion, and anomaly detection, where the harmful activity needs to be discovered quickly. Further, we see that system behavior, in terms of input event rate, is relatively predictable over long periods of time (such as the marked 17 day period). This indicates that we can highly benefit from an optimizer that produces a lowest-latency dataflow plan and/or assignment of operators to nodes.

On the other hand, there are periodic *shifts*, where system characteristics change significantly, motivating the need for query re-optimization, updating latency estimates reported to end users, and re-provisioning for the changed load — again requiring the ability to estimate latency *a priori*.

2) Prediction has to be quick: An optimizer needs to quickly evaluate the merit of a plan, while searching through a vast space of possible plans, during *plan selection*. Further, when we run the dataflow on multiple nodes in a data center, the optimizer also needs to perform *operator placement*, i.e., choose the “best” assignment of operators to nodes that minimizes latency. Thus, we need the ability to quickly evaluate and compare the merit (in terms of latency) of many different choices *without being able to actually run them*.

3) Latency is hard to predict: Even during the stable period marked in Figure 1, there are short-term variations in event rates (e.g., due to diurnal trends), that make it difficult to estimate the latency of a particular plan/assignment. Latency is challenging to predict reliably, because of the complexity of the dataflow plan and the non-trivial interactions between components of a distributed system. Note that we cannot simply measure latency at runtime, as it is not possible to run all possible plans/placements for a query. Further, system provisioning requires that we be able to predict the effect of changes such as the availability of more nodes. It is usually unfeasible to try out such new deployments without procuring the additional cycles/cores/machines *a priori*.

4) We need to predict actual latency: Given the difficulty of predicting latency, we may wish to consider the use of existing commonly used metrics such as resource usage [9], number

of intermediate tuples [18], load correlation [32], and feasible set size [33] as a “proxy” for latency. Some of these metrics are correlated with latency (in other words, minimizing such a metric reduces latency). However, a proxy metric cannot be used for provisioning, admission control, or user reporting based on latency — such a number cannot detect that a specific constraint (e.g., *worst-case latency should be less than 10 secs*) may be violated, nor can users derive any meaning out of the number beyond the ability to compare two values in a relative sense. Thus, we need a technique to directly predict latency in seconds.

In summary, we seek a latency estimation technique with these desirable properties: (1) it directly estimates latency, an intuitive and meaningful metric for a distributed EPS; (2) it is easy and quick to compute without introducing complexity into the system; (3) it is generally applicable for multi-node operator placement as well as for comparing plans on a single machine; (4) it corresponds precisely to actual latency, on a provable theoretical basis (beyond intuition).

Contributions We make the following contributions.

- We describe (§ III, IV, and VI) a novel solution for cost estimation, and the associated cost metric called *Mace* (*Maximum cumulative excess*). Mace is based on a unique variation of amortized analysis, and satisfies all the desirable properties outlined earlier.
- We propose a low-overhead scheduling policy and show (§ V) that with this scheduling policy, Mace *provably* corresponds closely to worst-case latency. Our solution can also be used to estimate latency beyond worst-case, including average and 99th percentile (cf. § V-B).
- Recognizing that an EPS may wish to use a different scheduling policy for other reasons, a tuning parameter w allows us to trade-off latency estimation accuracy while allowing such variations (§ VII-A). We focus on data centers, with the extension to general networks in § VII-B.
- We evaluate our solution (§ VIII) with real data on Microsoft StreamInsight [5] deployed over 1–13 nodes in a data center. Results validate the equivalence (within 4% error) between Mace and latency in all cases.
- Our solution is general. We describe two applications in § IX — plan selection and distributed operator placement, for which we propose a new algorithm called Mace-HC. We evaluate the optimizer’s ability to perform good plan selection, and also demonstrate, with a simulation of 400 nodes, that Mace-HC is an order-of-magnitude quicker than competing schemes at finding lower-latency placements.

We cover several extensions in [12], including handling multi-cores and handling query priorities. Mace and its relation to latency is applicable to any distributed queue-based workflow with well-understood operators (tasks) and control over the scheduling policy. However, we present our findings in the context of distributed event processing systems.

II. PRELIMINARIES

A. The Distributed Event-Based Dataflow Model

Each dataflow or CQ plan, similar to a database query plan, consists of a DAG of operators. Each operator consumes events from one or more input queues, performs computation, and produces new events to be output or placed on the input queue of other operators. Operators generate load on their host nodes by consuming CPU cycles. We target the important practical scenario where all nodes are located in a data center having one or more shared-nothing nodes with a high-bandwidth fast interconnect. Our experiments (cf. Section VIII) using a real data center validate the expectation that network link costs are relatively insignificant in such a cluster deployment of an EPS; an extension to handle high-latency or low-bandwidth networks is discussed in Section VII-B. As is common in data center deployments [8], we assume that clocks are synchronized using standard protocols such as NTP [29].

Definition 1 (EPS and Query Graph). *An EPS consists of a set of n nodes $\mathcal{N} = \{N_1, N_2, \dots, N_n\}$, a set of m operators $\mathcal{O} = \{O_1, O_2, \dots, O_m\}$, and a partitioning of the m operators into n disjoint subsets $\mathcal{S} = \{S_1, \dots, S_n\}$ such that S_i is the set of operators assigned to node N_i . The assignment of operators to nodes is called the operator placement. Note that each of the m operators may belong to a different CQ.*

The query graph \mathcal{G} is a DAG over \mathcal{O} where the roots of the graph are referred to as sources, and the leaves of the graph are referred to as sinks. The operators that an operator O_j is reachable from (in \mathcal{G}) are said to be upstream of O_j . Let each node N_i have a total available CPU of C_i cycles per time unit.

For example, Figure 3(a) shows an EPS query graph with 3 nodes (N_1, N_2 , and N_3), each having available CPU of $C_i = 1$ cycle/second. The partitioning is $S_i = \{O_i\} \forall 1 \leq i \leq 3$.

B. Latency

Latency denotes the delay that is introduced by the EPS from the point of event arrival to result generation. We observe that no matter how quickly the EPS processes events, an operator cannot produce an event e before all its “contributing” events (or punctuations [27]) have arrived from sources *outside* the EPS. Latency is intuitively the time interval from this point of time until the time when e is actually produced by the operator. As in data warehousing [15], we refer to the set of contributing source events for event e as the *lineage* of e .

We focus on worst-case latency as our estimation goal (other quantiles and averages are covered in Section V-B). Worst-case metrics are popular in applications with strict real-time needs, since they provide an upper bound on system misbehavior, which is often more useful than averages. For example, users may not want stock trades, fraud/intrusion detection, or anomaly detection in critical infrastructure, to be delayed by more than a specified time. It is also common practice in large systems [16] to optimize for the worst-case or 99.9th percentile rather than the average case.

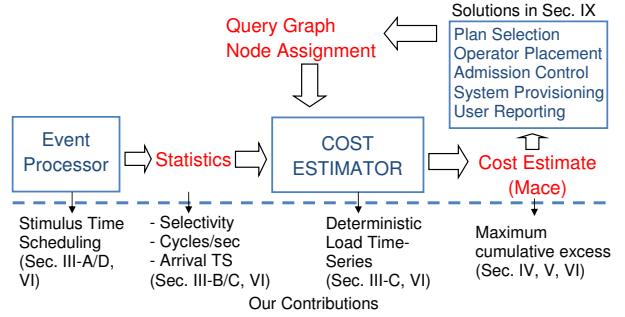


Fig. 2. Cost estimation architecture.

C. System Architecture

Figure 2 shows our system architecture. The *cost estimator* uses statistics measured by the *event processor*, such as selectivity and input event rates. Statistics are based on historical observations or trial runs on a small input sample. The estimator accepts a specification of a query graph and an operator placement, and outputs a latency estimate. The estimate is used as part of solving problems such as plan selection, operator placement, provisioning, etc. Figure 2 also provides a referenced overview of our solutions in this paper.

III. LATENCY ESTIMATION IN A DISTRIBUTED EPS

We now present our new building blocks for latency estimation. In Section IV, we will define Mace and leverage these building blocks to show the equivalence of Mace to latency.

A. Handling Events Deterministically

Each *source event* (i.e., from sources outside the EPS) has a well-known *timestamp*, the wall-clock time at the instant it arrives at the EPS. However, intermediate and output events may be produced at arbitrary wall-clock times, depending on when individual operators process their input queues. We use event lineage from Section II to formalize the notion of a deterministic *stimulus time* for events.

Definition 2 (Stimulus Time). *Each source event is assigned a new field called stimulus time, which is the wall-clock time of its arrival at the EPS from outside. The stimulus time of an event e produced by an operator O_j is the maximum timestamp across all source events in the lineage of e (i.e., the moment all source events in its lineage have arrived at the EPS).*

In practice, stimulus times can easily be set by an operator (when it generates a new event), to be the stimulus time of the associated latest incoming event. Note that this new field does not affect existing event timestamps or CQ processing semantics, and is only used to compute latency. We can now define latency precisely in a distributed EPS.

Definition 3 (Latency). *For each output event e produced by a sink in query graph \mathcal{G} , its latency is the difference between the event's egress time (the wall-clock time when it exits the EPS) and its stimulus time.*

We collect and compute statistics by dividing time into equal-width segments. More precisely, a time interval

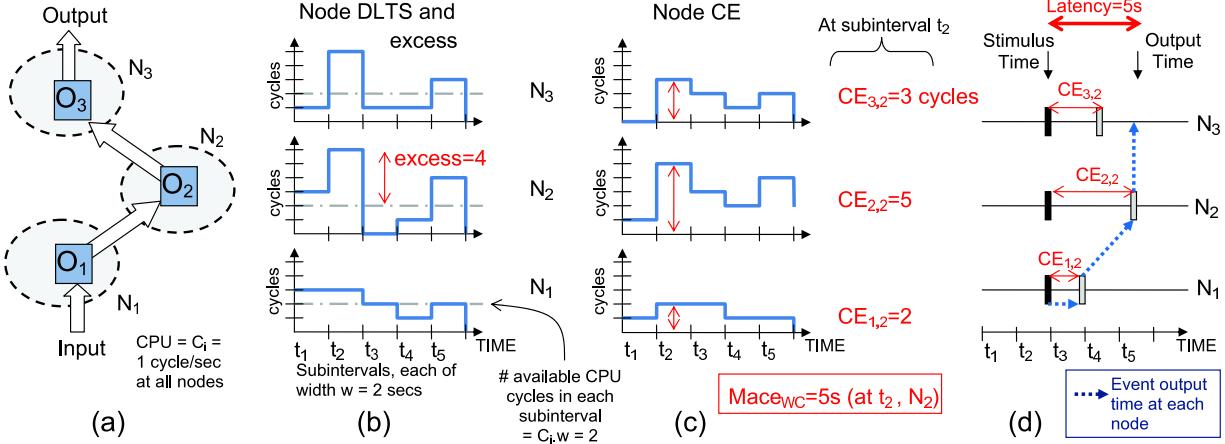


Fig. 3. (a) An example CQ plan on 3 nodes. (b) DLTS for each of the nodes, over 5 subintervals. Here, subinterval width is $w = 2$ secs and CPU on each node is $C_i = 1$ cycle/sec. (c) CE for each of the nodes, with CE at subinterval t_2 highlighted. Worst-case CE ($Mace_{WC}$) is shown. (d) Lifetime of an event which enters EPS at the end of subinterval t_2 . Latency of $e = Mace_{WC} = 5$ secs.

$[t_1, t_{d+1}]$ is partitioned into d discrete *subintervals* (or *buckets*) $[t_1, t_2], \dots, [t_d, t_{d+1}]$ each of width w time units. For brevity, we will refer to a particular subinterval $[t_p, t_{p+1}]$ simply by its left endpoint t_p . Thus, time is represented as a set of subintervals $\tau = \{t_1, \dots, t_d\}$. Figure 3(b) shows an example set of subintervals, each of width $w = 2$ seconds.

An event with stimulus time $t \in [t_p, t_{p+1}]$ is said to *belong to* subinterval t_p . Note that each incoming event (and its “child events” spawned by operators) belongs to a unique subinterval. We are interested in the maximum latency over all events belonging to a subinterval. This results in a time-series of such maximum latency values, that we formalize next.

Definition 4 (Maximum Latency). Maximum latency is a time-series $Lat_{1\dots d}$ defined over the set of discrete subintervals. The maximum latency Lat_p for subinterval t_p is the maximum latency across all output events which belong to subinterval t_p , i.e., whose stimulus times lie in t_p . The overall worst-case latency Lat_{WC} is simply the maximum latency seen over the entire time period. More formally, $Lat_{WC} = \max_{t_p \in \tau} Lat_p$.

B. Modeling Operators

In between the re-optimization points of Figure 1, we model our dataflow operators similarly to prior work [33], with two parameters maintained per single-input operator O_j :

- Selectivity (σ_j), the average number of events generated by the operator in response to each input event to the operator.
- Cycles/event (ω_j), the average CPU cycles consumed by the operator, for each input event to the operator.

In case of operators with q inputs, we maintain these parameters separately for each input, as $\sigma_{j,1\dots q}$ and $\omega_{j,1\dots q}$. Note that this model assumes that operators have a linear relationship between input and output. The generalization to non-linear operators is covered in Section VI-C.

C. Handling Load Deterministically

The input (from outside) to a EPS is one or more sources of events, each with time-varying event rates. The *event arrival*

time-series of source Z is a time-series whose value at each subinterval t_p is simply the number of Z events arriving between time t_p and t_{p+1} . The event arrival time-series may be known in advance, or we can measure it using observed data, e.g., during periods of repeatable load in between query re-optimizations as in Figure 1.

The actual CPU load imposed by operators during execution is difficult to model accurately because it is dependent on various factors including actual queue lengths, scheduling decisions, and runtime conditions. For example, the introduction of a new query can change the actual load time-series imposed by existing operators. This dynamic and hard-to-control nature makes maintaining them or using them to provide hard guarantees difficult. Moreover, such variability and system dependence makes our goal of estimating latency accurately difficult. We therefore adopt an alternate definition called *deterministic load time-series (DLTS)*. Surprisingly, this definition not only makes computation of Mace (in Section IV) easier, but is also crucial in provably establishing the equivalence of our Mace metric to latency.

Definition 5 (Operator DLTS). The DLTS of an operator O_j is a time-series $l_{j,1\dots d}$ whose value $l_{j,p}$ at each subinterval $t_p \in \tau$ equals the total CPU cycles required to process exactly all input events to O_j that belong to subinterval t_p , i.e., whose stimulus times lie within subinterval t_p .

We can view the DLTS of an operator O_j as the load imposed by O_j assuming perfect upstream, i.e., assuming that all operators upstream of O_j process events and produce results instantaneously. Intuitively, we can interpret this assignment of load to deterministic buckets based on event lineage, as a novel form of amortized analysis for our estimation problem.

DLTS is (by design) typically very different from the actual load pattern imposed by O_j during runtime; we will later show how DLTS can nevertheless be used to accurately estimate the actual latency. In practice, we can regard $l_{j,p}$ as the product of (1) the cycles/event parameter (ω_j), and (2) the number of input events to O_j that belong to t_p . Thus, DLTS is independent of runtime system behavior. We now define

node DLTS.

Definition 6 (Node DLTS). *The DLTS of a node N_i is a time-series $L_{i,1\dots d}$, whose value $L_{i,p}$ at each subinterval t_p is the sum of the deterministic load (at t_p) of all operators assigned to that node. More formally, $L_{i,p} = \sum_{O_j \in S_i} l_{j,p}$.*

Again, note that node DLTS is defined assuming a perfect upstream, and is typically different from the actual load pattern imposed on the node during runtime. Figure 3(b) shows the DLTS time-series for three nodes (subintervals are also indicated). In case of node N_2 , for example, we have $L_{2,1} = 3$, $L_{2,2} = 6$, and so on.

D. Stimulus Time Scheduling

We propose a low-overhead *operator scheduling policy* for the EPS. An EPS typically has one scheduler per core, that schedules operators to process events according to some policy. For example, the scheduler may maintain a list of operators with non-empty queues and use heuristics like round-robin or longest-queue-first to schedule operators for execution.

Our scheduling policy is called *stimulus time scheduling*. The basic idea is that each operator is assigned a priority based on the earliest stimulus time amongst all events in its input queue. The scheduler always chooses to execute the operator having the event with earliest stimulus time.

Definition 7 (Stimulus Time Scheduling). *Each node N_i may execute one operator from S_i at a time, and has a scheduler which schedules operators amongst S_i for execution according to stimulus time scheduling: At any given moment, the executing operator is processing the event with earliest stimulus time amongst all input events to operators in S_i .*

Stimulus time scheduling ensures that the events which have older stimuli get priority over events with newer stimuli. Note that since stimulus times become deterministic at the point of entry into the system, scheduling is no longer dependent on more operational characteristics such as queue lengths.

As the following theorem shows, stimulus time scheduling is usually an improvement, in terms of latency, over the round robin based approaches often used in practice.

Theorem 1. *On a single node EPS, stimulus time scheduling is the optimal scheduling policy to minimize worst-case latency.*

Proof: (Intuition) At any given time t , an event with stimulus time t' has already incurred a latency of $t - t'$. Thus, the event (say e) with earliest stimulus time is the one with highest as-yet incurred latency. Scheduling any event other than e only serves to increase the total latency of e , and hence the worst-case system latency. ■

We find that in practice, stimulus time scheduling also works very well in multi-node deployments. It does not require global knowledge that can be difficult to identify and maintain, but is necessary for optimal multi-node scheduling for worst-case latency. Crucially, we will see in Section VI-A that stimulus time scheduling can be implemented very efficiently, with constant time event enqueue and dequeue.

IV. MACE: MAXIMUM CUMULATIVE EXCESS

We now present Mace, our proposed cost metric for an EPS. We assume the use of stimulus time scheduling in the EPS (Section VII-A discusses how this assumption can be relaxed).

Recall that every subinterval t_p is associated with $C_i \cdot w$ cycles of CPU capacity on node N_i . Further, assuming a perfect upstream, each subinterval is associated with $L_{i,p}$ cycles of work to be performed. We define *ideal excess* for subinterval t_p as the excess cycles of work assigned to t_p beyond the CPU capacity ($L_{i,p} - C_i \cdot w$).

Excess work beyond the CPU capacity for a subinterval, will spill over to the next subinterval. The *cumulative excess* (CE) for a node at subinterval t_p is the cumulative amount of pending cycles of work associated with subinterval t_p in the perfect upstream case. We formalize this concept below.

Definition 8 (CE). *Cumulative Excess (CE) of a node N_i is a time-series $CE_{i,1\dots d}$ whose value $CE_{i,p}$ at each subinterval t_p is defined iteratively as follows: $CE_{i,0} = 0$; $CE_{i,p} = \max\{0, CE_{i,p-1} + L_{i,p} - C_i \cdot w\} \quad \forall 1 \leq p \leq d$*

In other words, CE tracks the cumulative pending work over the DLTS and gets reset to 0 when there is no excess. It reflects the amount of work that node N_i would be “behind” at subinterval t_p if the load imposed by N_i were the node DLTS. It is important to note that we use DLTS (which is based on the perfect upstream assumption) to compute CE —thus, CE does not (by design!) refer to the actual overload experienced by the node during operation.

Figures 3(b) and 3(c) illustrate the relationship between DLTS, CPU capacity, and CE for 3 nodes. For example, in case of N_2 , we have $CE_{2,1} = CE_{2,0} + L_{2,1} - C_2 \cdot w = 0 + 3 - 2 = 1$, while $CE_{2,2} = CE_{2,1} + L_{2,2} - C_2 \cdot w = 1 + 6 - 2 = 5$. In other words, N_2 has 5 cycles worth of work associated with subinterval t_2 , and will need $CE_{2,2}/C_2 = 5$ secs to process this old work before it can process newer events associated with the next subinterval¹. We now formalize the notion of *maximum cumulative excess*.

Definition 9 (Mace). *Mace is a time-series $Mace_{1\dots d}$ whose value $Mace_p$ at each subinterval t_p is the greatest cumulative excess (normalized by node CPU capacity) across all nodes for that subinterval, i.e., $Mace_p = \max_{N_i \in \mathcal{N}} CE_{i,p}/C_i$. The overall worst-case Mace ($Mace_{WC}$) is the greatest Mace across subintervals, i.e., $Mace_{WC} = \max_{t_p \in \tau} Mace_p$.*

Observe that Mace is a *deterministic* function of the CPU capacity and DLTS of all nodes. In Figure 3(c), we see that the Mace time-series is $\{Mace_1 = 1, Mace_2 = 5, Mace_3 = 3, Mace_4 = 2, Mace_5 = 4\}$, while $Mace_{WC} = 5$.

A. On $Mace_{WC}$ and Worst-Case Latency

Surprisingly, it turns out that $Mace_{WC}$ computed using DLTS is provably almost equal to the *actual* worst-case latency Lat_{WC} experienced by the distributed EPS during operation,

¹Note that processing the new events earlier is worse for latency, because older events are delayed even longer.

regardless of the actual loads imposed by individual operators and nodes at runtime. The following theorem formalizes this relationship. The proof (in Section V) is interesting in and of itself and provides fundamental clarity to the subtle inter-node and inter-operator interactions in an EPS. A stronger version extending the relation beyond worst-case (to averages and quantiles across time) is also covered in Section V.

Theorem 2 ($\text{Mace}_{\text{WC}} \approx \text{Lat}_{\text{WC}}$). *Given an EPS which executes the query graph \mathcal{G} according to stimulus time scheduling, and assuming that the clocks at all nodes are synchronized,*

$$\text{Mace}_{\text{WC}} \leq \text{Lat}_{\text{WC}} \leq \text{Mace}_{\text{WC}} + w + \epsilon$$

where ϵ is a small number (see proof in Section V for details).

We now develop some intuition behind this result, for a simple example scenario. Assume that there are three nodes (N_1, N_2, N_3) and three operators (O_1, O_2, O_3) in the EPS, with each operator O_i assigned to node N_i as in Figure 3(a). Let the CPU capacity of each node be $C_i = 1$ cycle per second, and the subinterval width be $w = 2$ seconds. Thus, the available CPU at each subinterval is $C_i \cdot w = 2$ cycles. The DLTS and CE of each node are shown in Figures 3(b) and 3(c).

For subinterval t_2 , note that the cumulative excess (CE) for nodes N_1, N_2 , and N_3 are 2, 5, and 3 cycles respectively. Thus, N_2 has the maximum CE of $\text{Mace}_2 = \text{CE}_{2,2}/C_2 = 5$ seconds. Let an event e arrive from outside at the end of subinterval t_2 (i.e., stimulus time is t_3). Figure 3(d) shows the progress of event e through the operators. We consider two phases separately (since $C_i = 1$, we drop this term below for clarity):

N_2 and upstream node N_1 : Event e will normally get processed at N_1 and reach N_2 at time $t_3 + \text{CE}_{1,2}$. In general, if there were more nodes upstream, e will reach N_2 at time $\leq t_3 + \text{CE}_{2,2}$ since $\text{CE}_{2,2} \geq \text{CE}_{*,2}$. Further, due to stimulus time scheduling, we know that as long as the e reaches N_2 at or before $t_3 + \text{CE}_{2,2}$, it will get processed at N_2 at time $t_3 + \text{CE}_{2,2} = t_3 + 5$. This is because scheduling at N_2 depends only on e 's stimulus time and not the time when e actually reaches N_2 .

N_2 and downstream node N_3 : Event e will get processed at N_2 and reach N_3 at time $t_3 + \text{CE}_{2,2} = t_3 + 5$. Since $\text{CE}_{2,2} \geq \text{CE}_{*,2}$, we know that at N_3 (and further downstream nodes if any), this event is guaranteed to have the earliest stimulus time (because $\text{CE}_{2,2}$ is maximum). Due to stimulus time scheduling, e will get processed at N_3 immediately and will be output at time $t_3 + \text{CE}_{2,2} = t_3 + 5$. We see that e 's latency (which is Lat_{WC}) of 5 seconds corresponds exactly to $\text{CE}_{2,2}$ and Mace_{WC} .

V. MACE'S EQUIVALENCE TO MAXIMUM LATENCY

We prove that using DLTS and stimulus time scheduling, Mace_{WC} equals Lat_{WC} to within a small margin of error. The main theorem is stated and proved in Section V-A, followed by a stronger version of the theorem which can be proved in a similar manner. Table I summarizes our notations. We make the following assumptions:

- (A-1) Without loss of generality, $t_1 = 0$ and $C_i = 1 \forall i$. Hence, all loads can be described directly in time units. During each subinterval, a node may perform w units of work.
- (A-2) For each input source, within each subinterval t_p , we assume that events have a constant inter-arrival time α , where the first event arrives at t_p , and the last event arrives at $t_{p+1} - \alpha$.
- (A-3) Within a particular subinterval, each operator O_j requires a constant amount of load ($\omega_{j,q}$ cycles) to process each event from its q^{th} input queue, which belongs to that subinterval.

Before stating the theorem, we introduce two lemmas:

Lemma 1 (Single Node Case). *Given the most latent event e with stimulus time t_p and latency Lat_p in a system with one node N_i ,*

$$0 \leq \text{Lat}_p \leq \text{CE}_{i,p-1} + L_{i,p}.$$

Also, if $\text{CE}_{i,p-1} + L_{i,p} - w > 0$,

$$\text{CE}_{i,p-1} + L_{i,p} - w \leq \text{Lat}_p.$$

Proof: (Sketch) We explain the bounds in the lemma:

Lower Bound If $\text{CE}_{i,p-1} + L_{i,p} - w > 0$, then we are unable to fully process the input during t_p . This implies that the most latent event, if it arrived at the last possible instant, could have as little latency as the amount of work left after t_p is over. This quantity is the previous excess ($\text{CE}_{i,p-1}$), plus the time to process the new load ($L_{i,p}$), minus the processing time (w) consumed during the current time interval.

Upper Bound In the worst case, the most latent event is guaranteed to have latency less than the latency it would have had if all input events belonging to t_p arrived at t_p . In this situation, the latency would be the time it takes to process the previous excess ($\text{CE}_{i,p-1}$), plus the time to process the new load ($L_{i,p}$). ■

Lemma 2 (Bottleneck Lemma). *Given a particular subinterval t_p , an operator O_j , the only operator running on node N_i , with q input queues and their associated load per event quantities (see A-3) for that subinterval $\omega_{j,1\dots q}$, if the operators which feed and consume events from O_j all reside on nodes with $\text{CE} \geq \text{CE}_{i,p}$, then O_j introduces at most:*

$$\lambda = \sum_{c=1\dots q} \omega_{j,c}$$

additional latency to the most latent event belonging to t_p .

Proof: (Intuition) Due to the constant inter-arrival time assumption (A-2), on an individual input queue basis, work associated with processing that input is equally spread across each time interval. If this work was scheduled to execute in a perfectly spread out fashion, no additional latency would be introduced by O_j since (1) any upstream operator (all residing on nodes with $\text{CE} \geq \text{CE}_{i,p}$) would feed work to O_j no faster than O_j could process it, and (2) all downstream operators (also all residing on nodes with $\text{CE} \geq \text{CE}_{i,p}$) would be unable to process their load faster than O_j could deliver work.

However, since events are scheduled to execute at discrete times, and fully utilize the processor while executing, events

TABLE I
SUMMARY OF MAIN TERMINOLOGY.

Symbol	Description	Reference
$\{N_1, \dots, N_n\}$	Set of nodes (machines) in the EPS	Def. 1
$\{O_1, \dots, O_m\}$	Set of operators in the EPS	Def. 1
C_i	Available CPU cycles per time unit, on node N_i	Def. 1
$\{t_1, \dots, t_d\}$	Division of time into segments	Sec. III-A
$\text{Lat}_{1\dots d}$	Max. latency across events in each subinterval	Def. 4
Lat_{WC}	Worst-case latency in EPS	Def. 4
$\sigma_{j,1\dots q}$	Selectivity of operator O_j , q^{th} input queue	Sec. III-B
$\omega_{j,1\dots q}$	Cycles/event imposed by operator O_j , q^{th} input	Sec. III-B
$t_{j,1\dots d}$	Deterministic Load Time-Series for operator O_j	Def. 5
$L_{i,1\dots d}$	Deterministic Load Time-Series for node N_i	Def. 6
$\text{CE}_{i,1\dots d}$	Cumulative Excess (cycles) time-series for node N_i	Def. 8
$\text{Mace}_{1\dots d}$	Maximum cumulative excess time-series	Def. 9
Mace_{WC}	Worst-case Maximum cumulative excess	Def. 9

may execute till a slightly later time than they would in the more continuous model described above. More specifically, in the worst case, each input other than the one with the most latent event e might process an event just prior to the proper processing time for e . Each of these events would then monopolize the CPU while being processed, following which event e gets processed. This sequence of actions results in the bound in the proof. Note that this sum λ is a very small number, as the typical time for an operator to process an event is on the order of microseconds. ■

A. Statement and Proof of Theorem 2

Given an EPS which executes the query graph \mathcal{G} according to stimulus time scheduling, and assuming that the clocks at all nodes are synchronized,

$$\text{Mace}_{\text{WC}} \leq \text{Lat}_{\text{WC}} \leq \text{Mace}_{\text{WC}} + w + \epsilon.$$

where ϵ is a small number which will be precisely specified later in the proof.

Proof: (By Contradiction) Assume the existence of an event with maximum latency Lat_{WC} s.t. $\text{Lat}_{\text{WC}} < \text{Mace}_{\text{WC}}$ or $\text{Lat}_{\text{WC}} > \text{Mace}_{\text{WC}} + w + \epsilon$.

Part 1 (Assume that $\text{Lat}_{\text{WC}} < \text{Mace}_{\text{WC}}$)

Let N_i to be the node with highest Mace (Mace_{WC}). Further, assume that all other nodes process their input instantly, and therefore introduce no latency beyond that introduced by N_i . We may now treat the EPS as a single-node system, where all operators on other nodes have been moved to N_i and have zero latency. Note that the latency experienced by any event on such a system will at most be equal to the worst-case latency on the original multi-node system, although Mace_{WC} is unchanged. As a result, a contradiction to our assumption ($\text{Lat}_{\text{WC}} < \text{Mace}_{\text{WC}}$) on this system implies a contradiction on our original multi-node EPS.

Let e be the event with stimulus time t_p = time of worst-case Mace with highest latency Lat_p :

$$\text{CE}_{i,p} = \max\{0, \text{CE}_{i,p-1} + L_{i,p} - w\} \quad (\text{by Definition 8})$$

Therefore, if $(\text{CE}_{i,p-1} + L_{i,p} - w) \leq 0$ then

$$\text{CE}_{i,p} = 0, \text{Lat}_p \geq 0 \implies \text{Lat}_p \geq \text{CE}_{i,p} = \text{Mace}_{\text{WC}}$$

Otherwise, if $(\text{CE}_{i,p-1} + L_{i,p} - w) > 0$ then

$$\text{CE}_{i,p} = \text{CE}_{i,p-1} + L_{i,p} - w,$$

$$\text{Lat}_p \geq \text{CE}_{i,p-1} + L_{i,p} - w \quad (\text{by Lemma 1}) \implies$$

$$\text{Lat}_p \geq \text{CE}_{i,p} = \text{Mace}_{\text{WC}}$$

Combining the above with Definition 4, we get:

$$\text{Lat}_{\text{WC}} \geq \text{Lat}_k \geq \text{Mace}_{\text{WC}}$$

A contradiction has been reached.

Part 2 (Assume that $\text{Lat}_{\text{WC}} > \text{Mace}_{\text{WC}} + w + \epsilon$)

Let e be the event with maximum latency $\text{Lat}_{\text{WC}} > \text{Mace}_{\text{WC}} + w + \epsilon$. Let t_p be the stimulus time for e . Also, let N_i be the node with maximum CE at time t_p . If all nodes other than N_i introduced no latency (the operators had zero latency), this would be equivalent to a single-node EPS with node N_i and with worst-case latency at most that of the original system. For this alternate system, we have

$$\begin{aligned} \text{Lat}_{\text{WC}} &\leq \text{CE}_{i,p-1} + L_{i,p} \quad (\text{by Lemma 1}) \implies \\ \text{Lat}_{\text{WC}} - w &\leq \text{CE}_{i,p-1} + L_{i,p} - w \implies \\ \text{Lat}_{\text{WC}} - w &\leq \max\{0, \text{CE}_{i,p-1} + L_{i,p} - w\} \implies \\ \text{Lat}_{\text{WC}} - w &\leq \text{CE}_{i,p} \quad (\text{by Definition 8}) \implies \\ \text{Lat}_{\text{WC}} &\leq \text{CE}_{i,p} + w \end{aligned}$$

Note that since $\text{Mace}_{\text{WC}} \geq \text{CE}_{i,p}$, we have

$$\text{Lat}_{\text{WC}} \leq \text{Mace}_{\text{WC}} + w.$$

We now begin a process of “activating” nodes other than N_i by allowing them and their associated operators to contribute to worst-case latency. To avoid a contradiction, the accumulated latency from activating these nodes must exceed some small value ϵ . Therefore, if we specify a small value ϵ which is guaranteed to bound this added latency, we have reached a contradiction.

Consider that, without affecting latency, we may treat all operators on a given node as a single operator with many inputs and many outputs. In this fashion, this “fused” operator may be considered a single operator running exclusively on its associated node. Note that this is a precondition for Lemma 2. Now, consider activating the nodes in descending CE order. Note that this implies that at the time a node is activated, all inputs and outputs go to nodes with equal or higher CE, and Lemma 2 informs us that activating the node may not introduce more than some small amount (λ) of latency. This amount is accumulated as all nodes are activated, and ϵ assigned the result. We have reached a contradiction. ■

B. Estimating Latency Averages/Quantiles

Theorem 3 (Stronger Version of Theorem 2). *Given an EPS which executes a query graph \mathcal{G} according to stimulus time scheduling, assuming synchronized clocks at all nodes, and assuming that Lat_p is the highest latency of any output with stimulus time t_p ,*

$$\text{Mace}_p \leq \text{Lat}_p \leq \text{Mace}_p + w + \epsilon.$$

Proof: Omitted for brevity. Since CE is self-containing (i.e., each subinterval incorporates the effects of earlier subintervals), the proof uses a similar reasoning as in Theorem 2. ■

In other words, if we divide output events into sets based on the subinterval they belong to, we can accurately estimate the maximum latency for each set (subinterval). Thus, we can estimate the average and quantiles (across time) of maximum latency. Note that if stimulus times are unique, as w gets smaller, the above estimates of average and quantiles (across time) of maximum latency converge to the actual average and quantiles (across events) of latency.

VI. IMPLEMENTATION DETAILS

A. Stimulus Time Scheduling

An EPS scheduler typically runs on a single thread per CPU core, and chooses operators for execution on that core. When an event enters the EPS from outside, we attach the current wall-clock time to the event as its stimulus time. When an operator receives an event or punctuation with stimulus time t , any output produced by the operator as a consequence is attached a stimulus time of t . Note that stimulus times are retained without modification across machine boundaries.

Efficient Implementation The naive method of achieving stimulus time scheduling is to use *priority queues* (*PQs*) ordered by stimulus time, to implement event queues. This gives $O(\lg n)$ enqueue and dequeue, where n is the number of events in the queue. We reduce the cost to a constant using the following technique. Each event queue is implemented as a collection of k FIFO queues, where k is the number of unique paths from this queue (edge) to the sources in the query graph. Note that k is at most a small constant known at plan compilation time. Event enqueue translates into an enqueue into the correct FIFO queue (based on the event’s path), while dequeue is similar to a k -way merge over the head elements of the k FIFO queues. Both are $O(\lg k)$ operations using a small tree and min-heap respectively. Correctness follows from the fact that operators process input in stimulus time order, causing each FIFO queue to always be in stimulus time order.

The scheduler maintains a priority queue (ordered by earliest event stimulus time) of active operators. When invoked, the scheduler schedules the operator having the event with lowest stimulus time in its queue. Batching of events amortizes the scheduler cost incurred at the time of selecting an operator for execution, without causing our latency estimate to diverge by a significant amount.

B. Computing Statistics

We first derive the external event arrival time-series; this can be obtained by observing event arrivals in the past or may be inferred based on models of expected input arrival distribution. We also maintain statistics for each operator O_j in the query graph as follows. Operator cycles/event (ω_j) is determined by measuring the time taken for each call to the operator and number of events processed during the call. Scheduling overhead is incorporated into the operator cost. Operator selectivity (σ_j) is measured by maintaining counters for the number of input and output events. Note that all our parameters are independent of the actual operator-node mapping and node CPU cost, which makes them particularly suited to operator placement, system provisioning, and user reporting. We cover the details of estimating operator parameters for unseen dataflow plans (during plan selection) in Section IX-A.

C. Computing DLTS and Mace

Let us first assume that each operator has only one input queue. For each operator O_j , we first derive the *input stimulus time-series* $A_{j,1\dots d}$ — the value $A_{j,p}$ at each subinterval t_p

is simply the number of input events to O_j that belong to subinterval t_p . We compute $A_{j,1\dots d}$ in a bottom-up fashion starting from the source operators. For a source operator O_s , $A_{s,1\dots d}$ is simply the corresponding external event arrival time-series. For an operator O_j whose upstream parent operator is $O_{j'}$, we have $A_{j,1\dots d} = A_{j',1\dots d} \cdot \sigma_{j'}$. Now, the DLTS of any operator O_j is easy to calculate as $l_{j,1\dots d} = A_{j,1\dots d} \cdot \omega_j$. Once we have the DLTS for each operator, CE and Mace are easy to compute by directly applying Definitions 6, 8, and 9. The overall complexity of these computations is $O(d \cdot m)$, for d subintervals and m operators.

In case of an operator with multiple inputs, statistics are maintained for each input separately; we use a function (usually a linear combination) to derive the DLTS of the operator and the input stimulus time-series for its child operators.

Note that the model presented here assumes, for each operator, linearity in both the output rate and CPU load relative to input rates. Clearly this is a poor choice for some operators (e.g. joins can be quadratic [31]). For these operators, more complex models involving non-linear terms are needed. Fortunately, since we are basing the fitting of these models on a great deal of input data, there is no risk of overfitting. Note that assigning these models to relational operators is a well-researched area in database query optimization.

VII. EXTENSIONS

A. Incorporating Other Scheduling Policies

We use stimulus time scheduling to establish the equivalence of Mace to latency. While this scheduling policy is very attractive (as discussed in Section III-D), we recognize that an EPS may wish to use different policies to achieve other system properties (e.g., taking operator priorities into account). We can extend the Mace-Lat relationship so that: (1) we enforce stimulus time scheduling only *across* subintervals, i.e., all events in subinterval t_i are scheduled before any event in subinterval t_j , $\forall j > i$; (2) for events within the same subinterval, we can use any scheduling policy.

In this case, Lemma 2 guarantees a higher λ (and hence ϵ). Specifically, we have $\lambda = \sum_{c=1\dots q} L_{i,p}$. Thus, by making the subinterval width w larger, we can incorporate other scheduling policies. In practice, no scheduling policy would delay event processing beyond some reasonable time, so w does not have to be arbitrarily large. The tradeoff is a looser bound for latency. Thus, w serves as a tuning parameter to balance latency predictability against any advantages that another scheduling policy might give the EPS.

B. Handling Low-Bandwidth & High-Latency Network Links

In this paper, we have focused on EPSs running inside data centers with high-bandwidth and low-latency interconnects. Our experiments in Section VIII using a real data center deployment validate this assumption in practical scenarios. We now extend our solution to relax these assumptions.

Link Capacity Link capacity is just another resource that introduces latency due to queuing of events. In network-

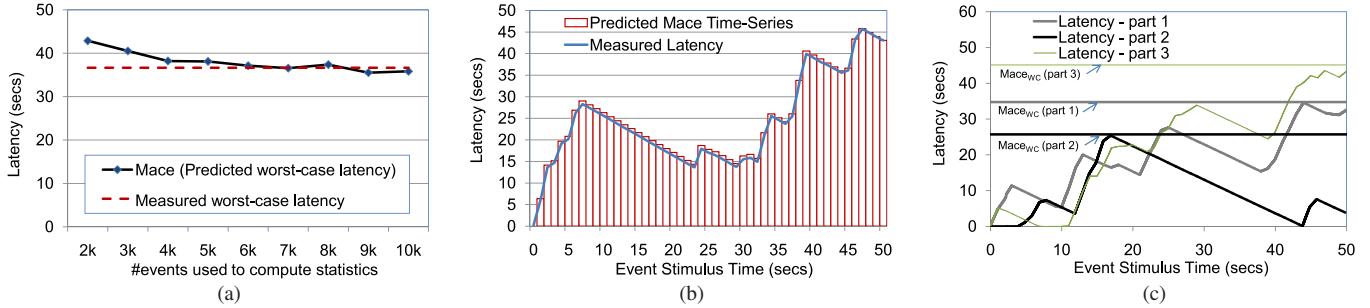


Fig. 4. (a) Estimated vs. actual Lat_{WC} , increasing training size. (b) Latency and Mace time-series. (c) Latency and Mace_{WC} for different data chunks.

constrained scenarios, we treat link capacity (bytes/sec) like CPU capacity, and take into account how load (bytes) accumulates at network links when computing Mace. Thus, the techniques work unmodified, except for the addition of new nodes corresponding to network links in the query graph.

Propagation Delay Refer to Definition 1. Let the *machine graph* be a graph over \mathcal{S} ; it is derived from the query graph by merging all operators S_i on the same machine N_i into one vertex. We set the weight of an edge from S_i to S_j in the machine graph to the propagation delay (PD) between nodes N_i and N_j . For each machine N_j , we pre-compute P_{\min}^j and P_{\max}^j as the minimum and maximum cost paths, over all paths in the graph from a source to a sink, that contain vertex S_j . During estimation, we calculate Mace for each machine using our techniques, taking PD into account when computing DLTS. The adjusted Mace (called Mace') for a node N_j has bounds $\text{Mace} + P_{\min}^j \leq \text{Mace}' \leq \text{Mace} + P_{\max}^j$. These per-machine Mace' bounds are used to modify the Lat_{WC} bounds of Thm. 2; the lower bound for Lat_{WC} uses the maximum (across nodes) lower bound of Mace', while the upper bound for Lat_{WC} uses the maximum (across nodes) upper bound of Mace'.

VIII. EVALUATION

We now show that despite using a seemingly simple system model, we achieve highly accurate latency estimation results in practice. This validates our provable result for real datasets using a commercial EPS running in a multi-node data center.

Setup We use StreamInsight [5] to run all experiments. We modified the EPS as described in Section VI. Single-node experiments (the default) were performed on a 3.0GHz Intel Core 2 Duo PC with 3GB of main memory, running Windows Vista. Multi-node experiments were performed on a cluster of 13 2.33GHz Intel Xeon machines with 4GB memory, running Windows Server 2008 and connected over Gigabit Ethernet.

Event Workload We use real Web clickstream data (see Example 1) as input to our experiments. Each event is a report including the timestamp of the click, source and destination URLs, and other information including user demographics (gender, age, etc.). The data is partitioned by source URL domain, giving five datasets each with around 200k events. In order to show the effects of different *event arrival patterns*, we feed events to the system using a load generator motivated by the popular On-Off model [3]. Briefly, events are generated as a sequence of high and low load periods. The ratio between

average high load and average low load is set to 100 (to model burstiness), while the ratio between the average durations of high and low load is set to 0.33. The duration of each period is drawn from an exponential distribution. During each period, events are generated with exponentially distributed inter-arrival times. Results using the original arrival patterns directly were identical and slightly less insightful; they are omitted for space.

Query Workload We use a set of one to five queries. Each query outputs results for a particular domain, indicating the percentage of male and female visitors. We use a sliding window with results reported periodically. The query has 14 operators, including projects, joins, selects, windowing, input, output, and custom operators such as those to extract parts of the URL and perform user-defined computations. Such a query is useful for demographic targeting in Web advertising.

1) Varying Amount of Training Data We run a single query with 14 operators over a small number of events (training data) to estimate operator statistics of selectivity and cycles/event. We then use these statistics to compute Mace_{WC} for a dataset of 75k events, for a particular event arrival pattern generated by our workload generator. Subinterval width is set to 1 second. We compare our Mace_{WC} to the measured worst-case latency (Lat_{WC}) incurred by actually executing our EPS for that event workload. Figure 4(a) shows the quality of our cost estimate as we increase the number of events used to compute statistics. We see that with as few as 6k events (8% of total events), our computed metric of Mace_{WC} estimates Lat_{WC} with an error of less than 3%.

2) Estimating Latency Across Time We use the same setup as before and first compute the operator statistics. We then compute the DLTS for a generated event arrival pattern, which is used to compute the entire Mace time-series. We then execute the query to measure the actual latency for each output event. In Figure 4(b), we plot both the Mace time-series and the measured latencies as a function of event stimulus time. We see that latency rises and falls with time, but Mace for each subinterval tracks the highest latency within that subinterval very closely. For simplicity, we will focus on worst-case Mace (Mace_{WC}) for the remaining experiments.

3) Predicting Mace_{WC} for Different Data Chunks We split the event dataset into three parts corresponding to different time chunks. We use the first chunk to compute operator statistics. We then compute Mace_{WC} assuming a different event arrival pattern applied to each of the three chunks. Fig-

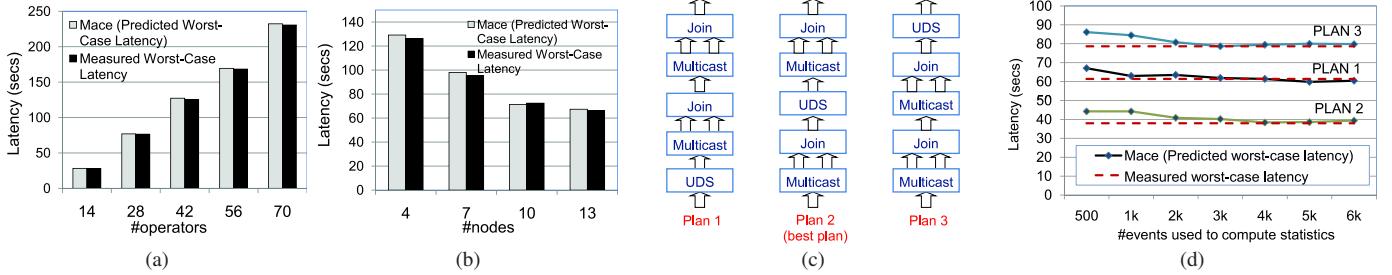


Fig. 5. (a) Increasing number of simultaneous CQs. (b) Increasing number of nodes in cluster. (c) Three CQ plans – only 5 of 14 operators shown for brevity. (d) Estimated vs. actual latency, increasing training size.

ure 4(c) shows the computed Mace_{WC} and the actual measured latency variations with time, for each portion of the dataset. We see that using our techniques, Mace_{WC} estimates worst-case latency accurately (within 4%) for different portions of the workload experiencing different event arrival patterns, given knowledge of only the original operator statistics and the expected event arrival workload.

4) Scale-Up to Multiple Operators We increase the number of operators running on a single node from 14 to 70, by running multiple query instances. Each query uses a dataset for clicks from a different domain, and a different event arrival pattern. We divide the dataset into two portions, derive operator statistics using the first portion, and make estimations for the second portion. Figure 5(a) reports the estimated and actual worst-case latencies as we increase the number of operators. Even with 70 operators, our estimate of worst-case latency closely matches the measured value.

5) Scale-Up to Multiple Nodes We increase the number of nodes in the cluster from 4 to 13. For each setting, we choose a random partitioning of 42 operators (running queries on the real dataset) across the machines. We first estimate worst-case latency for that partitioning using our technique, and then measure the actual worst-case latency on the cluster. Figure 5(b) shows that our estimate of worst-case latency closely matches the measured value (with less than 3% error) even in the highly distributed scenario.

IX. APPLICATIONS OF LATENCY ESTIMATION

We discuss how Mace can be used for the applications of plan selection and operator placement. We assume that latency is the metric to be minimized during selection and placement. It is important to note that if the EPS wishes to optimize for other metrics in combination with latency (e.g., a combination of throughput, resiliency, and latency), this could easily be incorporated during the optimization process. Other applications such as admission control, system provisioning, and user reporting are covered in our technical report [12].

A. Plan Selection

Our goal during plan selection is to choose the plan with lowest worst-case latency. Based on Theorem 2, we can formulate plan selection as the optimization problem: *Find the plan that minimizes Mace_{WC}*. We can formulate similar problems for other latency goals too, e.g., *Find the plan that*

minimizes average or 99th percentile (across time) of Mace.

Parameter Estimation In order to predict latency for a plan, we need to estimate selectivity and cycles/event for each operator. One alternative is to adapt techniques from traditional databases, such as building statistics on incoming event data and estimating statistics using knowledge of operator behavior. For example, we can estimate the selectivity of a filter using a histogram on the column being filtered. Another approach that works well in an EPS is to actually execute the plan over a small subset of incoming data, and measure the statistics. Our experiments below show that the latter approach works very well for plan selection, finding the best plan using a sample of just 500 events (< 1% of the total events).

Navigating the Search Space We have a search space of CQ plans obtained using techniques such as query rewriting, reordering joins and predicates, operator fusing (replacing inter-operator queues with function calls), etc. Navigating this search space can use traditional schemes like branch-and-bound or dynamic programming [26].

We can estimate the “goodness” of a plan by assuming a single node² and computing Mace_{WC} using the techniques described in Section VI, in time $O(d \cdot m)$. Note that due to the long-running nature of CQs and the potentially high reward of good plans, an EPS can adopt an iterative approach of *periodic re-optimization*, similar to techniques proposed for traditional databases [24]. Re-optimization can be performed when the statistics have been detected to have changed significantly (e.g., using techniques proposed in [21]) — for instance, at the *re-optimization points* indicated in Figure 1.

A.1) Evaluation of Plan Selection

We wish to validate our hypothesis that operator statistics derived using a small portion of events can be used for accurate latency prediction by an optimizer. We use a single machine in conjunction with synthetic data (75k events) on stock trades. Each event contains a price, volume, and review. Prices are modeled as a random walk with 80% probability of increasing, while volumes are drawn from a truncated uniform random distribution. We use the following CQ: *Apply a user-defined select (UDS) to the reviews of pairs of falling trades (i.e., trades with price lower than the immediately preceding trade) within a one minute window, which have similar volumes*.

²While the best plan could depend to a limited extent on the operator placement in a distributed setting, we treat these independently for simplicity, and discuss operator placement in Section IX-B.

```

1 Mace-HC(time-budget  $b$ ) begin
2    $s \leftarrow \text{CURRENTTIME}()$ ; // start time
3    $m \leftarrow \infty$ ; // Worst-case Mace
4   while CURRENTTIME() -  $s < b$  do
5      $p \leftarrow \text{random placement};$ 
6     Hill-climb  $p$  to local optimum;
7      $m' \leftarrow \text{Mace}_{\text{WC}}(p);$ 
8     if  $m' < m$  then  $m \leftarrow m';$ 
9     if insignificant improvement for many
       iterations then break;
10    return  $m;$ 
11 end

```

(a)

parameter	value
# independent sources	5
Prob. operator is anti-correlated	0.1
Ratio avg. rate high / low load	10
Ratio avg. duration high / low load	0.25
Skew for #ops per independent source	1.0
Min/Max operator selectivity	0.2/2
Skew for operator selectivity	1.5
Number of selectivity ranges	20
Avg. system load (idealistic)	0.75
Max. #hill-climb steps per iteration	10,000

(b)

Fig. 6. (a) Operator placement algorithm Mace-HC. (b) Summary of parameters used in experiments. (c) Mace-HC vs. other operator placement schemes.

The optimizer explores three CQ plans for this query, as shown in Figure 5(c). It first measures statistics using a small sample of events, and then computes Mace_{WC} for comparing the plans. In Figure 5(d), we show Mace_{WC} for each plan, as we increase the event sample size. We also execute each of the alternate plans and show the measured worst-case latency in the figure. We note that: (1) using just 500 events to compute statistics gives enough accuracy to differentiate between the plans, and (2) the best plan is one where the expensive filter is neither at the source nor at the sink of the CQ.

B. Operator Placement

Given a query graph \mathcal{G} , we wish to find an assignment of operators in \mathcal{G} to nodes, that minimizes worst-case latency. As before, we can formulate operator placement as an optimization problem, e.g., “*Find the operator placement that minimizes Mace_{WC}* ”. Operator placement is a dominant [32, 33] form of query optimization in an EPS.

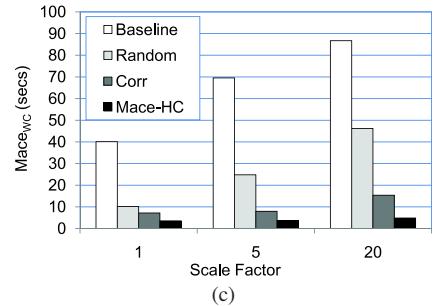
We can show that operator placement to minimize Mace_{WC} is NP-hard, by a reduction from *vector scheduling* [13] (our technical report [12] has the details). However, it turns out that a simple probabilistic placement algorithm that assigns each operator uniformly at random to a node achieves a very good approximation ratio, is very fast, and does well when there are many more operators than nodes. Based on this observation, we propose an algorithm called Mace-HC in Figure 6(a), that repeatedly performs randomly seeded hill-climbing until a time budget is exhausted or there is insignificant improvement after many iterations. Since the goal is to minimize Mace_{WC} , each hill-climbing step starts from a random operator placement and greedily moves operators away from the bottleneck node (one with the highest Mace_{WC}), such that overall Mace_{WC} improves.

Runtime Complexity Assume that we have m operators, n nodes, and d subintervals. Random placement has complexity $O(m)$. The complexity of hill climbing depends on the number of successful operator migration steps. At each step, it costs $O(n \cdot d)$ to find the bottleneck node and the target node. In the worst case, the algorithm has to try all operators on the node, giving a total runtime complexity of $O(m \cdot n \cdot d)$.

B.1) Evaluation of Operator Placement

We evaluate our placement algorithm that directly minimizes worst-case latency, against the following:

- **Baseline**, a simple randomized algorithm that places oper-



ators uniformly at random without hill-climbing. Baseline was run 10,000 times to obtain the distribution of runtime and result quality for simple random operator placements.

- **Random**, which is an advanced version of Baseline. It performs several rounds of random placements and remembers the best placement (with lowest Mace_{WC}) seen so far.
- **Corr**, the load correlation based placement algorithm [32]. It seeks to minimize load variance on each node and maximize correlation of the load time-series between different nodes, while balancing average load across nodes. Note that Corr optimizes for a different metric; thus, the primary intention of comparing to Corr is to validate our expectation that by not optimizing directly for latency, we may produce placements that are suboptimal for latency.

Setup We simulate the event arrival pattern at each operator, and only compare the placement algorithms (run at a central server) in terms of convergence speed and the Mace_{WC} value of the best placement found. Mace_{WC} is a reasonable basis for comparison since it was shown to be equivalent to worst-case latency. We simulate up to 400 nodes in our experiments. Our server is an Intel Core 2 Duo 2.4GHz PC with 2GB of main memory running Windows Vista.

Workload We generate multiple event sources as in Section VIII. Operators are assigned to the sources using a Zipf distribution with parameter 1.0. To model the effect of upstream operators, we multiply load by an operator-specific *selectivity factor*, drawn uniformly from a range that is chosen out of several predefined ranges using a Zipf distribution. The default parameters are shown in Figure 6(b). To make the algorithms comparable, we give them the same time budget, determined by the runtime of Corr until an average load correlation of $\theta = 0.8$ (used in [32]) is reached.

Increasing Network Nodes and Operators We increase the number of nodes and operators, and compare the algorithms. Scale factor X corresponds to $20 \cdot X$ nodes and $200 \cdot X$ operators. Figure 6(c) shows Mace_{WC} for each algorithm, for scale factors 1, 5, and 20. We see that Baseline and Random perform badly. Corr is better than the simpler schemes, while Mace-HC gives the lowest worst-case latency.

Runtime and Result Quality We studied how fast Mace-HC converges to a good operator placement, and found that across the various scale factors, Mace-HC quickly converges and produces low-latency placements at least an order of magnitude faster than simpler schemes (see [12] for details).

X. RELATED WORK

Cost Metrics Many cost metrics have been proposed in the past for event-based systems. *Intermediate tuples* [18] tries to find a plan that reduces the number of tuples flowing between join operators. *Feasible set size* [33] tries to maximize the set of input rate combinations that do not result in overload. *Load correlation* [32] tries to reduce average latency indirectly by minimizing load variance and maximizing load correlation across nodes. Other metrics proposed include *resource usage* [9] and *output rate* [31, 6]. *SBON* [25] and *SAND* [4] use network bandwidth-delay product as the metric. We observe that each solution excels in particular areas.

On the other hand, Mace has the unique property of being a *provably* accurate estimator of latency, an intuitive and significant user metric for many applications. Interestingly, the above systems can leverage Mace to either incorporate system latency into their optimization goal, or provide accurate latency reporting alongside their own metric.

Queuing Systems Queuing theory [19] has provided valuable insights into scheduling decisions in multi-operator and multi-resource queuing systems, but results are usually limited by high computational cost and strong assumptions about underlying data and processing cost distributions. We make no assumptions about the underlying distribution, with a provable latency result while remaining in the discrete domain, which makes our solution efficient and practical to implement.

Traditional Solutions Query optimization in databases is a well-studied problem [24, 26]. In addition, there have been studies on load balancing in traditional distributed and parallel systems [17, 22]. These techniques do not carry over directly to event processing [1, 14], because our queries are long running, disk I/O is not the bottleneck, operator scheduling is different, there is greater value to periodic re-optimization, and per-tuple load balancing decisions are too costly.

Scheduling and QoS Classic scheduling schemes such as FIFO and SJF generally do not focus on the problem of continuously scheduling rapidly arriving short jobs to achieve predictable and low worst-case latency. Some proposed scheduling techniques for EPSSs [7, 11] may have a side-effect of improving latency. Scheduling schemes in real-time and main-memory databases [2, 20, 23] are related, but deal with a different scenario and do not usually focus on worst-case latency. QoS-aware load shedding [28] has been proposed, while Tu et al. [30] handle QoS using adaptation and admission control. Our work is complementary — we propose a cost estimation solution with a provably close relation to latency, and use it to solve important applications.

XI. CONCLUSIONS

Latency is a metric that is significant to users in many commercial real-time applications. In this paper, we first suggest a stimulus time based scheduling policy that works very well for such applications. We propose a new latency estimation technique that produces a metric called Mace. We show that Mace is provably equivalent to maximum system latency

in a complex distributed event processing system. Mace is intuitive, easy to compute, and applicable to problems such as optimization, placement, provisioning, admission control, and user reporting of system misbehavior. Experiments using real datasets and a cluster deployment with StreamInsight validate our ability to estimate latency at high accuracy, and the use of Mace in applications such as plan selection and distributed operator placement. Finally, we note that Mace’s relation to latency is more generally applicable to any event-queue-based distributed workflow with control over scheduling.

REFERENCES

- [1] D. Abadi et al. The design of the Borealis stream processing engine. In *CIDR*, 2005.
- [2] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. *ACM TODS*, 1992.
- [3] A. Adas. Traffic models in broadband networks. *IEEE Comm.*, 1997.
- [4] Y. Ahmad and U. Çetintemel. Network-aware query processing for stream-based applications. In *VLDB*, 2004.
- [5] M. Ali et al. Microsoft CEP Server and Online Behavioral Targeting. In *VLDB*, 2009 (demonstration).
- [6] A. Ayad and J. Naughton. Static optimization of conjunctive queries with sliding windows over infinite streams. In *SIGMOD*, 2004.
- [7] B. Babcock et al. Chain: Operator scheduling for memory minimization in data stream systems. In *SIGMOD*, 2003.
- [8] M. Balazinska et al. Fault-tolerance in the Borealis distributed stream processing system. In *SIGMOD*, 2005.
- [9] M. Cammert et al. A cost-based approach to adaptive resource management in data stream systems. *IEEE TKDE*, 2008.
- [10] D. Carney et al. Monitoring streams — a new class of data management applications. In *VLDB*, 2002.
- [11] D. Carney et al. Operator scheduling in a data stream manager. In *VLDB*, 2003.
- [12] B. Chandramouli et al. Accurate latency estimation in a distributed event processing system. Technical report, Microsoft Research (MSR-TR-2010-146).
- [13] C. Chekuri and S. Khanna. On multi-dimensional packing problems. In *SODA*, 1999.
- [14] C. Cranor et al. Gigascope: A stream database for network applications. In *SIGMOD*, 2003.
- [15] Y. Cui et al. Tracing the lineage of view data in a warehousing environment. *ACM TODS*, 2000.
- [16] G. DeCandia et al. Dynamo: Amazon’s Highly Available Key-Value Store. In *SOSP*, 2007.
- [17] M. Garofalakis and Y. Ioannidis. Multi-dimensional resource scheduling for parallel queries. In *SIGMOD*, 1996.
- [18] L. Golab and M. T. Ozsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, 2003.
- [19] D. Gross and C. Harris. *Fundamentals of Queueing Theory*. Wiley-Interscience, 1998.
- [20] J. Haritsa et al. Earliest deadline scheduling for real-time database systems. In *IEEE RTSS*, 1991.
- [21] D. Kifer, S. Ben-David, and J. Gehrke. Detecting change in data streams. In *VLDB*, 2004.
- [22] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 2000.
- [23] S. Listgarten and M. Neimat. Modelling costs for a MM-DBMS. In *RTDB*, 1996.
- [24] V. Markl et al. Robust Query Processing Through Progressive Optimization. In *SIGMOD*, 2004.
- [25] P. Pietzuch et al. Network-aware operator placement for stream-processing systems. In *ICDE*, 2006.
- [26] P. Selinger et al. Access path selection in a relational database management system. In *SIGMOD*, 1979.
- [27] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *PODS*, 2004.
- [28] N. Tatbul et al. Load shedding in a data stream manager. In *VLDB*, 2003.
- [29] The Network Time Protocol. <http://www.ntp.org/>.
- [30] Y. Tu, Y. Xia, and S. Prabhakar. Quality of service adaptation in data stream management systems: A control-based approach. In *VLDB*, 2004.
- [31] S. Viglas and J. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD*, 2002.
- [32] Y. Xing et al. Dynamic load distribution in the Borealis stream processor. In *ICDE*, 2005.
- [33] Y. Xing et al. Providing resiliency to load variations in distributed stream processing. In *VLDB*, 2006.