Efficiently Measuring Bandwidth at All Time Scales

Frank Uyeda* Luca Foschini† Fred Baker‡ Subhash Suri† George Varghese*
*U.C. San Diego †U.C. Santa Barbara ‡Cisco

Abstract

The need to identify correlated traffic bursts at various, and especially fine-grain, time scales has become pressing in modern data centers. The combination of Gigabit link speeds and small switch buffers have led to "microbursts", which cause packet drops and large increases in latency. Our paper describes the design and implementation of an efficient and flexible end-host bandwidth measurement tool that can identify such bursts in addition to providing a number of other features. Managers can query the tool for bandwidth measurements at resolutions chosen after the traffic was measured. The algorithmic challenge is to support such *a posteriori* queries without retaining the entire trace or keeping state for all time scales. We introduce two aggregation algorithms, Dynamic Bucket Merge (`DBM`) and Exponential Bucketing (`EXPB`). We show experimentally that `DBM` and `EXPB` implementations in the Linux kernel introduce minimal overhead on applications running at 10 Gbps, consume orders of magnitude less memory than event logging (hundreds of bytes per second versus Megabytes per second), but still provide good accuracy for bandwidth measures at any time scale. Our techniques can be implemented in routers and generalized to detect spikes in the usage of any resource at fine time scales.

1 Introduction

How can a manager of a computing resource detect bursts in resource usage that cause performance degradation without keeping a complete log? The problem is one of extracting a needle from a haystack; the problem gets worse as the needle gets smaller (as finer-grain bursts cause drops in performance) and the haystack gets bigger (as the consumption rate increases). While our paper addresses this general problem, we focus on detecting bursts of bandwidth usage, a problem that has received much attention [6, 16, 18] in modern data centers.

The simplest definition of a *microburst* is the transmission of more than B bytes of data in a time interval t on a single link, where t is in the order of 100's of microseconds. For input and output links of the same speed, bursts must occur on several links at the same time to overrun a switch buffer, as in the Incast problem [8, 16]. Thus, a more useful definition is the sending of more than B bytes in time t over *several* input links that are destined to the same output switch port. This general definition requires detecting bursts that are correlated in time across several input links.

Microbursts cause problems because data center link speeds have moved to 10 Gbps while commodity switch buffers use comparatively small amounts of memory (Mbytes). Since high-speed buffer memory contributes significantly to switch cost, commodity switches continue to provision shallow buffers, which are vulnerable to overflowing and dropping packets. Dropped packets lead to TCP retransmissions which can cause millisecond latency increases that are unacceptable in data centers.

Administrators of financial trading data centers, for instance, are concerned with the microburst phenomena [4] because even a latency advantage of 1 millisecond over the competition may translate to profit differentials of $100 million per year [14]. While financial networks are a niche application, high-performance computing is not. Expensive, special-purpose switching equipment used in high-performance computing (e.g. Infiniband and FiberChannel) is being replaced by commodity Ethernet switches. In order for Ethernet networks to compete, managers need to identify and address the fine-grained variations in latencies and losses caused by microbursts. At the core of this problem is the need to identify the bandwidth patterns and corresponding applications causing these latency spikes so that corrective action can be taken.

Efficient and effective monitoring becomes increasingly difficult as faster links allow very short-lived phenomenon to overwhelm buffers. For a commodity 24-port 10 Gbps switch with 4 MB of shared buffer, the buffer can be filled (assuming no draining) in 3.2 msec by a single link. However, given that bursts are often correlated across several links and buffers must be shared, the time scales at which interesting bursts occur can be ten times smaller, down to 100's of μs. Instead of 3.2 msec, the buffer can overflow in 320 μs if 10 input ports each receive 0.4 MB in parallel. Assume that the strategy to identify correlated bursts across links is to first identify bursts on single links and then to observe that they are correlated in time. The single link problem is then to efficiently identify periods of length t where more than B bytes of data occur. Currently, t can vary from hundreds of microseconds to milliseconds and B can vary from 100's of Kbytes to a few Mbytes. Solving this problem *efficiently* using minimal CPU processing and logging bandwidth is one of the main concerns of this paper.

Although identifying "bursts" on a single link for a range of possible time scales and byte thresholds is chal-

lenging, the ideal solution should do two more things. First, the solution should efficiently extract flows responsible for such bursts so that a manger can reschedule or rate limit them. Second, the tool should allow a manager to detect bursts correlated in time across links. While the first problem can be solved using heavy-hitter techniques [15], we briefly describe some new ideas for this problem in our context. The second problem can be solved by archiving bandwidth measurement records indexed by link and time to a relational database which can then be queried for persistent patterns. This requires an efficient summarization technique so that the archival storage required by the database is manageable.

Generalizing to Bandwidth Queries: Beyond identifying microbursts, we believe that modeling traffic at fine time scales is of fundamental importance. Such modeling could form the basis for provisioning NIC and switch buffers, and for load balancing and traffic engineering at fine time scales. While powerful, coarse-grain tools are available, the ability to flexibly and efficiently measure traffic at different, and especially fine-grain, resolutions is limited or non-existent.

For instance, we are unable to answer basic questions such as: what is the distribution of traffic bursts? At which time-scale did the traffic exhibit burstiness? With the identification of long-range dependence (LRD) in network traffic [9], the research community has undergone a mental shift from Poisson and memory-less processes to LRD and bursty processes. Despite its widespread use, however, LRD analysis is hindered by our inability to estimate its parameters unambiguously. Thus, our larger goal is to use fine-grain measurement techniques for fine-grain traffic modeling.

While it is not difficult to choose a small number of preset resolutions and perform measurements for those, the more difficult and useful problem is to support traffic measurements for *all time scales*. Not only do measurement resolutions of interest vary with time (as in burst detection), but in many applications they only become critical *after the fact*, that is, after the measurements have already been performed. Our paper describes an end-host bandwidth measurement tool that succinctly summarizes bandwidth information and yet answers general queries at arbitrary resolutions without maintaining state for all time scales.

Some representative queries (among many) that we wish such a tool to support are the following:

1. What is the *maximum* bandwidth used at time scale t?
2. What is the *standard deviation* and 95th percentile of the bandwidth at time scale t?
3. What is the *coarsest* time scale at which bandwidth exceeds threshold L?

In these queries, the query parameters t or L are chosen *a posteriori* — after all the measurements have been performed, and thus require supporting all possible resolutions and bandwidths.

Existing techniques: All the above queries above can be easily answered by keeping the entire packet trace. However, our data structures take an order of magnitude less storage than a packet trace (even a sampled packet trace) and yet can answer flexible queries with good accuracy. Note that standard summarization techniques (including simple ones like SNMP packet counters [1]) and more complex ones (e.g., heavy-hitter determination [13]) are very efficient in storage but must be targeted towards a particular purpose and at a fixed time scale. Hence, they cannot answer flexible queries for arbitrary time scales.

Note that sampling 1 in N packets, as in Cisco NetFlow [2], does not provide a good solution for bandwidth measurement queries. Consider a 10 Gbps link with an average packet size of 1000 bytes. This link can produce 10 million packets per second. Suppose the scheme does 1 in 1000 packet sampling. It can still produce 10,000 samples per second with say 6 bytes per sample for timestamp and packet size. To identify bursts of 1000 packets of 1500 bytes each (1.5 MB), any algorithm would look for intervals containing 1 packet and scale up by the down sampling factor of 1000. The major problem is that this causes false positives. If the trace is well-behaved and has no bursts in any specified period (say 10 msec), the scaling scheme will still falsely identify 1 in 1000 packets as being part of bursts because of the large scaling factor needed for data reduction. Packet sampling, fundamentally, takes no account of the passage of time.

From an information-theoretic sense, packet traces, are inefficient representations for bandwidth queries. Viewing a trace as a time series of point masses (bytes in each packet), it is more memory-efficient to represent the trace as a series of *time intervals* with bytes sent per interval. But this introduces the new problem of choosing the intervals for representation so that bandwidth queries on any interval (chosen after the trace has been summarized) can be answered with minimal error.

Our first scheme builds on the simple idea that for any fixed sampling interval, say 100 microseconds, one can easily compute traffic statistics such as $\max$ or *Standard Deviation* by a few counters each. By exponentially increasing the sampling interval, we can span an aggregation period of length T, and still compute statistics at all time scales from microseconds to milliseconds, using only $O(\log T)$ counters. We call this approach Exponential Bucketing (`EXPB`). The challenge in `EXPB` is to avoid updating all $\log T$ counters on each packet arrival and to prove error bounds.

Our second idea, dubbed Dynamic Bucket Merge

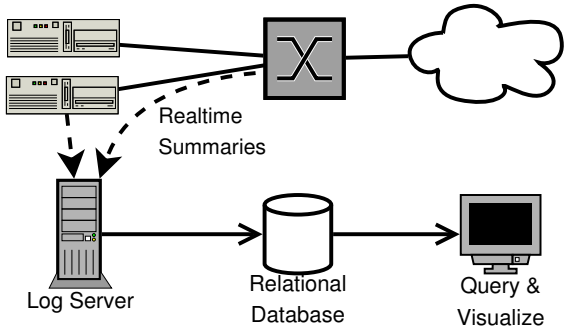


Figure 1: Example Deployment. End hosts and network devices implementing EXPB and DBM push output data over the network to a log server. Data at the server can be monitored and visualized by administrators then collapsed and archived to long-term, persistent storage.

(DBM), constructs an approximate streaming histogram of the traffic so that bursts stand out as peaks in this histogram. Specifically, we adaptively partition the traffic into k intervals/buckets, in such a way that the periods of heavy traffic map to more refined buckets than those of low traffic. The time-scales of these buckets provide a "visual history" of the burstiness of the traffic—the narrower the bucket in time, the burstier the traffic. In particular, DBM is well-suited for identifying not only whether a burst occurred, but *how many bursts*, and *when*.

System Deployment: Exponential Bucketing and Dynamic Bucket Merge have low computational and storage overheads, and can be implemented at multi-gigabit speeds in software or hardware. As shown in Figure 1, we envision a deployment scenario where both end hosts and network devices record fine-grain bandwidth summaries to a centralized log server. We argue that even archiving to a single commodity hard disk, administrators could pinpoint, to the second, the time at which correlated bursts occurred on given links, even up to a year after the fact.

This data can be indexed using a relational database, allowing administrators to query bandwidth statistics across links and time. For example, administrators could issue queries to "Find all bursts that occurred between 10 and 11 AM on all links in Set S". Set S could be the set of input links to a single switch (which can reveal Incast problems) or the path between two machines. Bandwidth for particular links can then be visualized to further delineate burst behavior. The foundation for answering such queries is the ability to efficiently and succinctly summarize the bandwidth usage of a trace in real-time, the topic of this paper.

We break down the remainder of our work as follows. We begin with a discussion of related algorithms and systems in Section 2. Section 3 illustrates the Dynamic Bucket Merge and Exponential Bucketing algorithms, both formally and with examples. We follow with our evaluations in Section 4, describe the implications for a system like Figure 1 in Section 5, and conclude in Section 6.

2 Related Work

Tcpdump [5] is a mature tool that captures a full log of packets at the endhost, which can be used for a wide variety of statistics, including bandwidth at any time scale. While flexible, tcpdump consumes too much memory for continuous monitoring at high speeds across every link and for periods of days. Netflow [2] can capture packet headers in routers but has the same issues. While sampled Netflow reduces storage, configurations with substantial memory savings cannot detect bursts without resulting in serious false positives. SNMP counters [1], on the other hand, provide packet and byte counts but can only return values at coarse and fixed time scales.

There are a wide variety of summarization data structures for traffic streams, many of which are surveyed in [15]. None of these can directly be adapted to solve the bandwidth problem at all time scales, though solutions to quantile detection do solve some aspects of the problem [15]. For example, classical heavy-hitters [13] measures the heaviest traffic flows during an interval. By contrast, we wish to measure "heavy-hitting subintervals across time", so to speak. However, heavy-hitter solutions are complementary in order to identify flows that cause the problem. The LDA data structure [12] is for a related problem – that of measuring average *latency*. LDA is useful for directly measuring latency violations. Our algorithms are complementary in that they help analyze the bandwidth patterns that *cause* latency violations.

DBM is inspired by the adaptive space partitioning scheme of [11], but is greatly simplified, and also considerably more efficient, due to the time-series nature of packet arrivals.

3 Algorithms

Suppose we wish to perform bandwidth measurements during a time window $[0, T]$, assuming, without loss of generality, that the window begins at time zero. We assume that during this period N packets are sent, with p_i being the byte size of the ith packet and t_i being the time at which this packet is logged by our monitoring system, for $i = 1, 2, \ldots, N$. These packets are received and processed by our system as a *stream*, meaning that the ith packet arrives before the jth packet, for any $i < j$.

The *bandwidth* is a rate, and so converting our observed sequence of N packets into a quantifiable bandwidth usage requires a *time scale*. Since we wish to measure bandwidth at different time scales, let us first make precise what we mean by this. Given a time

scale (or *granularity*) Δ, where $0 < \Delta < T$, we divide the measurement window $[0, T]$ into sub-intervals of length Δ, and aggregate all those packets that are sent within the same interval. In this way, we arrive at a sequence $S_\Delta = \langle s_1, s_2, \ldots, s_k \rangle$, where s_i is the sum of the bytes sent during the sub-interval $((i-1)\Delta, i\Delta]$, and $k = \lceil T/\Delta \rceil$ is the number of such intervals.[1]

Therefore, every choice of Δ leads to a corresponding sequence S_Δ, which we interpret as the bandwidth use at the temporal granularity Δ. All statistical measurements of bandwidth usage at time scale Δ correspond to statistics over this sequence S_Δ. For instance, we can quantify the statistical behavior of the bandwidth at time scale Δ by measuring the *mean, standard deviation, maximum, median, quantiles*, etc. of S_Δ.

In the following, we describe two schemes that can estimate these statistics for *every a posteriori* choice of the time scale Δ. That is, after the packet stream has been processed by our algorithms, the users can *query* for an arbitrary granularity Δ and receive provable quality approximations of the statistics for the sequence S_Δ.

Our first scheme, DBM, is time scale agnostic, and essentially maintains a streaming histogram of the values $s_1, s_2, \ldots, s_k$, by adaptively partitioning the period $[0, T]$. Our second scheme EXPB explicitly computes statistics for *a priori* settings of Δ, and then uses them to approximate the statistics for the queried value of Δ.

Since the two schemes are quite orthogonal to each other, it is also possible to use them both in conjunction. We give worst-case error guarantees for both of the schemes. Both schemes are able to compute the mean with perfect accuracy and estimate the other statistics, such as the maximum or standard deviation, with a bounded error. The approximation error for the DBM scheme is expressed as an additive error, while the EXPB scheme offers a multiplicative relative error. In particular, for the DBM scheme, the estimation of the maximum or standard deviation is bounded by an error term of the form $O(\varepsilon B)$, where $0 < \varepsilon < 1$ is a user-specified parameter dependent on the memory used by the data structure, and $B = \sum_{i=1}^{N} p_i$ is the total packet mass over the measurement window. In the following, we describe and analyze the DBM scheme, followed by a description and analysis of the EXPB scheme.

3.1 Dynamic Bucket Merge

DBM maintains a partition of the measurement window $[0, T]$ into what we call *buckets*. In particular, a m-bucket partition $\{b_1, b_2, \ldots, b_m\}$, is specified by a sequence of time instants $t(b_i)$, with $0 < t(b_i) \leq T$, with the interpretation that the bucket b_i spans the interval $(t(b_{i-1}),\ t(b_i)]$. That is, $t(b_i)$ marks the time when the ith bucket ends, with the convention that $t(b_0) = 0$, and $t(b_m) = T$. The number of buckets m is controlled by the memory available to the algorithm and, as we will show, the approximation quality of the algorithm improves linearly with m. In the following, our description and analysis of the scheme is expressed in terms of m. Each bucket maintains $O(1)$ information, typically the statistics we are interested in maintaining, such as the total number of bytes sent during the bucket. In particular, in the following, we use the notation $p(b)$ to denote the total number of data bytes sent during the interval spanned by a bucket b.

The algorithm processes the packet stream $p_1, p_2, \ldots, p_N$ in arrival time order, always maintaining a partition of $[0, T]$ into at most m buckets. (In fact, after the first m packets have been processed, the number of buckets will be exactly m, and the most recently processed packet lies in the last bucket, namely, b_m.) The basic algorithm is quite straightforward. When the next packet p_j is processed, we place it into a new bucket b_{m+1}, with time interval (t_{j-1}, T)—recall that t_{j-1} is the time stamp associated with the preceding packet p_{j-1}. We also note that the right boundary of the predecessor bucket b_m now becomes t_{j-1} due to the addition of the bucket b_{m+1}. Since we now have $m + 1$ buckets, we merge two *adjacent* buckets to reduce the bucket count down to m. Several different criteria can be used for deciding which buckets to merge, and we consider some alternatives later, but in our basic scheme we merge the buckets based on their *packet mass*. That is, we merge two adjacent buckets whose sum of the packet mass is the smallest over all such adjacent pairs. A pseudo-code description of DBM is presented in Algorithm 1.

Algorithm 1: DBM

```
1 foreach p_j ∈ S do
2     Allocate a new bucket b_i and set p(b_i) = p_j
3     if i == m + 1 then
5         Merge the two adjacent b_w, b_{w+1} for which
          p(b_w) + p(b_{w+1}) is minimum;
6     end
7 end
```

3.1.1 DBM Example

To clarify the operation of DBM we give the following example, illustrated in Figure 2.

Suppose that we run DBM with 4 buckets ($m = 4$), each of which stores a *count* of the number of buckets that have been merged into it, the *sum* of all bytes belonging to it, and the *max* number of bytes of any bucket merged into it. Now suppose that 4 packets have arrived

[1] To deal with the boundary problem properly, we assume that each sub-interval includes its right boundary, but not the left boundary. If we assume assume that no packet arrives at time 0, we can form a proper non-overlapping partition this way.

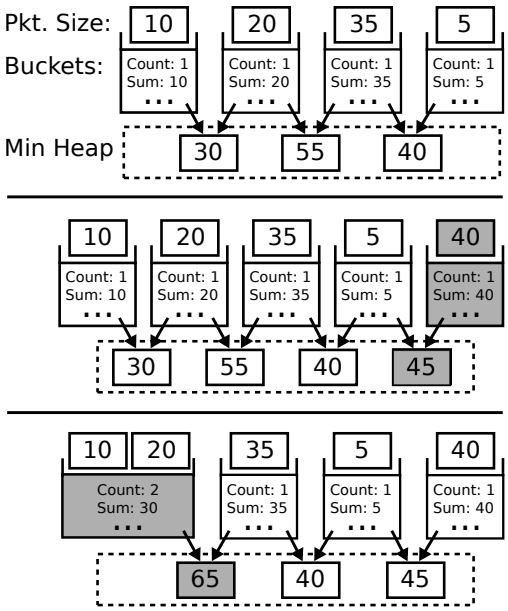


Figure 2: Dynamic Bucket Merge with 4 buckets. Initially each bucket contains a single packet and the min heap holds the sums of adjacent bucket pairs. When a new packet (value = 40) arrives, a 5th bucket is allocated and a new entry added to the heap. In the merge step, the smallest value (30) is popped from the heap and the two associated buckets are merged. Last, we update the heap values that depended on either of the merged buckets.

with masses 10, 20, 35, and 5, respectively. The state of DBM at this point is shown at the top of Figure 2. Note that Algorithm 1 required that we merge the buckets with the minimum combined sum. Hence, we maintain a min heap which stores the sums of adjacent buckets.

When a fifth packet with a mass of 40 arrives, DBM allocates a new bucket for it and updates the heap with the sum of the new bucket and its neighbor.

In the final step, the minimum sum is pulled from the heap and the buckets contributing to that sum are merged. In this example, the bucket containing mass 10 and 20 are merged into a single bucket with a new mass of 30 and a max bucket value of 20. Note that we also update the values in the heap which included the mass of either of the merge buckets.

3.1.2 DBM Analysis

The key property of DBM is that it can estimate the total number of bytes sent during *any* time interval. In particular, let $[t, t']$ be an arbitrary interval, where $0 \leq t, t' \leq T$, and let $p(t, t')$ be the total number of bytes sent during it, meaning $p(t, t') = \sum_{i=1}^{N} \{p_i \mid t \leq t_i \leq t'\}$. Then we have the following result.

Lemma 1. *The data structure* DBM *estimates $p(t, t')$ within an additive error $O(B/m)$, for any interval $[t, t']$, where m is the number of buckets used by* DBM *and $B = \sum_{i=1}^{N} p_i$ is the total packet mass over the measurement window $[0, T]$.*

Proof. We first note that in DBM each bucket's packet mass is at most $2B/(m-1)$, unless the bucket contains a single packet whose mass is strictly larger than $2B/(m-1)$. In particular, we argue that whenever two buckets need to be merged, there always exists an adjacent pair with total packet mass less than $2B/(m-1)$. Suppose not. Then, summing the sizes of all $(m-1)$ pairs of adjacent buckets must produce a total mass strictly larger than $2(m-1)B/(m-1) = 2B$, which is impossible since in this sum each bucket is counted at most twice, so the total mass must be less than $2B$.

With this fact established, the rest of the lemma follows easily. In order to estimate $p(t, t')$, we simply add up the buckets whose time spans intersect the interval $[t, t']$. Any bucket whose interval lies entirely inside $[t, t']$ is accurately counted, and so the only error of estimation comes from the two buckets whose intervals only partially intersect $[t, t']$—these are the buckets containing the endpoints t and t'. If these buckets have mass less than $2B/(m-1)$ each, then the total error in estimation is less than $4B/m$, which is $O(\frac{B}{m})$. If, on the other hand, either of the end buckets contains a single packet with large mass, then that packet is correctly included or excluded from the estimation, depending on its time stamp, and so there is no estimation error. This completes the proof. □

Theorem 1. *With* DBM *we can estimate the maximum or the standard deviation of S_Δ within an additive error εB, using memory $O(1/\varepsilon)$.*

Proof. The proof for the maximum follows easily from the preceding lemma. We simply query DBM for time windows of length Δ, namely, $(i\Delta, (i+1)\Delta]$, for $i = 0, 1, \ldots, \lceil T/\Delta \rceil$, and output the maximum packet mass estimated in any of those intervals. In order to achieve the target error bound, we use $m = \frac{4}{\varepsilon} + 1$ buckets.

We now analyze the approximation of the standard deviation. Recall that the sequence under consideration is $S_\Delta = \langle s_1, s_2, \ldots, s_k \rangle$, for some time scale Δ, where s_i is the sum of the bytes sent during the sub-interval $((i-1)\Delta, i\Delta]$, and $k = \lceil T/\Delta \rceil$ is the number of such intervals. Let $Var(S_\Delta)$, $E(S_\Delta)$, and $E(S_\Delta^2)$, respectively, denote the variance, mean, and mean of the squares for S_Δ. Then, by definition, we have

$$Var(S_\Delta) = E(S_\Delta^2) - E(S_\Delta)^2 = \frac{\sum_{i=1}^{k} {s_i}^2}{k} - E(S_\Delta)^2$$

Since DBM estimates each s_i within an additive error of εB, our estimated variance respects the following bound:

$$\leq \frac{\sum (s_i + \varepsilon B)^2}{k} - E(S_\Delta)^2$$

However, we can compute $E(S_\Delta)^2$ exactly, because it is just the square of the mean. In order to derive a bound on the error of the variance, we assume that $k > m$, that is, the size of the sequence S_Δ is at least as large as the number of buckets in DBM. (Naturally, statistical measurements are meaningless when the sample size becomes too small.) With this assumption, we have $2/k < 2/m$, and since $\varepsilon = 4/(m-1)$, we get that $2\frac{\sum s_i}{k} \leq \varepsilon B$, which, considering $k \geq 1$, yields the following upper bound for the estimated variance:

$$\leq \frac{\sum s_i^2}{k} - E(S_\Delta)^2 + \frac{k+1}{k}\varepsilon^2 B^2 \leq Var(S_\Delta) + 2\varepsilon^2 B^2$$

which implies the claim. □

Similarly, we can show the following result for approximating quantiles of the sequence S_Δ.

Theorem 2. *With* DBM *we can estimate any quantile of S_Δ within an additive error εB, using memory $O(1/\varepsilon)$.*

Proof. Let $s_1, s_2, \ldots, s_k$ be the sequence of data in the intervals $(i\Delta, (i+1)\Delta]$, for $i = 1, 2, \ldots, k = \lceil T/\Delta \rceil$, sorted in increasing order, and let $\hat{s}_1, \hat{s}_2, \ldots, \hat{s}_k$ be the sorted estimated sequence for the same intervals. We now compute the desired quantile, for instance the 95th percentile, in this sequence. Supposing the index of the quantile is q, we return $\hat{s}_q$. We argue that the error of this approximation is $O(\varepsilon B)$. We do this by estimating bounds on the s_i values that are erroneously (due to approximation) misclassified, meaning reported below or equal the quantile when they are actually larger or vice versa. If no s_i have been misclassified then $\hat{s}_q$ and s_q correspond to the same sample, and by Lemma 1 the estimated value $\hat{s}_q - s_q \leq \varepsilon B$, hence the claim follows. On the other hand, if a misclassification occurred, then the sample s_q is reported at an index different than q in the estimated sequence. Assume without loss of generality that the sample s_q has been reported as $\hat{s}_u$ where $u > q$. Then, by the pigeonhole principle, there is at least a sample s_h ($h > q$) that is reported as $\hat{s}_d$, $d \leq q$. By Lemma 1, $\hat{s}_d - s_h \leq \varepsilon B$. Since s_q and s_h switched ranks in the estimated sequence $\hat{s}$, by Lemma 1 it holds that $s_h - s_q \leq \varepsilon B$ and $\hat{s}_u - \hat{s}_d \leq \varepsilon B$. By assumption $u > q \geq d$, then it follows that $\hat{s}_u \geq \hat{s}_q \geq \hat{s}_d$ in the sorted sequence $\hat{s}$, which implies that $\hat{s}_q - \hat{s}_d \leq \varepsilon B$. The chain of inequalities implies that $\hat{s}_q - s_q \leq 3\varepsilon B$, which completes the proof. □

Algorithm 1 can be implemented at the worst-case cost of $O(\log m)$ per packet, with the heap operation being the dominant step. The memory usage of DBM is $\Theta(m)$ as each bucket maintains $O(1)$ information.

3.1.3 Extensions to DBM for better burst detection

Generic DBM is a useful oracle for estimating bandwidth in any interval (chosen after the fact) with bounded additive error. However, one can tune the merge rule of DBM if the goal is to pick out the bursts only. Intuitively, if we have an aggregation period with k bursts for small k (say 10) spread out in a large interval, then ideally we would like to compress the large trace to k high-density intervals. Of course, we would like to also represent the comparatively low traffic adjacent intervals as well, so an ideal algorithm would partition the trace into $2k+1$ intervals where the bursts and ideal periods are clearly and even visually identified. We refer to the generic scheme discussed earlier that uses *merge-by-mass* as DBM-mm, and describe two new variants as follows.

- merge-by-variance (DBM-mv): merges the two adjacent buckets that have the minimum aggregated packet mass variance
- merge-by-range (DBM-mr): merges the two adjacent buckets that have the minimum aggregated packet mass range (defined as the difference between maximum and minimum packet masses within the bucket)

These merge variants can also be implemented in logarithmic time, and require storing $O(1)$ additional information for each bucket (in addition to $p(b_i)$).

One minor detail is that DBM-mv and DBM-mr are sensitive to null packet mass in an interval while DBM-mm is not. For these reasons, we make the DBM-mr and DBM-mv algorithms work on the sequence defined by S_Δ, where Δ is the minimum time scale at which bandwidth measurements can be queried. Then DBM-mr and DBM-mv represents S_Δ as a histogram on m buckets, where each bucket has a discrete value for the signal. The goal of a good approximation is to minimize its predicted value versus the true under some error metric. We consider both the L_2 norm and the L_∞ norm for the approximation error.

$$E_2 = (\sum_{i=1}^{n} |s_i - \hat{s}_i|^2)^{\frac{1}{2}} \quad (1)$$

where $\hat{s}_i$ is the approximation for value s_i.

$$E_\infty = \max_{i=1}^{n} |s_i - \hat{s}_i| \quad (2)$$

We compare the performance of DBM-mr and DBM-mv algorithms with the optimal offline algorithms, that is, a bucketing scheme that would find the optimal partition of S_Δ to minimize the E_2 or the E_∞ metric. Then, the analysis of [7, 10] can be adapted to yield the following results that formally state our intuitive goal of picking out m bursts with $2m+1$ pieces of memory.

Theorem 3. *The L_∞ approximation error of the m-bucket* DBM-mr *is never worse than the corresponding error of an optimal $m/2$-bucket partition.*

Theorem 4. *The L_2 approximation error of the m-bucket* DBM-mv *is at most $\sqrt{2}$ times the corresponding error of an optimal $m/4$-bucket partition.*

3.2 Exponential Bucketing

Our second scheme, which we call Exponential Bucketing (EXPB), explicitly computes statistics for *a priori* settings of $\Delta_1, \ldots, \Delta_m$, and then uses them to approximate the statistics for the queried value for *any* Δ, for $\Delta_1 \leq \Delta \leq \Delta_m$. We assume that the time scales grow in powers of two, meaning that $\Delta_i = 2^{i-1}\Delta_1$. Therefore, we can assume that the scheme processes data at the time scale Δ_1, namely, the sequence $S_{\Delta_1} = (s_1, s_2, \ldots, s_k)$.

Conceptually, EXPB maintains bandwidth statistics for all m time scales $\Delta_1, \ldots, \Delta_m$. A naïve implementation would require updating $O(m)$ counters per (aggregated) packet s_i. However, by carefully orchestrating the accumulator update when a new s_i is available it is possible to avoid spending m updates per measurement as shown in Algorithm 2.

The intuition is as follows. Suppose one is maintaining statistics at 100 μs and 200 μs intervals. When a packet arrives, we update the 100 μs counter but not the 200 μs counter. Instead, the 200 μs counter is updated only when the 100 μs counter is zeroed. In other words, only the lowest granularity counter is updated on every packet, and coarser granularity counters are only updated when all the finer granularity counters are zeroed.

Algorithm 2: EXPB

```
1  sum=< 0, ..., 0 > (m times) ;
2  foreach s_i do
3      sum[0]=s_i;
4      j=0;
6      repeat
8          updatestat(j,sum[j]);
9          if j < m then
10             sum[j+1]+=sum[j];
11         end
12         sum[j]=0;
13         j++;
14     until i mod 2^j ≠ 0 or j ≥ m ;
15 end
```

3.2.1 EXPB Example

To better understand the EXPB algorithm we now present the example illustrated in Figure 3.

In this example, we maintain 3 buckets ($m = 3$) each of which stores statistics at time scales of 1, 2 and 4 time units. Each bucket stores the *count* of the intervals elapsed, the *sum* of the bytes seen in the current interval, and fields to compute max and standard deviation. We label the time units along the top and the number of bytes accumulated during each interval along the bottom.

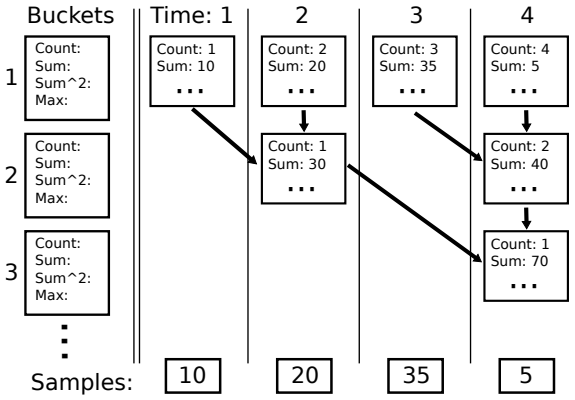


Figure 3: Exponential Bucketing Example. Each of the m buckets collects statistics at 2^{i-1} times the finest time scale. At the end of each time scale, Δ_i, buckets 1 to i must be updated. Before storing the new sum in a bucket j, we first add the old sum into bucket $j + 1$, if it exists.

In the first time interval 10 bytes are recorded in the first bucket and 10 is pushed to the sum of the second bucket. We repeat this operation when 20 is recorded in the second interval. Since 2 time units have elapsed, we also update the statistics for the Δ_2 time scale, and add bucket two's sum to bucket 3. In the third interval we update bucket 1 as before. Finally, at time 4 we update bucket 2 with the current sum from bucket 1, update bucket two's statistics, and push bucket two's sum to bucket 3. Finally, we update the statistics for Δ_3 with bucket three's sum.

3.2.2 EXPB Analysis

Algorithm 2 uses $O(m)$ memory and runs in $O(k)$ worst-case time, where $k = \lceil T/\Delta_1 \rceil$ is the number of intervals at the lowest time scale of the algorithm. The per-interval processing time is amortized constant, since the repeat loop starting at Line 6 simply counts the number of trailing zeros in the binary representation of i, for all $0 < i < k = T/\Delta$. The procedure $updatestat()$ called at Line 8 updates in constant time the $O(1)$ information necessary to maintain the statistics for each Δ_i, for $1 \leq i \leq m$.

We now describe and analyze the bandwidth estimation using EXPB. Given any query time scale Δ, we output the maximum of the bandwidth corresponding to the smallest index j for which $\Delta_j \geq \Delta$, and use the sum of squared packet masses stored for granularity Δ_j to compute the standard deviation. The following lemma bounds the error of such an approximation.

Lemma 2. *With* EXPB *we can return an estimation of the maximum or standard deviation of S_Δ that is between factor $1/2$ and 3 from the true value. The bound on the standard deviation holds in the limit when the ratio $E(S_\Delta^2)/E(S_\Delta)^2$ is large.*

Proof. We first prove the result for the statistic maximum, and then address the standard deviation. Let I be the interval $((i-1)\Delta, i\Delta]$ corresponding to the time scale Δ in which the maximum value is achieved, and let $p(I)$ be this value. Since $\Delta_j \geq \Delta$, there are at two most consecutive intervals I_i^j, I_{i+1}^j at time scale Δ_j that together cover I. By the pigeonhole principle, either I_i^j or I_{i+1}^j must contain at least half the mass of I, and therefore the maximum value at time scale Δ_j is at least 1/2 of the maximum value at Δ. This proves the lower bound side of the approximation. In order to obtain a corresponding upper bound, we simply observe that if I_i^j is the interval at time scale Δ_j with the maximum value, then I_i^j overlaps with at most 3 intervals of time scale Δ. Thus, the maximum value at time scale Δ_j cannot be more than 3 times the maximum at Δ proving an upper bound on the approximation.

The analysis for the standard deviation follows along the same lines, using the observation that $stddev_\Delta = \sqrt{E(S_\Delta^2) - E(S_\Delta)^2}$. An argument similar to the one used for the maximum value holds for the approximation of $E(S_\Delta^2)$. Then assuming the ratio $E(S_\Delta^2)/E(S_\Delta)^2$ to be a constant sufficiently greater than 1 implies the claim. We omit the simple algebra from this extended abstract. □

We note that there is a non-trivial extension of EXPB which allows it to work with a set of exponentially increasing time granularities whose common ratio can be any $\alpha > 1$. This can reduce average error. For a general $\alpha > 1$, Algorithm 2 cannot be easily adapted, so we need a generalization of it that uses an event queue while processing measurements to schedule when in the future a new measurement of length Δ_j must be sent to $updatestat()$. The details are omitted for lack of space.

3.3 Culprit Identification

As mentioned earlier, we do not want to simply identify bursts but also to *identify the flow* (e.g., TCP connection, or source IP address, protocol) that caused the burst so that the network manager can reschedule or move the offending station or application. The naive approach would be to add a heavy-hitters [13] data structure to each DBM bucket, which seems expensive in storage. Instead, we modify DBM to include two extra variables per bucket: a flowID and a flow count for the flowID.

The simple heuristic we suggest is as follows. Initially, each packet is placed in a bucket, and the bucket's flowID is set to the flowID of its packet. When merging two buckets, if the buckets have the same flowID, then that flowID becomes the flowID of the merged bucket and the flow counts are summed. If not, then one of the two flowIDs is picked with probability proportional to their flow counts. Intuitively, the higher count flows are more likely to be picked as the main contributor in each bucket as they are more likely to survive merges.

For EXPB, a simple idea is to use a standard heavy-hitters structure [13] corresponding to each of the logarithmic time scales. When each counter is reset, we update the flowID if the maximum value has changed and reinitialize the heavy-hitters structure for the next interval. This requires only a logarithmic number of heavy-hitters structures. Since there appears to be redundancy across the structures at each time scale, more compression appears feasible but we leave this for future work.

4 Evaluation

We now evaluate the performance and accuracy of DBM and EXPB to show that they fulfill our goal of a tool that efficiently utilizes memory and processing resources to faithfully capture and display key bandwidth measures. We will show that DBM and EXPB use significantly fewer resources than packet tracing and are suitable for network-wide measurement and visualization.

4.1 Measurement Accuracy

We implemented EXPB and the three variants of DBM as user-space programs and evaluated them with real traffic traces. Our traces consisted of a packets captured from the 1 Gigabit switch that connects several infrastructure servers used by the Systems and Networking group at U.C. San Diego, and socket-level send data produced by the record-breaking TritonSort sorting cluster [17].

Our "rsync" trace captured individual packets from an 11-hour period during which our NFS server ran its monthly backup to a remote machine using rsync. This trace recorded the transfer of 76.2 GB of data in 60.6 million packets, of which 66.6 GB was due to the backup operation. The average throughput was 15.4 Mbps with a maximum of 782 Mbps for a single second.

The "tritonsort" trace contains time-stamped byte counts from successful send system calls on a single host during the sorting of 500 GB of data using 23 nodes connected by a 10 Gbps network. This trace contains an average of 92,488 send events per second, with a peak of 123,322 events recorded in a single 1-second interval. In total, 20.8 GB were transferred over 34.24 seconds for an average throughput of 4.9 Gbps.

Ideally, our evaluation would include traffic from a mix of production applications running over a 10 Gbps network. While we do not have access to such a deployment, our traces provide insight into how DBM and EXPB might perform given the high bandwidth and network utilization of the "tritonsort" trace and the large variance in

bandwidth from second to second in the "rsync" trace.

For our accuracy evaluation, we used an aggregation period of 2 seconds. To avoid problems with incomplete sampling periods in EXPB, we must choose our time scales such that they all evenly divide our aggregation period. Since the prime factors of 2 seconds in nsec are 2^{11} and 5^{10} nsec, EXPB can use up to 11 buckets. Thus for EXPB, we choose the finest time scale to be $\Delta = 78.125$ μs (5^7 nsec) and the coarsest to be $\Delta =$ 80 msec ($2^{11}5^7$ nsec), which is consistent with the time scales for interesting bursts in data centers. For consistency, we also configure DBM to use a base sampling interval of 78.125 μs, but note that it can answer queries up to $\Delta = 2$ seconds.

To provide a baseline measurement, we computed bandwidth statistics for all of our traces at various time scales where $\Delta \geq 78.125$ μs. To ensure that all measurements in S_Δ are equal, we only evaluated time scales that evenly divided 2 seconds. In total, this provided us with ground-truth statistics at 52 different time scales ranging from 78.125 μs to 2 seconds. In the following sections we report accuracy in terms of error relative to these ground-truth measurements. While any number of values could be used for Δ and T in practice, we used these values across our experiments for the sake of a consistent and representative evaluation between algorithms.

4.1.1 Accuracy vs. Memory

We begin by investigating the tradeoff between memory and accuracy. At one extreme, SNMP can calculate average bandwidth using only a single counter. In contrast, packet tracing with tcpdump can calculate a wide range of statistics with perfect accuracy, but with storage cost scaling linearly with the number of packets. Both DBM and EXPB provide a tradeoff between these two extremes by supporting complex queries with bounded error, but with orders of magnitude less memory.

For comparison, consider the simplest event trace which captures a 64-bit timestamp and a 16-bit byte length for each packet sent or received. Using this data, one could calculate bandwidth statistics for the trace with perfect accuracy at a memory cost of 6 bytes *per event*. In contrast, DBM and EXPB require 8 and 16 bytes of storage per bucket used, respectively, along with a few bytes of meta data for each aggregation period.

To quantify these differences, we queried our traces for max, standard deviation, and 95th percentile (DBM only). For each statistic, we compute the average relative error of the measurements at each of our reference time scales and report the worst-case. To avoid spurious errors due to low sample counts, we omit accuracy data for standard deviations with fewer than 10 samples per aggregation period and 95th percentiles with fewer than 20 samples per aggregation period. We show the tradeoff between storage and accuracy in Table 1.

		Max of Avg. Rel. Error		
	Output	Max	S.Dev.	95th
trace (avg)	9.2 KBps			
(peak)	396 KBps	0%	0%	0%
DBM-mr	4 KBps	7.6%	14.7%	14.9%
EXPB	96 Bps	14.2%	5.9%	N/A

Table 2: We repeated our evaluation with the "rsync" trace and report accuracy results for our two best performing algorithms — DBM-mr and EXPB. We calculated the average relative error for each of our reference time scale and show the worst case.

While the simple packet trace gives perfectly accurate statistics, both DBM and EXPB consume memory at a fixed rate which can be configured by specifying the number of buckets and the aggregation period. In the presented configuration, both DBM and EXPB generate 4 KBps and 96 Bps, respectively — orders of magnitude less memory than the simple trace.

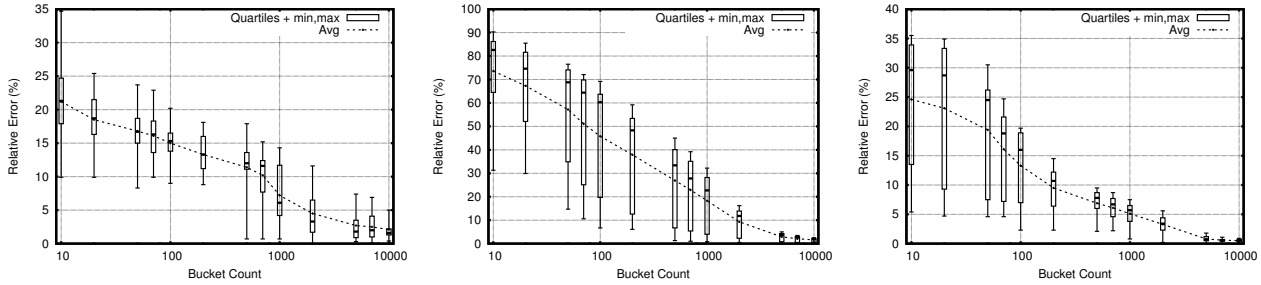
The cost of reduced storage overhead in DBM and EXPB is the error introduced in our measurements. However, we see that the range of average relative error rates is reasonable for max, standard deviation, and 95th percentile measurements. Further, of the DBM algorithms, DBM-mr gives the lowest errors throughout. While not shown, DBM's errors are largely due to under-estimation, but its accuracy improves as the query interval grows. EXPB gives consistent estimation errors for max across all of our reference points, but gradually degrades for standard deviation estimates as query intervals increase. Thus, for this trace, EXPB achieves the lowest error for query intervals less than 2msec. We have divided Table 1 to show the worst-case errors in these regions.

In Table 2 , we show the accuracy of DBM-mr and EXPB when run on the "rsync" trace with the same parameters as before. We note that again DBM-mr gives the most accurate results for larger query intervals, but now out-performs EXPB for query intervals greater than 160μs for max and 1msec for standard deviation.

To see the effect of scaling the number of buckets, we picked a representative query interval of 400 μs and investigated the accuracy of DBM-mr as the number of buckets were varied. The results of measuring the max, standard deviation and 95th percentile on the "tritonsort" trace are shown in Figure 4. We see that the relative error for all measurements decreases as the number of buckets is increased. However, at 4,000 buckets the curves flatten significantly and additional buckets beyond this do not produce any significant improvement in accuracy. While one might expect the error to drop to zero when the number of buckets is equal to the number of samples at S_Δ (5000 samples for 400μs), we do not see this since the trace is sampled at a finer granularity (78.125 μs) and

		Max of Avg. Relative Error					
		$\leq$ 2 msec			$>$ 2msec		
	Output Rate	Max	S.Dev.	95th	Max	S.Dev.	95th
packet trace (avg)	555 KBps						
(peak)	740 KBps	0%	0%	0%	0%	0%	0%
DBM−mm, 1000 buckets	4 KBps	25.9%	43.3%	18.3%	2.2%	5.7%	1.1%
DBM−mv, 1000 buckets	4 KBps	16.7%	58.9%	26.7%	7.2%	39.0%	10.4%
DBM−mr, 1000 buckets	4 KBps	14.0%	35.0%	16.1%	2.0%	4.1%	0.9%
EXPB, 11 buckets	96 Bps	2.7%	2.5%	N/A	2.8%	8.1%	N/A

Table 1: Memory vs. Accuracy. We evaluate the "tritonsort" trace with a base time scale of Δ =78.125 μs and a 2 second aggregation period. Data output rate is reported for a simple packet trace compared with the DBM and EXPB algorithms. For each statistic, we compute the max of the average relative error of measurements for each of our reference time scales.



(a) Max measurements (b) Std. Dev. measurements (c) 95th Percentile measurements

Figure 4: Relative error for DBM−mr algorithm shown for the 400 μs time scale with a varying number of buckets. The box plots show the range of relative errors from the 25th to 75th percentiles, with the median indicated in between. The box whiskers indicate the min and max errors.

the buckets are merged online. There is no guarantee that DBM will merge the buckets such that each spans exactly 400μs of the trace.

With approximations of the max and standard deviation with this degree of accuracy, we see both DBM and EXPB as an excellent, low-overhead alternative to packet tracing.

4.1.2 DBM Visualization

One unique property of the DBM algorithms is that they can be visualized to show users the shape of the bandwidth curves. Note that we proved earlier that DBM−mr is optimal in some sense in picking out bursts. We now investigate experimentally how all DBM variants do in burst detection.

In Figures 5 we show the output for a single, 2 second aggregation period from the "rsync" trace using DBM−mr. For visual clarity, we configured DBM−mr to aggregate measurements at a 4 msec base time scale (250 data points) using 9 buckets. Figure 5 shows the raw data points (bandwidth use in each 4 msec interval of the 2 second trace) with the DBM−mr output superimposed. Notice that DBM−mr picks out four bursts (the vertical lines). The fourth burst looks smaller than the 3.1 Mbps burst observable in the raw trace. This is because there were two adjacent measurement intervals in the raw trace with bandwidths of 3.1 and 2.2 Mbps, respectively. DBM−mr merged these measurements into a single bucket of with an average bandwidth of 2.65 Mbps for 8 msec.

We show the output for all DBM algorithms in a more clean visual form in Figures 6a, 6b and 6c. We have normalized the width of the buckets and list their start and end times on the x-axis. Additionally, we label each bucket with its mass (byte count). This representation compresses periods of low traffic and highlights short-lived, high-bandwidth events. From the visualization of DBM−mr in Figure 6c, we can quickly see that there were four periods of time, each lasting between 4 and 8 msec where the bandwidth exceeded 2.3 Mbps. Note that in Figure 6a, DBM−mm picks out only two bursts. The remaining bursts have been merged into the three buckets spanning the period from 1440 to 1636 msec, thereby reducing the bandwidth (the y-axis) because the total time of the combined bucket increases.

In practice, a network administrator might want to

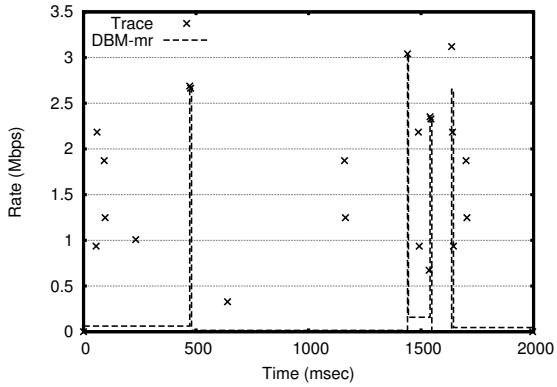


Figure 5: Visualization of events from a 2 second aggregation period overlaid with the output of DBM−mr using 9 buckets and a 4 msec measurement time scale.

quickly scan such a visualization and look for microburst events. To simulate such a scenario, we randomly inserted three bursts, each lasting 4 msec and transmitting between 4.0 and 4.4 MB of data. We show the DBM visualization for this augmented trace in bottom of Figure 6. DBM−mr and DBM−mm both allocate their memory resources to capture all three of these important events, even though they only represent 12 msec of a 2 second aggregation period. Again, DBM−mr cleanly picks out the three bursts.

4.1.3 Accuracy at High Load

As mentioned in Lemma 1, the error associated with the DBM algorithms increases with the ratio of total packet mass (total bytes) to number of buckets within an aggregation period. We now investigate to what extent increasing the mass within an aggregation period affects the measurement accuracy of DBM. To evaluate this, we first configured DBM to use a base time scale of $\Delta = 78.125$ μs and 1000 buckets, as before, but vary the mass stored in DBM by changing the aggregation period. Figures 7a & 7b show the change in average relative error for both max and standard deviation statistics in our high-bandwidth "tritonsort" trace at a representative query time scale (400 μs) as the aggregation period is varied between 1 and 16 seconds.

For DBM−mm and DBM−mv with 1000 buckets the relative error diverges significantly as the aggregation period is increased. In contrast, DBM−mr shows only a subtle degradation for max from 5.9% to 12.3%. For standard deviation, DBM−mv show consistently poor performance with average relative errors increasing from 32% to 64%, while both DBM−mm and DBM−mr trend together with DBM−mr's errors ranging from 9.8 to 31.7%.

We contrast DBM−mr's performance for these experiments with that of EXPB. We see that EXPB's average relative error in the max measurement gradually falls from 2.8% to 1.9% as the aggregation period increases. Further, the error in standard deviation falls from 1.4% at a 1 second aggregation period to 0.5% at 16 seconds.

These results indicate that degradation in accuracy does occur as the ratio of the total packet mass to bucket count increases, as predicted by Lemma 1. While DBM must be configured correctly to bound the ratio of packet mass to bucket count, EXPB's accuracy is largely unaffected by the packet mass or aggregation period.

4.2 Performance Overhead

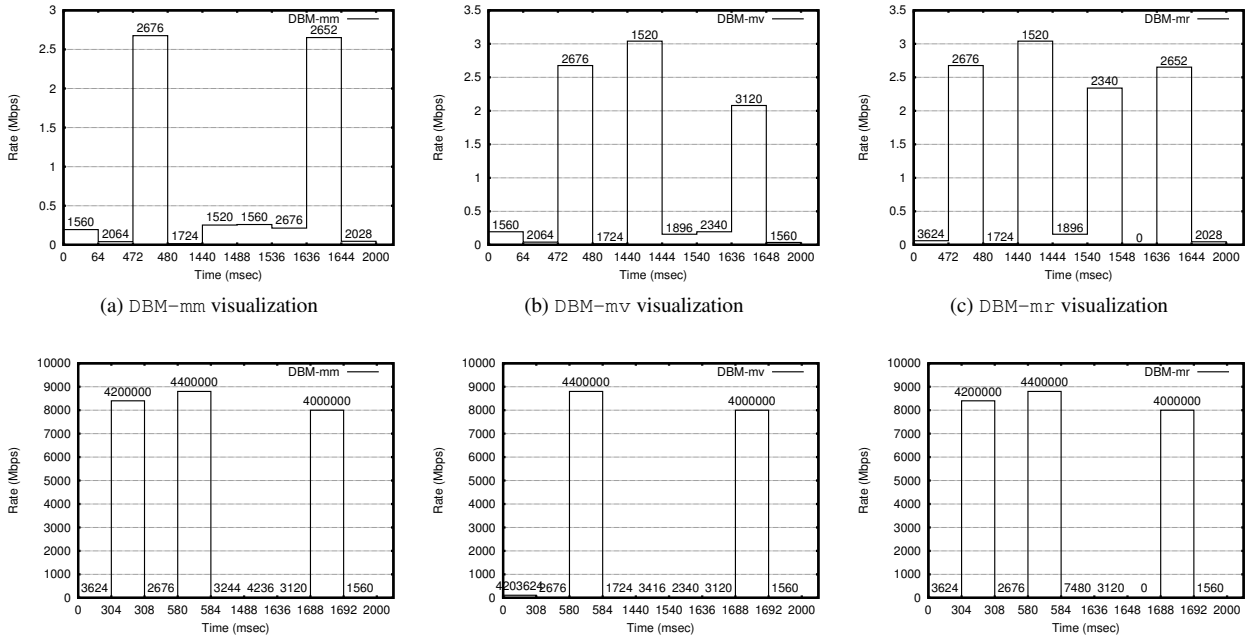
As previously stated, we seek to provide an efficient alternative to packet capture tools. Hence we compare the performance overhead of DBM and EXPB to that of an unmodified vanilla kernel, and to the well-established tcpdump[5].

We implemented our algorithms in the Linux 2.6.34 kernel along with a userspace program to read the captured statistics and write them to disk. To provide greater computational efficiency we constrained the base time scale and the aggregation period to be powers of 2. The following experiments were run on 2.27 GHz, quad-core Intel Xeon servers with 24 GB of memory. Each server is connected to a top-of-rack switch via 10 Gbps ethernet and has a round trip latency of approximately 100 μs.

To quantify the impact of our monitoring on performance, we first ran iperf [3] to send TCP traffic between two machines on our 10 Gbps network for 10 seconds. In addition, we instrumented our code to report the time spent in our routines during the test. We first ran the vanilla kernel source, then added different versions of our monitoring to aggregate 64 μs intervals over 1 second periods. We report both the bandwidth achieved by iperf and the average latency added to each packet at the sending server in Table 3. For comparison, we also report performance numbers for tcpdump when run with the default settings and writing the TCP and IP headers (52 bytes) of each packet directly to local disk. As DBM−mr is nearly identical to DBM−mm with respect to implementation, we omit DBM−mr's results.

As discussed in section 3.1, we see that the latency overhead per packet increases roughly as the log of the number of buckets. However, iperf's maximum throughput is not degraded by the latency added to each packet. Since the added latency per packet is several orders of magnitude less than the RTT, the overhead of DBM should not affect TCP's ability to quickly grow its congestion window. In contrast to DBM, tcpdump achieves 3.5% less throughput.

To observe the overhead of our monitoring on an application, we transferred a 1GB file using scp. We measured the wall-clock time necessary to complete the transfer by running scp within the Linux's time utility. To quantify the affects of our measurement on the total completion time, we measured the total overhead im-



(a) DBM−mm visualization (b) DBM−mv visualization (c) DBM−mr visualization

(d) DBM−mm visualization of bursty traffic (e) DBM−mv visualization of bursty traffic (f) DBM−mr visualization of bursty traffic

Figure 6: Visualization of DBM with 9 buckets over a single 2 second aggregation period. The start and end times for each bucket are shown on the x-axis, and each bucket is labeled with its mass (byte count). The top figures show the various DBM approximations of a single aggregation period, while the lower graphs show the same period with three short-lived, high bandwidth bursts randomly inserted.

posed on packets as they moved up and down the network stack. We report this overhead as a percentage of each experiment's average completion time (monitoring time divided by scp completion time). Each experiment was replicated 60 times and results are reported in Table 4. We see that although the cumulative overhead added by DBM grows logarithmically with the number of buckets, the time for scp to complete increases by at most 4.5%.

We see that our implementations of DBM and EXPB have a negligible impact on application performance, even while monitoring traffic at 10 Gbps.

4.3 Evaluation Summary

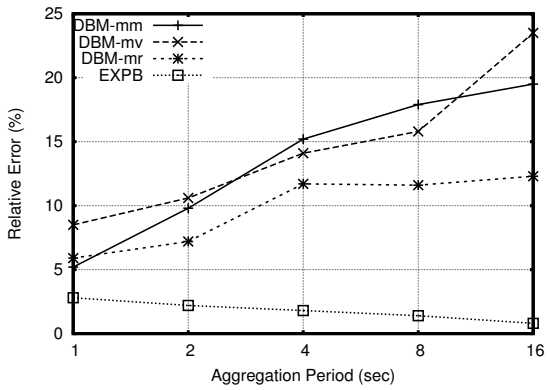
Our experiments indicate DBM−mr consistently provides better burst detection and has reasonable average case and worst case error for various statistics. When measuring at arbitrary time scales, EXPB has comparable or better average and worst-case error than DBM while using less memory. In addition, EXPB is unaffected by high mass in a given aggregation period. On other hand, DBM can approximate time series, which is useful for seeing how burst are distributed in time and for calculating more advanced statistics (i.e. percentiles). We recommend a parallel implementation where EXPB is used for Max and Standard Deviation and DBM−mr is used for all other queries.

5 System Implications

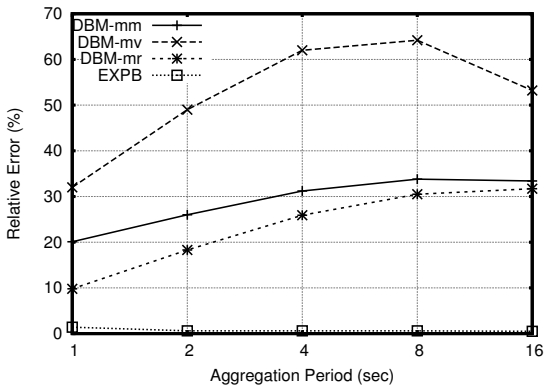
So far, we have described DBM and EXPB as part of an end host monitoring tool that can aggregate and visualize bandwidth data with good accuracy. However, we see these algorithms a part of a larger infrastructure monitoring system.

Long-term Archival and Database Support It is useful for administrators to retrospectively troubleshoot problems that are reported by customers days after the fact. At slightly more than 4 KBps, the data produced by both DBM and EXPB for a week (2.4 GB per link) could easily be stored to a commodity disk. With this data, an administrator can pinpoint traffic abnormalities at microsecond timescales and look for patterns across links. The data can be compacted for larger time scales by reducing granularity for older data. For example, one hour of EXPB data could be collapsed into one set of buckets containing max and standard deviation information at the original resolutions but aggregated across the hour.

With such techniques, fine-grain network statistics for hundreds of links over an entire year could be stored to a single server. The data could be keyed by link and time and stored in a relational database to allow queries across time (is the traffic on a single link becoming more bursty with time?) or across links (did a number of bursts cor-



(a) Max measurements with 1000 buckets



(b) Std. Dev. measurements with 1000 buckets

Figure 7: Average relative error for the DBM with 1000 buckets and EXPB with 11 buckets shown on the "tritonsort" trace for a 400 μs query interval and various aggregation periods.

Version	Buckets	Avg. BW	Overhead/Pkt
vanilla	N/A	9.053 Gbps	0.0 nsec
DBM−mm	10	9.057 Gbps	256.5 nsec
	100	9.010 Gbps	335.7 nsec
	1000	9.104 Gbps	237.5 nsec
	10000	8.970 Gbps	560.4 nsec
DBM−mv	10	9.043 Gbps	205.7 nsec
	100	8.986 Gbps	327.9 nsec
	1000	9.067 Gbps	432.2 nsec
	10000	9.067 Gbps	457.2 nsec
EXPB	14	9.109 Gbps	169.4 nsec
tcpdump	N/A	8.732 Gbps	N/A

Table 3: Average TCP bandwidth reported by iperf over 60 10-second runs. We also show the average time spent in the kernel-level monitoring functions for each packet sent. DBM and EXPB were run with a base time scale of $\Delta = 64$ μs and $T =$ 1 second aggregation period.

Version	Buckets	Time	Overhead
vanilla	N/A	14.133 sec	N/A
DBM−mm	10	14.334 sec	1.3%
	100	14.765 sec	1.9%
	1000	14.483 sec	2.7%
	10000	14.527 sec	2.5%
DBM−mv	10	14.320 sec	1.7%
	100	14.344 sec	2.3%
	1000	14.645 sec	2.9%
	10000	14.482 sec	3.1%
EXPB	14	14.230 sec	0.4%
tcpdump	N/A	15.253 sec	7.9%

Table 4: The time needed to transfer a 1GB file over scp. We measured the cumulative overhead incurred by our monitoring routines for all send and receive events. We report this overhead as a percentage of each experiment's total running time.

relate on multiple switch input ports?).

Hardware Implementation Both DBM and EXPB algorithms can be implemented in hardware for use in switches and routers. EXPB has an amortized cost of two bucket updates per measurement interval. Since bucket updates are only needed at the frequency of the measurement time scale, these operations could be put on a work queue and serviced asynchronously from the main packet pipeline. The key complication for implementing DBM in hardware is maintaining a binary heap. However, a 1000 bucket heap can be maintained in hardware using a 2-level radix-32 heap that uses 32-way comparators at 10 Gbps. Higher bucket sizes and speeds will require pipelining the heap. The extra hardware overhead for these algorithms in gates is minimal. Finally, the logging overhead is very small, especially when compared to NetFlow.

6 Conclusions

Picking out bursts in a large amount of resource usage data is a fundamental problem and applies to all resources, whether power, cooling, bandwidth, memory, CPU, or even financial markets. However, in the domain of data center networks, the increase of network speeds beyond 1 Gigabit per second and the decrease of in-network buffering has made the problem one of great interest.

Managers today have little information about how microbursts are caused. In some cases they have identified paradigms such as InCast, but managers need better visibility into bandwidth usage and the perpetrators of

microbursts. They would also like better understanding of the temporal dynamics of such bursts. For instance, do they happen occasionally or often? Do bursts linger below a tipping point for a long period or do they arise suddenly like tsunamis? Further, correlated bursts across links lead to packet drops. A database of bandwidth information from across an administrative domain would be valuable in identifying such patterns. Of course, this could be done by logging a record for every packet, but this is too expensive to contemplate today.

Our paper provides the first step to realizing such a vision for a cheap network-wide bandwidth usage database by showing efficient summarization techniques at links ($\sim$4 KB per second, for example, for running `DBM` and `EXPB` on 10 Gbps links) that can feed a database backend as shown in Figure 1. Ideally, this can be supplemented by algorithms that also identify the flows responsible for bursts and techniques to join information across multiple links to detect offending applications and their timing. Of the two algorithms we introduce, Exponential Bucketing offers accurate measurement of the average, max and standard deviation of bandwidths at arbitrary sampling resolutions with very low memory. In contrast, Dynamic Bucket Merge approximates a time-series of bandwidth measurements that can visualized or used to compute advanced statistics, such as quantiles.

While we have shown the application of `DBM` and `EXPB` to bandwidth measurements in endhosts, these algorithms could be easily ported to in-network monitoring devices or switches. Further, these algorithms can be generally applied to any time-series data, and will be particularly useful in environments where resource spikes must be detected at fine time scales but logging throughput and archival memory is constrained.

7 Acknowledgements

This research was supported by grants from the NSF (CNS-0964395) and Cisco. We would also like to thank our shepherd, Dave Maltz, and our anonymous reviewers for their insightful comments and feedback.

References

[1] A Simple Network Management Protocol (SNMP). http://www.ietf.org/rfc/rfc1157.txt.

[2] Cisco Netflow. www.cisco.com/web/go/netflow.

[3] iperf. http://http://iperf.sourceforge.net/.

[4] Performance Management for Latency-Intolerant Financial Trading Networks. *Financial Service Technology*, 9.

[5] tcpdump. http://www.tcpdump.org/.

[6] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). *SIGCOMM*, 2010.

[7] C. Buragohain, N. Shrivastava, and S. Suri. Space Efficient Streaming Algorithms for the Maximum Error Histogram. In *ICDE*, 2007.

[8] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph. Understanding TCP incast throughput collapse in datacenter networks. In *WREN 2009*.

[9] A. Erramilli, O. Narayan, and W. Willinger. Experimental queueing analysis with long-range dependent packet traffic. *IEEE/ACM Trans. Netw.*, 4(2):209–223, 1996.

[10] S. Gandhi, L. Foschini, and S. Suri. Space-efficient online approximation of time series data: Streams, amnesia, and out-of-order. In *ICDE*, 2010.

[11] J. Hershberger, N. Shrivastava, S. Suri, and C. Toth. Adaptive Spatial Partitioning for Multidimensional Data Streams. *Algorithmica*, 2006.

[12] R. R. Kompella, K. Levchenko, A. C. Snoeren, and G. Varghese. Every microsecond counts: tracking fine-grain latencies with a lossy difference aggregator. In *SIGCOMM 2009*.

[13] Y. Lu, M. Wang, B. Prabhakar, and F. Bonomi. ElephantTrap: A low cost device for identifying large flows. In *HOTI*, 2007.

[14] R. Martin. Wall Street's Quest To Process Data At The Speed Of Light. *Information Week*, April 23 2007.

[15] S. Muthukrishnan. *Data streams: Algorithms and applications*. now Publishers Inc., 2005.

[16] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan. Measurement and analysis of TCP throughput collapse in cluster-based storage systems. In *FAST 2008*.

[17] A. Rasmussen, G. Porter, M. Conley, H. V. Madhyastha, R. N. Mysore, A. Pucher, and A. Vahdat. TritonSort: A Balanced Large-Scale Sorting System. In *NSDI 2011*.

[18] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller. Safe and effective fine-grained TCP retransmissions for datacenter communication. *SIGCOMM 2009*.