Thread Cooperation in Multicore Architectures for Frequency Counting over Multiple Data Streams

Sudipto Das Shyam Antony Divyakant Agrawal Amr El Abbadi
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106-5110, USA
{sudipto, shyam, agrawal, amr}@cs.ucsb.edu

ABSTRACT

Many real-world data stream analysis applications such as *network monitoring*, *click stream analysis*, and others require combining multiple streams of data arriving from multiple sources. This is referred to as *multi-stream analysis*. To deal with high stream arrival rates, it is desirable that such systems be capable of supporting very high processing throughput. The advent of multicore processors and powerful servers driven by these processors calls for efficient parallel designs that can effectively utilize the parallelism of the multicores, since performance improvement is possible only through effective parallelism. In this paper, we address the problem of parallelizing *multi-stream analysis* in the context of multicore processors. Specifically, we concentrate on parallelizing frequent elements, top-k, and frequency counting over multiple streams. We discuss the challenges in designing an efficient parallel system for multi-stream processing. Our evaluation and analysis reveals that traditional "contention" based locking results in excessive overhead and wait, which in turn leads to severe performance degradation in modern multicore architectures. Based on our analysis, we propose a "cooperation" based locking paradigm for efficient parallelization of frequency counting. The proposed "cooperation" based paradigm removes waits associated with synchronization, and allows replacing locks by much cheaper atomic synchronization primitives. Our implementation of the proposed paradigm to parallelize a well known frequency counting algorithm shows the benefits of the proposed "cooperation" based locking paradigm when compared to the traditional "contention" based locking paradigm. In our experiments, the proposed "cooperation" based design outperforms the traditional "contention" based design by a factor of $2 - 5.5X$ for synthetic zipfian data sets.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Concurrency*; H.4.m [**Information Systems Applications**]: Miscellaneous

General Terms

Algorithms, Design, Experimentation, Performance.

Keywords

Multicore processors, parallel algorithms, data streams, concurrent structures, multi-streams, frequency counting, frequent elements, top-k.

1. INTRODUCTION

Data stream analysis forms an important class of applications where data is streaming in, and processing has to be done in real time. In traditional designs [25], data was considered to be arriving from a single source, and a plethora of sequential designs for the *single-stream* problem have been proposed [21, 22, 25]. But a number of common data stream applications require streams from multiple sources to be combined to answer queries, and we refer to this as *multi-stream* analysis. Multi stream analysis arises in many contexts, including large scale web advertising networks, the internet, sensor networks, and real time data analysis [8]. In large scale web-based advertising networks, clicks originating from different internet hosts result in multiple click streams that need to be merged at a single server for analysis and identification of suspicious publishers or IP addresses [23]. In the Internet, network monitors at core routers process streams of packets arriving from different origins for accounting purposes, and detection of malicious activities such as Distributed Denial of Service attacks [31]. In sensor networks, data streams from multiple sensors are typically combined at a single server for analysis, mining, and querying [2, 20]. Similarly, analysis of web server logs, monitoring calls in cellular networks, real time analysis of credit card and ATM transactions etc. require real time multi-stream analysis [4].

A number of high-throughput distributed stream processing systems have been proposed, such as Borealis [1] and Medusa [3]. These systems distribute the load on multiple distributed servers to ensure high throughput. But with the advent of multicore processors and extremely powerful servers driven by these processors, even centralized servers are now capable of delivering high throughput stream processing. For instance, the Sun SPARC Enterprise T5220 server powered by a UltraSPARC T2 processor [27] and supporting 1Gbps network connectivity has 64 hardware thread contexts which can be used for efficient and high throughput stream processing. Even standard workstations are now powered by processors with $4 - 8$ cores [18], and provide great potential for parallel processing. Efficient parallel designs are therefore needed to exploit such parallel hardware to support the processing of large numbers of streams with high and varying arrival rates. Note that stream processing systems have considerable data dependencies and shared data structures, and efficient parallel systems require thoughtful and efficient design.

Multi-Stream Analysis and Multicores: *Multi-stream analysis* has inherent parallel processing needs since stream elements are arriving from multiple sources at possibly variable rates, and the streams need to be combined for answering queries on the union. In the past when processors consisted of only one execution unit, the most efficient approach was to exploit some form of operating

system support such as `poll` or `select` to transform the problem into a *single stream* problem. It is widely acknowledged that these system calls introduce a significant overhead into the system. But in a sequential world, this was a preferred solution. But with the advent of multicore processors [18, 27], each processor now has multiple threads that can execute in parallel, and a system can improve its performance by exploiting the inherent parallelism to process each incoming stream using one or more threads. In this paper we investigate the challenges of such a parallel design.

Parallelizing Frequency Counting. Efficient parallel designs are the only means of improving performance in a multicore architecture. But, due to data dependencies arising from updates to a common structure, the problem is not "embarrassingly parallel". Thus, adding more threads will *not* lead to linear speedup. As it turns out, similar to many parallel algorithms and structures, parallelizing frequency counting is non-trivial, and actually it is very hard to eke out parallelism. Our experiments with a parallel frequency counting system designed using the traditional notion of "contention" based locking reveal the high overhead of a parallel system. Frequently, contending threads that share a resource need to *wait* for it if it is being exclusively held by some other thread. This results in wasted CPU cycles when the (hardware) threads do not perform useful work. To minimize the waits and the wasted CPU cycles, we propose a "cooperation" based locking paradigm where the threads "cooperate" (and not "contend") when sharing resources. Whenever a resource sought by a thread T_i is not available, then T_i's work is "delegated" to the thread T_j currently holding the resource, and T_i can move to its next job. T_i's request will eventually be processed by T_j. As a result, even though locks and shared resources are still present, *waits* associated with the locks have been eliminated. This allows efficient usage of the inherent parallelism, and results in higher processing throughput. Additionally, the proposed design paradigm allows replacing locks by much cheaper atomic synchronization primitives. In a previous work [10], we demonstrate the effectiveness of the "thread cooperation" paradigm for *intra-operator* parallelism of stream operations, and in this paper, we use this paradigm for *multi-stream analysis*. Concepts similar to "thread cooperation", such as the escrow transactional model [28], have earlier been used in the field of databases.

In this paper, we parallelize a standard frequency counting algorithm which forms the basis for frequent elements and top-k queries. Frequent elements [6, 21, 22] and top-k [9, 22, 26] queries are an important class of queries for stream analysis applications, and the research community has proposed several algorithms for answering such queries efficiently. A frequent elements query returns all the elements whose frequency of occurrence is above a certain threshold. For example, a query of the form "advertisements that are clicked more than 0.1% of the total clicks" is a frequent elements query. On the other hand, a top-k query returns the k elements with the highest frequency. Again, a query of the form "top-25 most clicked advertisements" is a top-k query.

Contributions:

- We propose and formalize the notion of "thread cooperation" for parallel analysis of multiple data streams. This concept of "thread cooperation" removes *waits* associated with locks, and will find use in various other "lock-based" designs.
- We propose a parallel design of the *Space Saving* algorithm [22], which Cormode et al. [6] have shown to have best performance amongst a number of *counter based* algorithms. Even though our design is based on *Space Saving*, the proposed paradigm can easily be augmented to accommodate other standard frequency counting algorithms such as *Lossy Counting* [21].
- We provide correctness proof sketches of the proposed design, and analyze the performance of the proposed "cooperation based" design. Our experiments show a factor of $2 - 5.5X$ improvement over traditional "contention" based locking paradigm.

Organization: Section 2 provides a survey of related work, and a brief description of the *Space Saving* [22] algorithm which we select to parallelize. Section 3 analyzes some common approaches towards designing a parallel system, and our analysis motivates the need for a new design paradigm. Section 4 formalizes the "thread cooperation" paradigm and describes the implementation details of the parallel system while providing arguments for correctness. Section 5 experimentally evaluates the proposed "cooperation" based design and compares it with traditional "contention" based design. Section 6 concludes the paper.

2. RELATED WORK

2.1 Data Stream Processing

The algorithms for answering frequent elements queries are broadly divided into two categories: *sketch based* and *counter based*. The *sketch based* techniques such as [5, 7] represent the entire stream's information as a "sketch" which is updated as the elements are processed. Since the "sketch" does not store per element information, the error bounds of these techniques are not very stringent, and have high processing costs. As pointed out in [6], these techniques are not very suitable for simple frequency counting problems.

On the other hand, the *counter based* techniques such as [22, 21, 11] monitor a subset of the stream elements and maintain an approximate frequency count of the elements. The goal is to guarantee high accuracy while having a small memory footprint. Different approaches use different heuristics to determine the set of elements to be monitored, and thus limit the space overhead. For example, in *Lossy Counting* [21], the stream is divided into rounds, and at the end of every round, potentially infrequent elements are deleted. This ϵ-approximate algorithm has a space bound of $O(\frac{1}{\epsilon}\log(\epsilon N))$, where N is the length of the stream and ϵ is the error bound. The *Space Saving* algorithm [22], on the other hand, uses a different heuristic to limit space, details of which are provided in Section 2.3.

Different solutions have also been suggested for answering top-k queries [9, 26]. Mouratidis et al. [26] suggest the use of geometrical properties to determine the k-skyband, and use this abstraction to answer top-k queries, whereas Das et al. [9] propose a technique which is capable of answering ad-hoc top-k queries, i.e., the algorithm does not need prior knowledge of the attribute on which the top-k queries have to be answered.

There has also been considerable research in distributed frequent elements [20], distributed top-k monitoring [2], and distributed stream processing systems [1, 3]. At first glance, it might seem that the multiple streams problem can be formulated as distributed streams, and the entire body of relevant literature can be used. But there is a subtle difference between distributed streams and the multiple streams problem we are considering. In distributed stream processing, since the streams are processed on distributed nodes, memory and processing is independent, and the onus is on reducing the communication overhead. On the other hand, in the multi-stream case, the streams are processed on the same machine sharing processing and memory, and the onus is on efficient processing

while effectively sharing the processor and memory among the different streams. Additionally, distributed stream processing systems have to deal with fault-tolerance, load balancing and many related issues of distributed systems [1], while these issues are not relevant in multi-stream analysis on a single powerful server.

The advent of modern multicore architectures [18, 27] have opened new frontiers, and their ubiquitous presence calls for algorithms that can efficiently exploit the parallelism inherent to these architectures. Although much research has been done in the stored database arena for exploiting parallelism [14, 15], very little research has focussed on stream operators. Gedik et al. [12] propose the use of Cell processors for parallel windowed stream joins. The proposed technique is targeted to the Cell architecture, and leverages specific features of Cell processors to improve performance. The growing demand for high throughput stream processing calls for designing other efficient parallel algorithms for stream processing, and this is the focus of this paper.

2.2 Query Model

In this section, we define the queries to be supported by the system. The queries can vary based on the type of answers sought (Queries 1, 2) or the frequency at which the queries need to be answered (Queries 3, 4).

QUERY 1. POINT QUERY*: This type of query is interested in a single element and is a boolean query of the form* IsElementFrequent(e) *or* IsElementInTopk(e).

QUERY 2. SET QUERY*: These queries report all the elements that are frequent, or all elements that are in the top-k. A frequent elements set query can be expressed formally in a language similar to SQL as:*

```
Select S.element
From Stream S
Where IsElementFrequent(S.element)
```

Even though a set query can be visualized as a combination of point queries for all the elements in the input alphabet, more efficient techniques can be employed provided the elements are sorted by their frequency.

QUERY 3. INTERVAL/DISCRETE QUERY*: These queries are posed as independent queries and consecutive queries are spaced out either with respect to time or the number of updates. A frequent elements interval set query can be expressed formally in a language similar to SQL as:*

```
Select S.element
From Stream S
Where IsElementFrequent(S.element)
Every 0.001s
```

QUERY 4. CONTINUOUS QUERY*: These queries are posed with "every update", i.e., as soon as a stream element is processed, the answer should be updated.*

When the stream elements are processed in parallel, the notion of "every update" is not as clear as in sequential processing of stream elements. If the sequential continuous query is mapped into the parallel processing scenario, there will be multiple concurrent queries at a particular instant, and the result from one query will be immediately updated by the next result. Most of the applications require the answer sets to be updated periodically and therefore, we only consider "Interval/Discrete" queries which can either be "point" or "set" queries.

Algorithm 2.1 *Space Saving* algorithm

1: *maxCounters* $\leftarrow 1/\epsilon$, *numMonitored* $\leftarrow 0$
2: **for** each element $\langle e \rangle$ in the stream **do**
3: /* Check if $\langle e \rangle$ is already being monitored */
4: **if** (LOOKUP($\langle e \rangle$)) **then**
5: *IncrementCounter*($\langle e \rangle$)
6: **else**
7: **if** (*numMonitored* $<$ *maxCounters*) **then**
8: *AddElement*(e, 1); *numMonitored*++
9: **else**
10: *Overwrite*(*Minimum frequency element*, e)

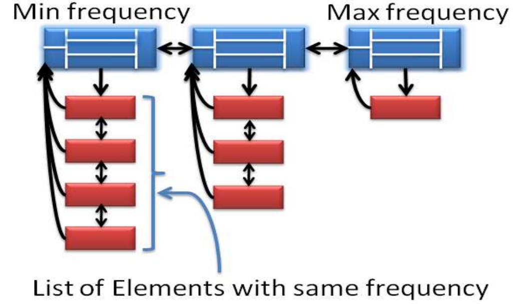
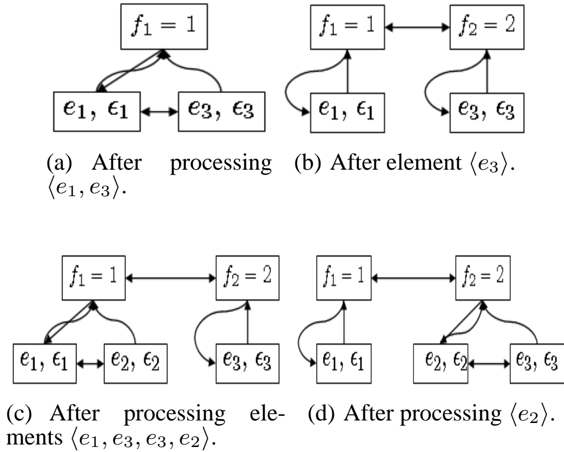


Figure 1: The *Stream Summary* structure.

2.3 Space Saving Algorithm

We select to parallelize the *Space Saving* algorithm [22] which has been demonstrated to have the best throughput amongst its class of frequency counting algorithms [6]. An interesting property of the algorithm is that it is deterministic and provides tight space bounds corresponding to the user specified error bound ϵ which determines the number of elements and counters that need to be monitored. *Space Saving* monitors only $O(\frac{1}{\epsilon})$ counters for providing ϵ-approximate answers. Algorithm 2.1 gives an overview of *Space Saving*. The main operations performed by the algorithm have been listed in Table 1. The algorithm monitors a subset of the stream elements which we refer to as the **Monitored Set**. If the element being processed is already being monitored, then its count is incremented (*IncrementCounter*). Otherwise, if the number of monitored elements is less than the maximum bound ($O(1/\epsilon)$), then the element is added to the monitored set (*AddElement*), else the current element *Overwrites* the element with minimum frequency and increments its count by one. Overwriting the minimum frequency element is a heuristic used by the algorithm to limit space. The intuition being that the minimum frequency element is least likely to be of interest to a frequent elements or top-k query. Thus, the space bound of the algorithm is $min(O(\frac{1}{\epsilon}, |A|))$, where $|A|$ is the size of the alphabet of the stream.

For *Overwriting*, this algorithm needs to track the minimum frequency element. The *Stream Summary* structure [11, 22] is used for this purpose. This structure consists of a *doubly-linked list* of frequency buckets which are sorted by frequency. Each bucket contains a list of elements which has the same frequency as that of the bucket. A nice property of this structure is that it maintains the elements sorted by frequency in $O(1)$ time per element. This allows answering both frequent elements and top-k queries using the same structure. For every element being processed, the algorithm looks up an element in the *Search Structure* (LOOKUP), and then updates the element in the *Stream Summary* structure. For lookup, the algorithm needs to have an efficient *search structure* that can be integrated with the *Stream Summary* structure, and a hash table is most suited for this purpose. The **Monitored Set** is thus repre-



(a) After processing $\langle e_1, e_3 \rangle$. (b) After element $\langle e_3 \rangle$.

(c) After processing elements $\langle e_1, e_3, e_3, e_2 \rangle$. (d) After processing $\langle e_2 \rangle$.

Figure 2: This figure illustrates the *Stream Summary* data structure for an example stream of elements $\langle\, e_1, e_3, e_3, e_2, e_2 \,\rangle$. The elements can be kept sorted in constant time per element.

Table 1: Main Operations in Space Saving

Operation	Description
LOOKUP(e)	Check whether element e is being monitored
IncrementCounter(e)	Increment the frequency of e
AddElement(e, *freq*)	Add element e with frequency $freq$
Overwrite(*min*, e)	Overwrite the minimum frequency element with e

sented by a combination of *Search Structure* and *Stream Summary*. Therefore, LOOKUP has to be supported by the hash table and the rest of the operation in Table 1 must be supported by *Stream Summary*.

3. INTUITIVE PARALLEL DESIGNS

In this section, we start by examining several possible intuitive designs for parallelizing frequency counting for *multi-stream analysis*. In these designs, each stream is assigned a thread which processes the elements. Recall that the *Space Saving* algorithm updates a structure (**Monitored Set**) while processing the elements. So the manner in which the threads share the structure determines the design of the system. We explore two straightforward designs, provide a brief experimental evaluation of these techniques, and explore some extensions to the two basic techniques. Our experimental setup uses an Intel quad core processor [18], and synthetically generated data following Zipfian distribution [32] where the zipfian factor α is varied from 1.5 to 3.0 in steps of 0.5. The parameter α controls the distribution, smaller α corresponds to a uniform distribution while higher α corresponds to a more skewed distribution. More details of the experimental setup appear in Section 5.

3.1 Independent Structures

This straightforward design corresponds to the *shared nothing* paradigm, where threads do not share any data or state information, and each thread has its own independent local structure. The idea is to simulate sequential execution, and run multiple copies of the same algorithm executing on different streams. Each stream has a local structure, and therefore there is no need for synchronization. If there are n different streams, then there are n different local

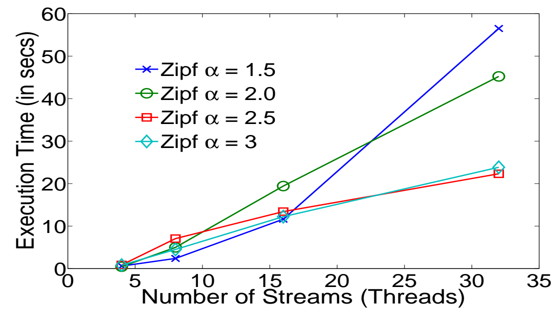


Figure 3: Execution time for a independent design using hierarchical merge. Each stream contains 1 million elements and is processed by a single thread. The query rate was set to 1 query every 50,000 elements.

structures, and these structures need to be merged periodically into a global structure so that queries can be answered from the global structure. Even though each thread processes only a fraction of the entire stream, since the minimum element in the local structure is not guaranteed to be the global minimum element, if the local structure is not large enough, it might lead to higher error. Lemma 3.1 provides a lower bound for the size of the local structures in order that the global error bound is preserved.

LEMMA 3.1. *For ϵ-approximate answers, the size of every local structure must be at least $O(1/\epsilon)$.*

PROOF. For a stream of N elements, by maintaining $O(1/\epsilon)$ counters, *Space Saving* ensures that every element with frequency greater than ϵN is reported [22]. Now in the multi-stream case, let N be the total length of the n different streams combined. The claim is that every thread must maintain at least $O(1/\epsilon)$ counters. Let us assume w.l.o.g that each stream is of length $\lceil N/n \rceil$ elements and let each thread maintain $1/\epsilon'$ counters where $\epsilon < \epsilon'$, while guaranteeing ϵ-approximate answers. Thus, an element e_i with frequency $f_i < \epsilon' \lceil N/n \rceil$ will not be monitored in the local structure. In an adversarial distribution, the element e_i appears in all the streams s_j such that $f_i^j < \epsilon' \lceil N/n \rceil$. Thus, this element will not be monitored in any of the local structures and hence not in the global structure as well. But since $\epsilon < \epsilon'$, it is possible that $\epsilon N < \sum_{\forall j \in \{1,\ldots,n\}} f_i^j < \epsilon' N$, in which case, *Space Saving* should have monitored e_i, thereby leading to a contradiction. □

COROLLARY 3.2. *Using independent structures for n streams, the total space complexity for ϵ-approximate answers is $O(n/\epsilon)$.*

Therefore, it follows from Corollary 3.2 that the independent design has high space overhead. In addition, the local structures need to be merged periodically to obtain the global structure from where the queries can be answered. Since the *Stream Summary* structure can be used to efficiently answer both frequent elements as well as top-k queries, in order for the merged global structure to retain those properties while remaining ϵ-approximate, all elements of every local structure need to be merged. Different heuristics can be used to reduce the merge overhead. For example, if we are looking up whether a specific element is frequent (*point query* or Query 1), then the query can be posed on each of the individual structures, and the results can be combined for the final query, but this only applies for a specific element and the process has to be repeated for any other frequent elements query or top-k query. On the other hand, if we are looking for whether an element is in the top-k elements, then some filtering of the results can be done to limit the

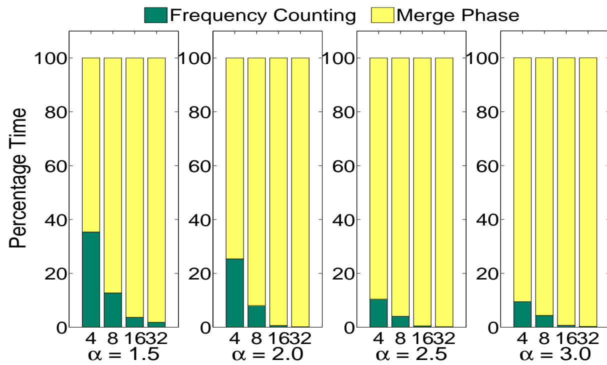


Figure 4: Profiling of the independent design using hierarchical merge. Along x-axis is the number of streams (and hence threads) and along y-axis is the time spent for an operation as percentage of total time. The query rate was set to 1 query every 50,000 elements.

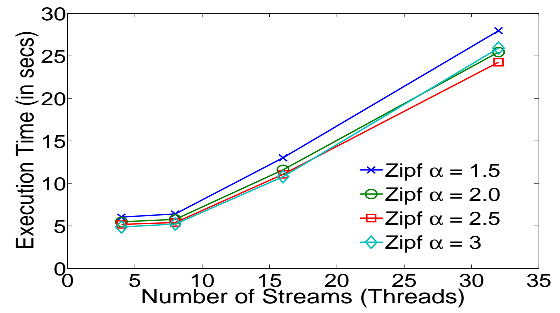


Figure 5: Execution time for a shared design using "contention" based locking. Each stream contains 1 million elements and is processed by a single thread.

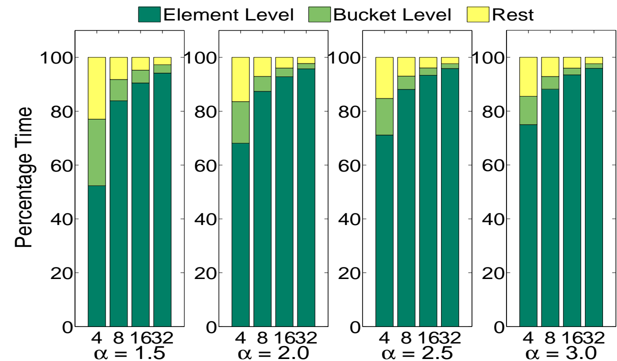


Figure 6: Profiling of the shared design using "contention" based locking. Along x-axis is the number of streams (and hence threads) and along y-axis is the time spent for an operation as percentage of total time.

number of entries that need to be merged (similar to techniques in [2]), but again, the merged results cannot be used for answering the frequent elements query. Another heuristic for answering the frequent elements query (*set query* or Query 2) is to pose the same query on the individual structures, and then merge the results from these individual queries. But since each structure only reflects a part of the entire stream, an element which is frequent globally might be infrequent locally, and hence the error is not bounded by ϵ. Therefore, we can see that different heuristics can lead to lowering of the merge overhead, but then either the error increases, or the properties of *Stream Summary* are not preserved. Compared to these techniques, if the complete merge of the local structures is done, then the same merged structure can be used to answer both *point queries* and *set queries* for frequent elements and top-k without any loss of accuracy of the results. Furthermore, the merge overhead is dependent on the query frequency – the greater the query frequency, the higher the number of merges and higher is the overhead. The merge overhead also increases with any increase in the number of parallel streams and threads, and for smaller values of the error bound ϵ. Thus, it is intuitive that even though the frequency counting part of the independent design would scale, the space overhead and the high merge overhead makes this design inefficient with increasing number of streams and query frequency, and our experimental evaluation confirms this.

Figure 3 provides experimental evaluation of the independent design. Along x-axis we plot the number of streams and along y-axis we plot the execution time in seconds, and different lines correspond to different values of zipfian α. Since the number of elements increases in proportion to the number of threads (each stream is assigned a processing thread), and ideal system with linear scaling would be a horizontal line in Figure 3. But since the problem is not "embarrassingly parallel", any practical system will have a positive slope, and the smaller the slope, the better is the performance and scalability of the parallel system. The high slope of the graphs as seen in Figure 3 demonstrate the overhead in the independent design. A profile of the time spent (Figure 4) reveals that with increase in the number of threads, the merge overhead increases as well, and in spite of the frequency counting part exhibiting good scalability, the merge overhead dominates the total cost. We compared two different types of merge algorithms: *serial merge*, where a single dedicated thread merges all the local structures into a global structure; and *hierarchical merge*, which uses a merge tree structure similar to the merge phase of the merge sort algorithm. In *hierarchical merge*, if there are n structures, then a thread merges two local structures, and in the first level, there are $n/2$ threads merging, in the second $n/4$ and so on. Therefore, there are $\log_2 n$ levels in the merge phase. In most runs, the *hierarchical merge* outperformed the serial, except for the case of small number of streams. But the times of *serial merge* is also comparable. In these experiments, we report the times using *hierarchical merge*. From Figure 3, it is evident that this approach of independent structures is efficient for smaller number of streams.

3.2 Shared Structures

The limitations of the independent design are high space overhead and merge overhead. To reduce the overhead, the threads can share the **Monitored Set**, and this forms the basis for the **Shared design** which we discuss in this subsection. Since multiple threads are accessing the same structure, the threads must synchronize, and this synchronization needs to be done at two levels:

- **Element Level Synchronization**: Multiple threads operating on the same element must be serialized so that there is only one thread operating on the element in the *Stream Summary*. This is achieved by acquiring a lock in the *search structure* on the element being processed. *Element level synchronization therefore serializes all threads processing the same element in the stream.*

- **Bucket Level Synchronization**: Since an increment or overwrite operation needs to move an element from one frequency bucket to another bucket in the *Stream Summary* structure, a

thread performing this operation needs to obtain locks on the source and the destination buckets. Since there can be several elements within a frequency bucket, a lock on a bucket prevents other threads from operating on any element belonging to that bucket. *Bucket-level synchronization therefore serializes accesses to a frequency bucket.*

The queries (which are only readers) also need to obtain locks so that the writers are blocked while a reader is processing a bucket. In this design, since a single global structure is used, the space complexity remains $O(\frac{1}{\epsilon})$ and there is no merge overhead. Thus, the *shared* design saves on space and merge cost. But since multiple locks need to be acquired for every element being processed, and due to multiple levels of synchronization, using the traditional "contention" based locking paradigm results in high contention overhead.

Figure 5 provides experimental evaluation of the shared design. Along x-axis we plot the number of streams and along y-axis we plot the execution time in seconds, and different lines correspond to different values of zipfian α. The high slope of the graphs in Figure 5 demonstrate the overhead in the "shared" design. Even though the number of threads in the system is increased, the processing time increases considerably as the number of threads increase. Figure 6 provides a profile of the time spent by the shared design. Along x-axis is the number of streams (and hence threads) and along y-axis is the time spent for an operation as percentage of total time. Figure 6 reveals that a majority of the time is spent contending either for synchronization at the *element-level* or at the *bucket-level*. Thus, even though the *shared* design has low space and merge overhead, the overhead due to contention and wait for shared resources makes this design inefficient. Multicore architectures have a large number of CPU cores, and threads waiting for shared resources waste cycles of idle CPU cores, which could have otherwise done useful work.

3.3 Discussion

Extensions of the straightforward techniques: There are several other simple schemes that extend the techniques proposed above. One possible approach can be to maintain a combination of local and global counters (i.e., a **Hybrid** Structure) to limit the contention (by maintaining local counters for frequent elements and a global structure for infrequent elements) as well as space overhead (no need to replicate relatively infrequent elements). Analysis reveals that this approach is also not very efficient. For a relatively uniform input, most of the items will hit the global structure (since most items are infrequent), thereby degenerating into the *shared design*. For a relatively skewed input, very few elements will hit the global structure (since most elements are infrequent and hence stored locally to reduce contention), thus reducing to the *independent design*. Another extension is a **hash based scheme** where elements are hashed to threads, i.e., same elements go to same threads. Even though this design will need less synchronization, load balancing among threads will be an issue when the input distribution is skewed, as some threads are overloaded, while others are lightly loaded.

Characteristics of a Good Design: Ideally, we would like a design with the following properties:
- *Small memory footprint and no merge overhead* similar to the shared design.
- *Low contention overhead* similar to the independent design.

In the next section, we show how "contention" in locking can be reduced by "thread cooperation" so that shared structures can also be used efficiently.

4. THREAD COOPERATION PARADIGM

In this section, we formalize and explain in detail the "cooperation" based threading paradigm to optimistically exploit the inherent parallelism in modern processor architectures, explain the parallel design of the *Space Saving* algorithm using the proposed design paradigm, propose parallel designs for the two different structures and argue their correctness.

4.1 Formalization

As observed in Section 3.3, instead of threads contending for shared resources, they can rather *cooperate*, thereby boosting each other and in turn improving the overall system performance. In this paradigm, even though locks are not eliminated completely, "waits" associated with locks have been eliminated. The delegated request is "queued", and the type of queueing depends on the semantics of the request being delegated.

Let us consider a system comprised of threads (hardware or software)[1] and the task to be completed is a set of *requests* that are *associative* and *commutative*. The system is said to *make progress* if the input set of *requests* is being consumed. The following principles formally express the *thread cooperation* model:

PRINCIPLE 4.1. ***Request Delegation**: If thread T_i is trying to acquire a shared resource $\mathbb{R}$ and it succeeds in acquiring $\mathbb{R}$, it will complete its request using $\mathbb{R}$. If it fails to acquire $\mathbb{R}$, it will "delegate" its request to the thread T_j that currently has exclusive access to $\mathbb{R}$, and "proceeds" to its next request. All other threads trying to acquire $\mathbb{R}$ will "delegate" their requests to T_j. Once T_j finishes its own request, and before it relinquishes control over $\mathbb{R}$, it will check for any pending requests on $\mathbb{R}$ and will relinquish $\mathbb{R}$ only when all pending delegated requests have been processed.*

PRINCIPLE 4.2. ***Minimal Existence**: Once a thread has acquired a shared resource, it will abstain from acquiring other shared resources.*

Note that *requests* are *delegated* in two different scenarios. *First*, when a shared resource is not *available*, and *second*, when a thread has already acquired a shared resource, and needs another shared resource. In either case, the threads rely on *cooperation* to make *progress*. To guarantee correctness, an implementation must satisfy the following invariant:

INVARIANT 4.1. ***Fulfillment Guarantee**: Once a thread has* delegated *a request, the request is neither* lost *nor left* unfulfilled.

There are multiple benefits of the *cooperation* based approach:

- **Removal of Waits:** Whenever there is contention for a shared resource, the threads follow Principle 4.1 and do not *wait*. This prevents wastage of useful computational resources and allows threads to make progress.
- **Arbitration Overhead Removal:** Since the threads do not contend for shared resources, this paradigm reduces the overhead of arbitration of locks among the contending threads.

The "thread cooperation" model logically separates the requests (tasks to be performed by a thread) from the execution threads. This allows the system to be viewed as a set of threads processing a set of requests, and it does not matter which thread processes

[1] A *hardware thread* corresponds to an execution unit with its own set of registers and local resources and is different from a *software thread*. Multiple *software threads* can be mapped to a *hardware thread* to time-share the resources.

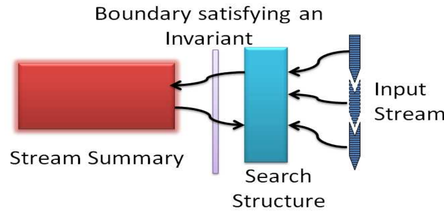


Figure 7: Overview of the parallel design of *Space Saving* using *thread cooperation*.

which request. This is analogous to how the operating system (OS) schedules jobs on the hardware execution threads. *Delegation* ensures that only the request is "blocked" and the associated thread can detach itself from the request and move forward to process the next request. This is analogous to the OS blocking I/O requests and not the actual CPU that was processing the job. Note that threads sharing resources do need synchronization, but the *waits* associated with synchronization primitives have been eliminated. In addition, as described later, cooperation based synchronization allows replacing locks with atomic operations which are much cheaper synchronization primitives.

4.2 Parallelizing Space Saving

Once we have formalized the *thread cooperation* paradigm, we now show how we can use this paradigm for parallelizing *Space Saving*. Figure 7 provides a high level overview of the system design for parallelizing *Space Saving* using the *thread cooperation* paradigm. The boundary conceptually separates the *Stream Summary* and the *Search Structure*, and these structures interact with each other through a well-defined interface, and need not be aware of the implementation details. The parallel stream processing system should guarantee that the following invariant holds:

INVARIANT 4.2. ***Boundary Invariant:*** *If Thread T_i processing element $\langle e \rangle$ has crossed the boundary into the* Stream Summary *structure, then it is the* only *thread active in the* Stream Summary *that is processing the element $\langle e \rangle$.*

The *Search Structure* should guarantee that Invariant 4.2 holds and this provides the **element level synchronization** as discussed in Section 3.2. As long as the desired properties and the invariants hold, this framework is independent of the choice of the different structures involved and the actual algorithm used to process the stream elements. The "cooperation" based model does not guarantee any ordering of the requests, but this does not impact the correctness of the *Space Saving* algorithm.

LEMMA 4.1. ***Order Invariance:*** *The final outcome of the* Space Saving *algorithm is invariant to the order in which the stream elements are processed.*

This follows directly from the definition of the frequency counting problem. Given a stream of N elements, the *Space Saving* algorithm reports all the elements whose frequency is above ϵN, where ϵ is the user specified error bound [22]. Therefore, as long as an element appears more than ϵN times in the input stream, *Space Saving* guarantees that the element will be reported, irrespective of where the element appeared in the stream.

COROLLARY 4.2. ***Correctness of the Cooperation based parallel design:*** *Reordering of requests due to* request delegation *does not affect the correctness of* Space Saving.

Table 2: Basic atomic primitives.

Operation	Description
CAS($addr$, $expected$, new)	Atomically compare the $expected$ value to the stored value at $addr$, and swap with new if successful
IAF($addr$)	Atomically increment the location $addr$ and read the incremented value
SWP($addr$, new)	Atomically swap the present value at $addr$ with the passed new value and return the old value

Corollary 4.2 follows directly from Lemma 4.1. In general, most stream processing algorithms are invariant to the order of processing, and the *thread cooperation* paradigm can be applied to them.

4.3 Implementation Details

As depicted in Figure 7, the system can be viewed as two different components which interact with each other through an interface that ensures that Invariant 4.2 holds. Recall that when processing the elements, a thread first accesses the *Search Structure* and this structure directs the thread to the appropriate location in the *Stream Summary*. In the implementation, an element in the hash table (*search structure*) points to the corresponding element in the *Stream Summary* structure, and this element in the *Stream Summary* in turn points to the bucket to which it belongs. Our implementation uses a number of atomic instructions. Refer to Table 2 for a brief description of the atomic operations used in the implementation. Only the CAS operation need to supported by the underlying hardware, and the rest of the operations can be implemented using the CAS primitive. Note that the atomic operations supported by the hardware are much cheaper than acquiring locks. We now discuss the two different components in Figure 7 in detail, and provide sketches of correctness proof of the methods. Our approach for proving correctness is as follows: *The design is* **correct** *if all the invariants are satisfied. Therefore, for each individual operations, if all the invariants are satisfied as a precondition for the operation, the operation is said to be correct if it ensures that all the invariants are satisfied as a postcondition.*

4.3.1 Search Structure or Hash table

The hash table lookup is equivalent to the LOOKUP operation in Table 1. Even though there have been proposals for different lock-free hash table implementations (such as Shalev et al. [30]), the hash table used by the frequency counting algorithms has certain special requirements and characteristics. We design a simple concurrent hash table where separate chaining is used to resolve hash collisions, and bucket-level locks are used to synchronize insertions to the same chain. We modify the conventional design of a chained hash table [19] to suit the requirements specific to the application.

Lookup and insertion: The hash table is designed with minimal locking: the "readers" in the hash table do not need locks. Locks are needed only to serialize insertions to the same hash bucket. In this design, threads are blocked only when they are trying to insert into the same hash bucket. If a moderately robust hash function (such as *multiplicative hashing* [19]) is used, then the likelihood of two "writers" mapping to the same hash bucket is very rare and hence the associated wait is also small. In order for the "readers" to traverse the list "lock-free", the following structural invariant must be satisfied at all times:

INVARIANT 4.3. *At all times during execution, a pointer should either be pointing to a valid node or be set to* null.

Deletion: Deletion of a hash table entry is done lazily – the entry

to be deleted is atomically marked as deleted, and this implies logical removal. Once an entry is removed logically, it can either be physically removed by some other thread or be reused. In this design, the location of the entry is reused for future insertions into the chain, and the following invariant guarantees correctness. In the implementation, deletions are marked by setting the $value$ field of the $\langle key, value\rangle$ pair in the hash table to null.

INVARIANT 4.4. *Once a location has been marked as deleted, the key in the location has been removed from the hash table and any subsequent lookup for that key should return failure.*

INVARIANT 4.5. ***Unique key:** At no point should two valid locations in the hash table contain the same key.*

The invariants listed above correspond to basic properties of the hash table and any concurrent design must satisfy them. In addition, there are certain application specific requirements. To guarantee **element level synchronization**, the most common approach would be to have the hash table ensure this level of synchronization, because every thread processing a stream element first accesses the hash table. Therefore, if thread T_i has *exclusive access* to element e, then any thread $T_j (j \neq i)$ processing e must wait. Since the elements (which form keys in the hash table) are a point of contention in the hash table, Principle 4.1 can be used to delegate the request and allow T_j to proceed with other requests while T_i guarantees to process the *delegated* requests as per Invariant 4.1. Since the requests in a frequency counting algorithm are increments to the count, "queueing" a delegation in the hash table is simple, and can be achieved through a field that stores a pending *count* corresponding to the element. If the pending *count* is 0, this implies that no thread owns the element and there are no pending requests. The first thread that increments the *count* acquires *exclusive access* on the element, and all other threads delegate the request by atomically incrementing the count. The following invariant guarantees **element level synchronization** and hence satisfies Invariant 4.2:

INVARIANT 4.6. *A thread obtains exclusive access to a key in the hash table if the* `IAF` *operation on count corresponding to the* key *returns* 1.

Relinquishing an element: Once a thread has processed a stream element, it has to relinquish *exclusive access* to the element. But before relinquishing access, it has to ensure that there are no delegated requests so that Invariant 4.1 is satisfied. To relinquish, the thread first performs a `CAS` of 1 (expected value) with 0 (new value) on the $count$ field corresponding to the entry in the hast table. If this succeeds, there are no delegated requests, and the element has been relinquished. Any other thread performing a subsequent `IAF` for acquiring the element will read *count* as 1, and hence will be able to acquire the element (Invariant 4.6). Failure implies pending delegated requests and the thread will do a `SWP` with 1 which will return the number of delegated requests, and set *count* to 1. This ensures that the owning thread still has exclusive access to the element and prevents other threads from crossing the boundary. The thread then subtracts 1 from this value (which corresponds to the request for this thread in the earlier round) and crosses the boundary with the subtracted value as increment to the frequency of the element. This ensures that both Invariants 4.1 and 4.2 are satisfied. Note that as a result of delegated requests accumulating inside the hash table, increments to counts of elements are processed in bulk, and we refer to this as **bulk increment**.

Based on the application requirements, the list of operations supported by the hash table is listed in Table 3. These operations are

Table 3: Application specific hash table operations.

Operation	Description
IncrementCount(e)	If e is not present in the hash table, insert e. "Log" request for e and acquire if e is available.
TryRelinquish(e)	Relinquish e if there are no pending requests on e.
TryRemove(e)	Remove e if it is available and there are no pending requests on e.

Algorithm 4.1 *IncrementCount*

```
1: Procedure IncrementCount(key, new, acquired, value)
2: index := hash(key), ptr := buckets[index].head
3: while (ptr ≠ null) do
4:    if (ptr.value ≠ null && ptr.key = key) then
5:       count := ⟨ IAF(ptr.value.count) ⟩
6:       (count = 1) ? acquired := true : acquired := false
7:       new := false
8:       return ptr.value
9:    ptr := ptr.next
10: buckets[index].lock()
11: ptr := buckets[index].head, emptyLoc := null
12: while (ptr ≠ null) do
13:    if (ptr.value ≠ null && ptr.key = key) then
14:       buckets[index].unlock()
15:       count := ⟨ IAF(ptr.value.count) ⟩
16:       (count = 1) ? acquired := true : acquired := false
17:       new := false
18:       return ptr.value
19:    else if (ptr.value = null) then
20:       emptyLoc := ptr
21:    prev := ptr, ptr := ptr.next
22: if (emptyLoc ≠ null) then
23:    emptyLoc.count := 1
24:    emptyLoc.key := key
25:    emptyLoc.value := value
26: else
27:    emptyLoc := getNewNode()
28:    emptyLoc.count := 1
29:    emptyLoc.key := key
30:    emptyLoc.next := null
31:    emptyLoc.value := value
32:    prev.next = emptyLoc
33: buckets[index].unlock(), acquired := true, new := true
34: return value
35: end Procedure IncrementCount
```

in addition to the operations of a traditional hash table. Some operations are prefixed with *Try* to signify that the success of these operations are contingent on certain conditions being met (analogous to a *trylock* operation on a mutex lock).

We now provide details of implementation of the important hash table operations, and provide sketches that prove the correctness of these operations.

Implementation of *IncrementCount* and its Correctness

The *IncrementCount* operation is critical as it combines the *Lookup* and *Insert* operations while ensuring that Principle 4.1 and Invariant 4.6 are satisfied, which in turn satisfied Invariant 4.2. Algorithm 4.1 provides an overview of this operation. This operation first performs the equivalent of the *Lookup* operation, and if the key is already present, its count is atomically incremented (line 5) which is equivalent to "logging" the request. The conditional in line 6 ensures that Invariant 4.6 is satisfied and only one thread has exclusive access. This is because, if the value of $count$ was initially 0, of all concurrent increments, only one thread will atomically read 1, and that thread will acquire the element, while all

Algorithm 4.2 *TryRelinquish*

```
1:  Procedure TryRelinquish(key, status, incr)
2:  index := hash(key), ptr := buckets[index].head
3:  while (ptr ≠ null) do
4:      if (ptr.value ≠ null && ptr.key = key) then
5:          if (⟨CAS(ptr.value.count, 1, 0)⟩) then
6:              status := success
7:              return ptr.value
8:          else
9:              status := pending_increment
10:             incr = ⟨SWP(value.count, 1)⟩ − 1
11:             return ptr.value
12:     ptr := ptr.next
13: status := failure
14: return null
15: end Procedure TryRelinquish
```

other threads "delegate" their request (Principle 4.1). Again, the same logic of mutual exclusion and delegation applies for the operations in lines 15 and 16. If the key is not present, the operation atomically inserts the $\langle key, value\rangle$ pair, and the inserting thread is given exclusive access to the newly inserted value (lines 23 and 28). The insert analogue here is equivalent to an *Insert* operation. In order to ensure that duplicates are not inserted, the operation first performs a *Lookup* (line 12). A failure implies the key is not present. Any other *Insert* operation on this hash bucket is serialized as the thread has already acquired a lock on the bucket (line 10) and the lock also prevents insertion of duplicates and guarantees satisfaction of Invariant 4.5. If any location has the $value$ field set to null then that location is available (Invariant 4.4) and can be reused, and the thread keeps track of available locations. The order of assignments in lines 24 and 25, and lines 29 and 31 are important to make sure that a parallel *Lookup* (which traverses "lock-free") sees the insertion only when the $value$ field is pointing to a valid location. Again, the assignments in lines 30 and 32 ensure that Invariant 4.3 is satisfied. It is thus evident that the *IncrementCount* operation does not violate any invariants, and hence is correct.

Implementation of *TryRelinquish* and its Correctness

The *TryRelinquish* operation is used to relinquish exclusive access on a key after the request on the key has been completed. Algorithm 4.2 provides an overview of the implementation. If there are no pending requests on the key, then the CAS in line 5 would succeed, and the *count* is set to 0 thereby reflecting that key is now *available*. If there is a concurrent *IncrementCount* request on the same key, if the increment succeeds before the CAS, then the CAS will fail. If the CAS operation succeeds (the present thread has relinquished), the *IncrementCount* request will read the incremented count as 1 and thus the invoking thread is given exclusive access to key. This guarantees that Invariant 4.6 is satisfied and only a single thread has exclusive access to an element. If the CAS operation fails, this implies that there are pending requests and then the element is still marked as *busy* and the SWP operation in line 10 reads the accumulated counts and writes 1 preventing other threads to acquire the key. Thus, a thread that has exclusive access to key can only relinquish the key if there are no pending requests and this satisfies Invariant 4.1. All other invariants of the hash table are satisfied through the precondition.

Implementation of *TryRemove* and its Correctness

The *TryRemove* operation is the most complicated of all the operations. This operation will remove a key from the hash table only if (i) it is present in the hash table, (ii) is not acquired by some other thread, and (iii) there are no pending requests on key. In addition, the field for checking availability of the key (the $count$ field) is different from the field to mark the key as deleted (the $value$ field). Thus a single atomic operation cannot achieve the remove operation. In addition, the *IncrementCount* operation does not synchronize with other operations unless there is an *Insert*, and hence acquiring a lock to guarantee atomicity of this operation also does not work. Algorithm 4.3 provides the pseudocode of an implementation of the operation. The operation is implemented "optimistically", and if at any point before completion it is detected that interaction with other threads have changed the state of the entry, then the operation is undone or compensated accordingly (lines 13 – 33).

Algorithm 4.3 *TryRemove*

```
1:  Procedure TryRemove(key, status, incr)
2:  index := hash(key), ptr := buckets[index].head
3:  if ((ptr := Lookup(key)) = null) then
4:      status := failure
5:      return null
6:  value := ptr.value
7:  if (⟨CAS(value.count, 0, 1)⟩) then
8:      ptr.value := null
9:      if (⟨CAS(value.count, 1, 0)⟩) then
10:         status := success
11:         return value
12:     else
13:         incr = ⟨SWP(value.count, 1)⟩ − 1
14:         buckets[index].lock()
15:         ptr := buckets[index].head, emptyLoc := null
16:         while (ptr ≠ null) do
17:             if (ptr.value ≠ null && ptr.key = key) then
18:                 buckets[index].unlock()
19:                 count := ⟨AAF(ptr.value.count, incr)⟩
20:                 if (count = incr) then
21:                     ⟨SWAP(ptr.value.count, 1)⟩
22:                     status := success_and_increment
23:                     return ptr.value
24:                 else
25:                     status := success
26:                     return value
27:             else if (ptr.value = null) then
28:                 emptyLoc := ptr
29:             prev := ptr, ptr := ptr.next
30:         status := pending_increment
31:         Insert into List
32:         buckets[index].unlock()
33:         return emptyLoc.value
34: else
35:     status := busy
36:     return null
37: end Procedure TryRemove
```

The operation first performs a *Lookup* to ascertain the presence of key (line 3) and this ensures that Invariant 4.5 is satisfied. If the lookup succeeds, then it is checked for availability (line 7). If this check fails, then the operation fails and returns busy (line 35). If the CAS in line 7 succeeds, this implies that the key is available, and the operation now marks it as busy by placing 1 in the $count$ field as a result of the successful CAS and this satisfies Invariant 4.6. The element is logically removed by the assignment in line 8, and now if there are still no pending requests on key, then the CAS in line 9 would succeed and the *TryRemove* operation succeeds as well. But if there are any pending requests for key ("delegated" while the present thread was holding it), then the CAS fails, and so the logical deletion has to be undone in order to satisfy Invariant 4.1. The pending count is read through the SWP in line 13, and the element is marked as busy by writing 1 in the corresponding $count$. Since the entry was logically removed, the location is now free to be reused (as per Invariant 4.4) and may or may not be avail-

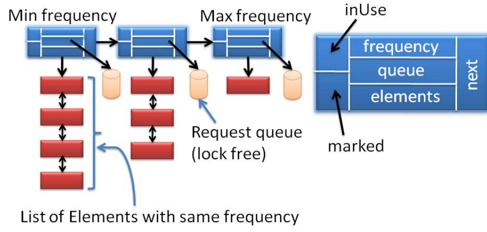


Figure 8: *Concurrent Stream Summary* structure based on "thread cooperation".

able. Since there can be a concurrent *Insert* equivalent operation on the same key, the reinsert must synchronize, and this synchronization is achieved through the lock at the bucket. To ensure that while the key was logically removed, the key was not reinserted by some other thread, a lookup is performed (line 16) and this ensures that Invariant 4.5 is satisfied. If the key has not been inserted, then this thread will atomically reinsert the key, mark it as busy, signal failure and return the pending requests (line 31). If on the other hand, the key was inserted by some other thread, then this implies that the remove was successful (because assuming success of this remove, some other thread has already inserted the key, and therefore this remove cannot be aborted). Based on whether the newly reinserted element is *available* or *busy*, the pending requests (read in line 13) are delegated (line 19) or processed by the current thread (line 21). In either case, the pending requests are preserved (satisfying Invariant 4.1) and only a single thread has exclusive access to a key (satisfying Invariant 4.6). Thus, *TryRemove* ensures that all invariants remain satisfied.

4.3.2 Concurrent Stream Summary

In this section, we discuss the implementation of the *Concurrent Stream Summary* structure based on the proposed *thread cooperation* paradigm. The concurrent analogue of *Stream Summary* is very similar to the original *Stream Summary* structure in [22]. This modified structure is shown in Figure 8. The structure consists of a singly linked list of *frequency buckets*. Each bucket maintains a list of elements which have the same frequency as that represented by the bucket. The buckets are in ascending order of frequency (from left to right), and the elements traverse through this structure of buckets as their frequency changes while the stream is being processed. Each bucket has a *queue* of requests for holding requests delegated to the bucket, and we use a standard "lock-free" queue implementation [29]. As shown in Figure 8, each bucket in the linked list consist of different fields. The $next$ field points to the next node in the list. The *inUse* field is used to guarantee **bucket level synchronization** and if set to 1, indicates that the bucket is already owned by a thread. When a thread wants to access a bucket, it performs a CAS of 0 (expected value) with 1 (new value) on the *inUse* field. Success implies that the bucket was available and now *inUse* is set to 1, and the thread performing the operation gets exclusive access to it. On failure, the thread appends the request to the bucket's queue, and moves on to the next stream element. The $marked$ field is used to represent logical presence, and if it is set to true, then the node is considered to have been *logically removed* from the list and marked for *garbage collection*.

Optimistic Traversal: Multiple "reader" threads might traverse the list of frequency buckets, and the list has been designed to allow *optimistic traversal*. In other words, the readers traverse the list without acquiring locks, and abort the traversal if something goes wrong. To ensure correctness and safety, the following invariant should always be satisfied:

INVARIANT 4.7. *At no point of execution should a "reader" read invalid locations (garbage), i.e., all pointers in the list of frequency bucket should point to valid buckets or be set to* null*.*

Maintenance of sort order: The sort order of the buckets should be maintained at all times according to the following invariant:

INVARIANT 4.8. *If m and n are two buckets such that $m.next = n$, and $m.marked =$ false $\wedge$ $n.marked =$ false, then: $m.frequency < n.frequency$.*

Since the readers traverse "lock-free", if empty buckets are reused for different frequencies, it could lead to violation of the sort order. The following invariant prevents this:

INVARIANT 4.9. *Once a bucket is assigned a frequency, its frequency can never be changed.*

Lazy Removal: If a bucket does not contain any elements and there are no pending requests, its $marked$ field is atomically set to true and is *logically removed* from the structure. Physical removal is done lazily. Although marked buckets are still present in the structure, any operation on these buckets is not-defined, so any "reader" accessing a marked bucket should abort the read and restart the process. To ensure correctness, the following invariant should hold:

INVARIANT 4.10. *A bucket with $marked$ set to* true *cannot be reused. It can only eventually be removed from the structure. Any operation on a marked bucket is undefined and should fail.*

Once a bucket is logically removed, physical removal from the structure is done by a thread which acquires the bucket immediately preceding the *marked* bucket. If a sequence of buckets are marked, the entire sequence can be removed by the thread owning the bucket immediately preceding the sequence. This comprises the garbage collection process. During physical removal, only the next pointer of the bucket immediately preceding the sequence of marked buckets is changed. The thread owning the preceding bucket can therefore garbage collect the buckets without acquiring additional locks.

INVARIANT 4.11. *A marked bucket or a sequence of marked buckets can only be physically removed from the structure by a thread that is the owner of the bucket immediately preceding the sequence of buckets. If there are no preceding buckets, then the owner of the bucket can itself physically remove the bucket.*

As "readers" traverse lock free, once a sequence of buckets is removed, there might still be readers traversing these buckets. The structure of the sequence is therefore preserved so that the readers can eventually rejoin the main structure. A mechanism similar to reference counting in Java, or hazard pointers [24] can be used to reclaim a physically removed bucket without resulting in dangling pointers in threads that might still have a pointer to the reclaimed memory. We now explain the individual operations of *Stream Summary* which implements the operations in Table 1.

Implementation of AddElement and its Correctness

An *AddElement* request arrives at a bucket either when a new element is being added to the **Monitored set** (corresponding to *AddElement* in Table 1), or an existing element's count is being incremented (during increment and overwrite). Owing to the *delegation*

Algorithm 4.4 Processing the *AddElement* Request

```
1: Procedure AddElement(element, bucket)
2: if (element.frequency = bucket.frequency) then
3:    bucket.AddToList(element)
4: else if (element.frequency < bucket.frequency) then
5:    if (bucket = minFreq) then
6:       newBucket := getNewBucket(element.frequency)
7:       newBucket.AddToList(element)
8:       newBucket.next := bucket
9:       minFreq := newBucket
10:   else
11:      DelegateRequestToBucket(minFreq)
12: else
13:   FindDestBucket(bucket, element)
14: end Procedure AddElement
```

of the requests, the *Add* request to a bucket can be of three types: (i) an element with the same frequency as that of the bucket, (ii) an element with frequency greater than that of the bucket (*delegation* of the add request for a not existent bucket immediately following the current bucket), and (iii) an element with frequency less than the present bucket (add request of new element with frequency 1 being added to the **Monitored set** and *delegated* to a minimum frequency bucket whose frequency is greater than 1). Algorithm 4.4 provides a high level overview of how the add request is processed, and this operation subsumes the *AddElement* operation in Table 1. If the addition is to the same bucket, it can be processed right away, while addition to a higher frequency bucket is handled by the *FindDestBucket*. For inserting elements with frequency less than the current minimum frequency, the request is *delegated* to the minimum frequency bucket.

To argue the correctness of Algorithm 4.4, we must show that it does not violate any of the structural invariants. In line 6, when the new node is created, the creator is the only thread with access to the bucket and hence is free to perform any operation on the bucket[2]. The new bucket is first made to point to the old minimum frequency (line 8) and then it is made *public*[3] by the assignment in line 9. Thus Invariant 4.7 is satisfied at all times. Again, the conditional at line 5 ensures that Invariant 4.8 is maintained. Therefore, if *FindDestBucket* does not violate any invariants, then all other invariants are satisfied because this function does not affect those properties. Therefore, this operation is correct.

Implementation of FindDestBucket and its Correctness

Finding the next bucket when incrementing the count of an element is an important operation for both *AddElement* and *IncrementCounter*. This function finds the appropriate bucket for inserting an element into the structure. If the current next node is not the destination for the element, then either a new bucket needs to be inserted after the present bucket, or the list of buckets needs to be traversed. List traversal is necessary to handle **bulk increments** resulting from delegations in the hash table (Section 4.3.1). In either case, the design of the structure allows optimistic traversal. If at any point during traversal through the list, the “reader” finds that it is accessing a node that has been marked for garbage collection, then it aborts the present run, and starts the traversal again. From an efficiency perspective, this failure and abort will be rare, as this case would arise when dealing with **bulk increments**, and this is common only for the high frequency elements, which are generally towards the extreme right of the **Concurrent Stream Summary** structure. For the less frequent elements in the middle of the structure, the next node will be the destination in most cases.

[2]Such a node is called *private* and only the creator can access it.

[3]Making *public* implies that other threads can access the bucket and is now treated as a shared resource.

Algorithm 4.5 Finding the Destination bucket for an element

Require: Invoking thread has exclusive access to startBkt.
Ensure: The element has either been added, or addition has been delegated.
Ensure: *Garbage Collected* any candidate bucket next to startBkt.

```
1: Procedure FindDestBucket(startBkt, e)
2: GarbageCollectCandidateBuckets(startBkt)
3: nextBkt := startBkt.next
4: if (nextBkt = null || nextBkt.frequency > e.frequency) then
5:    newBkt := getNewBucket(e.frequency)
6:    nextBkt.AddToList(e)
7:    nextBkt.next := nextBkt
8:    startBkt.next := nextBkt
9: else if (nextBkt.frequency = e.frequency) then
10:   DelegateRequestToBucket(nextBkt)
11: else if (nextBkt.frequency < e.frequency) then
12:   /*Process a Bulk Increment.*/
13:   repeat
14:      prevBkt := startBkt
15:      nextBkt := prevBkt.next
16:      repeat
17:         if (!(nextBkt.marked)) then
18:            prevBkt := nextBkt
19:         nextBkt := nextBkt.next
20:      until (nextBkt ≠ null && nextBkt.freq ≤ e.freq)
21:      returnStatus := DelegateRequestToBucket(prevBkt)
22:      /*If returnStatus is false, then the read has to rolled back and
         restarted.*/
23:   until (returnStatus ≠ true)
24: else
25:   startBkt.AddToList(e)
26: end Procedure FindDestBucket
```

Algorithm 4.6 Processing the *IncrementCounter* Request

```
1: Procedure IncrementCounter(element, increment)
2: bucket.RemoveFromList(element)
3: element.frequency += increment
4: FindDestBucket(bucket, element)
5: if (bucket.isEmpty() && bucket.noPendingRequests()) then
6:    gcStatus := bucket.atomicMarkGarbageCollected()
7:    if (gcStatus = true && bucket = minFreq) then
8:       newMinFreq := findNewMinFreqBucket(minFreq)
9:       minFreq := newMinFreq
10: end Procedure IncrementCounter
```

Again, to prove correctness, we show that this operation does not violate any of the structural invariants. According to the precondition of the operation, the invoking thread must have exclusive access to $startBkt$ and hence garbage collecting any marked nodes starting from $startBkt.next$ (line 2) satisfies Invariant 4.11. Similar to *AddElement*, the bucket created at line 5 is *private* and before it is made *public* in line 8, the next pointer is set to a valid location (line 7). Thus Invariant 4.7 is satisfied, and the conditional check in line 4 ensures that Invariant 4.8 is satisfied. If the element is to be added to the next bucket, then that bucket being a shared resource (and since the thread is already holding $startBkt$), the request is delegated (line 10) in accordance to Principle 4.2. When processing a bulk increment (lines 16–23), the thread might encounter a marked bucket when trying to delegate the request (line 21). In case of failure, the thread can rollback and restart the process. This ensures that Invariant 4.10 is satisfied. Al other invariants are satisfied as precondition as hence, this operation is also correct.

Implementation of IncrementCounter and its Correctness

The *IncrementCounter* request (Table 1) increments the count of an element while maintaining the sort order of the elements, and is

Algorithm 4.7 Processing the *Overwrite* Request

```
1: /* minFreq is the pointer to minimum frequency bucket. */
2: Procedure Overwrite(element, increment)
3: bucket := minFreq
4: if (deferAllOverwrites) then
5:    /*There are no candidates to be overwritten, so defer the requests.*/
6:    DelegateRequestToBucket(bucket)
7: curElement := bucket.firstElement
8: while (curElement ≠ NULL) do
9:    hashtable.tryRemove(curElement.element, status, incr)
10:   if (status = success) then
11:      /* The current element was successfully removed from the hash
         table, so it can be overwritten. */
12:      bucket.removeElementFromList(curElement)
13:      element.error := bucket.frequency
14:      element.frequency := element.error + increment
15:      FindDestBucket(bucket, element)
16:      break
17:   else if status = pending_increment then
18:      IncrementCounter(curElement, incr)
19:   /* The current element is busy, so move to the next element. */
20:   curElement := curElement.next
21: if (bucket.isEmpty()) then
22:   /* Move all pending requests to the next available bucket. */
23:   Select new Minimum Frequency Bucket as shown in Increment-
      Counter (Algorithm 4.6)
24:   deferAllOverwrites := false
25: if (curElement = NULL) then
26:   /* The OVERWRITE request was not processed because there are
      no candidates to be overwritten. So append the request to the end of
      the queue and defer all further overwrites till we have some candi-
      date that can be overwritten. */
27:   DelegateRequestToBucket(bucket)
28:   deferAllOverwrites := true
29: end Procedure Overwrite
```

one of the most frequent operations for the application under consideration. Algorithm 4.6 provides an overview of how the operation is implemented. After an element has been incremented (i.e., removed from the original bucket), if the bucket becomes empty and has no pending requests, then the bucket should be atomically marked for garbage collection.

As above, correctness of the operation is proved by showing it does not violate any of the structural invariants. Each of the constituent operations satisfy the invariants (as argued in the previous sections). If after the increment, the bucket becomes empty, and there are no pending requests, then the thread will atomically mark for garbage collection (line 6) thus satisfying Invariant 4.9. If the minimum frequency bucket is marked for garbage collection, since there are no other buckets preceding this bucket the current thread removes the bucket (line 9) thus satisfying Invariant 4.11. The *findNewMinFreqBucket* is similar to the *FindDestBucket* operation and returns the first valid bucket immediately after the current *minFreq* bucket, and in the process removes any marked nodes if present. Since this find minimum operation is similar to the *FindDestBucket* operation, its correctness is similar to the correctness proof of *FindDestBucket*. Other invariants are satisfied because this operation does not alter these properties.

Implementation of Overwrite and its Correctness

Overwrite is specific to the *Space Saving* algorithm which uses this operation to limit the number of counters monitored. *Overwrite* implements *Space Saving* algorithm's heuristic of replacing a minimum frequency element with a new un-monitored incoming element. It selects a candidate element from the minimum frequency bucket, overwrites it with the current element being processed, and then increments the count of the new element. To select a candidate for overwriting, the thread will follow the principle of *Minimal Existence* (Principle 4.2) and will not block on any shared resource. It will start from the first element in the minimum frequency bucket, and to overwrite the element, the corresponding entry in the *Search Structure* should be deleted. This deletion is non-blocking, and failure implies that some other thread is trying to increment that element, and the increment request should be in the request queue of the minimum frequency bucket. The thread then moves on to the next element in the bucket, and the process is repeated. If all elements are busy and none of them could be overwritten, this implies that all these elements have pending increment requests. The overwrite request is thus *deferred* till all the increment requests have been processed. Since a single thread will process all these requests, the processing can be highly optimized as the thread now has more knowledge about the requests.

The correctness of this operation is also very straightforward. Since the thread executing this operation has exclusive access to the minimum frequency bucket, and changes made inside the bucket (lines 10–18) does not violate any invariant. When the bucket falls empty (line 21), then a logic similar to that of Algorithm 4.6 is used, and the correctness argument follows from there. If the remove request in the hash table returns pending increments (line 17), then the corresponding *IncrementCounter* call (line 18) ensures that Invariant 4.1 is satisfied. Since all other operations do not make any changes to the structure, no invariant is violated by this operation, and hence is correct.

4.3.3 Auto Tuning and Throttling

The crux of the proposed "thread cooperation" model is the use of request *delegation* to remove "waits". When a request is delegated inside the hash table (*Search Structure*), the counts accumulate, and all delegated requests are processed in bulk. But when a request is delegated in the *concurrent stream summary* structure, each delegated request adds to the queue of requests, and the thread currently owning the bucket has to process all the requests to satisfy Invariant 4.1. Thus, if many requests are delegated to the same bucket, this will lead to an excessive increase in the length of the request queue, and in turn may lead to performance degradation. This happens predominantly for uniform streams where most of the elements are infrequent, and therefore a large number of elements are deleted from the **Monitored Set**. Since all deletions happen at the minimum frequency bucket, this bucket becomes a "hot-spot". In such a scenario, parallelism is limited by the structure, and additional threads are simply adding to the overhead of the system.

To avoid performance degradation in such a scenario, we propose a mechanism of *auto tuning and throttling* of threads. Whenever a thread delegates a request in the *stream summary*, it monitors the queue size of the bucket to which the request is delegated. If the queue size is above a threshold ρ, then this bucket is a potential "hot-spot" and continuous additions to its request queue might lead to performance degradation. Therefore, this thread will "throttle" itself and sleep for a small time slice. This can be treated as another form of "thread cooperation" where a thread abstains from overloading an already overloaded bucket. This feedback loop dramatically improves the performance when dealing with uniform streams, but also introduces some overhead due to the necessity of monitoring the queue sizes. In Section 5 we evaluate the benefits as well as overhead resulting from *auto tuning and throttling*.

4.3.4 Space Analysis

The parallel implementation of *Space Saving* as described earlier uses a single shared *concurrent stream summary*. But additional space is required to "queue" requests that have been delegated. If

an element has been queued, at any instant, there is only one request corresponding to that element. This is equivalent to the element being in the input stream, which would anyway have to be allocated space to be buffered. Thus instead of the element being present in the input buffer, it is encapsulated as a request in the queue. Additional space is required for locks and synchronization, which is again of the same order as the number of counters. Thus, the space complexity for the parallel algorithm is of the same order as that of the sequential algorithm, i.e., $O(\frac{1}{\epsilon})$.

4.3.5 Answering Queries

The *Concurrent Stream Summary* structure maintains the elements in a sorted order, so that queries can efficiently traverse this structure to find the appropriate elements. Queries can be processed with high efficiency because the elements of interest will have high frequencies and reside in the rightmost end of the structure, and the low frequency elements will be cluttered in the leftmost end (assuming frequencies increase from left to right). Therefore, as the queries start from the minimum frequency, they can very quickly prune out the low frequency elements and reach the region of interest. Again, queries can be answered without acquiring any locks.

Point Queries: These queries verify whether the query point is frequent or is contained in the top-k. Frequent elements queries are straightforward and can be answered directly from the *Search Structure* without accessing the *Concurrent Stream Summary*. This is done by looking up the query point in the search structure and reporting it as frequent if its frequency is above the query threshold. For top-k queries, the frequency of the kth element needs to be determined, and this can be done by a traversal through the structure, reading the buckets for the number of elements and the request queue statistics, thereby counting the number of elements that are present to the right of the present bucket. With a count of the number of elements in the structure, the bucket to which the kth element belongs can be determined. Once this is known, if the frequency of the query point is above the kth frequency, then the point is in the top-k, otherwise the answer is negative.

Reporting the Answer Set: These queries report the entire set of candidates that qualify the frequent elements query for the specified support or the top-k query for a given value of k. Answering these queries are costlier since they need to traverse through the elements in the bucket. Again, once the appropriate buckets are determined through a traversal of the list of buckets, the elements can be reported by a traversal of the list of elements in the bucket.

4.4 Generalization of the Framework

In this paper, we explain the application of the proposed framework for parallelizing the *Space Saving* algorithm. One of the reasons for selecting *Space Saving* was that a recent work [6] compared a number of frequency counting algorithms and demonstrated that *Space Saving* outperforms similar algorithms in terms of processing speed and accuracy (expressed as precision and recall). Another reason is that *Space Saving* has a constant space bound of $O(\frac{1}{\epsilon})$ and this simplifies the algorithm implementation.

The *thread cooperation* based framework is not limited to the *Space Saving* algorithm and can be easily extended to other frequency counting algorithms. It is evident from the description of the *IncrementCounter* operation that the framework can handle arbitrary increments in frequency of the elements. Thus, any frequency counting algorithm in which the frequency of the counters *increases monotonically* (algorithms such as *Lossy Counting* [21]), can be adapted to the framework. For adapting other algorithms, most of the operations (except the *Overwrite* operation) will remain unchanged. For example, the *Lossy Counting* [21] algorithm divides the stream into multiple rounds and at the end of the round removes elements which are infrequent. Therefore, for adaptation into the framework, only the *Overwrite* request in *Space Saving* has to be replaced by a request that removes the minimum frequency bucket at round boundaries.

5. EXPERIMENTAL EVALUATION

In this section, we experimentally evaluate the proposed framework. The experiments have been performed on an Intel Core 2 Quad Q6600 processor [18]. This processor has 4 cores, each corresponding to a hardware thread and operating at a clock speed of 2.4GHz, and the cores share a $L2$ Cache of 4MB. The machine has 3.2 GB available main memory and runs Fedora Core Linux with kernel 2.6.26.6-49.fc8. All algorithms and the framework have been implemented in C++ and compiled using GNU C++ compiler with Level 2 optimization. The data set is synthetically generated and follows zipfian distribution which is very close to realistic data distribution [32]. The zipfian factor α determines whether the distribution is uniform or skewed. The frequency of the elements in the distribution varies as $f_i = \frac{N}{i^\alpha \zeta(\alpha)}$ where $\zeta(\alpha) = \sum_{i=1}^{|A|} \frac{1}{i^\alpha}$ where N is the length of the stream, $|A|$ is the size of the alphabet, and f_i represents the frequency of the ith frequent element. Smaller values of α represent lesser skew in the distribution with $\alpha = 0$ representing uniform distribution. As the value of α increases, the skew of the data distribution also increases. The data set has a total of 50 million elements and an alphabet of 5 million. GCC built-in atomic primitives were used for performing the atomic operations, and *pthread* library was used for threads and locks. In our first set of experiments, we choose data with α in the range 1.5 to 3.0. The lower α values have not been evaluated because the frequent elements and top-k elements are more interesting and meaningful in a skewed distribution, than in a uniform distribution. In the case of a uniform data (such as $\alpha \leq 1.0$) there are a lot of infrequent elements, and so in *Space Saving*, there will be a lot of *Overwrite* requests, thereby making the minimum frequency bucket a "hot spot". Thus, such a distribution will not benefit from parallelization. For such a scenario, we propose *auto tuning and throttling* (Section 4.3.3), and the effectiveness of the scheme is demonstrated by the experiments in the latter part of this section for data sets with $\alpha \leq 1$. In all our experiments, the number of streams is varied from 4 up to 32 in multiples of 2, and each stream has 1 million elements, i.e., in an experiment with 32 streams, the total number of elements processed is 32 million. The performance of algorithms does not vary significantly for smaller values of error bound ϵ, while larger values of ϵ result in performance degradation. Larger ϵ also results in smaller number of buckets, and hence lesser chance for exploiting parallelism. In our experiments, we set $\epsilon = 0.0001$. In this section, for a fair evaluation, we compare the *cooperative* design to the *shared design*. The *independent* design is not included in the comparison since its performance is heavily dependent on the query rate and ϵ, whereas the remaining two are more stable.

5.1 Saturated Workload

In this experiment, we evaluate the maximum throughput of the proposed design by using a *saturated workload* wherein a thread always has some element to process whenever it has finished processing the earlier element, or in other words, a thread is *saturated* with work. Figure 9 provides a comparison of the execution times of the "cooperation" based design (referred to as **Coop** in this graph as well as the rest of the graphs in this section) and the "contention"

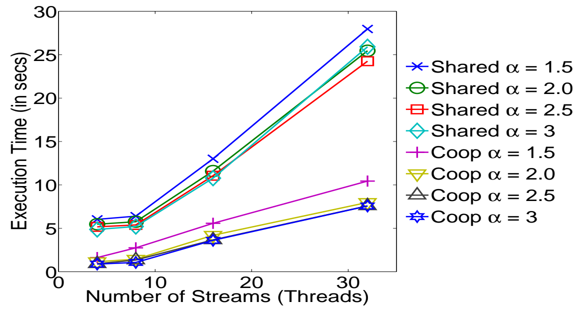


Figure 9: Execution time for saturated workload.

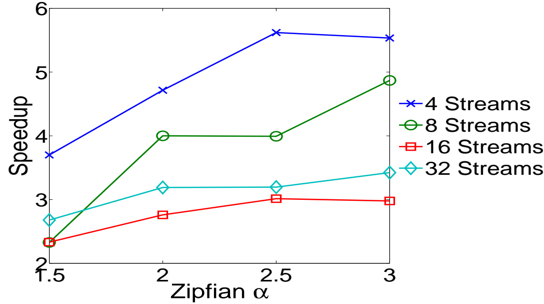


Figure 10: Speedup of the "cooperation" based design compared to the "contention" based design for saturated workload.

based design (referred to as **Shared**). Along the x-axis, we plot the number of streams (and hence the number of threads), and along the y-axis we plot the execution time in seconds. Different lines in the graph correspond to different values of α. An ideal system with linear speed-up will have a horizontal line, and the smaller the slope of the graphs, the better it is. As is evident from Figure 9, not only does **Coop** outperform **Shared** in terms of execution time, **Coop** demonstrates better scalability since the slope of the lines corresponding to **Coop** is considerably smaller than that of **Shared**. The benefits of **Coop** with respect to execution time can be seen from Figure 10 which plots the speedup of **Coop** compared to **Shared**. In this figure, the x-axis represents the varying zipfian α and the y-axis corresponds to the speedup. For a particular zipfian α, the speedup is computed as: $T^{Shared}_{(i,\alpha)}/T^{Coop}_{(i,\alpha)}$, where i represents the number of streams (4, 8, etc.). As is evident from Figure 10, **Coop** outperforms **Shared** by a factor ranging from $2 - 5.5X$. This experiment demonstrates the superiority of the performance of the proposed "cooperation" based design.

5.2 Unsaturated Workload

In many realistic applications, the workload might not be saturated, i.e., due to variable arrival rates, when a thread is ready to process the next stream element, there is a likelihood that the next element has not yet arrived. In such a case, we say that the read "failed" and the thread needs to wait for the arrival of the next element when the read "succeeds". We model this behavior by allowing a read to "fail" with certain probability. In other words, a failure probability of 10% means that with every read, there is a 10% chance that the read will fail. Thus the saturated workload considered in the previous subsection has a failure probability of 0%. That experiment would demonstrate the overhead on the systems at higher arrival rates. When a read "fails", the thread does not access the **Monitored Set**. This might have two implications: $i)$ if the structure is already overloaded, this will lead to a reduction in

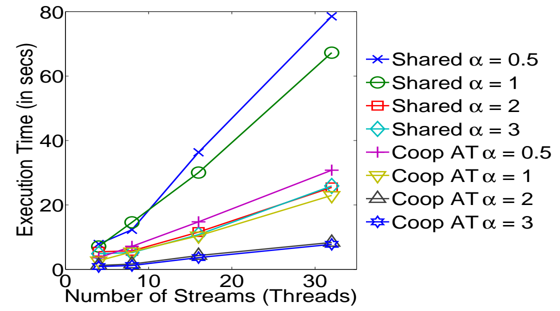


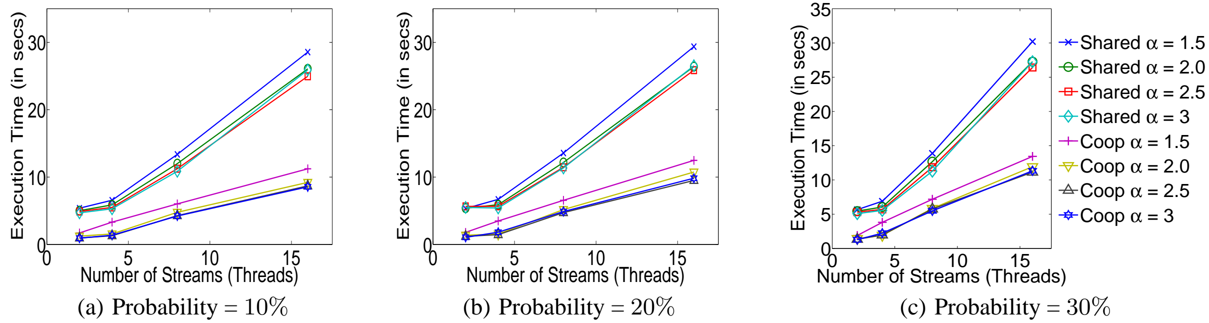
Figure 13: Comparison of the execution time for saturated workload. In this experiment, the "cooperation" based design with "auto-tuning" is compared with the "shared" design.

the load and can be treated as "load-shedding" and this might result in reduced execution times, $ii)$ if the structure is lightly loaded, it has the potential to accommodate more concurrency, and hence a failed read implies failed opportunity and this will lead to increased execution time. Thus, for a system that is performing at its peak for a saturated workload, an unsaturated workload will result in an increase in total time of completion due to slower arrival rate. But for a system which is overloaded for a saturated workload, an unsaturated workload might lead to lesser execution time.

Figure 11 plots the execution time of the two designs as the failure probability is varied. Along the x-axis we plot the number of streams and along the y-axis, we plot the execution time in seconds. The different lines correspond to different values of zipfian α and the different designs. The zipfian α of the input distribution is varied from 1.5 to 3.0 and the experiment is repeated for different probabilities of not finding a stream element when a thread is ready. It is evident from Figure 11 that the behavior of the designs is almost similar to that with the saturated case (Figure 9) except that with increasing failure probability, the graphs for **Coop** and **Shared** are getting closer. This is due to the fact that the low contention overhead in **Coop** allows more concurrency compared to that of **Shared**. In **Shared**, the contention overhead is very high (Figure 6), and if a thread "fails" a read and does not access the structure, it contributes to reducing the contention overhead on the structure, and hence the overall execution time does not increase much. The increased execution time of **Coop** with increasing failure probability is further evident from Figure 12 which plots the speedup corresponding to the times in Figure 11. In this figure, we again plot the failure probability along the x-axis and the speedup along the y-axis. The speedup is computed similar to the way described in the previous subsection, but instead of computing at a particular α, here it is computed at a particular value of the probability. As is evident from Figure 12, as the failure probability increases, the speedup decreases. These experiments further strengthen the claim that the "cooperation" based design has low contention overhead and allows higher concurrency, as a result of which decreased packet arrival rate increases execution time which is bound to happen in a system operating at its peak.

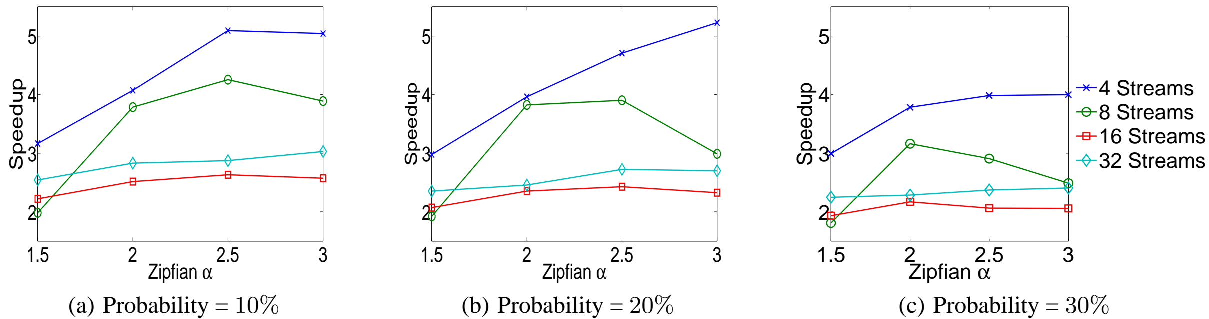
5.3 Evaluation of Auto Tuning

In Section 4.3.3, we discussed the potential for using *auto tuning and throttling* to deal with "hot-spots" when processing uniform data distributions. In all the previous experiments, we considered distributions for zipfian $\alpha \geq 1.5$, and in this section, we evaluate the impact of *auto tuning* for $\alpha \leq 1.0$. Due to too many overwrites of minimum frequency element in the case of uniform distributions, the minimum frequency bucket becomes a "hot-spot". Profiling ex-



(a) Probability = 10% (b) Probability = 20% (c) Probability = 30%

Figure 11: Comparison of the execution time for an unsaturated workload. The zipfian α of the input distribution is varied from 1.5 to 3.0 and the experiment is repeated for different probabilities of not finding a stream element when a thread is ready.



(a) Probability = 10% (b) Probability = 20% (c) Probability = 30%

Figure 12: Speedup of the "cooperation" based design compared to the "contention" based design for an unsaturated workload. The zipfian α of the input distribution is varied from 1.5 to 3.0 and the experiment is repeated for different probabilities of not finding a stream element when a thread is ready.

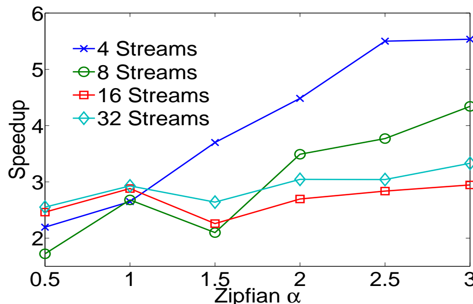


Figure 14: Speedup of the "cooperation" based design with "auto-tuning" compared to the "contention" based design for saturated workload.

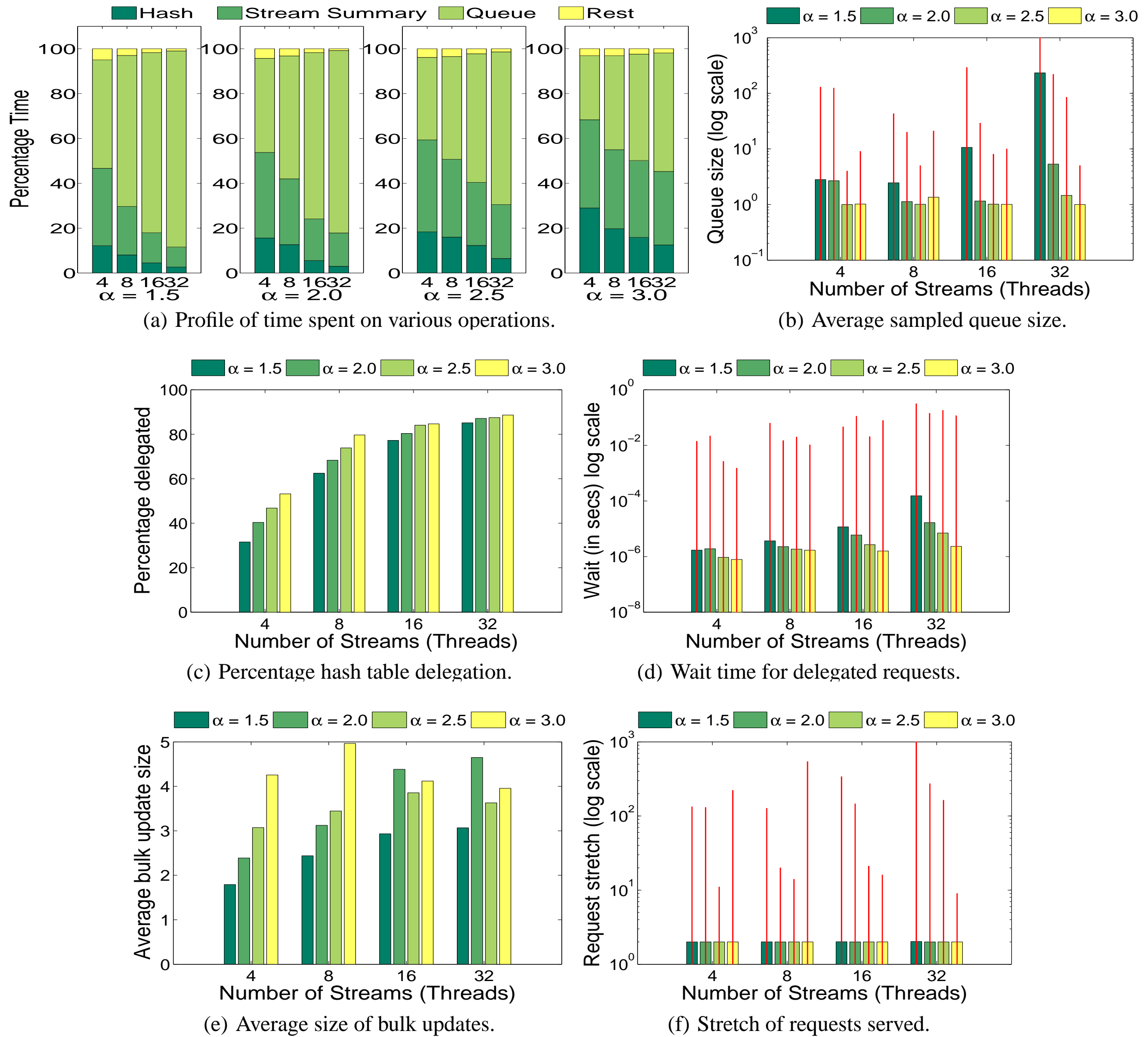
periments revealed that without *auto-tuning*, for $\alpha = 0.5$, the maximum sampled queue size grows to about $3-4$ orders magnitude higher than in the case with $\alpha \geq 1.5$. This results in severe performance degradation since a majority of the work is delegated to a single thread, thereby reducing the system to a sequential system with all the overhead of a concurrent system. *Auto tuning and throttling* prevents queues from building up rapidly, and allows better distribution of the work. Our experimental samples using $\alpha \geq 1.5$ showed maximum queue size of the order of hundreds and average queue size of the order of tens. So, in our experiments, we set the queue size threshold ρ to 500. Therefore, for skewed data, the threads will hardly be throttled and can perform at their peak rate, while for almost uniform data, threads will be repeatedly throttled incorporating admission control and better load distribution.

In Figure 13, we plot the execution time of the Cooperation based design using *auto tuning* and compare it with **Shared**. Along the x-axis, we plot the number of streams and along the y-axis, we plot the execution time in seconds. Different lines correspond to different values of α, and **Coop AT** represents "cooperation" based design with *auto tuning*. In this experiment, we use a saturated workload. To improve clarity of the graph, we only provide the lines corresponding to $\alpha = 0.5$ and 1 (our new data sets) and $\alpha = 1.5$ and 2.5 (representatives from the previously used data sets). As is evident from the figure, **Coop AT** performs reasonably for lower values of α (almost uniform) where **Coop AT** outperforms **Shared** by a factor of $\sim 2X$. For higher values (skewed), its performance is almost similar to that of the "cooperation" based design without *auto tuning*.

Auto tuning introduces the overhead of profiling of queue sizes, and when the threads are not throttled (in the skewed case), this is pure overhead compared to **Coop**. But our experiments reveal that this overhead is limited to $\sim 5\%$ (for example, for $\alpha = 2.0$ and number of streams as 16, the average execution time for **Coop** is 4.195586s and that of **Coop AT** is 4.299514 resulting in an overhead of $\sim 3.29\%$). Thus for a very small overhead, we have an algorithm that performs well for the entire spectrum of α. If the system designer knows that the stream to be processed has a high skew, then *auto tuning* can be turned off allowing higher performance, but turning on *auto tuning* only marginally degrades performance while allowing greater flexibility.

5.4 Profiling the Cooperation based design

In this section, we provide an in depth analysis of the proposed "cooperation" based parallel design with and without *auto tuning*



(a) Profile of time spent on various operations.

(b) Average sampled queue size.

(c) Percentage hash table delegation.

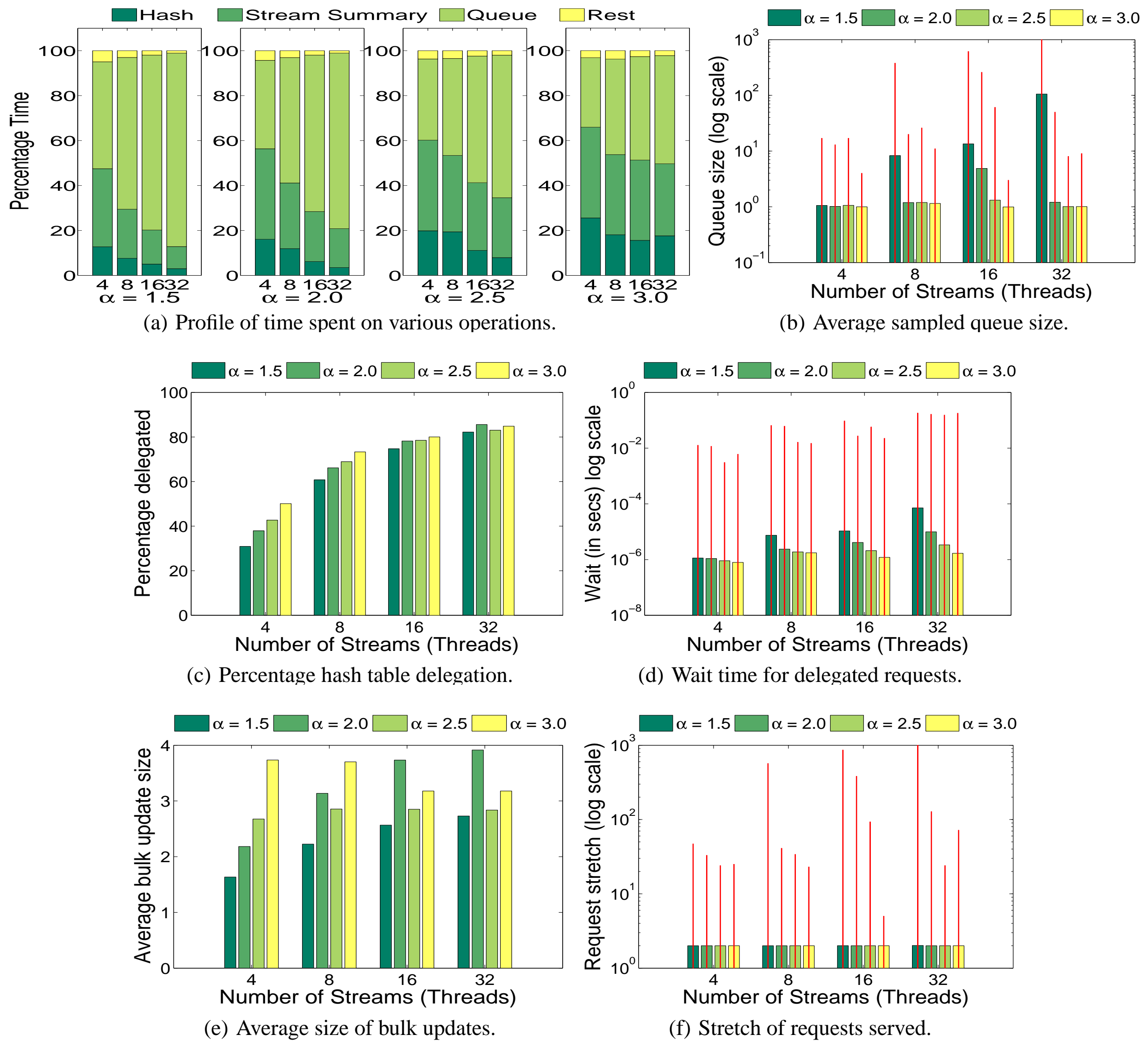
(d) Wait time for delegated requests.

(e) Average size of bulk updates.

(f) Stretch of requests served.

Figure 15: Detailed profile of cooperation based design without auto tuning while processing a saturated workload.

for both saturated and unsaturated workloads. Figures 15 and 16 provide results for the design without *auto tuning* for saturated and unsaturated workloads respectively, while Figures 17 and 18 provide the corresponding results for the design with *auto tuning*. For unsaturated workloads, we select failure threshold 20% as a representative graph. The experimental parameters are similar to that used in the experiments of the previous set of experiments. The number of streams and hence threads are varied from 4 to 32 in multiples of 2, and the number of elements is each stream is set to 1 million. For the design without *auto tuning*, the zipfian α is varied from 1.5 to 3.0, while for the design with *auto tuning*, α is varied from 0.5 to 3.0. In all the figures, the x-axis plots the number of streams and hence the number of threads in the system. For all figures, sub figure (a) plots the profile of the time spent on the various important operations expressed as a percentage of the total time of execution. The y-axis plots the time spent on operations as a percentage of total time. *Hash* refers to the time spent for the hash table operations, *Stream Summary* refers to the time spent for the *Stream Summary* operations, *Queue* refers to the sum of time spent in appending the delegated requests to the queues associated with buckets in *Stream Summary* and removing the requests from the queue when processing them. Sub figure (b) plots the sampled queue size and the y-axis plots the queue size in log scale. The bars correspond to the average of the sampled queue sizes, while the thin lines correspond to the maximum queue size. Sub figure (c) plots the percentage of requests delegated in the hash table, where the percentage is plotted along the y-axis. Sub figure (d) plots the wait times for the requests that are delegated in the queues corresponding to the buckets in the *stream summary structure*. Recall that when a frequency bucket in the *stream summary* structure is not available, the request is delegated by enqueuing the request in the queue for the bucket. The wait time corresponds to the time gap between the instant when the request is enqueued to the instant at which the request is processed by the thread owning the bucket. The y-axis plots the wait time in seconds (plotted in log scale) and the bars correspond to the average wait times while the thin lines overlaid on the bars correspond to the maximum sampled wait time. Sub figure (e) plots the average bulk update size. Again recall that requests delegated in the hash table result in accumulated counts which are processed in bulk (**bulk increments** in Section 4.2), and



(a) Profile of time spent on various operations.

(b) Average sampled queue size.

(c) Percentage hash table delegation.

(d) Wait time for delegated requests.

(e) Average size of bulk updates.

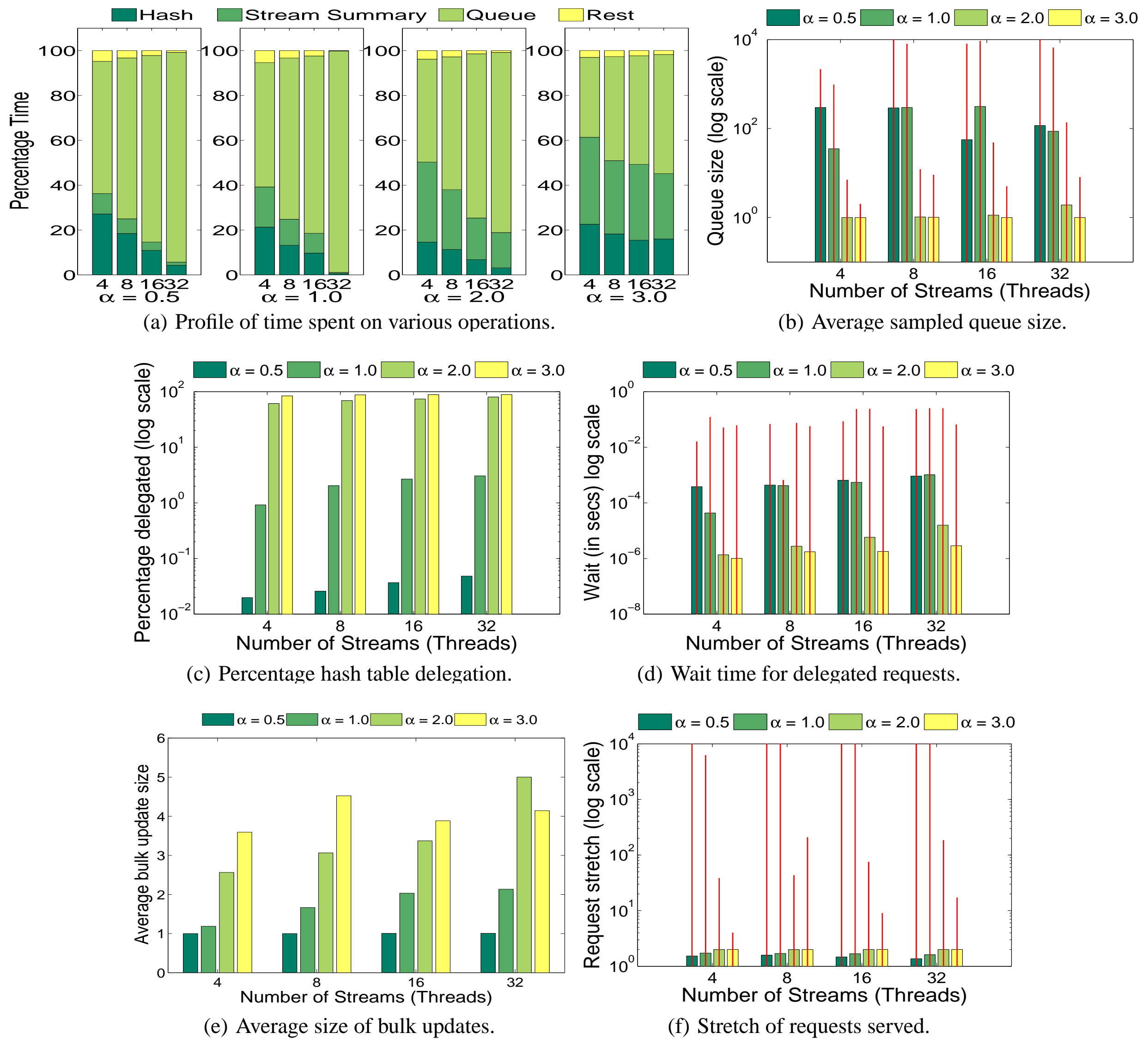
(f) Stretch of requests served.

Figure 16: Detailed profile of cooperation based design without auto tuning while processing an unsaturated workload with a miss probability of 20%.

the y-axis plots the average bulk updates. Finally, sub figure (f) plots the stretch of requests served. Whenever a thread acquires a frequency bucket, it continues to serve all the requests delegated to the buckets, and can only relinquish the bucket once all the requests have been served. A stretch is defined as the number of requests which a thread serves between acquiring a bucket and subsequently releasing it, and the y-axis plots the stretch of requests served (in log scale). As earlier, the bars correspond to the average stretch of requests, while the overlaid lines correspond to the maximum stretch of requests.

Figures 15(a), 16(a), 17 and 18(a) demonstrate that very little time is spent in the hash table for **element level synchronization** and the *Stream Summary* for **bucket level synchronization** (in contrast to Figure 6). A considerable portion of the time is spent on the operations related to the request queue, and a more scalable queue implementation would thus result in further improvement in performance of the proposed design. Our present queue implementation does not allocate memory wisely, and this is in part responsible for the high overhead of the queue operations. Note that the time for the queue operations is higher for smaller values of zipfian α. This is because for more uniform distributions, different threads are generally processing different elements, and hence would not be blocked at the hash table, therefore resulting in large number of accesses to the *Stream Summary*. Since the summary structure has limited parallelism due to **bucket level synchronization**, a lot of requests are delegated in this structure. The increase in number of concurrent accesses to the request queues result in the increase in time for queue operations.

The above analysis is further supported by Figures 15(b), 16(b), 17(b), 18(b) and Figures 15(c), 16(c), 17(c), 18(c) which plot the average sampled queue size and percentage of requests delegated in the hash table respectively. As discussed earlier, for smaller values of zipfian α, the minimum frequency bucket becomes a "hot spot", and this is reflected by the considerably larger queue sizes for $\alpha = 1.5$ in Figures 15(b) and 16(b). The queue sizes for $\alpha = 0.5$ and 1.0 in Figures 17(b) and 18(b) demonstrate the success of *auto tuning* in limiting the queue sizes even for uniform distributions, and explains the improved performance of **Coop AT** as



(a) Profile of time spent on various operations.

(b) Average sampled queue size.

(c) Percentage hash table delegation.

(d) Wait time for delegated requests.

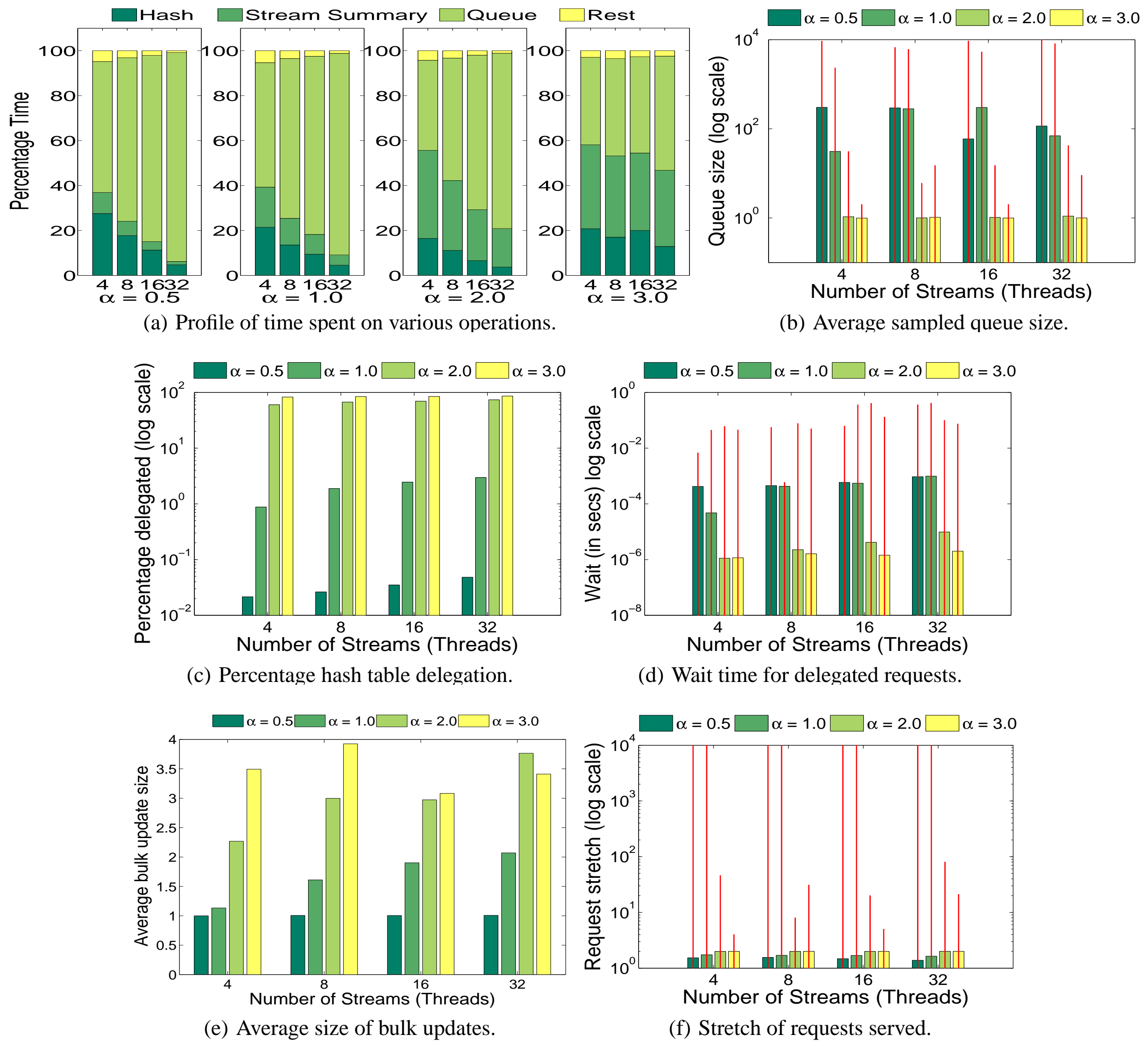
(e) Average size of bulk updates.

(f) Stretch of requests served.

Figure 17: Detailed profile of cooperation based design with auto tuning while processing a saturated workload.

shown in Figure 13. Finally, Figures 15(c), 16(c), 17(c) and 18(c) show the successful removal of *waits* associated with **element level synchronization** by plotting the requests delegated in the hash table as a percentage of the total number of requests. For higher α (skewed streams) and increasing number of threads, more threads are likely to be processing the same elements (since in a skewed distribution, few elements have dominant presence in the stream). Therefore, more requests are delegated in the hash table due to **element level synchronization**. The number of hash delegations increase with the number of threads and the increase in α. Note that in spite of the large number of requests delegated in the hash table, neither is the hash table a point of contention, nor is a significant portion of execution time spent synchronization in the hash table (Figures 15(a) and 17(a)). This demonstrates the efficiency of the proposed *cooperation* based design in reducing the *waits* associated with synchronization.

Figures 15(c) and 15(e) together explains the improved performance of the design for skewed distributions of the input streams. As can be seen, for skewed streams, not only does a large percentage of requests delegated in the hash table (recall that delegation in the hash table is much cheaper when compared to delegation in the stream summary structure), but the requests are processed in larger bulks, resulting in the effect similar to multiple stream elements being processed in a single pass. A similar behavior can be observed for the rest of the experiments plotted in Figures 16(e), 17(e), and 18(e). Note that in Figures 17(e), and 18(e), the average bulk updates for zipfian $\alpha = 0.5$ is almost equal to 1, which coupled with the low percentage of hash table delegations (Figures 17(c) and 18(c)) and increasing queue sizes (Figures 17(b) and 18(b)) further supports the previous analysis that for uniform streams, the *stream summary* structure, and especially the minimum frequency bucket becomes the "hot-spot". Additionally, as demonstrated by Figures 15(d), 16(d), 17(d), and 18(d), the requests delegated in the *stream summary* structure have an average wait time of about a micro second, and maximum wait times of about 10ms. This shows that *delegation* does not introduce huge waits for the enqueued requests, while removing the necessity of threads waiting for locks. Additionally, sub figures (b), (d), and (f) of Figures 15, 16, 17, and 18, show the effectiveness of the proposed *auto tuning* approach in limiting the building up of the



(a) Profile of time spent on various operations.

(b) Average sampled queue size.

(c) Percentage hash table delegation.

(d) Wait time for delegated requests.

(e) Average size of bulk updates.

(f) Stretch of requests served.

Figure 18: Detailed profile of cooperation based design with auto tuning while processing an unsaturated workload with a miss probability of 20%.

request queues for uniform data distributions, and as a result even when the structure becomes a hot spot, performance is still reasonable and does not degrade.

Now evaluating the impact of unsaturated workloads, if we refer to Figures 15 and 16, it can be seen that the reduced workload results in decrease in the percentage of requests delegated in the hash table, and the average size of bulk updates. This further supports the performance results where unsaturated workload resulted in deterioration of performance of the *cooperation* based design, since the proposed design supports more parallelism, and unsaturation leads to lost opportunity of processing more data. Similar trends can be observed in Figure 17 and 18 even when the system uses *auto tuning*.

These results explain the major performance benefits observed in the previous section where performance of the proposed design was evaluated, and further assert the efficiency of the proposed *cooperation* based design in removing waits associated with *contention* based locking designs.

6. CONCLUDING REMARKS

In this paper, we consider the problem of analyzing multiple data streams and propose a parallel algorithm for the problem in the context of multicore processors. The proposed design uses the concept of *thread cooperation* to remove *waits* associated with locks. Removal of the *waits* is particularly significant in the context of multicore processors where a waiting thread results in wasted CPU cycles. Additionally, the proposed model allows synchronization using only atomic operations which are much cheaper than locks required for synchronizing in the “contention” based design. The proposed design conceptually segregates the requests from the execution threads, and whenever a shared resource required by the request is not available, only the request is blocked (*delegated*), and the thread can move on to process the next request. This is extremely important in the context of parallelism in multicore processors, and the benefits of the proposed design is evident from the experimental results which show that the proposed “cooperation” based design outperforms the traditional “contention” based design by a factor of $2 - 5.5X$ over the entire spectrum of uni-

form to skewed data sets. In spite of *multi-stream* analysis systems being extremely hard to parallelize, our parallel design effectively uses the inherent parallelism in multicores with significantly lower overhead. In addition to the gains for comparatively skewed data, our design is also efficient for uniform data sets where achieving parallelism is a much harder problem. In this paper, we implement the *Space Saving* algorithm, but the framework is general enough to accommodate other frequency counting algorithms [21].

During the course of implementation of this system, we gained useful experience and learned several important lessons which apply to parallel systems in general. *First*, even though individual lock operations are not very expensive, acquiring and releasing millions of locks per second results in significant overhead, which is in part responsible for the inefficiency of the *shared design*. *Second*, a lot of operations use locks only for mutual exclusion, and do not require the serialization order imposed by locks. For such applications, mutual exclusion can be achieved by using cheaper atomic operations, while *delegation* allows removal of waits. Exploiting these application specific characteristics might result in significant improvement in performance. *Third*, frequent system calls are also expensive, and introduce significant overhead. Memory allocation is one such expensive call, and frequent memory allocation can considerably deteriorate performance. As a result, spending time with a custom memory allocator is often useful.

In the last few years, there has been growing interest in different non-conventional parallel processing architectures such as Graphics processors [13, 16], cell broadband engine [12, 17], etc. These processors were originally designed for different application domains, but recent results have shown promise in the use of these processors for data management operations. The present *cooperation based* design does not differentiate the processing capabilities of the threads, and hence is suitable for *homogeneous* conventional multicore architectures such as chip multiprocessors (CMP) [18, 27]. Heterogeneous processing capability in combination with specialized instructions, such as vector operations, supported by Graphics and Cell processors open new challenges. In the future, we would like to explore the possibility of extension of the proposed paradigm to these other novel architectures. Additionally, experiments in this paper concentrate on a specific type of CMP architecture [18] known as the *fat camp* processors [14]. The *fat camp* processors are characterized by fewer cores where each core has been optimized for extremely efficient single thread performance. Another class of CMPs, referred to as *lean camp* processors [14], are characterized by large number of cores and hardware threads [27], but each hardware thread is not optimized for efficient single thread performance. In the future, we would also like to evaluate the performance of the proposed design on these *lean camp* processors.

Acknowledgements

We would like to thank the anonymous reviewers, our shepherd Qiong Luo, and Pamela Bhattacharya for their insightful comments that has helped improve this paper. This work is partly supported by NSF Grant IIS-0744539.

7. REFERENCES

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. The Design of the Borealis Stream Processing Engine. In *CIDR*, pages 277–289, 2005.

[2] B. Babcock and C. Olston. Distributed top-k monitoring. In *SIGMOD*, pages 28–39, 2003.

[3] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Load Management and High Availability in the Medusa Distributed Stream Processing System. In *SIGMOD*, pages 929–930, 2004.

[4] A. Bulut and A. K. Singh. A unified framework for monitoring data streams in real time. In *ICDE*, pages 44–55, 2005.

[5] M. Charikar, K. Chen, and M. Farach-Colton. Finding Frequent Items in Data Streams. In *ICALP*, pages 693–703, 2002.

[6] G. Cormode and M. Hadjieleftheriou. Finding frequent items in data streams. *Proc. VLDB Endow.*, 1(2):1530–1541, 2008.

[7] G. Cormode and S. Muthukrishnan. What's Hot and What's Not: Tracking Most Frequent Items Dynamically. *ACM Trans. Database Syst.*, 30(1):249–278, 2005.

[8] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: A Stream Database for Network Applications. In *SIGMOD*, pages 647–651, 2003.

[9] G. Das, D. Gunopulos, N. Koudas, and N. Sarkas. Ad-hoc top-k query answering for data streams. In *VLDB*, pages 183–194, 2007.

[10] S. Das, S. Antony, D. Agrawal, and A. El Abbadi. CoTS: A Scalable Framework for Parallelizing Frequency Counting over Data Streams. In *ICDE*, pages 1323–1326, 2009.

[11] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Frequency Estimation of Internet Packet Streams with Limited Space. In *ESA*, volume 2461, pages 348–360, 2002.

[12] B. Gedik, P. S. Yu, and R. Bordawekar. Executing Stream Joins on the Cell Processor. In *VLDB*, pages 363–374, 2007.

[13] N. K. Govindaraju, N. Raghuvanshi, and D. Manocha. Fast and approximate stream mining of quantiles and frequencies using graphics processors. In *SIGMOD*, pages 611–622, 2005.

[14] N. Hardavellas, I. Pandis, R. Johnson, N. G. Mancheril, A. Ailamaki, and B. Falsafi. Database Servers on Chip Multiprocessors: Limitations and Opportunities. In *CIDR*, pages 339–350, 2007.

[15] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, pages 981–992, 2008.

[16] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *SIGMOD*, pages 511–524, 2008.

[17] S. Héman, N. Nes, M. Zukowski, and P. Boncz. Vectorized data processing on the cell broadband engine. In *DaMoN*, pages 1–6, 2007.

[18] Intel® Core™ 2 Quad Processor Overview. http://www.intel.com/products/processor/core2quad/index.htm; (Accessed 06/08/2009), 2009.

[19] D. E. Knuth. *The Art of Computer Programming, Sorting and Searching*, volume 3. Addison-Wesley, Cambridge, MA, 1997.

[20] A. Manjhi, V. Shkapenyuk, K. Dhamdhere, and C. Olston. Finding (Recently) Frequent Items in Distributed Data Streams. In *ICDE*, pages 767–778, 2005.

[21] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *VLDB*, pages 346–357, 2002.

[22] A. Metwally, D. Agrawal, and A. El Abbadi. An integrated efficient solution for computing frequent and top-k elements in data streams. *ACM Trans. Database Syst.*, 31(3):1095–1133, 2006.

[23] A. Metwally, D. Agrawal, A. El Abbadi, and Q. Zheng. On hit inflation techniques and detection in streams of web advertising networks. In *ICDCS*, page 52, 2007.

[24] M. M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *PODC*, pages 21–30, 2002.

[25] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query Processing, Resource Management, and Approximation in a Data Stream Management System. In *CIDR*, 2003.

[26] K. Mouratidis, S. Bakiras, and D. Papadias. Continuous monitoring of top-k queries over sliding windows. In *SIGMOD*, pages 635–646, 2006.

[27] Sun UltraSPARC T2 Processor – The industry's first true system on chip. http://www.sun.com/processors/UltraSPARC-T2/index.xml; (Accessed 06/08/2009), 2007.

[28] P. E. O'Neil. The escrow transactional method. *ACM Trans. Database Syst.*, 11(4):405–430, 1986.

[29] W. N. Scherer III, D. Lea, and M. L. Scott. Scalable synchronous queues. In *PPoPP*, pages 147–156, 2006.

[30] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53(3):379–405, 2006.

[31] S. Venkataraman, D. Song, P. Gibbons, and A. Blum. New streaming algorithms for fast detection of superspreaders. In *NDSS*, pages 149–166, 2005.

[32] G. K. Zipf. *Human Behavior and The Principle of Least Effort*. Addison-Wesley, Cambridge, MA, 1949.