Hyperion: High Volume Stream Archival for Retrospective Querying

Peter J. Desnoyers
Department of Computer Science
University of Massachusetts

Prashant Shenoy
Department of Computer Science
University of Massachusetts

Abstract

Network monitoring systems that support data archiving and after-the-fact (retrospective) queries are useful for a multitude of purposes, such as anomaly detection and network and security forensics. Data archiving for such systems, however, is complicated by (a) data arrival rates, which may be hundreds of thousands of packets per second on a single link, and (b) the need for online indexing of this data to support retrospective queries. At these data rates, both common database index structures and general-purpose file systems perform poorly.

This paper describes Hyperion*, a system for archiving, indexing, and on-line retrieval of high-volume data streams. We employ a write-optimized* stream file system *for high-speed storage of simultaneous data streams, and a novel use of* signature file *indexes in a distributed multi-level index.*

We implement Hyperion on commodity hardware and conduct a detailed evaluation using synthetic data and real network traces. Our streaming file system, StreamFS, is shown to be fast enough to archive traces at over a million packets per second. The index allows queries over hours of data to complete in as little as 10-20 seconds, and the entire system is able to index and archive over 200,000 packets/sec while processing simultaneous on-line queries.

1 Introduction

Motivation: Network monitoring by collecting and examining packet headers has become popular for a multitude of management and forensic purposes, from tracking the perpetrators of system attacks to locating errors or performance problems. Networking monitoring systems come in two flavors. In *live* monitoring, packets are captured and examined in real-time by the monitoring system. Such systems can run continual queries on the packet stream to detect specific conditions [20], compute and continually update traffic statistics, and proactively detect security attacks by looking for worm or denial of service signatures [9]. Regardless of the particular use, in live monitoring systems, captured packet headers and payloads are discarded once examined.

However, there are many scenarios where it is useful to retain packet headers for a period of time. Network forensics is one such example—the ability to "go back" and retrospectively examine network packet headers is immensely useful for network troubleshooting (e.g., root-cause analysis), to determine how an intruder broke into a computer system, or to determine how a worm entered a particular administrative domain. Such network monitoring systems require *archival storage* capabilities, in addition to the ability to query and examine live data. Besides capturing data at wire speeds, these systems also need to archive and index data at the same rates. Further, they need to efficiently retrieve archived data to answer *retrospective* queries.

Currently, there are two possible choices for architecting an archiving system for data streams. A relational database may be used to archive data, or a custom index may be created on top of a conventional file system.

The structure of captured information—a header for each packet consisting of a set of fields—naturally lends itself to a database view. This has led to systems such as GigaScope [6] and MIND [18], which implement a SQL interface for querying network monitoring data.

A monitoring system must receive new data at high rates: a single gigabit link can generate hundreds of thousands of packet headers per second and tens of Mbyte/s of data to archive, and a single monitoring system may record from multiple links. These rates have prevented the use of traditional database systems. MIND, which is based on a peer-to-peer index, extracts and stores only flow-level information, rather than raw packet headers. GigaScope is a stream database, and like other stream databases to date [20, 27, 1] supports continual queries on live streaming data; data archiving is not a design concern in these systems. GigaScope, for instance, can process continual queries on data from some of the highest-speed links in the Internet, but relies on external mechanisms to store results for later reference.

An alternative is to employ a general-purpose file system to store captured packet headers, typically as log files, and to construct a special-purpose index on these files to support efficient querying. A general-purpose file system, however, is not designed to exploit the particular characteristics of network monitoring applications, resulting in lower system throughput than may be feasible. Unix-like file systems, for instance, are typically optimized for writing small files and reading large ones sequentially, while network monitoring and querying writes very large files at high data rates,

while issuing small random reads. Due to the high data volume in these applications, and the need to bound worst-case performance in order to avoid data loss, it may be desirable to optimize the system for these access patterns instead of relying on a general-purpose file system.

Thus, the unique demands placed by high-volume stream archiving indicate that neither existing databases nor file systems are directly suited to handle their storage needs. This motivates the need for a new storage system that runs on commodity hardware and is specifically designed to handle the needs of high-volume stream archiving in the areas of disk performance, indexing, data aging, and query and index distribution.

Research Contributions: In this paper, we present *Hyperion*[1], a novel stream archiving system that is designed for storing and indexing high-volume packet header streams. Hyperion consists of three components: (i) *StreamFS*, a stream file system that is optimized for sequential immutable streaming writes, (ii) a multi-level index based on signature files, used in the past by text search engines, to sustain high update rates, and (iii) a distributed index layer which distributes coarse-grain summaries of locally archived data to other nodes, to enable distributed querying.

We have implemented Hyperion on commodity Linux servers, and have used our prototype to conduct a detailed experimental evaluation using real network traces. In our experiments, the worst-case StreamFS throughput for streaming writes is 80% of the mean disk speed, or almost 50% higher than for the best general-purpose Linux file system. In addition, StreamFS is shown to be able to handle a workload equivalent to streaming a million packet headers per second to disk while responding to simulateneous read requests. Our multi-level index, in turn, scales to data rates of over 200K packets/sec while at the same time providing interactive query responses, searching an hour of trace data in seconds. Finally, we examine the overhead of scaling a Hyperion system to tens of monitors, and demonstrate the benefits of our distributed storage system using a real-world example.

The rest of this paper is structured as follows. Section 2 and 3 present design challenges and guiding design principles. Sections 4, 5, and 6 present the design and implementation of Hyperion. We present experimental results in Section 7, related work in Section 8, and our conclusions in Section 9.

2 Design Challenges

The design of a high-volume archiving and indexing system for data streams must address several challenges:

Archive multiple, high-volume streams. A single heavily loaded gigabit link may easily produce monitor data at a rate of 20Mbyte/sec[2]; a single system may need to monitor several such links, and thus scale far beyond this rate. Merely storing this data as it arrives may be a problem, as a commodity hardware-based system of this scale must necessarily be based on disk storage; although the peak speed of such a system is sufficient, the worst-case speed is far lower than is required. In order to achieve the needed speeds, it is necessary to exploit the characteristics of modern disks and disk arrays as well as the sequential append-only nature of archival writes.

Maintain indices on archived data — The cost of exhaustive searches through archived data would be prohibitive, so an index is required to support most queries. Over time, updates to this index must be stored at wireline speed, as packets are captured and archived, and thus must support especially efficient updating. This high update rate (e.g. 220K pkts/sec in the example above) rules out many index structures; e.g. a B-tree index over the entire stream would require one or more disk operations per insertion. Unlike storage performance requirements, which must be met to avoid data loss, retrieval performance is not as critical; however, our goal is that it be efficient enough for interactive use. The target for a highly selective query, returning very few data records, is that it be able to search an hour of indexed data in 10 seconds.

Reclaim and re-use storage. Storage space is limited in comparison to arriving data, which is effectively infinite if the system runs long enough. This calls for a mechanism for reclaiming and reusing storage. Data aging policies that delete the oldest or the least-valuable data to free up space for new data are needed, and data must be removed from the index as it is aged out.

Coordinate between monitors. A typical monitoring system will comprise multiple monitoring nodes, each monitoring one or more network links. In network forensics, for instance, it is sometime necessary to query data archived at multiple nodes to trace events (e.g. a worm) as they move through a network. Such distributed querying requires some form of coordination between monitoring nodes, which involves a trade-off between distribution of data and queries. If too much data or index information is distributed across monitoring nodes, it may limit the overall scale of the system as the number of nodes increase; if queries must be flooded to all monitors, query performance will not scale.

Run on *commodity hardware.* The use of commodity processors and storage imposes limits on the processing and storage bandwidths available at each monitoring node, and the system must optimize its resource usage to scale to high data volumes.

[1] Hyperion, a Titan, is the Greek god of observation.

[2] 800Mbit/sec traffic, 450 byte packets, 90 bytes captured per packet

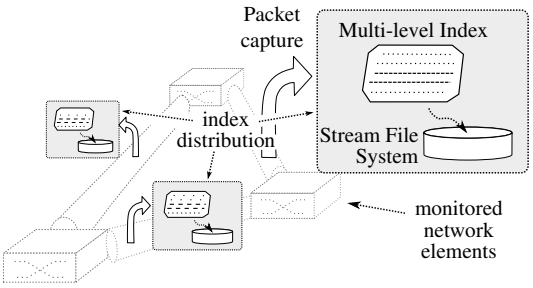


Figure 1: Components of the Hyperion network monitoring system.

3 Hyperion Design Principles

The challenges outlined in the previous section result in three guiding principles for our system design.

P1: *Support queries, not reads:* A general-purpose file system supports low-level operations such as reads and writes. However, the nature of monitoring applications dictates that data is typically accessed in the form of queries; in the case of Hyperion, for instance, these queries would be predicates identifying values for particular packet header fields such as source and destination address. Consequently, a stream archiving system should support data accesses at the level of queries, as opposed to raw reads on unstructured data. Efficient support for querying implies the need to maintain an index and one that is particularly suited for high update rates.

P2: *Exploit sequential, immutable writes:* Stream archiving results in continuous sequential writes to the underlying storage system; writes are typically immutable since data is not modified once archived. The system should employ data placement techniques that exploit these I/O characteristics to reduce disk seek overheads and improve system throughput.

P3: *Archive locally, summarize globally.* There is an inherent conflict between the need to scale, which favors local archiving and indexing to avoid network writes, and the need to avoid flooding to answer distributed queries, which favors sharing information across nodes. We "resolve" this conflict by advocating a design where data archiving and indexing is performed locally and a coarse-grain summary of the index is shared between nodes to support distributed querying without flooding.

Based on these principles, we have designed *Hyperion*, a stream archiving system that consists of three key components: (i) a *stream file system* that is highly optimized for high volume archiving and retrospective querying, (ii) a *multi-level index structure* that is designed for high update rates while retaining reasonable lookup performance, and (iii) a *distributed index* layer that distributes a coarse-grain summary of the local indices to enable distributed queries (see Figure 1) The following sections present the rationale for and design of these components in detail.

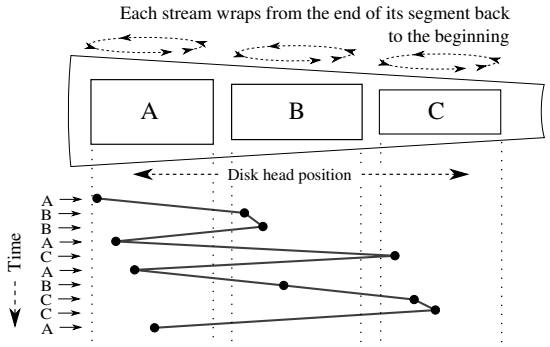


Figure 2: Write arrivals and disk accesses for single file per stream. Writes for streams A, B, and C are interleaved, causing most operations to be non-sequential.

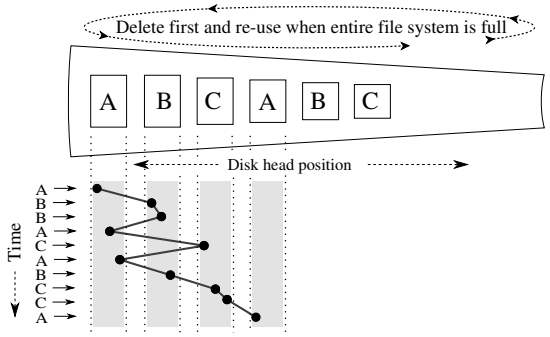


Figure 3: Logfile rotation. Data arrives for streams A, B, and C in an interleaved fashion, but is written to disk in a mostly sequential order.

4 Hyperion Stream File System

The requirements for the Hyperion storage system are: storage of multiple high-speed traffic streams without loss, re-use of storage on a full disk, and support for concurrent read activity without loss of write performance. The main barrier to meeting these requirements is the variability in performance of commodity disk and array storage; although storage systems with best-case throughput sufficient for this task are easily built, worst-case throughput can be three orders of magnitude worse.

In this section we first consider implementing this storage system on top of a general-purpose file system. After exploring the performance of several different conventional file systems on stream writes as generated by our application, we then describe StreamFS, an application-specific file system

for stream storage.[3]

In order to consider these issues, we first define a stream storage system in more detail. Unlike a general purpose file system which stores *files*, a stream storage system stores *streams*. These streams are:

- *Recycled:* when the storage system is full, writes of new data succeed, and old data is lost (i.e. removed or over-written in a circular buffer fashion). This is in contrast to a general-purpose file system, where new data is lost and old data is retained.
- *Immutable:* an application may append data to a stream, but does not modify previously written data.
- *Record-oriented:* attributes such as timestamps are associated with ranges in a stream, rather than the stream itself. Optionally, as in StreamFS, data may be written in records corresponding to these with boundaries which are preserved on retrieval.

This stream abstraction provides the features needed by Hyperion, while lacking other features (e.g. mutability) which are un-needed for our purposes.

4.1 Why Not a General Purpose Filesystem?

To store streams such as this on a general purpose file system, a mapping between streams and files is needed. A number of such mappings exist; we examine several of them below. In this consideration we ignore the use of buffering and RAID, which may be used to improve the performance of each of these methods but will not change their relative efficiency.

File-per-stream: A naïve stream storage implementation may be done by creating a single large file for each data stream. When storage is filled, the beginning of the file cannot be deleted if the most recent data (at the end of the file) is to be retained, so the beginning of the file is over-written with new data in circular buffer fashion. A simplified view of this implementation and the resulting access patterns may be seen in Figure 2. Performance of this method is poor, as with multiple simultaneous streams the disk head must seek back and forth between the write position on each file.

Log files: A better approach to storing streams is known as *logfile rotation*, where a new file is written until it reaches some maximum size, and then closed; the oldest files are then deleted to make room for new ones. Simplified operation may be seen in Figure 3, where files are allocated as single extents across the disk. This organization is much better at allocating storage flexibly, as allocation decisions may be revised dynamically when choosing which file to delete. As shown in the figure, fairly good locality will be maintained when first filling the volume; with continued use, however, consecutively created files and extents may be located far apart on disk, degrading throughput to that of the previous method.

[3] Specialized file systems for application classes (e.g. streaming media) have a poor history of acceptance. However, file systems specific to a single application, often implemented in user space, have in fact been used with success in a number of areas such as web proxies [25] and commercial databases such as Oracle. [21]

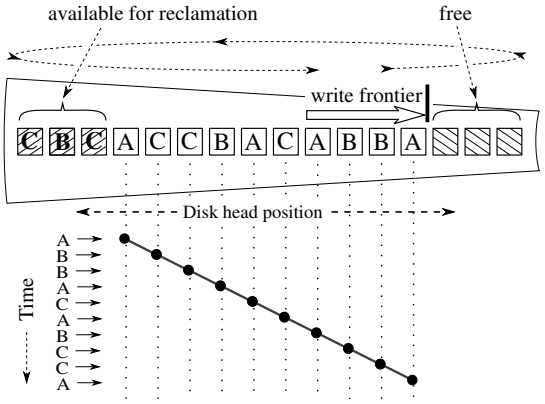


Figure 4: Log allocation - StreamFS, LFS. Data arrives in an interleaved fashion and is written to disk in that same order.

Log-Structured File System: The highest write throughput will be obtained if storage is allocated sequentially as data arrives, as illustrated in Figure 4. This is the method used by Log-structured File Systems (LFS) such as [22], and when logfile rotation is used on such a file system, interleaved writes to multiple streams will be allocated closely together on disk.

Although write allocation in log-structured file systems is straightforward, *cleaning*, or the garbage collecting of storage space after files are deleted, has however remained problematic [24, 32]. Cleaning in a general-purpose LFS must handle files of vastly different sizes and lifetimes, and all existing solutions involve copying data to avoid fragmentation. The FIFO-like Hyperion write sequence is a very poor fit for such general cleaning algorithms; in Section 7 our results indicate that it results in significant cleaning overhead.

4.2 StreamFS Storage Organization

The Hyperion stream file system, StreamFS, adopts the log structured write allocation of LFS; as seen in Figure 4, all writes take place at the *write frontier*, which advances as data is written. LFS requires a garbage collector, the *segment cleaner* to eliminate fragmentation which occurs as files are deleted; however, StreamFS does not require this, and *never copies data in normal operation.* This eliminates the primary drawback of log-structured file systems and is made possible by taking advantage of both the StreamFS storage reservation system and the properties of stream data.

The trivial way to do this would be to over-write all data as the write frontier advances, implicitly establishing a sin-

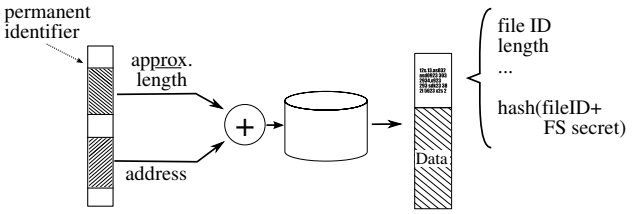


Figure 5: Random read operation.

gle age-based expiration policy for all streams. Such a policy would not address differences between streams in both rate and required data retention duration. Instead, StreamFS provides a storage *guarantee* to each stream; no records from a stream will be reclaimed or over-written while the stream size (i.e. retained records) is less than this guarantee. Conversely, if the size of a stream is larger than its guarantee, then only that amount of recent data is protected, and any older records are considered *surplus*.

The sum of guarantees is constrained to be less than the size of the storage system minus a fraction; we term the ratio of guarantees to volume size the volume *utilization*.[4] As with other file systems the utilization has a strong effect on performance.

StreamFS avoids a segment cleaner by writing data in small fixed-length blocks (default 1MB); each block stores data from a single stream. As the write frontier advances, it is only necessary to determine whether the next block is surplus. If so, it is simply overwritten, as seen in Figure 4, and if not it is skipped and will expire later; no data copying or cleaning is needed in either case. This provides a flexible storage allocation mechanism, allowing storage reservation as well as best-effort use of remaining storage. Simulation results have shown [8] this "cleaning" strategy to perform very well, with no virtually no throughput degradation for utilizations of 70% or less, and no more than a 15% loss in throughput at 90% utilization.

4.3 Read Addressing via Persistent Handles

Hyperion uses StreamFS to store packet data and indexes to that data, and then handles queries by searching those indexes and retrieving matching data. This necessitates a mechanism to identify a location in a data stream by some sort of pointer or *persistent handle* which may be stored in an index (e.g. across system restart), and then later used to retrieve the corresponding data. This value could be a byte offset from the start of the stream, with appropriate provisions (such as a 64-bit length) to guard against wrap-around. However, the pattern of reads generated by the index is highly non-sequential, and thus translating an offset into a disk location may require multiple accesses to on-disk tables. We therefore use a mechanism similar to a SYN cookie [2], where the information needed to retrieve a record (i.e. disk location and approximate length) is safely encoded and given to the application as a handle, providing both a persistent handle and a highly optimized random read mechanism.

[4]This definition varies slightly from that used for general-purpose file systems, as much of the "free" space beyond the volume utilization may hold accessible data.

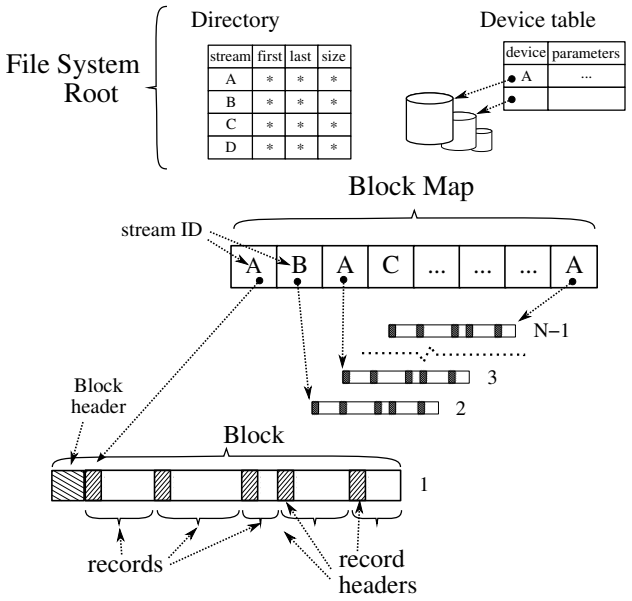


Figure 6: StreamFS metadata structures: record header for each record written by the application, block header for each fixed-length block, block map for every N (256) blocks, and one file system root.

Using application-provided information to directly access the disk raises issues of robustness and security. Although we may ignore security concerns in a single-application system, we still wish to ensure that in any case where a corrupted handle is passed to StreamFS, an error is flagged and no invalid data is returned. This is done by using a *self-certifying* record header, which guarantees that a handle is valid and that access is permitted. This header contains the ID of the stream to which it belongs and the permissions of that stream, the record length, and a hash of the header fields (and a *file system secret* if security is of concern) allowing invalid or forged handles to be detected. To retrieve a record by its persistent handle, StreamFS decodes the handle, applies some simple sanity checks, reads from the indicated address and length, and then verifies the record header hash. At this point a valid reader has been found; permission fields may then be checked and the record returned to the application if appropriate.

4.4 StreamFS Organization

The record header used for self-certifying reads is one of the StreamFS on-disk data structures illustrated in Figure 6. These structures and their fields and functions are as follows:

- *record:* Each variable-length record written by the application corresponds to an on-disk record and *record header*. The header contains validation fields described above, as well as timestamp and length fields.
- *block:* Multiple records from the same stream are combined in a single fixed-length block, by default 1Mbyte in length. The *block header* identifies the stream to which the block belongs, and record boundaries within the block.
- *block map:* Every N^{th} block (default 256) is used as a *block map*, indicating the associated stream and an in-stream sequence number for each of the preceding $N-1$ blocks. This map is used for write allocation, when it must be determined whether a block is part of a stream's guaranteed allocation and must be skipped, or whether it may be overwritten.
- *file system root:* The root holds the stream directory, metadata for each stream (head and tail pointers, size, parameters), and a description of the devices making up the file system.

4.5 Striping and Speed Balancing

Striping: StreamFS supports multiple devices directly; data is distributed across the devices in units of a single block, much as data is striped across a RAID-0 volume. The benefits of single disk write-optimizations in StreamFS extend to multi-disk systems as well. Since successive blocks (e.g., block i and $i+1$) map onto successive disks in a striped system, StreamFS can extract the benefits of I/O parallelism and increase overall system throughput. Further, in a d disk system, blocks i and $i+d$ will map to the same disk drive due to wrap-around. Consequently, under heavy load when there are more than d outstanding write requests, writes to the same disk will be written out sequentially, yielding similar benefits of sequential writes as in a single-disk system.

Speed balancing: Modern disk drives are *zoned* in order to maintain constant linear bit density; this results in disk throughput which can differ by a factor of 2 between the innermost and the outermost zones. If StreamFS were to write out data blocks sequentially from the outer to inner zones, then the system throughput would drop by a factor of two when the write frontier reached the inner zones. This worst-case throughput, rather than the mean throughput, would then determine the maximum loss-less data capture rate of the monitoring system.

StreamFS employs a balancing mechanism to ensure that system throughput remains roughly constant over time, despite variations across the disk platter. This is done by appropriately spreading the write traffic across the disk and results in an increase of approximately 30% in worst-case throughput. The disk is divided into three[5] zones R, S and T, and each zone into large, fixed-sized regions $(R_1,\ldots,R_n),(S_1,\ldots,S_n),(T_1,\ldots,T_n)$. These regions are then used in the following order: $(R_1,S_1,T_n,R_2,S_2,T_{n-1},\ldots,R_n,S_n,T_1)$; data is written sequentially to blocks within each region. The effective throughput is thus the average of throughput at 3 different points on the disk, and close to constant.

When accessing the disk sequentially, a zone-to-zone seek will be required after each region; the region size must thus be chosen to balance seek overhead with buffering requirements. For disks used in our experiments, a region size of 64MB results in one additional seek per second (degrading disk performance by less than 1%) at a buffering requirement of 16MB per device.

5 Indexing Archived Data

An Hyperion monitor needs to maintain an index which supports efficient retrospective queries, but also which may be created at high speed. Disk performance significantly limits the options available for the index; although minimizing random disk operations is a goal in any database, here multiple fields must be indexed in records arriving at a rate of over 100,000 per second per link. To scale to these rates, Hyperion relies on index structures that can be computed online and then *stored immutably*. Hyperion partitions a stream into intervals and computes one or more *signatures* [13] for each interval. The signatures can be tested for the presence of a record with a certain key in the associated data interval. Unlike a traditional B-tree-like structure, a signature only indicates whether a record matching a certain key is present; it does not indicate where in the interval that record is present. Thus, the entire interval needs to be retrieved and scanned for the result. However, if the key is not present, the entire interval can be skipped.

Signature indices are computed on a per-interval basis; no stream-wide index is maintained. This organization provides an index which may be streamed to disk along with the data—once all data within an interval have been examined (and streamed to storage), the signature itself can also be streamed out and a new signature computation begun for the next interval. This also solves the problem of removing keys from the

[5] The original choice of 3 regions was selected experimentally, but later work [8] demonstrates that this organization results in throughput variations of less than 4% across inner-to-outer track ratios up to 4:1.

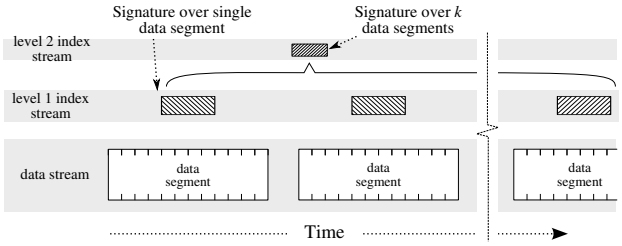


Figure 7: Hyperion multi-level signature index, showing two levels of signature index plus the associated data records.

index as they age out, as the signature associated with a data interval ages out as well.

5.1 Multi-level Signature Indices

Hyperion uses a multi-level *signature index*, the organization of which is shown in detail in Figure 7. A signature index, the most well-known of which is the Bloom Filter [3], creates a compact signature for one or more records, which may be tested to determine whether a particular key is present in the associated records. (This is in contrast to e.g. a B-tree or conventional hash table, where the structure provides a map from a key to the location where the corresponding record is stored.) To search for records containing a particular key, we first retrieve and test only the signatures; if any signature matches, then the corresponding records are retrieved and searched.

Signature functions are typically inexact, with some probability of a *false positive*, where the signature test indicates a match when there is none. This will be corrected when scanning the actual data records; the signature function cannot generate *false negatives*, however, as this will result in records being missed. Search efficiency for these structures is a trade-off between signature compactness, which reduces the amount of data retrieved when scanning the index, and false positive rate, which results in unnecessary data records being retrieved and then discarded.

The Hyperion index uses Bloom's hash function, where each key is hashed into a b-bit word, of which k bits are set to 1. The hash words for all keys in a set are logically ORed together, and the result is written as the signature for that set of records. To check for the presence of a particular key, the hash for that key h_0 is calculated and compared with the signature for the record, h_s; if any bit is set in h_0 but not set in h_s, then the value cannot be present in the corresponding data record. To calculate the false positive probability, we note that if the fraction of 1 bits in the signature for a set of records is r and the number of 1 bits in any individual hash is k, then the chance that a match could occur by chance is $1-(1-r)^k$; e.g. if the fraction of 1 bits is $\frac{1}{2}$, then the probability is 2^{-k}.

Multi-level index: Hyperion employs a two-level index [23], where a level-1 signature is computed for each data interval, and then a level-2 signature is computed over k data intervals. A search scans the level-2 signatures, and when a match is detected the corresponding k level-1 signatures are retrieved and tested; data blocks are retrieved and scanned only when a match is found in a level-1 signature.

When no match is found in the level-2 signature, k data segments may be skipped; this allows efficient search over large volumes of data. The level-2 signature will suffer from a higher false positive rate, as it is k times more concise than the level-1 signature; however, when a false positive occurs it is almost always detected after the retrieval of the level-1 signatures. In effect, the multi-level structure allows the compactness of the level-2 signature, with the accuracy of the level-1 signature.

Bit-sliced index: The description thus far assumes that signatures are streamed to disk as they are produced. When reading the index, however, a signature for an entire interval—thousands of bytes—must be retrieved from disk in order to examine perhaps a few dozen bits.

By buffering the top-level index and writing it in *bit-sliced* [12] fashion we are able to retrieve only those bits which need to be tested, thus possibly reducing the amount of data retrieved by orders of magnitude. This is done by aggregating N signatures, and then writing them out in N-bit *slices*, where the i^th slice is constructed by concatenating bit i from each of the N signatures. If N is large enough, then a slice containing N bits, bit i from each of N signatures, may be retrieved in a single disk operation. (although not implemented at present, this is a planned extension to our system.)

5.2 Handling Range and Rank Queries

Although signature indices are very efficient, like other hash indices they are useful for exact-match queries only. In particular, they do not efficiently handle certain query types, such as range and rank (top-K) queries, which are useful in network monitoring applications.

Hyperion can use certain other functions as indices, as well. Two of these are interval bitmaps [26] and aggregation functions.

Interval bitmaps are a form of what are known as bitmap indices [4]; the domain of a variable is divided into b intervals, and a b-bit signature is generated by setting the one bit corresponding to the interval containing the variable's value. These signatures may then be superimposed, giving a summary which indicates whether a value within a particular range is present in the set of summarized records.

Aggregate functions such as *min* and *max* may be used as indexes as well; in this case the aggregate is calculated over a segment of data and stored as the signature for that data.

Thus a query for $x < X_0$ can use aggregate minima to skip segments of data where no value will match, and a query for x with $COUNT(x) > N_0$ can make use of an index indicating the top K values [17] in each segment and their counts.

Of these, *min* and *max* have been implemented in the Hyperion system.

5.3 Distributed Index and Query

Our discussion thus far has focused on data archiving and indexing locally on each node. A typical network monitoring system will comprise multiple nodes and it is necessary to handle distributed queries without resorting to query flooding. Hyperion maintains a distributed index that provides an integrated view of data at all nodes, while storing the data itself and most index information locally on the node where it was generated. Local storage is emphasized for performance reasons, since local storage bandwidth is more economical than communication bandwidth; storage of archived data which may never be accessed is thus most efficiently done locally.

To create this distributed index, a coarse-grain summary of the data archived at each node is needed. The top level of the Hyperion multi-level index provides such a summary, and is shared by each node with the rest of the system. Since broadcasting the index to all other nodes would result in excessive traffic as the system scales, an *index node* is designated for each time interval $[t_1, t_2)$. All nodes send their top-level indices to the index node during this time-interval. Designating a different index node for successive time intervals results in a temporally-distributed index. Cross-node queries are first sent to an index node, which uses the coarse-grain index to determine the nodes containing matching data; the query is then forwarded to this subset for further processing.

6 Implementation

We have implemented a prototype of the Hyperion network monitoring system on Linux, running on commodity servers; it currently comprises 7000 lines of code.

The StreamFS implementation takes advantage of Linux asynchronous I/O and raw device access, and is implemented as a user-space library. In an additional simplification, the file system root resides in a file on the conventional file system, rather than on the device itself. These implementation choices impose several constraints: for instance, all access to a StreamFS volume must occur from the same process, and that process must run as root in order to access the storage hardware. These limitations have not been an issue for Hyperion to date; however, a kernel implementation of StreamFS is planned which will address them.

The index is a two-level signature index with linear scan of the top level (not bit-sliced) as described in Section 5.1. Multiple keys may be selected to be indexed on, where each key may be a single field or a composite key consisting of multiple fields. Signatures for each key are then superimposed in the same index stream via logical OR. Query planning is not yet implemented, and the query API requires that each key to be used in performing the query be explicitly identified.

Packet input is supported from trace files and via a special-purpose gigabit ethernet driver, sk98_fast, developed for nProbe at the University of Cambridge [19]. Support for Endace DAG hardware is planned, as well.

The Hyperion system is implemented as a set of modules which may be controlled from a scripting language (Python) through an interface implemented via the SWIG wrapper toolkit. This design allows the structure of the monitoring application to be changed flexibly, even at run time—as an example, a query is processed by instantiating data source and index search objects and connecting them. Communication between Hyperion systems is by RPC, which allows remote query execution or index distribution to be handled and controlled by the same mechanisms as configuration within a single system.

7 Experimental Results

In this section we present operational measurements of the Hyperion network monitor system. Tests of the stream file system component, StreamFS, measure its performance and compare it to that of solutions based on general-purpose file systems. Micro-benchmarks as well as off-line tests on real data are used to test the multi-level indexing system; the micro-benchmarks measure the scalability of the algorithm, while the trace-based tests characterize the search performance of our index on real data. Finally, system experiments characterize the performance of single Hyperion nodes, as well as demonstrating operation of a multi-node configuration.

7.1 Experimental Setup

Unless specified otherwise, tests were performed on the following system:

Hardware	2 × 2.4GHz P4 Xeon, 1 GB memory
Storage	4 × Fujitsu MAP3367NP Ultra320 SCSI, 10K RPM
OS	Linux 2.6.9 (CentOS 4.4)
Network	SysKonnect SK-9821 1000mbps

File system tests wrote dummy data (i.e. zeros), and ignored data from read operations. Most index tests, however, used actual trace data from the link between the University of Massachusetts and the commercial Internet [31]. These

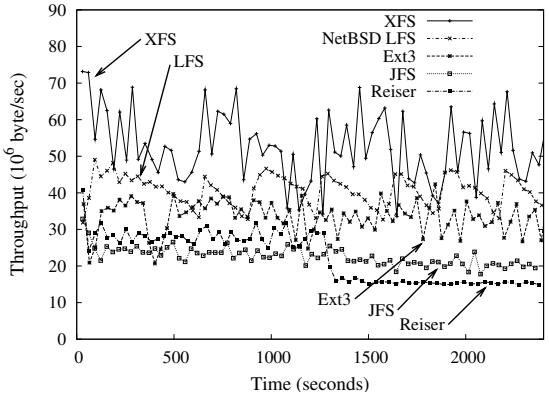


Figure 8: Streaming write-only throughput by file system. Each trace shows throughput for 30s intervals over the test run.

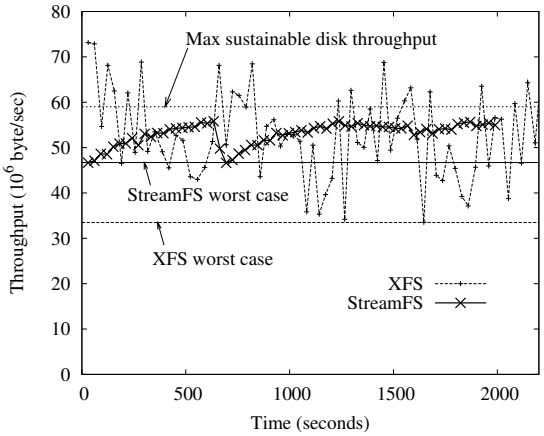


Figure 9: XFS vs. StreamFS write only throughput, showing 30 second and mean values. Straight lines indicate disk throughput at outer tracks (max) and inner tracks (min) for comparison. Note the substantial difference in worst-case throughput between the two file systems.

trace files were replayed on another system by combining the recorded headers (possibly after modification) with dummy data, and transmitting the resulting packets directly to the system under test.

7.2 File Systems and Databases

Our first tests establish a baseline for evaluating the performance of the Hyperion system. Since Hyperion is an application-specific database, built on top of an application-specific file system, we compare its performance with that of existing general-purpose versions of these components. In particular, we measure the speed of storing network traces on both a conventional relational database and on several conventional file systems.

Database Performance: We briefly present results of bulk loading packet header data on Postgres 7.4.13. Approximately 14.5M trace data records representing 158 seconds of sampled traffic were loaded using the COPY command; after loading, a query retrieved a unique row in the table. To speed loading, no index was created, and no attempt was made to test simultaneous insert and query performance. Mean results with 95% confidence intervals (8 repetitions) are as follows:

data set	158 seconds	14.5M records
table load	252 s ($\pm$7 s)	1.56 $\times$ real-time
query	50.7 s ($\pm$0.4 s)	0.32 $\times$ real-time

Postgres was not able to load the data, collected on a moderately loaded (40%) link, in real time. Query performance was much too slow for on-line use; although indexing would improve this, it would further degrade load performance.

Baseline File system measurements: These tests measure the performance of general-purpose file systems to serve as the basis for an application-specific stream database for Hyperion. In particular, we measure write-only performance with multiple streams, as well as the ability to deliver write performance guarantees in the presence of mixed read and write traffic. The file systems tested on Linux are ext3, ReiserFS, SGI's XFS [29], and IBM's JFS; in addition LFS was tested on NetBSD 3.1.

Preliminary tests using the naïve single file per stream strategy from Section 4.1 are omitted, as performance for all file systems was poor. Further tests used an implementation of the log file strategy from Section 4.1, with file size capped at 64MB. Tests were performed with 32 parallel streams of differing speeds, with random write arrivals of mean size 64KB. All results shown are for the steady state, after the disk has filled and data is being deleted to make room for new writes.

The clear leader in performance is XFS, as may be seen in Figure 8. It appears that XFS maintains high write performance for a large number of streams by buffering writes and writing large extents to each file – contiguous extents as large as 100MB were observed, and (as expected) the buffer cache expanded to use almost all of memory during the tests. (Sweeney *et al.* [29] describe how XFS defers block assignment until pages are flushed, allowing such large extents to be generated.)

LFS has the next best performance. We hypothesize that a key factor in its somewhat lower performance was the significant overhead of the segment cleaner. Although we were not able to directly measure I/O rates due to cleaning, the system CPU usage of the cleaner process was significant: approximately 25% of that used by the test program.

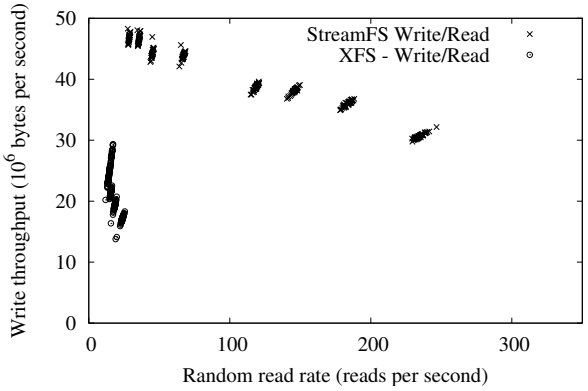


Figure 10: Scatter plot of StreamFS and XFS write and read performance.

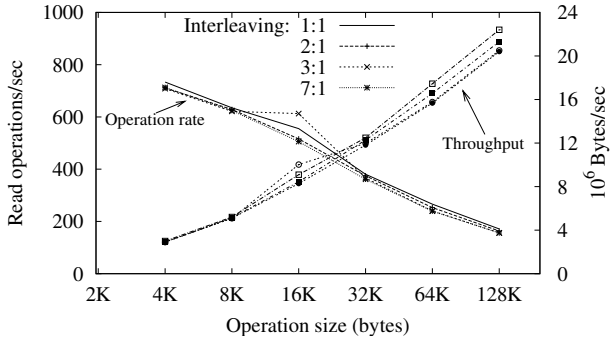


Figure 11: Streaming file system read performance. Throughput (rising) and operations/sec (falling) values are shown. The interleave factor refers to the number of streams interleaved on disk - i.e. for the 1:1 case the stream being fetched is the only one on disk; in the 1:7 case it is one of 7.

7.3 StreamFS Evaluation

In light of the above results, we evaluate the Hyperion file system StreamFS by comparing it to XFS.

StreamFS Write Performance: In Figure 9 we see representative traces for 32-stream write-only traffic for StreamFS and XFS. Although mean throughput for both file systems closely approaches the disk limit, XFS shows high variability even when averaged across 30 second intervals. Much of the XFS performance variability remains within the range of the disk minimum and maximum throughput, and is likely due to allocation of large extents at random positions across the disk. A number of 30s intervals, however, as well as two 60s intervals, fall considerably below the minimum disk throughput; we have not yet determined a cause for these drop-offs in performance. The consistent performance of StreamFS, in turn, gives it a worst-case speed close to the mean — almost 50% higher than the worst-case speed for XFS.

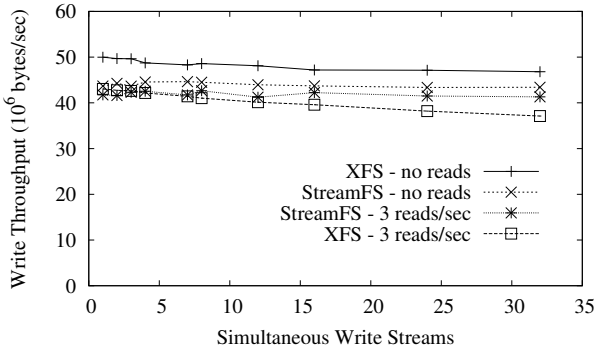


Figure 12: Sensitivity of performance to number of streams.

Read/Write Performance: Useful measurements of combined read/write performance require a model of read access patterns generated by the Hyperion monitor. In operation, on-line queries read the top-level index, and then, based on that index, read non-contiguous segments of the corresponding second-level index and data stream. This results in a read access pattern which is highly non-contiguous, although most seeks are relatively small. We model this non-contiguous access stream as random read requests of 4KB blocks in our measurements, with a fixed ratio of read to write requests in each experiment.

Figure 10 shows a scatter plot of XFS and StreamFS performance for varying read/write ratios. XFS read performance is poor, and write performance degrades precipitously when read traffic is added. This may be a side effect of organizing data in logfiles, as due to the large number of individual files, many read requests require opening a new file handle. It appears that these operations result in flushing some amount of pending work to disk; as evidence, the mean write extent length when reads are mixed with writes is a factor of 10 smaller than for the write-only case.

StreamFS Read Performance: We note that our prototype of StreamFS is not optimized for sequential read access; in particular, it does not include a read-ahead mechanism, causing some sequential operations to incur the latency of a full disk rotation. This may mask smaller-scale effects, which could come to dominate if the most significant overheads were to be removed.

With this caveat, we test single-stream read operation, to determine the effect of record size and stream interleaving on read performance. Each test writes one or more streams to an empty file system, so that the streams are interleaved on disk. We then retrieve the records of one of these streams in sequential order. Results may be seen in Figure 11, for record sizes ranging from 4KB to 128KB. Performance is dominated by per-record overhead, which we hypothesize is due to the lack of read-ahead mentioned above, and interleaved traffic has little effect on performance.

Sensitivity testing: These additional tests measured changes in performance with variations in the number of streams and of devices. Figure 12 shows the performance of StreamFS and XFS as the number of simultaneous streams varies from 1 to 32, for write-only and mixed read/write operations. XFS performance is seen to degrade slightly as the number of streams increases, and more so in the presence of read requests, while StreamFS throughput is relatively flat.

Multi-device tests with StreamFS on multiple devices and XFS over software RAID show almost identical speedup for both. Performance approximately doubles when going from 1 to 2 devices, and increases by a lesser amount with 3 devices as we get closer to the capacity of the 64-bit PCI bus on the test platform.

7.4 Index Evaluation

The Hyperion index must satisfy two competing criteria: it must be fast to calculate and store, yet provide efficient query operation. We test both of these criteria, using both generated and trace data.

Signature Index Computation: The speed of this computation was measured by repeatedly indexing sampled packet headers in a large ($\gg$ cache size) buffer. on a single CPU. Since the size of the computation input — i.e. the number of headers indexed – is variable, linear regression was used to determine the relationship between computation parameters and performance.

In more detail, for each packet header we create N indices, where each index i is created on F_i fields (e.g. source address) totaling B_i bytes. For index i, the B_i bytes are hashed into an M-bit value with k bits set, as described in Section 5.1. Regression results for the significant parameters are:

variable	coeff.	std. error	t-stat
index (N)	132 ns	6.53 ns	20.2
bytes hashed (B_i)	9.4 ns	1.98 ns	4.72
bit generated (k)	43.5 ns	2.1 ns	21.1

As an example, if 7 indices are computed per header, with a total of 40 bytes hashed and 60 signature bits generated, then index computation would take $7 \cdot 132 + 40 \cdot 9.4 + 60 \cdot 43.5 = 3910$ns or 3.9 μs/packet, for a peak processing rate of 256,000 headers per second on the test CPU, a 2.4GHz XeonAlthough not sufficient to index minimum-sized packets on a loaded gigabit link, this is certainly fast enough for the traffic we have measured to date. (e.g. 106,000 packets/sec on a link carrying approximately 400Mbit/sec.)

Signature Density: This next set of results examines the performance of the Hyperion index after it has been written to disk, during queries. In Figure 13 we measure the signature *density*, or the fraction of bits set to 1, when summarizing addresses from trace data. On the X axis we see the number of addresses summarized in a single hash block, while the different traces indicate the precision with which each address is summarized. From Bloom [3] we know that the efficiency of this index is maximized when the fraction of 1 (or 0) bits is 0.5; this line is shown for reference.

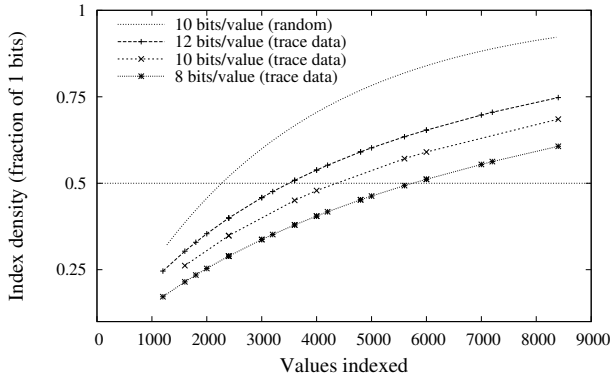


Figure 13: Signature density when indexing source/destination addresses, vs. number of addresses hashed into a single 4KB index block. Curves are for varying numbers of bits (k) per hashed value; top curve is for uniform random data, while others use sampled trace data.

From the graph we see that a 4KB signature can efficiently summarize between 3500 and 6000 addresses, depending on the parameter k and thus the false positive probability. The top line in the graph shows signature density when hashing uniformly-distributed random addresses with $k = 10$; it reaches 50% density after hashing only half as many addresses as the $k = 10$ line for real addresses. This is to be expected, due to repeated addresses in the real traces, and translates into higher index performance when operating on real, correlated data.

Query overhead: Since the index and data used by a query must be read from disk, we measure the overhead of a query by the factors which affect the speed of this operation: the volume of data retrieved and the number of disk seeks incurred. A 2-level index with 4K byte index blocks was tested, with data block size varying from 32KB to 96KB according to test parameters. The test indexed traces of 1 hour of traffic, comprising 26GB, $3.8 \cdot 10^8$ packets, and $2.5 \cdot 10^6$ unique addresses. To measure overhead of the index itself, rather than retrieval of result data, queries used were highly selective, returning only 1 or 2 packets.

Figures 14 and 15 show query overhead for the simple and bit-sliced indices, respectively. On the right of each graph, the volume of data retrieved is dominated by sub-index and data block retrieval due to false hits in the main index. To the left (visible only in Figure 14) is a domain where data retrieval is dominated by the main index itself. In each case, seek overhead decreases almost linearly, as it is dominated by skipping from block to block in the main index; the number of these blocks decreases as the packets per block increase. In each case there is a region which allows the 26GB of data

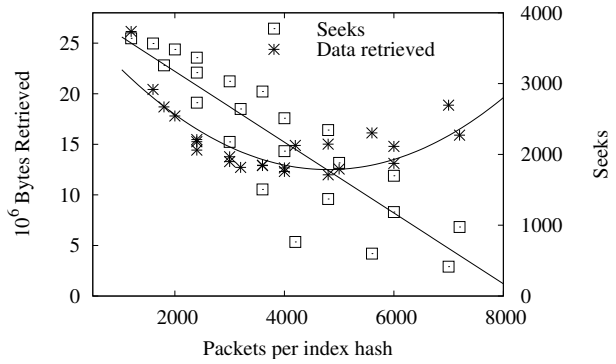


Figure 14: Single query overhead for summary index, with fitted lines. *N* packets (X axis) are summarized in a 4KB index, and then at more detail in several (3-6) 4KB sub-indices. Total read volume (index, sub-index, and data) and number of disk seeks are shown.

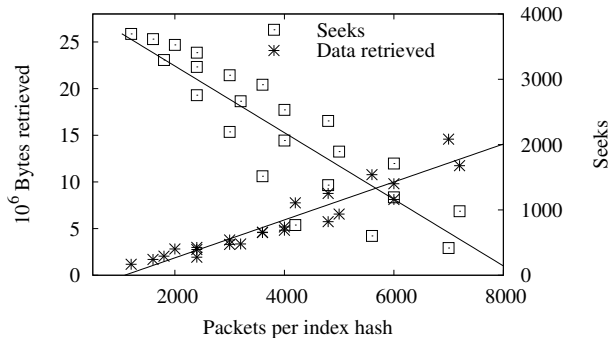


Figure 15: Single query overhead for bit-sliced index. Identical to Figure 14, except that each index was split into 1024 32-bit slices, with slices from 1024 indices stored consecutively by slice number.

to be scanned at the cost of 10-15 MB of data retrieved, and 1000-2000 disk seeks.

7.5 Prototype Evaluation

After presenting test results for the components of the Hyperion network monitor, we now turn to tests of its performance as a whole. Our implementation uses StreamFS as described above, and a 2-level index without bit-slicing. The following tests for performance, functionality, and scalability are presented below:

- performance tests: tests on single monitoring node which assess the system's ability to gather and index data at network speed, while simultaneously processing user queries.

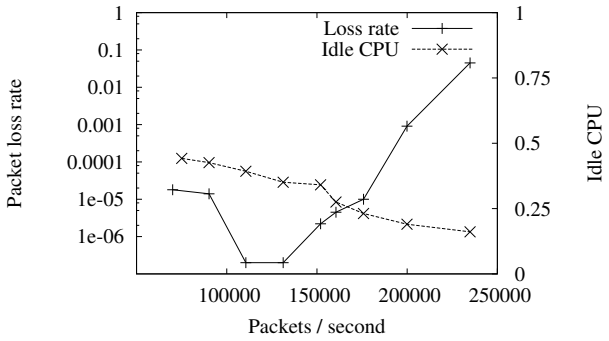


Figure 16: Packet arrival and loss rates.

- functionality testing: three monitoring nodes are used to trace the origin of simulated malicious traffic within real network data.

- scalability testing: a system of twenty monitoring nodes is used to gather and index trace data, to measure the overhead of the index update protocol.

Monitoring and Query Performance: These tests were performed on the primary test system, but with a single data disk. Traffic from our gateway link traces was replayed over a gigabit cable to the test system. First the database was loaded by monitoring an hour's worth of sampled data — $4 \cdot 10^8$ packets, or 26GB of packet header data. After this, packets were transmitted to the system under test with inter-arrival times from the original trace, but scaled so as to vary the mean arrival rate, with simultaneous queries. We compute packet loss by comparing the transmit count on the test system with the receive count on Hyperion, and measure CPU usage.

Figure 16 shows packet loss and free CPU time remaining as the packet arrival rate is varied.[6] Although loss rate is shown on a logarithmic scale, the lowest points represent zero packets lost out of 30 or 40 million received. The results show that Hyperion was able to receive and index over 200,000 packets per second with negligible packet loss. In addition, the primary resource limitation appears to be CPU power, indicating that it may be possible to achieve significantly higher performance as CPU speeds scale.

System Scalability: In this test a cluster of 20 monitors recorded trace information from files, rather than from the network itself. Tcpdump was used to monitor RPC traffic between the Hyperion processes on the nodes, and packet and byte counts were measured. Each of the 20 systems monitored a simulated link with traffic of approximately 110K pkts/sec, with a total bit rate per link of over 400 Mbit/sec. Level 2 indices were streamed to a *cluster head*, a position

[6] Idle time is reported as the fraction of 2 CPUs which is available. Packet capture currently uses 100% of one CPU; future work should reduce this.

which rotates over time to share the load evenly. A third level of index was used in this test; each cluster head would store the indices received, and then aggregate them with its own level 2 index and forward the resulting stream to the current *network head*. Results are as follows:

Hyperion communication overhead overhead in K bytes/sec

	leaf	cluster head	net head
transmit	102 KB/s	102 KB/s	
receive		408 KB/s	510 KB/s

From these results we see that scaling to dozens of nodes would involve maximum traffic volumes between Hyperion nodes in the range of 4Mbit/s; this would not be excessive in many environments, such as within a campus.

Forensic Query Case Study: This experiment examines a simulated 2-stage network attack, based on real-world examples. Packet traces were generated for the attack, and then combined with sampled trace data to create traffic traces for the 3 monitors in this simulation, located at the campus gateway, the path to target A, and the path to B respectively.

Abbreviated versions of the queries for this search are as follows:

1 `SELECT p WHERE src=B,dport=SMTP,t`$\leq T_{now}$
Search outbound spam traffic from B, locating start time T_0.

2 `SELECT p WHERE dst=B,t`$\in T_0 \cdots T_0 + \Delta$
Search traffic into B during single spam transmission to find control connection.

3 `SELECT p WHERE dst=B,t`$\in T_0 - \Delta \cdots T_0$
Find inbound traffic to B in the period before T_0.

4 `SELECT p WHERE s/d/p=Z/B/`P_x`, syn, t`$\leq T_0$
Search for SYN packet on this connection at time T_{-1}.

5 `SELECT p WHERE dst=B,t`$\in T_{-1} - \Delta \cdots T_{-1}$
Search for the attack which infected B, finding connection from A at T_2.

6 `SELECT p WHERE dst=A,t`$\in T_{-2} - \Delta \cdots T_{-2} + \Delta$
Find external traffic to A during the A-B connection to locate attacker X.

7 `SELECT p WHERE src=X,syn,t`$\leq T_{-2}$
Find all incoming connections from X

We note that additional steps beyond the Hyperion queries themselves are needed to trace the attack; for instance, in step 3 the search results are examined for patterns of known exploits, and the results from steps 5 and 6 must be joined in order to locate X. Performance of this search (in particular, steps 1, 4, and 7) depends on the duration of data to be searched, which depends in turn on how quickly the attack is discovered. In our test, Hyperion was able to use its index to handle queries over several hours of trace data in seconds. In actual usage it may be necessary to search several days or more of trace data; in this case the long-running queries would require minutes to complete, but would still be effective as a real-time forensic tool.

8 Related Work

Like Hyperion, both PIER [15] and MIND [18] are able to query past network monitoring data across a distributed network. Both of these systems, however, are based on DHT structures which are unable to sustain the high insertion rates required for indexing packet-level trace data, and can only index lower-speed sources such as flow-level information. The Gigascope [6] network monitoring system is able to process full-speed network monitoring streams, and provides a SQL-based query language. These queries, however, may only be applied over incoming data streams; there is no mechanism in GigaScope itself for *retrospective queries*, or queries over past data. StreamBase [28] is a general-purpose stream database, which like GigaScope is able to handle streaming queries at very high rates. In addition, like Hyperion, StreamBase includes support for persistent tables for retrospective queries, but these tables are conventional hash or B-tree-indexed tables, and are subject to the same performance limitations.

A number of systems such as the Endace DAG [10] have been developed for wire-speed collection and storage of packet monitoring data, but these systems are designed for off-line analysis of data, and provide no mechanisms for indexing or even querying the data. CoMo [16] addresses high-speed monitoring and storage, with provisions for both streaming and retrospective queries. Although it has a storage component, however, it does not include any mechanism for indexing, limiting its usefulness for querying large monitor traces.

The log structure of StreamFS bears a debt to the original Berkeley log-structured file system [22], as well as the WORM file system from the V System [5]. There has been much work in the past on supporting streaming reads and writes for multimedia file systems (e.g. [30]); however, the sequential-write random-read nature of our problem results in significantly different solutions.

There is an extensive body of literature on Bloom filters and the signature file index used in this system; two useful articles are a survey by Faloutsos [11] and the original article by Bloom [3]. Multi-level and multi-resolution indices have been described both in this body of literature (e.g. [23]) as well as in other areas such as sensor networks [14, 7].

9 Conclusions

In this paper, we argued that neither general-purpose file systems nor common database index structures meet the unique needs imposed by high-volume stream archiving and indexing. We proposed Hyperion, a novel stream archiving system that consists of StreamFS, a write-optimized stream file system, and a multi-level signature index that handles high up-

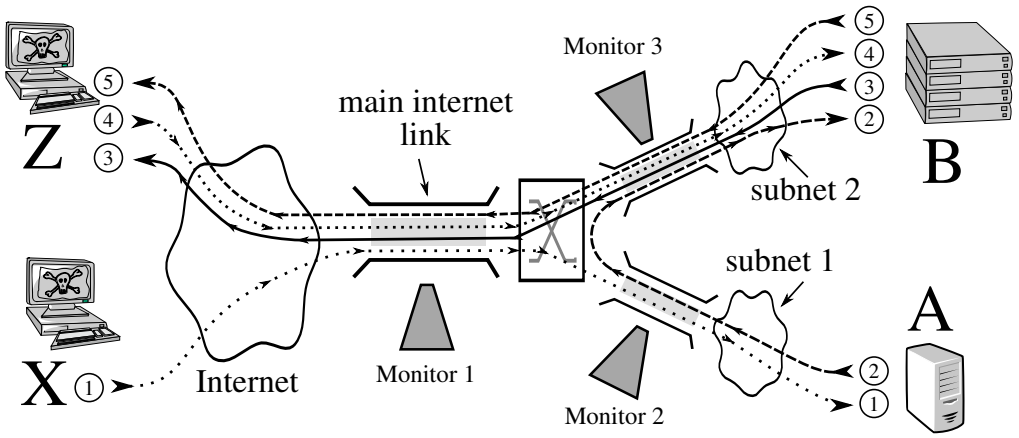


Figure 17: Forensic query example: (1) attacker X compromises system A, providing an inside platform to (2) attack system B, installing a bot. The bot (3) contacts system Z via IRC, (4) later receives a command, and begins (5) relaying spam traffic.

date rates and enables distributed querying. Our prototype evaluation has shown that (i) StreamFS can scale to write loads of over a million packets per second, (ii) the index can support over 200K packet/s while providing good query performance for interactive use, and (iii) our system can scale to tens of monitors. As part of future work, we plan to enhance the aging policies in StreamFS and implement other index structures to support richer querying.

10 Acknowledgments

This work was funded in part by NFS grants EEC-0313747, CNS-0323597, and CNS-0325868. The authors would also like to thank Deepak Ganesan and the anonymous reviewers for contributions to this paper.

References

[1] ABADI, D. J., AHMAD, Y., BALAZINSKA, M., ÇETINTEMEL, U., CHERNIACK, M., HWANG, J.-H., LINDNER, W., MASKEY, A. S., RASIN, A., RYVKINA, E., TATBUL, N., XING, Y., AND ZDONI, S. The Design of the Borealis Stream Processing Engine. In *Proc. Conf. on Innovative Data Systems Research* (Jan. 2005).

[2] BERNSTEIN, D. Syn cookies. Published at `http://cr.yp.to/syncookies.html`.

[3] BLOOM, B. Space/time tradeoffs in hash coding with allowable errors. *Communications of the ACM 13*, 7 (July 1970), 422–426.

[4] CHAN, C.-Y., AND IOANNIDIS, Y. E. Bitmap index design and evaluation. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (June 1998), pp. 355–366.

[5] CHERITON, D. The V Distributed System. *Communications of the ACM 31*, 3 (Mar. 1988), 314–333.

[6] CRANOR, C., JOHNSON, T., SPATASCHEK, O., AND SHKAPENYUK, V. Gigascope: a stream database for network applications. In *Proc. 2003 ACM SIGMOD Intl. Conf. on Management of data* (2003), pp. 647–651.

[7] DESNOYERS, P., GANESAN, D., AND SHENOY, P. TSAR: a two tier sensor storage architecture using interval skip graphs. In *Proc. 3rd Intl. Conf on Embedded networked sensor systems (SenSys05)* (2005), pp. 39–50.

[8] DESNOYERS, P., AND SHENOY, P. Hyperion: High Volume Stream Archival for Retrospective Querying. Tech. Rep. TR46-06, University of Massachusetts, Sept. 2006.

[9] DREGER, H., FELDMANN, A., PAXSON, V., AND SOMMER, R. Operational experiences with high-volume network intrusion detection. In *Proc. 11th ACM Conf. on Computer and communications security (CCS '04)* (2004), pp. 2–11.

[10] ENDACE INC. Endace DAG4.3GE Network Monitoring Card. available at `http://www.endace.com`, 2006.

[11] FALOUTSOS, C. Signature-Based Text Retrieval Methods: A Survey. *IEEE Data Engineering Bulletin 13*, 1 (1990), 25–32.

[12] FALOUTSOS, C., AND CHAN, R. Fast Text Access Methods for Optical and Large Magnetic Disks: Designs and Performance Comparison. In *VLDB '88: Proc. 14th Intl. Conf. on Very Large Data Bases* (1988), pp. 280–293.

[13] FALOUTSOS, C., AND CHRISTODOULAKIS, S. Signature files: an access method for documents and its analytical performance evaluation. *ACM Trans. Inf. Syst. 2*, 4 (1984), 267–288.

[14] GANESAN, D., ESTRIN, D., AND HEIDEMANN, J. Dimensions: Distributed multi-resolution storage and mining of networked sensors. *ACM Computer Communication Review 33*, 1 (January 2003), 143–148.

[15] HUEBSCH, R., CHUN, B., HELLERSTEIN, J. M., LOO, B. T., MANIATIS, P., ROSCOE, T., SHENKER, S., STOICA, I., AND YUMEREFENDI, A. R. The Architecture of PIER: an Internet-Scale Query Processor. In *Proc. Conf. on Innovative Data Systems Research (CIDR)* (Jan. 2005).

[16] IANNACCONE, G., DIOT, C., MCAULEY, D., MOORE, A., PRATT, I., AND RIZZO, L. The CoMo White Paper. Tech. Rep. IRC-TR-04-17, Intel Research, Sept. 2004.

[17] KARP, R. M., SHENKER, S., AND PAPADIMITRIOU, C. H. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst. 28*, 1 (2003), 51–55.

[18] LI, X., BIAN, F., ZHANG, H., DIOT, C., GOVINDAN, R., HONG, W., AND IANNACCONE, G. Advanced Indexing Techniques for Wide-Area Network Monitoring. In *Proc. 1st IEEE Intl. Workshop on Networking Meets Databases (NetDB)* (2005).

[19] MOORE, A., HALL, J., KREIBICH, C., HARRIS, E., AND PRATT, I. Architecture of a Network Monitor. *Passive & Active Measurement Workshop 2003 (PAM2003)* (2003).

[20] MOTWANI, R., WIDOM, J., ARASU, A., BABCOCK, B., BABU, S., DATAR, M., MAKU, G., OLSTON, C., ROSENSTEIN, J., AND VARMA, R. Query processing, approximation, and resource management in a data stream management system. In *Proc. Conf. on Innovative Data Systems Research (CIDR)* (2003).

[21] NDIAYE, B., NIE, X., PATHAK, U., AND SUSAIRAJ, M. A Quantitative Comparison between Raw Devices and File Systems for implementing Oracle Databases. `http://www.oracle.com/technology/deploy/performance/WhitePapers.html`, Apr. 2004.

[22] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10*, 1 (1992), 26–52.

[23] SACKS-DAVIS, R., AND RAMAMOHANARAO, K. A two level superimposed coding scheme for partial match retrieval. *Information Systems 8*, 4 (1983), 273–289.

[24] SELTZER, M. I., SMITH, K. A., BALAKRISHNAN, H., CHANG, J., MCMAINS, S., AND PADMANABHAN, V. N. File System Logging versus Clustering: A Performance Comparison. In *USENIX Winter Technical Conference* (1995), pp. 249–264.

[25] SHRIVER, E., GABBER, E., HUANG, L., AND STEIN, C. A. Storage management for web proxies. In *Proc. USENIX Annual Technical Conference* (2001), pp. 203–216.

[26] STOCKINGER, K. Design and Implementation of Bitmap Indices for Scientific Data. In *Intl. Database Engineering & Applications Symposium* (July 2001).

[27] STONEBRAKER, M., CETINTEMEL, U., AND ZDONIK, S. The 8 requirements of real-time stream processing. *SIGMOD Record 34*, 4 (2005), 42–47.

[28] STREAMBASE, I. StreamBase: Real-Time, Low Latency Data Processing with a Stream Processing Engine. from `http://www.streambase.com`, 2006.

[29] SWEENEY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., AND PECK, G. Scalability in the XFS File System. In *USENIX Annual Technical Conference* (Jan. 1996).

[30] TOBAGI, F. A., PANG, J., BAIRD, R., AND GANG, M. Streaming RAID: a disk array management system for video files. In *Proc. 1st ACM Intl. Conf. on Multimedia* (1993), pp. 393–400.

[31] Umass trace repository. Available at `http://traces.cs.umass.edu`.

[32] WANG, W., ZHAO, Y., AND BUNT, R. HyLog: A High Performance Approach to Managing Disk Layout. In *Proc. 3rd USENIX Conf. on File and Storage Technologies (FAST)* (2004), pp. 145–158.

Appendix A Performance Analysis

The speed balancing optimization for zoned disks, the storage allocation policy and the signature file index were measured and evaluated analytically. Results are given in this appendix.

A.1 Disk Speed Balancing

The speed balancing mechanism described in Section 4.5 is designed to even out the variation in speed due to zoned bit recording,[7] which results in a linear relationship between track number and both throughput and track capacity. This variation in track capacity, in turn, results in a non-linear relationship between block number on a disk and throughtput.

The relation between disk position (in blocks) and track number, thus bit rate, is obtained by integrating capacity per track, giving a transfer rate proportional to $k_4 - t(a)$. This may be seen in Figure 18(a), where the measured transfer rate of a drive is compared with the curve determined from this equation and disk parameters.

Our balancing method results in a logical volume where if the raw transfer rate at an address a is $t(a)$, then the translated mean transfer rate is $\frac{1}{3}\left(t(a/3)+t(\frac{a}{3}+\frac{N}{3})+t(N-\frac{a}{3})\right)$. This is shown plotted in Figure 18(a) as well, and may be seen to result in near constant average throughput across the logical volume.

Measured performance of the disk speed balancing mechanism is shown in Figure 18(a); as may be seen, throughput is almost constant, resulting in a 25% increase in worst-case throughput.

Result 1 *Let transfer rate r decrease linearly from $1+d$ to d as track position t goes from 0 to 1, and logical block address a range from 0 to 1. Then:*

- *Transfer rate at address a is:* $r(a) = \sqrt{(1-a)d^2 + 2(1-a)d + 1}$
- *After the balancing algorithm in Section 4.5, the mean transfer rate at translated address a' is:* $\frac{1}{3}r\left(\frac{a'}{3}\right) + \frac{1}{3}r\left(\frac{1+a'}{3}\right) + \frac{1}{3}r\left(1-\frac{a'}{3}\right)$
- *If the ratio of outer to inner transfer rate is less than 4:1, the balanced throughput will vary from the mean by a ratio of no more than 1.038:1.*

Derivation of Result 1:

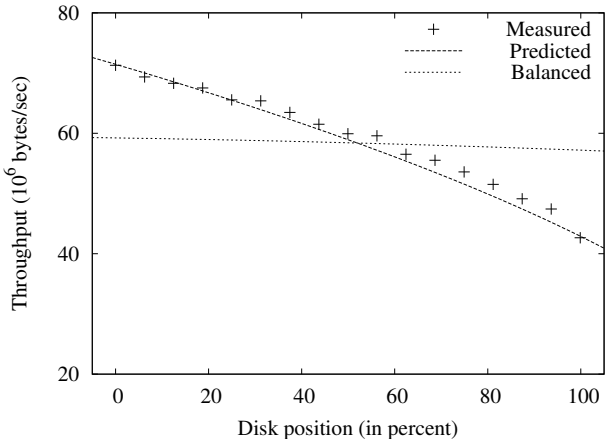
- The capacity of track t is equal to $\alpha(d+(1-t))$ where α is a proportionality constant.
- Integrating and setting α so that total capacity is 1, the capacity of tracks $0 \ldots t$ is thus

$$\frac{1}{d+1/2}\left((d+1)t - t^2/2\right)$$

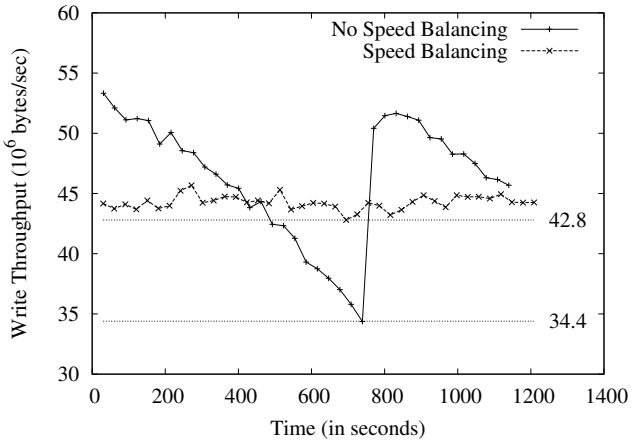
- Given a block address a, the corresponding track number $t(a)$ is thus

$$t(a) = (d+1) - \sqrt{(d+1)^2 - (2d+1)a}$$

[7] Zoned bit recording (ZBR) maintains an approximately constant linear bit density across tracks, by storing more bits on the outer tracks. The disk rotates at constant speed, and thus the bits on the outer tracks pass under the head more quickly than those on inner tracks.



(a) Disk speed vs. location



(b) Measured speed with and without balancing

Figure 18: Disk speed balancing (a) shows measured and predicted disk throughput vs. position, and predicted balanced performance; in (b) we see the effect of the balancing algorithm on actual stream file system throughput.

and transfer rate

$$\begin{aligned} r(a) &= [d+(1-t(a))] \\ &= \sqrt{(d+1)^2-(2d+1)a} \end{aligned}$$

- The balancing mechanism translates an address a in the range $0\cdots N-1$ to a' as follows:

$$a' = \begin{cases} a=0 \pmod 3: & a/3 \\ a=1 \pmod 3: & N/3+(a-1)/3 \\ a=2 \pmod 3: & N-(a-2)/3 \end{cases}$$

and we let the balanced throughput at a denote the mean of throughputs for $a-1$, a, and $a+1$:

$$r_{bal}(a) = \frac{1}{3}r(a/3)+\frac{1}{3}r((a+1)/3)+\frac{1}{3}r((3-a)/3)$$

- Substituting $n=\frac{d+1}{d}$, the ratio of outer-track to inner-track circumference, the maximum ratio of max to min balanced throughput for values of n in $1+\varepsilon\ldots 4$ is found at approximately $n=1.9567$, with $r_{max}(n)/r_{min}(n) = 1.0377$.

■

A.2 Storage Fragmentation

Result 2 *Let the total size of storage be S and the total arrival rate R; each stream i arrives at rate $r_i R$, and has a committed allocation of $c_i \cdot S$. Let $c=\sum_i c_i$; i.e. the total fraction of storage which is reserved, and require that (i) $c_i \le \frac{r_i}{k}$ and (ii) $c_i \ge \frac{c\cdot r_i}{\ell}$; i.e. the ratio of committed allocation to rate for each flow is bounded on both sides. Then:*

- *The probability p_{skip} that the next block at the write frontier will be skipped is less than*

$$\frac{(k-1)(l-1)}{\left(\frac{k}{c}-1\right)\ell+1}$$

- *If the reservation rate, c, is $\frac{1}{2}$, and $k=2$ and $\ell=2$, then: $p_{skip} < \frac{1}{7}$ or 0.1428*
- *For a block size of 1MB, a seek time of 4ms[8] and a disk throughput of 60MB/s, this results in a throughput loss of approximately 4%.*

Derivation of Result 2:

- Streams may be divided into "fast" and "slow" streams: a stream is fast if $r_i > c_i$, and slow if $r_i < c_i$. Note that blocks written by a fast stream will become available before the write frontier wraps around to them, while blocks written by a slow stream may need to be skipped when the write frontier reaches them.
- Without loss of generality we may assume that there is a single slow stream. In the worst case both (i) and (ii) above are equalities: $c_{k\neq i} = \frac{cr_k}{\ell}$ (thus $c_i = \frac{c(\ell-1)}{\ell}$).
- Let the fraction of blocks that must be skipped be α. This is equal to c_i minus the number of blocks written by stream i during one rotation of the write frontier, in which $(1-\alpha)S$ blocks are written:

$$\alpha = c_i - (1-\alpha)r_i$$

[8] For short seeks on a 10K RPM disk, 3ms of mean rotational latency plus 1ms of settling time.

- We note that $p_{skip} = \alpha$, and the results above follow directly.

■

A.3 Signature File Index

Result 3 *Given a two-level index, where records are of size S, index 1 consists of signatures of b_1 bits for every N_1 records, with k_1 bits per key, and index 2 signatures are b_2 bits for every N_2 (=$K \cdot N_1$) records, with k_2 bits set per key:*

- *The amount of data retrieved from index 1 plus the record data retrieved due to false positives is minimized for (approximately) $k_1 = \log_2(S) - 1$, and the cost (i.e. data retrieved) per N_1 records queried is $\frac{N\log_2(S)}{\ln(2)} + \frac{N_1}{2}$*
- *The data retrieved to scan index 2, including reads of index 1 and data records due to false positives, is minimized at approximately $k_2 = \log_2(\ln(S))$, and the amount of data retrieved to query over N_2 records is:*

$$\frac{\log_2(\ln(S))N_2}{ln(2)} + \frac{K}{\ln(S)}\frac{N\log_2(S)}{\ln(2)} + \frac{N_1}{2}$$

To analyze the cost of the index, we measure the total amount of data read from storage if no matching terms are found. Note that this ignores the per-operation cost of storage, which although significant is dependent on hardware details.

Derivation of Result 3:

- Bloom [3] derives the result that the smallest signature index for a given false positive probability is sized so that the probability of any bit in the index being set to 1 is $\frac{1}{2}$, and thus the false positive probability p_{false} is 2^{-k_1}. This will be the case when $\ln(2)b_1$ (not necessarily unique) bits have been set, or $k_1N_1 = \ln(2)b_1$.
- The cost per block search is the cost of reading the index ($\frac{1}{\ln(2)}k_1N_1$) plus 2^{-k_1} times the cost of reading the entire block, or

$$C_1 = \frac{1}{\ln(2)}k_1N_1 + 2^{-k_1}N_1S$$

  which is minimized at approximately $k_1 = \log_2(S) - 1$, for a total cost of $\frac{N_1\log_2(S)}{\ln(2)} + \frac{N_1}{2}$.
- If we group $M = N_2/N_1$ level one index blocks together, and generate a level two index with k_2 bits set per record, then the size of the level two index is $\frac{1}{\ln(2)}k_2N_2$ and the cost is

$$C_2 = \frac{1}{\ln(2)}k_2N_2 + 2^{-k_2}MC_1$$

which is minimized at $k_2 = \log_2(C) - \log_2(N_1) - 1$ or about $k_2 = \log_2(\ln(S))$. Substituting back we obtain the result above.

■