Hancock: A Language For Analyzing Transactional Data Streams

CORINNA CORTES, KATHLEEN FISHER, DARYL PREGIBON,
ANNE ROGERS, and FREDERICK SMITH
AT&T Labs

Massive transaction streams present a number of opportunities for data mining techniques. The transactions in such streams might represent calls on a telephone network, commercial credit card purchases, stock market trades, or HTTP requests to a web server. While historically such data have been collected for billing or security purposes, they are now being used to discover how the transactors, for example, credit-card numbers or IP addresses, use the associated services.

Over the past 5 years, we have computed evolving profiles (called *signatures*) of transactors in several very large data streams. The signature for each transactor captures the salient features of his or her behavior through time. Programs for processing signatures must be highly optimized because of the size of the data stream (several gigabytes per day) and the number of signatures to maintain (hundreds of millions). Originally, we wrote such programs directly in C, but because these programs often sacrificed readability for performance, they were difficult to verify and maintain.

Hancock is a domain-specific language we created to express computationally efficient signature programs cleanly. In this paper, we describe the obstacles to computing signatures from massive streams and explain how Hancock addresses these problems. For expository purposes, we present Hancock using a running example from the telecommunications industry; however, the language itself is general and applies equally well to other data sources.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Classifications—*Specialized application languages*; H.2.8 [**Database Management**]: Database Applications—*Data mining*; I.5.1 [**Pattern Recognition**]: Models—*Statistical*

General Terms: Languages, Performance

Additional Key Words and Phrases: Domain-specific languages, data mining, statistical models

1. INTRODUCTION

A transactional data stream is a sequence of records that log interactions between entities. For example, a stream of stock market transactions consists of buy or sell orders for particular securities from individual investors. A stream of credit card transactions contains records of purchases by consumers from

Authors' addresses: C. Cortes and D. Pregibon, currently at Google Labs, 1440 Broadway, New York, NY 10018; email: (corinna,daryl)@google.com; K, Fisher, AT&T Labs, Shannon Laboratory, 180 Park Avenue, Florham Park, NJ 07932; email: kfisher@research.att.com; A. Rogers, currently at University of Chicago, 1100 E. 58 St., Chicago, IL 60637; email: amr@cs.uchicago.edu; F. Smith, currently at The Mathworks, 3 Apple Hill Drive, Natick, MA 01760; email: fsmith@mathworks.com.

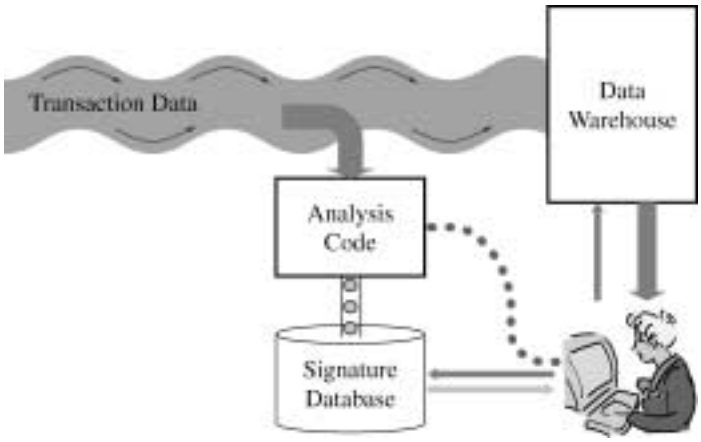

Fig. 1. Typical use of a fraud signature.

merchants. A stream of call-detail transactions contains records of telephone calls from an originating phone number to a dialed phone number. If such transactional data simply flow into a data warehouse, discovering the entities that are interesting can be difficult because of the sheer volume of data. Where should data analysts focus their attention?

One solution to this problem, which AT&T has used very effectively [Cortes and Pregibon 1998; Cortes et al. 2000], is to tap the transactional stream as it flows into the data warehouse and use the resulting information to build and maintain *signatures*, which are small profiles of the entities in the stream [Burge and Shawe-Taylor 1996; Cortes and Pregibon 1999; Denning 1987; Fawcett and Provost 1997]. Typically, the stream is processed in batches, the length of which often has some semantic meaning, for example, a day or an hour. As each batch is processed, the signatures for the entities mentioned in the transactions are updated. Analysts design these signatures to capture the essence of the entities present in the stream along desired dimensions with the hope that the features of the signatures will be general enough to support both important known applications and anticipated future interests. These signatures serve as a high-level summary of the contents of a transaction warehouse and allow analysts to focus their attention on finding interesting patterns in the signatures.

The space and time constraints in computing and using signatures are quite challenging. The volume of data is typically large in two dimensions: the number of transactions in a single batch and the number of entities to track over time. For example, in telecommunications data, a daily batch often describes hundreds of millions of calls involving tens of millions of distinct telephone numbers. Over time, such a data stream yields hundreds of millions of telephone numbers with signatures. The size of these signatures is bounded by the resources available to compute them. As more resources become available, data analysts revise their applications to store larger signatures.

Various uses of signature data impose strict time constraints. The first such use is in batch processing. During processing, the signatures for entities

mentioned in the batch are retrieved and updated. All the transactions within a single batch must be processed within less than half the batch window time to permit recovery in the case of failures (disks fail fairly frequently). Hence reading and writing all the necessary signatures must take less than this amount of time. In the telecommunications case, typically over 100M signatures must be updated daily. A second use is in viewing signatures through a web interface. Data analysts must be able to access the signature associated with a given entity within web-time, typically under a second.

The original signature programs were C programs carefully engineered to achieve the necessary performance. Data analysts wrote the original programs in C because they were comfortable with the language, it supported their performance goals, and it allowed them to manage the space usage of their applications through its support for user-control of data representation. In C, the performance requirements forced the analysts to structure their programs around managing the scale. The bulk of the code involved ensuring good locality of access to the signatures stored on disk to overcome the I/O bottleneck.

Although these original signature programs were very useful to AT&T, the data analysts were reluctant to work on new signatures. The programs were tedious to write in the first place because the "interesting" part, the per-entity code, was a small fraction of the required code, the bulk of which was dedicated to ensuring good performance. Once written, the programs were hard to maintain because it was difficult to see *what* was being computed through all the code specifying *how* to compute it.[1] Finally, despite significant engineering efforts, the data analysts were able to compute only 2-byte signatures[2] within the available resources. They desperately needed an easier way to write signature programs and to compute larger signatures.

At this point, we hypothesized that a domain-specific language for writing signature programs might be a possible solution. By designing appropriate abstractions, we could allow signature programs to be structured around the per-entity computations and hide issues of scale. The scaffolding code that dominated the earlier programs could be generated by the compiler, allowing the data analysts to focus on the part of the computation that interests them. The resulting programs would be easy to write and maintain, but still have good performance.

With this thesis, we undertook the design and implementation of Hancock, a domain-specific language for signature processing. We viewed this project as a case study in *practical* language design. Our evaluation criteria for the language was simple: did the data analysts use it? To succeed, we had to work closely with the analysts to ensure that we produced something that met all their constraints and that they were comfortable with. We had to generate programs that were no less efficient than the C programs they were already

[1]Maintenance is a particularly important aspect of signature programs in telecommunications because the federal government regulates how AT&T uses such transactional data. Periodically, the data analysts must review the programs to ensure that they comply with current federal regulations involving the uses of such data.

[2]Signature sizes are defined to be the size of the payload not the size of the payload plus the size of the key.

using. We also had to produce a working system quickly, as new signatures were desperately needed. Finally, we had limited resources to apply to the problem, both in terms of people to design and build the system (roughly two researchers and two summer students) and in terms of equipment, i.e., existing machines, disks, and memory.

Because of our success criteria and resource constraints, we adopted a bottom-up and iterative design philosophy. We added only those features and abstractions to Hancock that seemed essential, that is, those for which we had several motivating examples and that we knew we could implement efficiently. By adopting an iterative strategy, we ensured that from very early on we always had something working. We did not try to design the whole language before implementing it.

At the start of the project, we made a number of basic design decisions.

—*Not using a database*. Our users had had previous experience loading databases with transaction data and using database facilities to compute signatures. But because the patterns of usage of signature applications (high percentages of updates) did not match those for which databases are typically tuned (large numbers of concurrent reads) [Belanger et al. 1999], this approach did not yield acceptable performance. Consequently, the data analysts adopted a programmatic approach. We took this choice as a given, although subsequent experimental results have supported their preference [Fisher et al. 2002; Sullivan and Heybey 1998; Babcock et al. 2002; Carney et al. 2002; Hellerstein et al. 2000].

—*Designing a language*. Although portions of our design might have been amenable to a library-based design, we believed that overall the benefits of a language outweighed the learning-curve overhead and the lack of supporting tools (e.g., debugger, etc.). The primary benefit of a language over a library is that a language presents the user with a world view. Because Hancock is designed for a specific domain, the world view it presents makes it much easier to write efficient programs for that domain. In addition, the control-flow abstraction that is at the heart of batch processing (see Section 3.3) is awkward to express in a library.

—*Extending* C. We knew we wanted to extend a language in designing Hancock so we would not have to replicate the fundamental structures provided by every programming language. We chose C because it was already familiar to our target users and it has low run-time overhead.

—*Static and dynamic checking*. Finding bugs in signature programs can be very difficult: how do you determine that the seven gigabytes worth of data in a signature collection are the *right* seven gigabytes? It is impossible for a human being to look at a more than a tiny fraction of the data. To help prevent bugs, we chose to provide both static and dynamic checking to the extent possible in a C-based framework and given the all-important performance constraints.

—*Supporting parallelism*. An obvious way to parallelize a signature program is to divide the transaction stream into substreams containing ranges of entities and then to assign the task of computing the signatures for each

range of entities to a separate processor. Although we may provide support for this strategy in the future, we have not yet done so because in practice, the parallelism achieved by running each signature program on a separate processor was sufficient to obtain acceptable performance.

We designed and implemented the first version of Hancock in 1998. This version was specific to a particular form of call-detail transactional data [Bonachea et al. 1999]. Using this version of the language, data analysts designed a new signature program; shortly after, all existing signature programs were ported to Hancock. We then generalized the language to support arbitrary fixed-width transactional streams [Cortes et al. 2000], in the process improving various aspects of the implementation to support larger signatures. In response, the data analysts designed two new signature programs with signatures over 100 bytes per entity. All of these Hancock programs, and many supporting programs, have been running in production for several years. Thus, according to our evaluation criteria, the Hancock project has been a tremendous success.

This paper describes the design of Hancock. It expands upon Cortes et al. [2000] and incorporates recent additions to the design. We have structured the rest of the paper as follows. In Section 2, we describe the environment in which Hancock programs run and introduce a signature application that we use as a running example to motivate and explain the various features of Hancock. In the following sections, we present Hancock's various domain-specific abstractions, starting with abstractions related to transaction streams (streams, multi-unions, and the `iterate` statement) and then moving on to abstractions related to storing signatures persistently (maps, pickles, and directories). In Section 5, we describe how we extended our design with a parameterization mechanism to provide better support for some of the new signatures written in Hancock. We then give an implementation overview (Section 6), describe our experiences using Hancock in practice (Section 7), sketch some future directions for the language (Section 8), and end with some concluding thoughts (Section 9).

2. SETTING THE STAGE

The analysis done with Hancock on call-detail records at AT&T is only one part of a complex system. In this section, we sketch the pieces of that system that surround the Hancock programs to give the reader more context. We then introduce an example signature program that we will use throughout the paper to motivate and explain the design of Hancock.

2.1 Collecting Raw Data

Switches in the telephone network collect information about the calls they handle as part of normal call processing. They store this information in *AMA records*, a highly complex, industry-standard format. These records contain information needed for call signatures, such as the originating telephone number, as well as information not needed for signatures, such as how the call was routed through the network. As calls are completed, the switches transmit the corresponding AMA records to a collection machine, where a program extracts

fixed-width call-detail records from the AMA records. The extracted call-detail records are then sent on to a system called "the streamer." This system uses registered filters to create special-purpose buckets of records. All Hancock applications share the same filter, which accepts almost all records.

Once an hour, the records in the Hancock bucket are sorted in two different ways to produce two different files: one sorted primarily by the originating number and secondarily by the dialed number, the other sorted by the dialed and then by the originating number. Once a day, each of the two collections of sorted files are merged and fed to a suite of Hancock programs that update a family of signature collections.

These signature collections are accessed in several ways. Fraud representatives query them through a web interface using canned selection programs written in Hancock. Data analysts use a standard suite of Hancock selection programs to retrieve lists of signatures corresponding to given phone numbers. Finally, researchers (largely statisticians) use their own Hancock programs to explore the data and to develop new signatures.

With the exception of the Hancock programs, most parts of this system are either C programs or UNIX shell programs. Many systems geared toward analyzing stream data, Aurora [Carney et al. 2002], Telegraph [Hellerstein et al. 2000] and STREAMS [Babcock et al. 2002], to name a few, are designed to build architectures similar to the one described above, up to and including the streamer. That is, they solve the problem of how to get the records to the right application. Hancock solves a different problem: what to do with all the data once you have it.

2.2 Running Example: The Cell Tower Signature

In the rest of this section, we introduce an example signature program that we use throughout the paper to motivate and explain the design of Hancock. This example, called the *Cell Tower application*, mines information from a wireless call-detail stream. In particular, the application measures the mobility or "diameter" of mobile phone numbers (MPNs). Phone numbers that are used exclusively in one or a few neighboring cells have small diameters, while those used in larger regions have larger diameters. Such information is useful for fraud detection and for developing new location-based services.

The wireless call-detail stream consists of a sequence of records, each one of which describes a call made on the wireless network. Although these records contain many fields, only the following few are relevant for our purposes:

—originating phone number,
—dialed phone number,
—first cell tower (originating),
—last cell tower (originating),
—first cell tower (dialed), and
—last cell tower (dialed).

Either one (or both) of the originating and dialed phone numbers correspond to a mobile phone number (MPN). Information related to mobility appears in the

cell tower fields. If the originating phone number belongs to a mobile phone, then the first originating tower captures the first tower used to carry the call. If the phone moved significantly, then the last tower captures the final tower used during the call. The dialed towers carry the same information for the dialed phone number.

The Cell Tower application tracks for each MPN the five most frequently (and most recently) used cell towers and another value that captures the frequency with which calls placed to/from the MPN do not involve the top five cell towers. As one might expect, the top five list is dynamic, so the signature computation includes a probabilistic bumping algorithm that allows a new cell tower to enter the top five list as its frequency of use increases. More concretely, we will use the following `C` struct:

```
#define TOPN 5
typedef struct {
  unsigned int tower[TOPN];
  float count[TOPN];
  float other;
} profile_t;
```

as the type of the signature for each mobile phone number. The `tower` array stores the five most frequently used cell towers, while the parallel array `count` measures the frequency with which the corresponding tower is used. Field `other` measures how many calls are not reflected in the list of the top five towers.

In the Cell Tower application, we want to track such profiles over time; consequently, we must associate each mobile phone number with its profile persistently. This type of association is central to the applications in Hancock's target domain. As a result, Hancock includes an abstraction, called `maps`, for defining and managing such associations.

In the `profile_t struct`, we use an unsigned integer to represent cell towers. In the wireless call-detail stream, however, cell towers are represented as variable-length strings. Because types with fixed size are much more convenient for both computation and storage, we use a hash table to associate an (unsigned) integer hash key with each string and store the hash key in the profile instead of the string. Because the profile data is persistent, the mapping between a given hash key and a given cell tower name must be persistent as well. While this persistent data structure is important for this application, it is not a core part of most Hancock applications. As a result, Hancock does not provide persistent hash tables directly. Instead, Hancock provides an abstraction, called `pickles`, that allow users to define their own persistent data structures. The Cell Tower application uses this construct to define a persistent hash table.

We want to reflect the close semantic coupling between the signature collection and the persistent hash table in the application program to ensure that we use the two kinds of persistent data properly. This issue of persistently grouping various data arises in many Hancock applications. Consequently, Hancock provides a construct, called `directories`, for this purpose. We will use a directory to group the signature collection and the persistent hash table. In addition, to help identify and preserve the integrity of the application data, we will add a

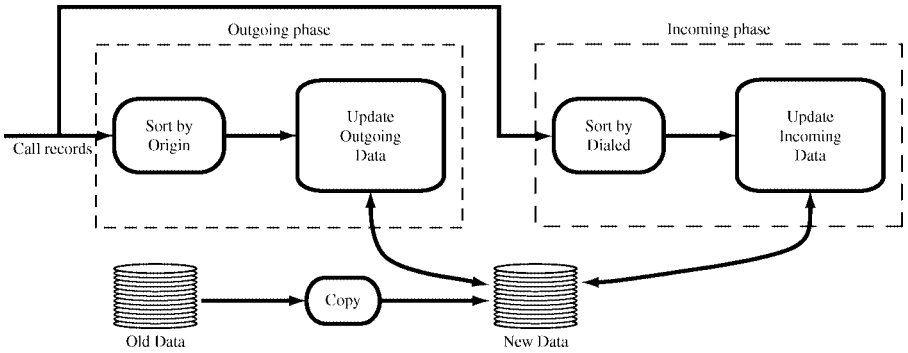

Fig. 2. High-level architecture of signature computations. The processing typically consists of several phases, each sorting the data in a different order and updating a different part of the signatures.

field to the directory tagging the data with the date of its most recent update. To support this kind of usage, Hancock directories provide automatic persistence for statically-sized C types and C strings.

The Cell Tower application uses a process flow typical for signature applications to compute the desired persistent information. Figure 2 depicts this flow: transaction records are collected for some time period, the length of which depends on the application (e.g., a day for marketing but just a few minutes for fraud detection). At the end of the time period, the records are processed to update the signatures. Before processing, the old signature data is copied to preserve a back-up for error-recovery purposes. During processing, several passes are made over the data to perform the updates.

Each pass over the stream of records has a standard structure as well. First, a user-supplied function translates each *physical* record into a *logical* representation, discarding invalid records in the process. During this translation, we convert cell tower names into their associated hash keys. Second, a user-supplied filter function discards valid records that are not interesting for our current purposes. In our sample application, we remove calls that do not involve a cell tower. Next, the records are sorted in some order, for example, according to the originating phone number for one pass and according to the dialed phone number for another.[3] Finally, the portion of each signature relevant to the given sort is retrieved from disk, updated, and then written back to disk. For example, after sorting by the originating phone number, the portion of a signature that tracks out-bound calling is updated; after sorting by the dialed number, the portion that tracks in-bound calling is modified. Sorting the stream ensures good locality for accesses to the signatures on disk and groups the information relevant to each phone number into a contiguous segment of the stream. Hancock's `stream` abstraction and `iterate` statement allow a programmer to describe and process transactional data streams using this structure easily.

[3]If the records have already been sorted, as in the streamer architecture described in Section 2.1, this step can be omitted.

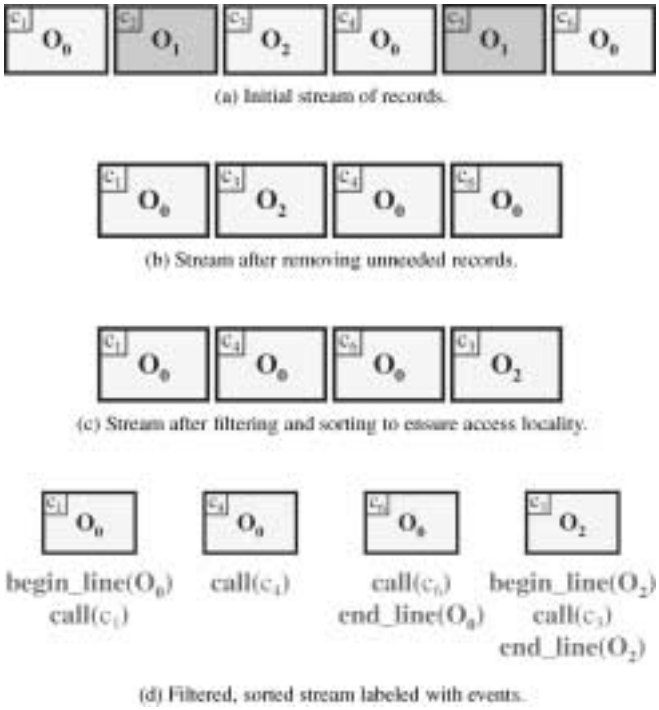

Fig. 3. Sample wireless call stream running from left to right. Each box represents a call from an originating phone number (for example, the first call was placed by O_0). Each box is labeled with a call number (c_1, for example). Light gray/yellow boxes represent calls that originated from mobile phone numbers. Dark gray/purple boxes represent calls that originated from land lines.

To improve the readability of code fragments in the remainder of the paper, we adopt the following naming convention: map types end with the `_m` suffix, pickles with `_p`, directories with `_d`, and streams with `_s`.

In summary, this application requires three abstractions for representing persistent data: one to associate signatures with keys (cf. Section 4.5), a second to describe the persistent hash table (cf. Section 4.7), and a third to group the most-recent-processing date, the signature collection, and the persistent hash table (cf. Section 4.6). It also needs abstractions for describing and processing stream data (cf. Section 3). We discuss how Hancock provides the abstractions necessary for this application in the next few sections.

3. STREAMS

Hancock's core abstractions include mechanisms for describing and computing with streams of transactions. In this section, we present Hancock's mechanism for defining stream types, its model of stream events, and its mechanism for consuming streams.

3.1 Describing Streams

The design of Hancock's stream abstraction arose directly from examining the original signature programs, which consumed streams of highly-encoded

call-detail records. These programs interleaved code for decoding the representation of stream records with signature processing code, which made it difficult to update the programs when the stream representation changed.

We designed Hancock's stream abstraction to separate the description of a stream type from its use and thereby encourage good programming practices. We also designed it to encourage the programmer to specify a separate *logical* representation of the stream records that is suitable for computation, as opposed to a physical representation that is suitable for storage. Separating the physical and logical representations allows one person to understand the physical representation (the expert on that data source) but many people to use the logical representation (the consumers of that data source). This division facilitates maintenance: if the physical representation changes, only the translation from the physical to the logical representation must be modified, presumably by the expert on that data source. The consumers need not modify their programs.

Another important aspect of Hancock's stream abstraction is the fact that during the translation from the physical representation to the logical, the physical record is validated. A fact of life of large-scale data processing is erroneous values in the data. Because translation requires examining the physical record, it easy and efficient to determine at that point if the record is meaningful. Removing buggy data simplifies downstream processing because later code can assume that all the logical records are meaningful.

Hancock programmers use stream type declarations to describe streams. Hancock supports two forms of this declaration: a specialized form for streams whose records are stored on disk in a fixed-width binary format and a general form for records stored in other formats. We designed the binary form initially because the call-detail records used by the original signature programs all had that form. We later extended the design to general records to accommodate a wider range of data sources. We retained the original form because it is more convenient to use in the frequently occurring special case of fixed-width binary data.

The declaration of a binary stream specifies both the physical and the logical representations for the records in the stream. It also specifies a function to convert from the encoded physical representation to the expanded logical representation, validating the record in the process. The Hancock runtime system handles all file I/O and calls the specified function once for each binary record in the stream.

The following declaration introduces the wireless call stream `binaryWireless_s`:

```
stream binaryWireless_s {
  getvalidWCRbinary : wcrPhy_t => wcrLog_t;
};
```

For this stream, the C type `wcrPhy_t` serves as the physical representation and `wcrLog_t` serves as the logical. The identifier `getvalidWCRbinary` names the function that validates records and specifies how to convert from the physical to the logical representation.

In the general case, the programmer handles the file I/O rather than the Hancock runtime system. In particular, a general stream declaration specifies a function that takes a file pointer as an argument, reads data from that file, and returns a valid logical record.

The records that constitute a stream are stored on disk, historically in collections of files grouped in a directory. To process these records in a Hancock program, we had to provide a mechanism whereby the programmer could connect this on-disk representation to the appropriate in-memory representation. We designed Hancock's *initializing declarations* to provide such a mechanism. These declarations extend C's declaration form with a path that indicates the location of the on-disk representation. Advantages of this form include familiarity to C programmers and the guarantee that a persistent variable is connected to its on-disk representation before that variable can be used. For example, programmers can declare a stream variable `calls` and connect it to the data stored in `data/call-detail.current` using the syntax:

```
wireless_s calls = "data/call-detail.current";
```

In the remainder of this paper, we use the term "record" to mean the logical representation of the elements in a stream, since stream definitions are the only place where the physical representation is needed.

3.2 Event-Based Programming Model

The original signature programs used an event-based programming model to process their data streams. The events were triggered by value changes in the telephone number fields in successive records in the data stream. For example, typical events included seeing a new area code (`npa_begin`), a new exchange (`nxx_begin`),[4] a new phone number (`line_begin`), an individual call record (`call`), the last record for a phone number (`line_end`), etc. Depending on the intended application, a given program would take different actions in response to these events. For example, when a program detects an `npa_begin` event in a stream, it may retrieve the time zone for the triggering area code. In response to an `nxx_begin` event, it may age[5] the signature values stored for the phone numbers in that newly seen exchange. For a `line_begin` event, it may initialize counters that it later increments in response to `call` events. The program may store the final values for these counters when a `line_end` event occurs.

This computational model is well suited to processing transactional streams, but C does not provide much help in realizing this program structure. Hence, although the original programs used this model, the underlying structure was not readily apparent in the C implementations, making the heart of each such program very difficult to code and maintain. In particular, the programs had no explicit notion of event and no event-detection code identified as such. The event-response code was intermingled with the event-detection code and buried

[4]An *exchange* is the first six digits of a 10-digit telephone number.

[5]Aging is a process that allows new information to replace old information gradually. It is typically implemented by multiplying old values by a small constant λ. New values are then multiplied $1-\lambda$ before merging them with old values.

in the scaffolding code necessary to sequence the various events. This scaffolding code was particularly difficult to read because it involved deeply nested and inverted loops. The nesting arises from the hierarchical nature of the events: within a given area code, one must detect the exchanges; within an exchange, one must detect the lines, within a line, one must detect the calls. The inversion of these loops occurs because when one detects a "begin" event, that is, a new area code, a new exchange, etc., one has also detected an "end" event, that is, the last call for the previous area code, exchange, etc. Hence each loop starts with the response code for the "end" event, followed by the response code for the "begin" event. This structure makes the loops hard to read because the code to clean up a piece of the computation precedes the related set-up code in the program text.

Our primary goal in designing Hancock was to make the stream processing code as transparent as possible; hence we added various abstractions to provide explicit support for the event-based computational model used in signature programs. To that end, we reified the notion of an event and introduced the `iterate` control-flow abstraction that separates event detection from event response and makes the scaffolding code implicit. As a result, the Hancock versions of the original signature programs are much shorter and the control-flow is very clear. The effect is to highlight the event-response code, which corresponds to the per-entity code that interests the data analysts.

In the following sections, we introduce our abstractions for events, event detection, and event response.

3.2.1 *Describing Events: Multi-Unions.* Originally, Hancock included hard-coded call-detail events. Each piece of a phone number (fields with names `npa`, `nxx`, `line`) in the logical representation of a call record automatically "came with" beginning and ending events. The events were triggered by changes in the field's values as the computation moved from record to record in the stream. This model worked well for our original data source. It was expressive enough to code the original signature programs, it required little work on the part of the programmer, and it allowed us to get the original applications working in Hancock very quickly.

As a general solution, however, this model was clearly lacking since it meant that Hancock programs could manipulate only one type of data. To address this problem, we introduced a way for programmers to describe the events associated with each record in a data stream. Because the events of interest for a given data source are essentially fixed, it made sense to have a declaration form in which programmers could give names to the various events and specify the type of value that those events would carry. For example, in the call-detail case, the `line_begin` event carries the phone number for which the given record is the first call, while the `call` event carries the entirety of the call record. Because each record in the data stream may trigger any subset of the possible events, we needed to provide values that were collections of events. To allow such event collections to be computed modularly, we required a way to merge event collections. Finally, we had to have a way to test if a particular event belonged to an event collection and if so, to extract the associated value.

With these properties in mind, we designed Hancock's `multi-union` abstraction. A multi-union declares a set of labels and associated types. Because of their common usage in Hancock programs, we often refer to these labels as *events*.[6] As an example, consider the declaration:

```
munion line_e {: long long line_begin,
                 wcrLog_t call,
                 long long line_end :};
```

This code creates a multi-union type `line_e` to describe the events we need for the Cell Tower application. A value with this type contains any subset of the declared labels, including the empty set, which we write `{: :}`. Each label in the set carries a value of the indicated type. If `l` is the current mobile phone number and `c` the current `wcrLog_t` call record in a stream, then the expression

```
{:  line_begin = l, call = c :};
```

creates a value with type `line_e`. This value would describe the events that occur when the first (but not the last) call record for mobile phone number `l` appears in the stream.

Hancock supports a variety of `munion` operators. For example, if `e1` and `e2` are multi-union values with the same type, then expression `e1:+:e2` produces a new value that contains the union[7] of the labels of `e1` and `e2`. If both `e1` and `e2` contain a given label, then `e2`'s value takes precedence. In addition to the union operator, Hancock also supports operations for membership test (`@`), value access ("`.`"), difference (`:-:`), and removal (`:\:`).

To maintain consistency with C's general philosophy, we allow programmers to access the value associated with a label in a multi-union without first checking that that label carries a value in the multi-union. In practice, deconstructing multi-union values with the value-access operator does not occur frequently. Typically multi-unions are deconstructed within the context of an `iterate` statement (cf. Section 3.3), which uses a pattern-matching-like construct to extract only the values actually present in the given multi-union.

3.2.2 *Detecting Events.* Allowing programmers to separate the events of interest from the logical representation meant that we had to provide a way for programmers to describe how to produce those events from the data stream. In studying the events used in signature programs, it was clear that the events to be associated with a single record could not be determined by examining the record in isolation. Some events required looking at the current record and the one that preceded it, while others required looking at the current record and the one that follows it. Typically "beginning" events require looking back, while "ending" events require looking forward. Depending on the events needed for an application, a program might have to look both ahead and back, or only ahead, or only back, or in some cases only at the current record. None of our

[6] Although we designed multi-unions to describe events, they are in fact a general construct, suitable for many purposes; hence we named their constituents *labels* instead of *events*.

[7] *Merge* might have been a better name for this operation, since the value associated with a label in `e1` will be lost if the same label appears in `e2`.

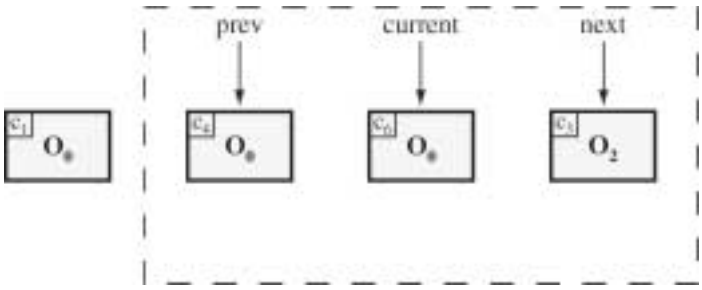

Fig. 4. Stream with window of type*wcrLog_t[3:1]. See Figure 3 for a description of the stream notation.

sample applications required looking forward or backward an arbitrary number of records, however.

To allow programmers to calculate the events to associate with the "current" record from a fixed segment of a data stream, we added the notion of an event-detection function. Such a function takes as a parameter a *window* onto the stream and returns a multi-union that describes the events detected for the current record in that window (see Figure 4).

A window type allows the programmer to specify statically both how many records in the stream can be viewed at once and the position of the "current" record. We chose not to support more dynamic window specifications because such expressiveness would require a more complicated and expensive implementation without providing any benefit to the desired applications.

Syntactically, a window is like an array with the added notion of a "current" record. For example, the declaration

```
wcrLog_t *w[3:1]
```

specifies that `w` is a window of size three onto a stream with records of type `wcrLog_t`. A pointer to the current record appears in the middle slot of the window, that is, in `w[1]`. Slots with lower indices (`w[0]`) store pointers to records earlier in the stream; slots with higher indices (`w[2]`) look ahead to records appearing later in the stream. The window contains pointers to the stream elements instead of the stream elements themselves so that when the window overlaps either the beginning or the end of the stream (or both), the slots with no corresponding record can be set to NULL.

Given multi-unions and windows, programmers can write event-detection functions. For example, the Cell Tower application uses the event detection function shown in Figure 5.

This function compares the previous record with the current one to determine if a `line_begin` event has occurred. It then compares the next record with the current one to determine if a `line_end` event has occurred. Function `originDetect` then uses the multi-union `:+:` operation to merge these intermediate results with the `call` event carrying the current record as its value. When called with the window in Figure 4, the `originDetect` function generates the value

$$\{: \texttt{call} = c_6, \texttt{line\_end} = O_o :\}$$

```
line_e originDetect (wcrLog_t *w[3:1]){
  wcrLog_t *prev = w[0];
  wcrLog_t *current = w[1];
  wcrLog_t *next = w[2];
  line_e b,e;

  if ((prev == NULL) || (prev->origin != current->origin))
    b = {: line_begin = current->origin :};
  else b = (wline_e){: :};
  if ((next == NULL) || (next->origin != current->origin))
    e = {: line_end = current->origin :};
  else e = (wline_e){: :};
  return b :+: {: call = *current :} :+: e;
}
```

Fig. 5. An event detection function for the Cell Tower application.

The Cell Tower application uses an additional event detection function to process incoming calls, called `dialedDetect`, which is structured similarly, although it reads values from the `dialed` field of the records in the window.

The runtime overhead of constructing multi-unions and calling event-detection functions does not significantly contribute to the running time of our signature programs, whose performance is mostly bounded by I/O and compression times.

3.3 Consuming a Stream

As in the original signature programs, Hancock's computation model is built around the notion of iterating over a stream of transaction records. The code that implements this computation is the heart of each Hancock signature program: it contains the per-entity code that implements the semantics of the signature. Consequently, it is crucial that this code be clear, concise, and efficient.

Conceptually, this code performs a number of tasks. It must

—identify the transaction stream;
—indicate any desired filter to discard unwanted records;
—specify how to sort the stream;
—indicate the desired event-detection function;
—give the event-response code for all relevant events.

Although it may seem counterintuitive to sort the records in a stream, sorting them is important semantically and crucial to obtaining good performance. Semantically, sorting by the entity around which signatures are constructed groups all the data relevant to each entity into a contiguous segment of the stream. This grouping ensures that events like "the first call for a phone number" make sense. Performance-wise, sorting by the entity ensures good locality of access to the persistent data associated with the entities. In practice, the time required to sort the stream is quickly recovered in reduced stream-processing

time. In fact, without this locality, signature programs will not run in a timely fashion [Fisher et al. 2002].

Hancock provides the `iterate` abstraction to group these pieces into a single coherent whole with an easily understandable control-flow. This statement has the following form:

```
iterate
  (over stream expression
   filteredby filter predicate
   sortedby sorting order
   withevents event detection function)
 {
   body
 };
```

The header specifies the initial stream, a set of transformations to prepare the stream for computation, and a function to detect events in the transformed stream. The body contains a set of event clauses that specify how to respond to the detected events. We describe some of these pieces in more detail below.

The `filteredby` clause specifies a predicate to remove unneeded records from the stream. Immediately removing such records improves the efficiency of sorting. It also simplifies event response code since that code can assume that it will only be given "interesting" records to process. For example, a `wireless_s` stream may include land-to-cell calls, which are not used to compute signatures for originating phone numbers in the Cell Tower application. Figure 3(b) shows a sample wireless call stream after filtering.

The `sortedby` clause describes a sorting order for the stream by listing the fields from the records in the stream that constitute the desired sorting key. For example, the clause

```
sortedby origin, dialed
```

produces a stream sorted primarily by the originating telephone number and secondarily by the dialed phone number. Figure 3(c) shows a sample call stream after the calls have been filtered and sorted by the originating number. If the records in the stream do not need to be sorted, as in the architecture described in Section 2.1, then this clause can be omitted.

The `withevents` clause specifies an event detection function. Figure 3(d) shows a sample wireless call stream labeled with events.

The *body* consists of a sequence of event clauses that specify code to execute when an event detection function triggers an event. Events that occur simultaneously (i.e., in the same multi-union value) are processed in the order they appear in these event clauses. Given this ordering information, Hancock generates the control-flow to sequence the response code. The name of each event clause corresponds to a label in the multi-union returned by the event detection function. Each event clause takes as a parameter the value carried by the

```
void doOrigin(char *streamLoc, cellTower_m m)
{
  wireless_s calls = streamLoc;
  profile_t p;

  iterate
    ( over calls
      filteredby originCellCall
      sortedby origin, dialed
      withevents originDetect ) {

      event line_begin(long long origin) {
        initProfile(&p);
      }

      event call(wcrLog_t c) {
        aggregate(&p, c, ORIGIN);
      }

      event line_end(long long origin) {
        profile_t op = m<:origin:>;       // Map read operation
        m<:origin:> = update(op,p);       // Map write operation
      }
  };
}
```

Fig. 6. Outgoing phase for the Cell Tower application.

corresponding label. For example, the mobile phone number that triggers the `line_begin` event is passed to the `line_begin` event clause. The body of each event clause is a block of Hancock/C code.

As an example, Figure 6 shows the Cell Tower code which processes the calls made *from* mobile phone numbers. The function `doOrigin`, which encapsulates this pass over the data, contains a single `iterate` statement that consumes a wireless call-record stream. It uses the predicate function `originCellCall` to remove noncellular calls from the stream. It sorts the filtered stream by the originating phone number. It uses the function `originDetect` to detect events in the sorted stream. The event clauses specify how to respond to the detected events. Section 4.5 describes the operations that appear in the `line_end` event response code.

Note that the code in Figure 6 is concise. It is easy to see what calls are being incorporated into the signatures by looking at the filter function, which is a key part of ensuring that a given program complies with federal legislation regarding data use. The control-flow is also easy to understand. The various events of interest are clearly marked, and for each record, the response code for each applicable event is executed from top to bottom. Because this structure highlights event-response code, it allows data analysts to focus on the per-entity

code when they write new signature programs, which was the primary criterion in designing both the `iterate` abstraction and Hancock as a whole.

3.4 Discussion

At various times, we have considered adding more complex stream operations, such as stream composition, and more sophisticated event processing, such as selecting only one matching event from each `munion`, to Hancock. We have resisted the temptation to add such features because we lack a compelling application that needs them at this time.

3.5 Related Work

Many systems support operations on streams. We restrict our attention to systems that either use an event-based model for processing streams or are designed to handle high-volume streams.

3.5.1 *Event-Based Processing.* Michael Jackson describes a design methodology similar to the one supported by Hancock's `iterate` abstraction [Jackson 1975]. Using his methodology, COBOL programmers identify the structure of the input data to an application and then specify the operations to be performed when processing each piece of that structure. COBOL does not provide explicit support for this programming model.

AWK [Aho et al. 1979] is a string processing language that is based on pattern-action pairs. Patterns can be arbitrary boolean combinations of regular and relational expressions. Although Hancock's event-detection functions could be used to implement AWK's patterns, AWK's light-weight nature makes it superior for simple text-file processing. On the other hand, AWK's string focus makes it less suitable for processing large amounts of binary data. In addition, AWK provides little support for managing persistent data over time.

SAX [SAX Project 2002] is an event-based API for processing XML documents. It converts an XML document into a stream. SAX programmers process the resulting stream by registering call-back functions for the following set of events: the beginning/ending of XML documents, the beginning/ending of XML elements, and groups of characters. SAX also allows programmers to generate output streams with new events and to build pipelines of event processors. Unlike tree-based approaches to processing XML, SAX allows programmers to process XML documents without constructing in-memory data structures for complete documents. Hancock and SAX share the idea of processing streams using events, but they proceed in very different ways. SAX provides a light-weight library that simplifies the process of writing XML applications but does not provide linguistic support beyond that of the host language. Hancock, on the other hand, provides much richer programmer support, at the cost of learning (and using) a new language.

3.5.2 *High-Volume Streams.* This section describes systems designed to handle high-volume stream data. The oldest of these systems, Tribeca [Sullivan and Heybey 1998], predated Hancock. More recently, high-volume stream processing has become an area of active interest in the database

community [SIGMOD 2002; VLDB 2002]. Aurora [Carney et al. 2002], Telegraph [Hellerstein et al. 2000], and STREAMS [Babcock et al. 2002] are all examples of systems under development for computing with high-volume streams. In contrast to Hancock, which has been deployed in production for several years, these newer systems are in various stages of prototyping. Further experience is necessary to determine how well these systems will scale. We briefly describe the focus of each of these projects.

Tribeca [Sullivan and Heybey 1998] is a system for monitoring network traffic. It provides a query language that includes operations for separating and recombining streams, operations for computing moving-aggregates over windows, and a restricted form of join. The separation and recombination operators might be used, for example, to convert a packet-level stream into a session-level stream. This conversion is done by splitting the packet-level data into separate streams (one for each session), converting each substream into a single record for a session, and then recombining the session records into one stream. Tribeca provides much more support for describing and manipulating streams than Hancock does, but it provides less support for computing with the individual elements in a stream.

Aurora [Carney et al. 2002] and Hancock are complementary. Aurora is a system designed to monitor stream data. It provides a general framework for producing architectures like the one described in Section 2.1. It supports queries over multiple streams of data and allows queries to join and leave the system over time. One can view the queries combined with the streams as a graph. At the end of any path in the graph is an application that consumes the resulting data; that application could be a Hancock program.

Telegraph [Hellerstein et al. 2000] is an adaptive dataflow system designed to compute continuous queries over streams of data. PSoup [Chandrasekaran and Franklin 2002], a system built on top of Telegraph, expands upon this model to allow the query mix to change over time. This system can be used to compute aggregates from stream data, such as how many music downloads occurred in a given subnet within the last hour. Like Aurora, Telegraph is not designed to provide direct support for integrating stream data into persistent structures, the essential operation in computing signatures.

The members of the STREAMS project [Babcock et al. 2002] are developing a system for executing continuous queries over multiple streams. The focus of this project is to develop fundamental models of stream data systems and efficient methods for managing resources in such systems. At present, their model explicitly excludes queries that can modify persistent data during computation. This restriction, which may be removed over time, eliminates signatures as a possible application for STREAMS.

4. PERSISTENT DATA

Given Hancock's various stream abstractions, we next had to develop mechanisms to store the computed signatures persistently. We approached this design problem in the same way we approached designing streams: we met with the domain experts frequently and studied the existing C-based programs to

understand precisely what they were computing and why. This process enabled us to come up with a set of design requirements for persistent data.

The overarching requirements were efficiency of the resulting programs, data safety, and fast prototyping at scale. This last point deserves further elaboration. Our users want to experiment with their ideas *at scale*. They are not interested in writing programs that use only a small amount of data because such applications may not be feasible on larger data sets. However, they do not want to waste significant time in coding an application that produces uninteresting data in the end, either. Hence, they want the ability to put together reasonably efficient prototypes for computations at scale with little effort. Once an idea has proven itself, though, they are willing to invest (some) effort to exploit application-specific knowledge to tune the performance of their applications and to craft specialized persistent types if necessary.

As our experience with signature programs grew, the design of Hancock's support for persistent data evolved from a single mechanism for supporting signature collections to a collection of mechanisms, which we call Hancock's *persistent data system*.

In this section, we discuss the persistent data requirements for signature applications. We then review related work, concluding that none of the existing solutions adequately meet the application requirements. Next, we describe the persistent data needed by the Cell Tower application to make the discussion more concrete. Finally, we present the details of Hancock's persistent data mechanisms.

4.1 Design Requirements: Maps

The primary data structure used in the original signatures programs persistently associated fixed-sized signatures with telephone-number keys. The performance of these structures was crucial to program performance. Typical applications stored 2 bytes for each of roughly 400 million keys. On disk these signature collections required approximately 2 Gbytes. The data analysts very much wanted to expand the size of the signature associated with each phone number, but performance considerations prevented them from doing so.

The operations necessary over these structures were quite limited: reading and writing the values associated with single keys, copying an entire structure, and iterating over the collection of all keys with values. The single-value reading and writing operations were used during the daily stream processing. The analysts copied the entire structure once a day and then used the fresh copy as the basis for that day's processing. This practice provided a coarse-grained rollback mechanism. If a failure occurred during processing, they discarded the modified copy and restarted the computation with (a fresh copy) of the previous day's collection. The analysts iterated over all the keys in a collection to "age" the signatures in the collection, that is, to modify the values to indicate that a day had passed.

Time constraints on these operations were fairly tight. The structures had essentially three modes of use: the daily signature computation, worklist selection, and web-based querying of the values associated with single keys. In the

daily computation, the signature program consumed a 5- to 10-Gbyte stream of call-detail records, in the process typically updating roughly 20–30% of the signature values in the collection, although some applications updated all of their signature values every day. The daily processing for a single application, including sorting the stream and copying the persistent structures, could take no more than a couple of hours so that all the signature programs would have time to complete (possibly with restarts to handle errors) before the next day's data arrived. Achieving this performance required carefully managing disk accesses to maximize locality of reference. In worklist selection, an analyst feeds a list of several hundred thousand phone numbers to a selection program and needs to have the results within a few minutes. Finally, in web-based querying, an analyst types a phone number into a web page and expects to have the associated signature in a few seconds. This requirement precludes course-grained compression techniques to improve processing performance because it can take hours to decompress gigabytes worth of data.

Another issue that arose in talking with the data analysts was the question of data safety. The scale of the data makes data safety crucial because it makes detecting errors extremely difficult. How does one determine that all the data in a large signature collection are correct? Because this problem is so difficult, it is essential that a persistent data system provide mechanisms to prevent inadvertent data corruption wherever possible and to stop corrupt data from tainting other data.

In response to these requirements, we designed the Hancock `map` abstraction, described in Section 4.5. This abstraction provides the operations described above, meets the indicated performance constraints, enabled larger signatures by improving upon the on-disk representation used in the original C programs, and where possible, supports data safety.

4.2 Design Requirements: Directories and Pickles

After building the first version of Hancock and porting the original production signatures to the new system, we realized that these programs needed additional abstractions for persistence. In particular, we needed a way to group related information persistently and we needed a way to support custom persistent structures that could be split across memory and disk.

Two related observations motivated the persistent grouping mechanism. First, the names of maps were often used to encode auxiliary information, such as the date of processing and the source of the data. Second, many applications involved a collection of maps and other structures that together constituted the persistent data for the application. An abstraction that permitted users to manipulate related data as a single unit would allow auxiliary information to be stored with a map without arcane naming conventions and would reduce the chance of errors from improperly mixing data from different time windows or from different applications. Furthermore, providing automatic serialization for basic C-types would mean that programmers would be able to store the auxiliary information quickly and easily. These considerations led us to the Hancock `directory` abstraction, described in Section 4.6.

The need for custom, partially memory-resident persistent structures became apparent in studying the auxiliary structures used in some signature applications. Some of these structures were quite large and could not fit comfortably in memory, but they were not good candidates for the map abstraction. This consideration led us to the Hancock `pickle` abstraction, described in Section 4.7.

4.3 Related Work

Many other languages provide support for persistent data. This support typically falls into one of three categories: pickle-based approaches, interfaces to databases, and orthogonal persistence systems. We briefly discuss each of these approaches.

The notion of pickling data structures dates back at least to Modula-3 [Nelson 1991], which provides pickles as a way to represent a value as a stream of bytes. Writing a value as a pickle and then reading it back produces a value "equivalent" to the original value. Similar mechanisms have been developed for other languages including C++ [Wang 1998], Java [Riggs et al. 1996], Python [van Rossum 2001], and SML-NJ [Appel 1990]. These byte-stream pickle mechanisms provide automatic persistence, but they do not support data structures that reside partially in-memory and partially on-disk. As a result, byte-stream pickles cannot be used to implement persistent data structures of the scale typically found in Hancock applications.

A second common approach to supporting persistence in a programming language is to provide an interface to a standard relational or object-oriented database. We rejected this approach for Hancock because we did not believe that a traditional database could handle the high-percentage of updates generated during daily stream processing [Belanger et al. 1999; Fisher et al. 2002; Sullivan and Heybey 1998; Babcock et al. 2002; Carney et al. 2002; Hellerstein et al. 2000]. Also, we were concerned that such an approach would not have integrated cleanly into the rest of Hancock.

Orthogonal persistence systems, such as `Oberon-D` [Knasmüller 1997], `PJava/PJama` system [Atkinson et al. 1996], and `Thor` [Liskov et al. 1999], represent a third approach to persistence.[8] Persistence in such systems is independent of the type of the data being preserved (the *orthogonality* property). The system automatically determines if a given piece of data must be persistent by starting from a collection of persistent roots and making all reachable data persistent (the *transitivity* property). Finally, in such systems there is no syntactic indication as to whether a given variable is persistent or not (the *independence* property). This approach is akin to automatic memory management, which can simplify programming, but at some performance cost. Because of the the tight space and time requirements of our domain and the absence of evidence that automatic techniques can work on large scale data, we adopted a more explicit technique for persistence in Hancock.

[8]M. Knasmüller [1997] discussed a variety of persistent object systems—ODE, GemStone, O_2, and ObjectStore—and how they relate to each other.

```
#define TOPN 5
typedef struct {
  unsigned int tower[TOPN];
  float count[TOPN];
  float other;
} profile_t;
```

Fig. 7. C `struct` for Cell Tower profiles.

4.4 Cell Tower Application: Persistent Data

In Section 2.2, we introduced the Cell Tower application, which tracks for each MPN the five most frequently (and most recently) used cell towers and another value that captures the frequency with which calls placed to/from the MPN do not involve the top five cell towers. Here we review the data structures needed to implement this application because we will use them to explain Hancock's persistent types in the following sections.

The Cell Tower application tracks profiles of type `profile_t` (cf. Figure 7) over time. It uses a Hancock map, called `cellTower_m` in the following, to associate each mobile phone number with its profile persistently.

In the `profile_t struct`, we use an unsigned integer to represent cell towers. In the wireless call-detail stream, however, cell towers are represented as variable-length strings. Because types with fixed size are much more convenient for both computation and storage, we use a hash table to associate an (unsigned) integer hash key with each string and store the hash key in the profile instead of the string. Because the profile data is persistent, the mapping between a given hash key and a given cell tower name must be persistent as well. Hancock does not support a built-in persistent hash table type, so the Cell Tower application uses a user-defined pickle type (`pHashTable_p`) to construct one.

We want to reflect the close semantic coupling between the signature collection and the persistent hash table in the application program to ensure that we use the two kinds of persistent data properly. We will use a directory (`cellTower_d`) to group the signature collection and the persistent hash table. In addition, to help identify and preserve the integrity of the application data, we will add a field (with type `char *`) to the directory tagging the data with the date of its most recent update.

In summary, this application requires three abstractions for representing persistent data: a map to associate signatures with keys (cf. Section 4.5), a pickle to describe the persistent hash table (cf. Section 4.7), and a directory to group the most-recent-update date, the signature collection, and the persistent hash table (cf. Section 4.6). The rest of this section describes the details of these data structures.

4.5 Signature Collections

The signature applications that were our original class of applications need a highly efficient persistent data structure for associating values with keys, so Hancock provides a persistent data type, called *maps*, to support these applications. Hancock maps meet the tight performance requirements in part by

providing only a limited set of operations: retrieving or updating the value associated with a key, removing a key from the map, asking if a given key has an associated value, and iterating over a range of keys with stored values. Hancock maps do not support transactions, locking, secondary indices, or declarative querying to avoid the associated overhead. In the remainder of this section, we describe maps in more detail.

4.5.1 *Map Type Declarations.* The programmer controls *what* a map stores using the map type declaration, while Hancock controls *how* the map stores its data by providing the in-memory and on-disk representations. The Cell Tower application, for example, uses the following declaration:

```
map cellTower_m {
  key 0..9999999999LL;
  split (10000, 100);
  value profile_t;
  default CT_DEFAULT;
  compress ctSqueeze;
  decompress ctUnsqueeze;
};
```

This declaration defines the map type named `cellTower_m`. We describe each of the clauses in this declaration in turn.

Keys typically represent some form of identification number, for example, telephone numbers, credit card numbers, or IP addresses. All keys are represented using the C type `long long`.[9] The **`key`** clause specifies the range of keys for the map. The example declaration specifies that valid mobile telephone numbers fall between `0` and `9999999999`.

Important performance issues for accessing map values are I/O and decompression times. The `split` specification allows programmers to tune these times by setting the *block* and *stripe* sizes. Intuitively, blocks serve as the unit of I/O, while stripes serve as the unit of compression. The first number in the `split` clause indicates the number of keys in a block. The second number specifies how many keys constitute a stripe. The **`split`** clause in the example specifies that there are 10,000 keys per block and 100 keys per stripe for `cellTower_m` maps. An earlier paper described the implementation of maps in detail and discusses efficient key decompositions for different data access patterns [Fisher et al. 2002].[10]

The **`value`** clause specifies the type of data to be associated with each key. This type can be any C type of statically known size. The Cell Tower application uses `profile_t`, the C `struct` defined in Section 4.4, as its value type. We refer to a key with a value in a given map as an *active* key and to an active key and its associated value as an *item*.

[9]We chose to use C's `long long` type to represent keys because representing telephone numbers requires more than 32 bits. The size of the key data type does not effect the size of a map on disk or in memory.

[10]Earlier versions of Hancock encoded both the range and split information into the key specification. Hancock now separates this information to simplify the specification and reduce errors.

The **default** clause specifies a value to be returned when a program requests data for an inactive key. Such requests are relatively frequent because large transaction streams often contain data for *fresh* keys, that is, whenever a new telephone number, credit card number or IP address is issued. Consequently, assigning meaningful values to new keys is an important element in Hancock programs. Isolating the default construction in the map declaration allows code that queries a map for the value associated with a given key to assume that it always receives a meaningful value, even for fresh keys. This assurance greatly simplifies transaction processing code (cf. Section 3.3).

Defaults come in two forms. A default may be a constant, as is the Cell Tower application, or a function. A constant default must be an expression that has the value type of the map. For the `cellTower_m` map, the default `profile_t CT_DEFAULT` specifies a constant as the default value for the cell-tower hash values and zeros for the counts. A function default is specified as the name of a function that computes a default value given a key as an argument. Such defaults are often used to query backup data sources.

Hancock allows programmers to specify functions to compress and decompress values before they are stored on disk because programmers can often exploit domain-specific knowledge to produce better compressors than the generic ones that Hancock supplies. Early prototypes for a given application may use default compression; once the ideas are validated, analysts may spend time writing custom compression functions to improve the performance of a proven application. The optional **compress** and **decompress** clauses name functions that compress (or decompress) a value of the map type.

4.5.2 *Map Variable Declarations.* Once the programmer has specified a map type, variables can be declared to have that type using C syntax: `cellTower_m cellData`. In addition, as noted earlier in Section 3.1, Hancock provides initializing declarations to allow the programmer to connect program variables to data on-disk.[11] An initializing declaration augments a standard C declaration with qualifiers and a location. For example, the declaration

```
new cellTower_m out = "cellData.current/ctOut";
```

augments the standard C declaration with a string, `"cellData.current/ctOut"`, which specifies the location of the on-disk representation of the map, and with a qualifier, **new**, which indicates that the map should not exist already on disk. As in stream variable declarations, the location is a UNIX path expression for the location of the data, in this case the map, on-disk.

Unlike stream data, maps are write-able. To promote data safety, Hancock provides three qualifiers for maps (and other persistent type declarations): **const**, **exists**, and **new**. These qualifiers help protect data by conveying the programmer's expectations about how a map will be used. The runtime system generates an error and halts the program when a property specified by a qualifier is violated at runtime. The **const** qualifier indicates that the map

[11] Hancock also supports a mechanism, called `sig_main`, to use command-line arguments to connect persistent data structures to their in-memory variable names.

```
event line_end(long long origin) {
    profile_t op = ct<:origin:>;
    ct<:origin:> = update(op,p);
}
```

Fig. 8. Map operations.

will not be updated during the computation, regardless of the program variable used to manipulate the structure. Unlike the **const** qualifier of plain C, the **const** qualifier in Hancock's initializing declarations cannot be cast away. The **exists** qualifier guards against inadvertently starting a computation from an empty persistent structure, producing the wrong data and wasting significant time in the process. Its presence indicates that the specified map must exist on disk at runtime. The qualifiers **const** and **exists** can be combined to indicate that the map will not be updated and must exist on disk. The **new** qualifier protects against inadvertently using data from an existing structure instead of creating a new one, potentially destroying valuable data. This qualifier indicates that the specified map must not exist at runtime. If the programmer does not specify either **exists** or **new**, then the runtime system assumes that the programmer does not care whether the map exists. In this case, the runtime system uses the existing map if one exists or creates a new one otherwise.

Hancock's qualifiers help programmers protect their data, but the qualifiers would have been more effective had maps been read-only by default with a **writable** qualifier instead of **const**. This alternative design would have protected data by default.

4.5.3 *Map Operations.* Because the signature programs we studied required only a few operations, Hancock provides only limited operations for manipulating individual items in maps: read, write, remove, test active key, and test key range. Hancock's indexing operator, written `<: ... :>`, can be used as an r-value (for reading) or as an l-value (for writing). Hancock's map remove operator, written `\<: ... :>`, removes an item from the map. Hancock's test active key operator, written `@<: ... :>`, queries a map to determine whether the key is active, that is, whether the key has a value associated with it in the map. It is a runtime error to pass an out-of-range key to any of these operations. To avoid triggering this runtime error, programmers can test if a given key is within the static key range of a map before performing other operations. In particular, the test key range operator, written `m?<: k :>`, returns true if the key `k` is within the declared key range for map `m`.

As an example, Figure 8 contains the code for responding to `line_end` events in the Outgoing phase of the Cell Tower application. It first reads the value for the key `origin` from map `m`, updates that value, and then writes the updated value into map `m` for the key `origin`.

In addition to the operations that manipulate individual items, Hancock also provides an operation that converts a map into a transaction stream with one stream entry for each active key from within a given range. The Hancock

expression

```
ct[startKeyExpr..stopKeyExpr]
```

generates a stream of active keys from map `ct` that fall between the values of the expressions `startKeyExpr` and `stopKeyExpr` inclusive.

Such map-to-stream expressions make it easy to write programs that update every value in a map or that evaluate queries that characterize the data. For example, the Cell Tower application uses map iteration to age each cell tower `count` every day, a process that allows new information to replace old information gradually. An auxiliary program for the Cell Tower application might generate a histogram that plots the number of MPNs with each possible number of recently-used cell towers by iterating over the map and assigning each active MPN to a histogram bucket based on its profile.

To support the coarse-grained rollback mechanism used to provide fault tolerance for signature programs, Hancock supports a generic persistent copy operator `:=:` for maps and the other persistent structures. The statement

```
newCellData :=: oldCellData
```

replicates the persistent data structure associated with variable `oldCellData` and associates `newCellData` with the copy. The variable `newCellData` should already have been connected to an on-disk location. The implementation ensures that any in-memory changes to `oldCellData` made prior to the copy operation are materialized in the copy. Changes made to `oldCellData` after the copy are not materialized in the copy.

4.6 Groups of Persistent Data

Conceptually, Hancock's directories correspond to Unix directories on disk and to (pointers to) C structs in memory. Each field corresponds to a file on disk (or a subdirectory in the case of a field that is itself a Hancock directory) and to a field in a C struct in memory. This structure allows us to use Unix tools to manipulate and examine the various components of a Hancock directory on disk, while using familiar C syntax to manipulate them in memory. The in-memory structure is initialized whenever an initializing declaration for the directory is executed. The in-memory representation is initialized from the on-disk representation if one exists, or from default values if not. Changes to the in-memory structure are propagated automatically to disk before the associated program terminates (precisely when is unspecified).

The following declaration defines the directory type `cellTower_d` used in the Cell Tower application:

```
directory cellTower_d {
  pHashTable_p ctHashTable;
  compressionTable_p ctab;
  cellTower_m ctOut;
  cellTower_m ctIn;
  char *lastUpdated default "never";
};
```

This declaration introduces a new type `cellTower_d` that groups together a persistent hash table (the type `pHashTable_p` is a pickle defined in Section 4.7), a compression table, two cell tower maps (one for outgoing calls, another for incoming calls),[12] and a string denoting the date of the last update to the data in the directory. It also introduces names for the group members (`ctHashTable`, `ctab`, `ctOut`, `ctIn`, and `lastUpdated`).

For each Hancock directory type, the Hancock compiler generates the C struct to store the in-memory representation of the directory. For the `cellTower_d` example, each value is represented as a pointer to a C `struct` defined as follows:

```
struct cellTowerDirectoryRep {
     pHashTable_p ctHashTable;
     compressionTable_p ctab;
     cellTower_m ctOut;
     cellTower_m ctIn;
     char *lastUpdated;
};
```

In general, directory fields may have Hancock `map`, `pickle`, or `directory` types or almost any C type of statically known size, including arrays and structs.[13] Hancock also permits fields with type `char *`, which it treats as strings.

Each field has its own in-memory and on-disk representation, which is determined by the type of the field. The representations used for fields with Hancock map, pickle, and directory types are determined by the underlying type. For fields with standard C types, Hancock determines the representations. In memory, Hancock uses the standard C representations; on disk, it uses an ASCII-based representation, which allows programmers to inspect the values easily. If Hancock's representation is not efficient enough for a particular field, say for a large integer vector, then programmers can use a pickle to customize the representation. This flexibility allows programmers to build their applications quickly using Hancock's built-in persistent types, while making it easy to switch to a more efficient representation later.

Fields with standard C types can specify an optional default value, which Hancock uses to initialize the field when creating a new directory. The type of the default must match the type of the field. For example, the `lastUpdated` field is declared to use the string `"never"` as a default.

4.6.1 *Discussion.* The design of Hancock's directory mechanism is a compromise between efficiency and flexibility. UNIX directories provided a convenient grouping and naming mechanism that was familiar to Hancock programmers and easy to manipulate using standard tools. Using one file per field makes it easy to integrate other Hancock types, such as maps and pickles, into directories because the implementation of those types need not understand anything about directories. Finally, providing an ASCII representation for standard C

[12] Separating the inbound and outbound cell usage into two maps is a design decision. An alternative design might have one map with an array of two `profile_t`'s as its signature type.

[13] Hancock does not support `long doubles`.

```
typedef struct {
  hashTable *ht;
  char readOnly;
  char updated;
} pht_rep;

int initPHT(Sfio_t *fp, pht_rep *data, char readOnly);
int flushPHT(Sfio_t *fp, pht_rep *data, char close);

pickle pHashTable_p {initPHT => pht_rep => flushPHT};
```

Fig. 9. Pickle type for persistent hash tables.

types allows programmers to add extra information to their data easily and quickly, while retaining the flexibility to use a more efficient representation should that be required.

4.7 User-Defined Persistent Data

Directories provide a simple form of persistence for standard C types, while maps provide an efficient mechanism for associating data with keys. But as we noted earlier, it is impossible to know in advance all the persistent data structures that might be needed by Hancock applications. We designed Hancock's pickle mechanism to provide programmers with a way to design custom persistent data structures that integrate smoothly with the rest of Hancock's persistent data system, in particular, with directories and initializing declarations. We also designed pickles to support large structures, which requires that they permit data structures that reside partially in-memory and partially on-disk and that they introduce minimal overhead. To ensure that pickles were sufficiently expressive, we set as a goal that programmers could code maps as pickles.

To define a pickle, the programmer uses the `pickle` declaration to specify the name of the new pickle type, the in-memory representation of the type, and a pair of functions: one for initializing the in-memory representation from the on-disk representation and another for writing in-memory changes back out to disk. Figure 9 shows the code for the `pHashTable_p` persistent hash table.

The initialization function fills the in-memory representation from the contents of the file containing the on-disk representation. The runtime system opens the file and verifies any qualifiers that were specified in the declaration. The initialization function is responsible for verifying that the *contents* of the file have the expected format and raising an error if they do not. If the file is empty, the function must either generate initial values or raise an error, as appropriate.

To permit the programmer to define partially memory-resident structures, the runtime system does not close the file associated with a pickle until it deallocates the pickle. Consequently, the initialization function for the pickle

can save the file pointer in the in-memory representation of the pickle. Whenever an operation on the pickle requires data not currently in memory, the pickle implementation can use the file pointer to load the appropriate data.

The write function flushes in-memory data to disk so that this data may be preserved across program executions. In general, the runtime system calls this function when closing the pickle prior to program termination. To protect persistent data, however, the runtime system never calls this function when the associated pickle is declared to be `const` in its initializing declaration.

Hancock provides only the copy operation on pickles, which it implements by calling the write function of the source pickle to flush any in-memory changes to disk, copying the associated file on disk, and then using the read operation to initialize the destination pickle from the data on-disk.

The pickle designer is responsible for providing all other operations on the pickle by writing functions that take values of the pickle type (i.e., pointers to `pht_rep structs`) as arguments. For example, the code

```
int insertPHT(pHashTable_p pht, char *s, unsigned int *h){
  if (!pht->readOnly){
    pht->updated = 1;
    return hashInsert(pht->ht, s, h);
  } else
    return READ_ONLY_ERROR;
}
```

inserts a new value into a persistent hash table and stores the associated hash code in variable `h`.

4.7.1 *Discussion.* Hancock's pickle model is very lightweight. It simply provides a mechanism to guarantee that a data structure is initialized before the first use and flushed back to disk on program termination. These guarantees along with the ability to package pickles in directories make them quite useful while introducing minimal overhead.

Hancock's pickles were designed to allow programmers to implement data structures, such as maps, that should not be materialized completely in-memory upon initialization. This approach allows applications that use only a small part of a data structure to avoid paying the cost of rebuilding the whole data structure in memory. Byte-stream pickles have no provision for partially materializing a data structure.

5. PARAMETERIZED TYPES

We implemented and used for several years the streams, maps, directories, and pickles described in the previous sections. While working with these programs, we identified several problems with the design described thus far.

First, it was not uncommon for the function constituents of the various types to rely on global variables to access and update information. For example, the stream declaration that maintains a hash table mapping between cell tower names and fixed-width representations of those names had to access the hash

table as a global variable. Similarly, default functions for maps used global variables to access their back-up data sources; compression functions used global compression tables. This practice suffered from the usual weaknesses of abusing global variables.

Second, we saw a proliferation of types that differed only in small ways. For example, as data analysts designed larger signatures in Hancock, it became desirable to parallelize some applications by partitioning the maps into smaller maps with contiguous key ranges. These partitioned maps might be accessed separately, for updating for example, or they might be accessed as a group, for selection purposes.

Because key ranges are a part of map types, this practice led to a number of map type declarations that differed only in their key ranges. Figure 10 contains a collection of map types (`part0_m` through `part3_m`) that illustrates this problem. The programmer could choose to use a single type for the partitions, such as the type `part_overlay_m`, but this approach has two problems. First, it would waste space because the map representation contains an index that uses space proportional to the size of the key range. Second, it would lose some of the benefits of Hancock's dynamic key range checks because keys that are valid for the full range would pass the check even though the intended range for a given partition might not include the key. A simple bug that miscalculated the partition associated with a key would be difficult to detect.

A related problem is that the programmer had to choose between replicating auxiliary functions that manipulated the partitioned map type or using an overlay type and casting to work around the limits of our type system.

In response to these observations, we added a parameterization mechanism to Hancock's stream, map, directory, and pickle abstractions. When declaring these types, programmers can specify a list of formal parameters that are in scope throughout the type declaration. The corresponding actual parameters are supplied in initializing declarations.

For example, the following refinement of the `wireless_s` stream takes a persistent hash table as a parameter:

```
stream wireless_s(pHashTable_p pht) {
  getValidCall(: pht :)  : Sfio_t => wcrLog_t;
};
```

The notation "`(: pht :)`"[14] indicates that expression `pht` should be passed to the function `getValidCall`.

Assuming that `my_pht` has type pHashTable_p, the following initializing declaration connects `my_pht` with the stream `calls`:

```
wireless_s calls(: my_pht :) = "data/call-detail.current"
```

All invocations of the translation function `getValidCall` for the stream `calls` receive as a parameter the persistent hash table `my_pht`. The translation function uses this table to convert the variable-width cell tower names into integers instead of making a reference to a global variable.

[14]We use the "`(:...:)`" notation for type parameter actuals to avoid a parsing ambiguity.

```
#define PARTSIZE MAXKEY/4

map part0_m {
  key (0..PARTSIZE);
  value int;
  default 0;
}

map part1_m {
  key (PARTSIZE+1..2*PARTSIZE);
  value int;
  default 0;
}

map part2_m {
  key (2*PARTSIZE+1..3*PARTSIZE);
  value int;
  default 0;
}

map part3_m {
  key (3*PARTSIZE+1..MAXKEY);
  value int;
  default 0;
}

int countActive0(part1_m m) { ... }
int countActive1(part2_m m) { ... }
int countActive2(part3_m m) { ... }
int countActive3(part4_m m) { ... }
```

(a) Example of partitioned map types and auxiliary functions.

```
map part_overlay_m {
  key (0..MAXKEY)
  value int;
  default 0;
}

int countActive(part_overlay_m m) { ... }
```

(b) Example overlay type and auxiliary function.

Fig. 10. Partitioned map types and auxiliary functions.

Using the same notation, programmers can parameterize maps, directories, and pickles. For example, the following map is parameterized by a compression table pickle and a pair of `long long` values that specify the upper and lower bounds for the key space for the map:

```
map cellTower_m(compressionTable_p ctab,
                long long minKey, long long maxKey)  {
  key (minKey .. maxKey);
  split (10000, 100);
  value profile_t;
  default CTM_DEFAULT;
  compress ctSqueeze(:ctab:);
  decompress ctUnsqueeze(:ctab:);
};
```

In general, parameters may be used in the expressions specifying the key ranges, the splits, and expression defaults. They may also be passed as extra arguments to default functions and to compression and decompression functions.

Because directories serve to connect related elements, we allow earlier fields in a directory to be in-scope for later fields. This cascading means we can tightly couple related persistent structures. For example, the following directory passes its compression table ctab to the cell tower maps:

```
directory cellTower_d(char * defaultString) {
  pHashTable_p        ctHashTable;
  compressionTable_p  ctab;
  cellTower_m         outMap(:ctab, MINKEY, MAXKEY:);
  cellTower_m         inMap(:ctab, MINKEY, MAXKEY:);
  char *              lastUpdated default defaultString;
};
```

This coupling ensures that after an initializing declaration such as

```
cellTower_d cellTowerData(:"never":) = "data/cellTower.current";
```

the compression table ctab has been initialized and passed to the two maps.

Type parameters are associated with program variables using initializing declarations, but once data on disk has been connected to a program variable, the runtime system supplies the parameters when needed. As a result, the programmer uses the unparameterized version of the type for function argument types. For example, a function that takes a cell tower directory as an argument simply uses the type cellTower_d. This design allows Hancock programmers to get the benefit of the parameters—tighter dynamic checks and fewer global variables—without having to define multiple instances of the same function.

Hancock's type parameters share some properties with statically sized arrays in C. In C, a programmer can declare an integer array of size 10 and then pass the array to a function that expects a pointer to an integer. This allows the programmer to use one function for many different array sizes just as using the unparameterized version of the Hancock type allows the programmer to pass maps with many different key ranges to a single function. One major difference between this mechanism in C and Hancock's type parameters is that type parameters in Hancock are sticky, that is, a map does not lose its key-range information when it is passed to a function.

To summarize, Hancock's type parameters allow a programmer to associate information with data dynamically when program variables are initialized.

Once the connection between a variable and data is made, the programmer can use the unparameterized version of the type because this information "sticks" with the data and is passed by the runtime system to the appropriate dynamic checks or user-supplied functions as needed.

6. IMPLEMENTATION

In this section, we give a brief overview of our implementation of Hancock. The Hancock compiler translates Hancock code into C. It invokes a platform-dependent C compiler to convert the resulting C code into machine code. Finally, it links this code to the Hancock runtime system to produce an executable.

To implement the translation, we modified CKIT [Chandra et al. 1999], a C-to-C translator written in ML that was designed to facilitate the implementation of C-based domain-specific languages. Our modifications include extending CKIT's YACC-based grammar with Hancock-related productions, using hooks in CKIT's parse-tree representation to specify representations for Hancock forms, and providing implementations for call-back functions that translate the extended parse tree into CKIT's abstract-syntax for C. Currently, we do not use CKIT's hooks for extending its abstract syntax. During the translation from the extended parse tree to the abstract syntax, we typecheck the various Hancock forms. After translation, we use CKIT's pretty-printer to produce C code.

The Hancock runtime system, which is written in C, manages the runtime aspects of Hancock's pickles, maps, directories, and streams. It locates and opens each of the files and directories associated with such values. It allocates the space to hold their in-memory representations. It calls type-specific read functions to initialize these representations and write functions to flush changes to disk. For pickles, the programmer supplies these read and write functions. For maps, the runtime system itself provides them. For directories, the compiler constructs them from directory declarations. For streams, the programmer supplies the read function, but no write function is necessary because streams are (currently) read-only.

In addition to the above roles, the runtime system provides the implementation for maps, which are essentially multilevel tables [Gupta et al. 1998; Lampson et al. 1999; Huang et al. 1999]. The compiler translates the source-level map operations described in Section 4.5.3 into the corresponding operations in the runtime system. An earlier paper provides a more detailed discussion of Hancock's map implementation [Fisher et al. 2002].

7. EXPERIENCES

In this section, we describe our experiences with Hancock programs in practice. We rewrote the original domestic long-distance signature programs in Hancock. These Hancock programs have been running every day in production for the past four years. Unlike the original C programs, the Hancock versions are easy to check periodically to ensure that they are compliant with current federal legislation because the Hancock programs are shorter and better organized. The Hancock programs are also easy to update in response to changes in the transaction data. For example, updating the programs to make them aware

that area codes 866 and 877 had become toll-free required changing only two lines of one header file and recompiling the programs. Currently, all of AT&T's domestic long-distance signatures are written in Hancock.

In addition to supporting the original applications better, Hancock's domain-specific abstractions and improved performance[15] enabled data analysts to craft new kinds of signatures. When analysts used C directly, they were able to store only two- to four-byte signatures. Because of this limited space, they had to make very rough approximations. Although this rough data was very useful for certain kinds of marketing applications, it was not suitable for many other kinds of applications, notably fraud detection. Hancock's abstractions and their efficient performance enabled analysts to build signatures containing over 100 bytes. With that level of detail, analysts could store sufficiently precise information to enable new applications previously thought to be infeasible.

Most existing Hancock programs manipulate long-distance call-detail data because the data analysts we work with focus on that domain. However, nothing in Hancock is specific to this domain; Hancock gives programmers full control over the description of their data sources. People have used Hancock to analyze data from various sources: wireless call records, calling-card call records, telephone numbers, TCPDUMP data, IP addresses, and even referee reports.

8. FUTURE WORK

Although the current design of Hancock works well for the most part, there are three areas that we would like to improve:

—*Variable-width data*. We would like to extend the design of Hancock to facilitate signature computations over variable-width data sources. For example, we would like to be able to build applications that process a stream of variable-width web-log records to compute signatures for the URLs mentioned in the stream. Currently, we can program these applications by using parameterization and a persistent hash table to convert the variable-width data into a fixed-width representation, but the resulting program structure is not as clean as we would like. A further goal would be to support variable-width signatures within maps. This problem seems harder, as it raises the question of memory management. Is the programmer or the runtime system responsible for managing the memory returned in response to map read operations?

—*Built-in compression*. Currently, Hancock's built-in compression functions do not provide good compression. They are fast, but do nothing beyond putting the data into a machine-independent format. Conversely, the programmer can supply compression functions with a much better compression ratio, but writing such functions is difficult and error-prone. We would like to investigate techniques for providing better automatic compression.

—*Declarative stream descriptions*. One of the most tedious parts of writing a Hancock application is in writing the stream description (if one does not

[15]Isolating the map implementation from its uses allowed us to experiment with various implementations. This experimentation led to a more compact representation.

already exist), particularly if the stream is in ASCII. The parsing code typically uses C's `scanf` function, with its `%c`, `%d`, `%u`, etc. conversion operations. Such code is slow and difficult to get right, and often contains errors that are machine-dependent, for example, errors that manifest themselves only on big-endian machines, or only when word sizes are 64-bits. Also, it is tedious to handle all the error cases, so often the programs only handle "correct" data, evolving over time to handle the error cases that have been seen. We would like to provide a way for programmers to describe the data declaratively and then use a tool to generate a Hancock stream description (and potentially a multi-union and an event detection function).

9. CONCLUSIONS

The volume of data in many transactional data streams is overwhelming. But this challenge provides an opportunity for data mining research to enter a new area. We believe that Hancock is a valuable tool for exploiting this opportunity.

Hancock has allowed us to improve our application base by replacing hard-to-maintain, hand-written C code with disciplined Hancock code. Because Hancock provides high-level, domain-specific abstractions, Hancock programs are easier to read and maintain than the earlier C programs. By careful design, these abstractions have efficient implementations, which allow Hancock programs to preserve the execution speed and data efficiency of the earlier C programs. Hancock gave domain experts the confidence to attack more challenging problems because it allowed them to concentrate on *what* to compute without worrying about *how* to manage the volume of data.

Hancock is publicly available for noncommercial use at

`www.research.att.com/projects/hancock`

We hope that others will join us in exploring the language and its functionality.

ACKNOWLEDGMENTS

We would like to thank Dan Bonachea for his contributions to the design of the earliest version of Hancock, Glenn Fowler, John Linderman, and Phong Vo for their assistance with the run-time system, Karin Högstedt for her work on implementing maps, Nevin Heintze and Dino Oliva for their help in using CKIT, Dan Suciu and Mary Fernandez for discussions that helped refine the stream model, and John Reppy for producing the pictures.

REFERENCES

AHO, A., KERNIGHAN, B., AND WEINBERGER, P. 1979. AWK—A pattern scanning and processing language. *Softw. Pract. Exp. 9*, 4 (April), 267–279.

APPEL, A. W. 1990. A runtime system. *Lisp Symbol. Computat. 4*, 3, 343–380.

ATKINSON, M., DAYNES, L., JORDAN, M., PRINTEZIS, T., AND SPENCE, S. 1996. An orthogonally persistent Java. *ACM SIGMOD Rec. 25*, 4.

BABCOCK, B., BABU, S., DATA, M., MOTWANI, R., AND WIDOM, J. 2002. Models and issues in data stream systems. In *Proceedings of the 2002 ACM Symposium on Principles of Database Systems* (PODS 2002). See the Stream Project homepage, `www-db.stanford.edu/stream`, for a complete list of papers.

BELANGER, D., CHURCH, K., AND HUME, A. 1999. Virtual data warehousing, data publishing, and call detail. In *Processings of Databases in Telecommunications 1999, International Workshop.* Also appears in Lecture Notes in Computer Science, vol. 1819, Springer-Verlag, Berlin, Germany, 106–117.

BONACHEA, D., FISHER, K., ROGERS, A., AND SMITH, F. 1999. Hancock: A language for processing very large-scale data. In *USENIX 2nd Conference on Domain-Specific Languages*. USENIX Association, Berkeley, CA, 163–176.

BURGE, P. AND SHAWE-TAYLOR, J. 1996. Frameworks for fraud detection in mobile telecommunications networks. In *Proceedings of the Fourth Annual Mobile and Personal Communications Seminar*. University of Limerick, Limerick, Ireland.

CARNEY, D., CETINEMEL, U., CHERNIACK, M., CONVEY, C., LEE, S., SEIDMAN, G., STONEBRAKER, M., TATBUL, N., AND ZDONIK, S. 2002. Monitoring streams—a new class of data management applications. In *Proceedings of the 28th VLDB Conference*. See the Aurora Project homepage, `www.cs.brown.edu/research/aurora/main.html`, for a complete list of papers.

CHANDRA, S., HEINTZE, N., MACQUEEN, D., OLIVA, D., AND SIFF, M. 1999. Pre-release of C-frontend library for SML/NJ. See `cm.bell-labs.com/cm/cs/what/smlnj`.

CHANDRASEKARAN, S. AND FRANKLIN, M. J. 2002. Streaming queries over streaming data. In *Proceedings of the 28th VLDB Conference.*

CORTES, C., FISHER, K., PREGIBON, D., ROGERS, A., AND SMITH, F. 2000. Hancock: A language for extracting signatures from data streams. In *Proceedings of the Sixth International Conference on Knowledge Discovery and Data Mining*. 9–17.

CORTES, C. AND PREGIBON, D. 1998. Giga mining. In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining.*

CORTES, C. AND PREGIBON, D. 1999. Information mining platform: An infrastructure for KDD rapid deployment. In *Proceedings of the Fifth International Conference on Knowledge Discovery and Data Mining.*

DENNING, D. E. 1987. An intrusion-detection model. *IEEE Trans. Softw. Eng. 13*, 2, 222–232.

FAWCETT, T. AND PROVOST, F. 1997. Adaptive fraud detection. *Data Min. Knowl. Disc. 1*, 291–316.

FISHER, K., GOODALL, C., HOGSTEDT, K., AND ROGERS, A. 2002. An application-specific database. In *Proceedings of 8th Biennial Workshop on Data Bases and Programming Languages* (DBPL '01). Lecture-Notes in Computer Science, vol. 2397. Springer-Verlag, Berlin, Germany, 213–227.

GUPTA, P., LIN, S., AND MCKEOWN, M. 1998. Routing lookups in hardware and memory access speeds. In *Proceedings of the 17th Annual Joint Conference of the IEEE Computer and Communications Societies*. Vol. 3, 1240–1247.

HELLERSTEIN, J., FRANKLIN, M., CHANDRASEKARAN, S., DESHPANDE, A., HILDRUM, K., MADDEN, S., RAMAN, V., AND SHAH, M. 2000. Adaptive query processing: Technology in evolution. *IEEE Data Eng. Bull.*, 7–18. See the Telegraph Project homepage, `telegraph.cs.berkley.edu`, for a complete list of papers.

HUANG, N.-F., ZHAO, S.-M., PAN, J.-Y., AND SU, C.-A. 1999. A fast IP routing lookup scheme for gigabit switching routers. In *Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies*. Vol. 3, 1429–1436.

JACKSON, M. A. 1975. *Principles of Program Design*. Academic Press, New York, NY.

KNASMÜLLER, M. 1997. Adding persistence to the Oberon system. In *Proceedings of the Joint Modular Languages Conference 97.*

LAMPSON, B., SRINIVASAN, V., AND VARGHESE, G. 1999. IP lookups using multiway and multicolumn search. *IEEE/ACM Trans. Network. 7*, 3, 324–334.

LISKOV, B., CASTRO, M., SHRIRA, L., AND ADYA, A. 1999. Providing persistent objects in distributed systems. In *Proceedings of the 13th European Conference on Object-Oriented Programming* (ECOOP '99).

NELSON, G., Ed. 1991. *Systems Programming with Modula-3*. Prentice Hall, Englewood, Cliffs, NJ.

RIGGS, R., WALDO, J., WOLLRATH, A., AND BHARAT, K. 1996. Pickling state in the Java system. In *Proceedings of the USENIX 1996 Conference on Object-Oriented Technologies* (COOTS).

SAX PROJECT. 2002. SAX home page. See `www.saxproject.org`.

SIGMOD. 2002. *Proceedings of the 21st ACM SIGMOD International Conference on Management of Data.*

SULLIVAN, M. AND HEYBEY, A. 1998. Tribeca: A system for managing large databases of network traffic. In *Proceedings of the USENIX Annual Technical Conference (No. 98)*.
VAN ROSSUM, G. 2001. Python library reference. See `python.sourceforge.net/devel-docs/lib/lib.html`.
VLDB. 2002. *Proceedings of the 28th International Conference on Very Large Data Bases*.
WANG, D. C. 1998. *The asdlGen Reference Manual*. See `www.cs.princeton.edu/zephyr/ASDL`.

Received April 2002; revised January 2003; accepted June 2003