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, *e.g.* credit-card numbers or IP addresses, use the associated services.

Over the past five 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 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: []:

General Terms:

Additional Key Words and Phrases:

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 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?

Author's address: AT&T Labs, Shannon Laboratory, 180 Park Avenue, Florham Park, NJ 07932 ({corinna,kfisher,daryl,amr}@research.att.com). Frederick Smith's current address: The Mathworks, 3 Apple Hill Drive, Natick, MA 01760 (fsmith@mathworks.com).

Submitted for publication

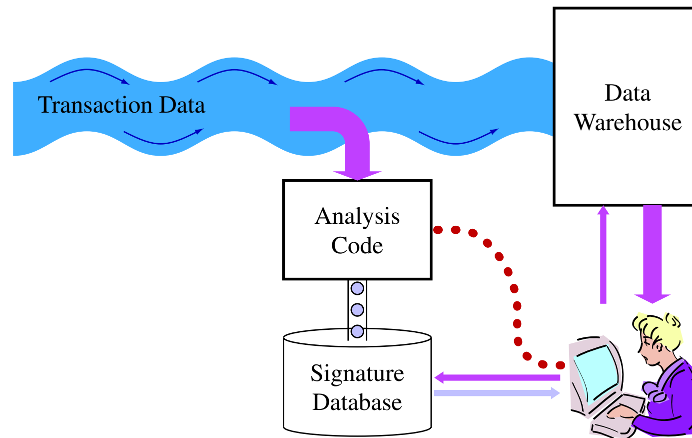


Fig. 1. Typical use of a fraud signature.

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]. Signatures evolve over time in response to new transactions. 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.

For example, a signature may capture characteristics of international calling behavior such as the average daily number of international calls, call duration, and call destination (*e.g.* region of the world). An application to detect international toll fraud can use these signatures to compare current calling behavior for each number. Figure 1 depicts how such a fraud system might work: a program taps the call-detail stream as it goes into a data warehouse and uses the tapped data to update the signatures of the telephone numbers with calls in the stream. If the new calls are inconsistent with the signature of a given telephone number, the system sends an alert to a network security representative. The representative then retrieves the relevant call-detail records from the warehouse to determine if the alert warrants further action. If the new calls are consistent with the account signature, they are used to update the signature to allow a smooth evolution of calling behavior. Since most accounts exhibit consistent behavior through time, an overwhelming majority of calls are incorporated into the appropriate signatures and stored in the warehouse without being examined manually.

The precise composition of a signature varies depending on the data stream, the intended and anticipated applications. For example, the signature for a phone number might contain directly measurable features such as the time when most telephone calls are placed from that number, the regions to which those calls are placed, and the time of the last call. The signature might also contain derived information such as the degree to which the calling

pattern from the number is "business-like" [Cortes and Pregibon 1998]. A signature for an IP address might contain direct features such as the number of packets it sends per day or derived features such as whether it looks like a client, a server, or a proxy.

Researchers at AT&T Labs analyze data streams from the telecommunications domain. The initial applications processed roughly five million (M) international call-detail records per day, generated by approximately 12M accounts. Subsequently, we have tackled larger and larger data streams, including the complete AT&T long distance data stream, which consists of approximately 300M records from 100M accounts per day.

Local anecdotal evidence suggested that traditional databases were not suited to the task of building and maintaining such signatures because they could not perform adequately at the necessary scale. In particular, traditional databases had difficulty loading the daily transactions in a timely fashion[Belanger et al. 1999].

Consequently, data analysts at AT&T did not use a database to implement the initial signature applications. Instead, they wrote the initial programs in C using an *ad hoc* representation for the signature data that was custom designed for each program. These representations were indexed only by the identifiers associated with the entities being tracked, while daily snapshots of the signature collections provided a course-grained roll-back mechanism. These implementations were well-tailored for the desired class of applications, and they met the necessary performance requirements.

Despite the success of the data produced by the initial signature programs, data analysts were not satisfied with the programs themselves because they were hard to write and maintain. Although the per-entity code was conceptually very simple (count the number of international calls, *etc.*), the code to manage the volume of data and ensure good performance was not. This infrastructure code swamped the per-entity code, making programs difficult to write and maintain. Maintenance is important in this domain because the analysts have to review the programs periodically to ensure that they comply with changing federal regulations and the analysts may want to enhance some of the signature features. Although analysts had written a handful of successful applications using this technique, the complexity of the programs dissuaded them from writing new and more challenging ones. Hence although analysts thought signatures very useful, they were at a loss as to how to compute them: traditional databases were too slow, and the *ad hoc* approaches were too complex to write and maintain.

In response to this problem, we designed and implemented Hancock, a C-based domain-specific programming language for computing signatures. By design, the language makes time- and space-efficient signature programs easy to read and to write, independent of the quantity of data involved. Because Hancock manages scaling issues, it allows data analysts to design new signatures quickly and to verify compliance with federal regulations easily.

This paper discusses the computational difficulties in writing efficient signature code for massive data streams, shows how Hancock alleviates these difficulties, and discusses the motivations behind the Hancock design. In an early paper about Hancock[Bonachea et al. 1999], we presented a preliminary version of the language that handled only one particular form of data (call-detail records from AT&T's long distance network). Another paper[Fisher et al. 2001] discussed the data structures used to implement maps, a persistent data structure for associating data with keys, and compared the performance of maps with that of a database. This paper, which expands upon Cortes *et. al.*[2000], describes a version of Hancock that supports a rich set of persistent data types and handles arbitrary data streams.

We have organized the rest of the paper as follows. Section 2 presents a signature application that we use as a running example to explain the features of Hancock. We introduce in Section 3 the various elements of Hancock's persistent data system: *directories*, *maps*, and *pickles*, *initializing declarations*, and *persistent type parameterization*. In Section 4 we describe Hancock's *view* mechanism, which helps programmers manage multiple representations of a single piece of data. We present Hancock's abstractions for describing and processing transaction streams in Section 5. Section 6 introduces `sig_main`, Hancock's version of C's `main` function. This construct provides automatic command-line processing and facilitates connecting persistent data structures to their in-memory representations. We briefly sketch our implementation of Hancock in Section 7 and describe our experiences with the language in Section 8. We summarize our design in Section 9 and conclude in Section 10.

2. RUNNING EXAMPLE: THE CELL TOWER APPLICATION

In this section, we introduce an example that we use throughout the paper to 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
— Primary cell tower (originating)
— Secondary cell tower (originating)
— Primary cell tower (dialed)
— Secondary 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 originating primary tower captures the first tower used to carry the call. If the phone moved significantly, then the secondary tower captures the last 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 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, *e.g.*,

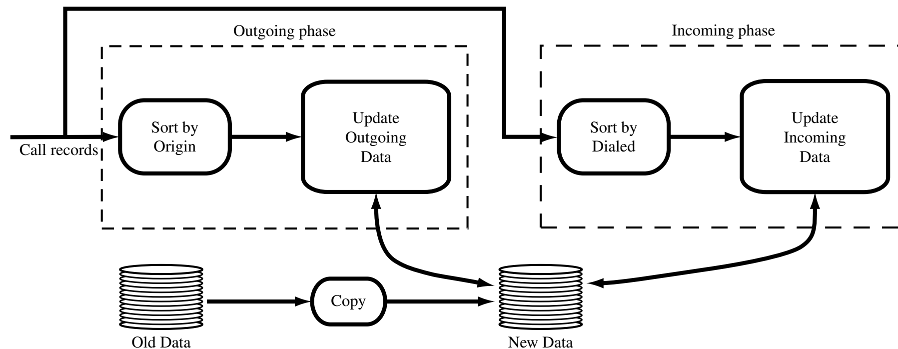


Fig. 2. High-level architecture of signature computations. The processing typically consists of several phases, each sorting the data in a different order and updating a different part of the signatures.

according to the originating phone number for one pass and according to the dialed phone number for another. 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 easily.

In summary, this application requires three abstractions for representing persistent data: one to associate signatures with keys (*cf.* Section 3.5), a second to describe the persistent hash table (*cf.* Section 3.6), and a third to group the most-recent-update date, the signature collection, and the persistent hash table (*cf.* Section 3.4). It also needs abstractions for describing and processing stream data (*cf.* Section 5). Finally, it needs abstractions for expressing high-level control flow (*cf.* Section 6). We discuss how Hancock provides the abstractions necessary for this application in the next few sections.

3. PERSISTENT DATA

The Cell Tower application highlights the need for persistent data types in Hancock. In this section, we start by reviewing related work and discussing the requirements for supporting persistent data in Hancock. We then present the details of Hancock's persistent data mechanisms.

3.1 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

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 meet the performance requirements of Hancock applications[Belanger et al. 1999; Fisher et al. 2001; Sullivan and Heybey 1998]. 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.[1] 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, we adopted a more explicit technique for persistence in Hancock.

3.2 Design Requirements

The design of Hancock's persistent data mechanisms has evolved from a single mechanism for supporting signature collections to a much broader collection, which we call a *persistent data system (PDS)*. The overarching goals of the design were efficiency and data safety.

Space and time efficiency are crucial concerns because of the scale of the data in our stream-analysis applications. Typical applications analyze streams of 5GB to 10GB of data per day. They use this information to update persistent analysis data daily. This data ranges in size from half a gigabyte to 11GBs. Hancock's target audience would never have used Hancock without an efficient implementation.

The scale of the data also makes data safety crucial because it makes detecting errors extremely difficult. How does one determine that all the data in a 10GB 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.

From these overriding goals, we derived several other goals: *ease-of-use*, *flexibility*, and *transparency*. We wanted Hancock's PDS to allow data analysts to try out ideas at scale with as little work as possible. Among other things, this goal called for a built-in mechanism for making C data types persistent. But we also knew that the system had to be flexible, because we cannot predict *a priori* all the persistent data structures that would be needed. Consequently, Hancock must allow programmers to leverage application-specific knowledge to improve the performance of their applications and to craft their own persistent types when the built-in types are insufficient. In particular, Hancock must provide a

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

mechanism for supporting data structures, beyond maps, that reside partially in memory and partially on disk. Finally, we wanted to allow programmers to isolate details related to data persistence from uses of that data. We call a system *transparent* if a programmer can ignore the persistent nature of the data once the connection between the in-memory and on-disk representations for a persistent data structure has been made.

With these properties in mind, we now describe Hancock's persistent data system, which includes three persistent data types: *directories*, *maps*, and *pickles*. Directories provide built-in support for persistence for statically-sized C types and a way to group related persistent structures. Maps supply a built-in abstraction for signature collections. Pickles allow programmers to design their own persistent data types. Hancock provides *initializing declarations* to connect program variables with persistent types to their on-disk representations. Finally, Hancock's persistent data types may be parameterized, allowing such types to be specialized at initialization time.

The rest of this section describes Hancock's PDS in detail. We start with an overview of the features common to directories, maps, and pickles. We then discuss the particular details of each of these types.

3.3 Overview

Hancock's persistent data types differ in their details, but they share a common overall structure. In particular, a persistent type must indicate how it should be represented on disk, how it should be represented in-memory, how the type can be customized, how the in-memory representation should be connected to the on-disk representation, and how it should be copied. For each persistent type, Hancock and the programmer work together to make these choices.

To make the ideas in the rest of this section more concrete, we briefly introduce the `directory` declaration form, deferring details about directories to Section 3.4. The following declaration defines the directory type `cellTower_d` used in the Cell Tower application:

```
directory cellTower_d(int level) {
  pht_p ctHashTable;
  compTable_p cTab(:level:);
  cellTower_m ctOut(:ctab:);
  cellTower_m ctIn(:ctab:);
  char *lastUpdated default "never";
};
```

This declaration introduces a new type `cellTower_d` that groups together a persistent hash table, a compression table, two cell tower maps (one for outgoing calls, another for incoming calls),[2] 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`).

3.3.1 *Representation issues.* Every persistent data structure needs both an in-memory representation to access the data during computation and an on-disk representation to store the data between (and perhaps, during) computations. These representations can be determined by the Hancock compiler (as for directories), the Hancock runtime system (as for

[2] 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.

maps), or the programmer (as for pickles). Hancock guarantees that the in-memory data for a persistent data structure is initialized from the on-disk representation before that data can be used and that the on-disk data is consistent with the dynamically-declared type of a variable. It also guarantees that changes to the data structure that might exist only in memory are flushed to disk on normal program termination.[3]

3.3.2 *Customization using type parameters.* Type parameters allow programmers to define a family of related types with one declaration. Such parameters may appear in the expressions or be passed to the functions that are part of persistent type declarations. For example, the `directory` type `cellTower_d` takes an integer parameter, `level`. This parameter is passed as an argument to the nested compression table. Examples of parameter use include defaults for fields with C types in directories or as key ranges or compression tables in maps. Type parameters reduce the number of distinct types that a programmer must specify without sacrificing the data-safety gained from Hancock's static and dynamic type checking.

3.3.3 *Connecting the in-memory and on-disk representations.* Once the programmer has specified a persistent type, variables can be declared to have that type using C syntax: `cellTower_d cellData`. This mechanism alone is not sufficient, however, because the programmer must be able to connect program variables to persistent data on-disk. Hancock provides two related mechanisms to make this connection: initializing declarations and the `sig_main` construct. We defer discussion of `sig_main` to Section 6.

An initializing declaration augments a standard C declaration with additional information, in particular, with qualifiers, actual parameters, and a location. For example, the following declaration:

```
new cellTower_d cellData (: 9 :) = "cellData.current";
```

augments the standard C declaration with a string, `"celltower.current"`, which specifies the location of the on-disk representation; with a qualifier, `new`, which indicates that the directory should not exist already on disk; and an actual parameter, `9`, which provides a value for the formal type parameter `level`.

The supplied location may be any string-valued expression. The runtime system interprets the value of this expression as a path in the machine's file system. If the on-disk representation exists, the runtime system verifies the format of the data and initializes the in-memory representation. If the on-disk representation does not exist, the runtime system creates it, in which case the on-disk and in-memory representations are initialized in type-specific ways.

To promote data safety, Hancock provides three qualifiers for persistent type declarations: `const`, `exists`, and `new`. These qualifiers convey the programmer's expectations about how a persistent data structure 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 persistent data structure 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 indicates that the specified on-disk representation must exist on disk at runtime. The `new` qualifier has the opposite effect; it indicates

[3] Such changes are not flushed to disk on error exits to avoid (further) corrupting persistent data.

that the specified on-disk representation 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 on-disk representation exists. In this case, the runtime system uses the existing representation if one exists or creates a new one otherwise. 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. The `new` qualifier protects against inadvertently using data from an existing structure instead of creating a new one, potentially destroying valuable data.

In addition to supplying a qualifiers and a location, the sample initializing declaration also specifies actual parameters for the variable's type. Hancock's type system ensures that the types of the actual parameters supplied by the programmer match the declared type of the formal parameters in the type declaration. To provide flexibility, programmers are allowed to omit actual parameters from initializing declarations. When omitted, the runtime system attempts to supply appropriate values, either through a default mechanism or from persistent data stored on disk. If it cannot find appropriate values, then the runtime system halts the program with an error.

3.3.4 *Persistent copying.* Each persistent data structure provides a set of operations suitable for that data structure, but Hancock supports a generic persistent copy operator `:=:` for all persistent structures. The statement

```
newCellData :=: oldCellData
```

replicates the persistent data structure associated with variable `newCellData` and associates `newCellData` with the copy. The variable `newCellData` should already have been connected to an on-disk location. The implementation ensures that in-memory changes to `oldCellData` are materialized in the copy.

3.4 Groups of persistent data

The previous section discussed the common structure of Hancock's three persistent data types using directories as an example. This section discusses the details of Hancock's `directory` data type, which provides persistence for statically-sized C types and strings and allows programmers to group related persistent structures.

Directories are represented in memory as pointers to C `structs`, the definitions of which are generated by the compiler from directory declarations. For the `cellTower_d` example, each value is represented as a pointer to a C `struct` defined as follows:

```
struct cellTowerDirectoryRep {
     pht_p ctHashTable;
     compTable_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`.[4] Hancock also permits fields with type `char *`, which it treats as strings.

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

Directories are stored on disk as UNIX directories. Each field of a Hancock directory appears on-disk in the associated UNIX directory as a file (or as a UNIX directory in the case of nested Hancock directories) with the name of the field.

Each field has its own in-memory and on-disk representation, which is determined by the type of the field. We discuss the representations of maps and pickles in Sections 3.5 and 3.6, respectively. For 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 using UNIX tools. 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 prototype 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.

Directory declarations can be parameterized. The declaration for `cellTower_d` takes one parameter, `level`, which it passes to the compression table pickle as an parameter. If an initializing declaration for a parameterized directory fails to supply parameters, the Hancock runtime system attempts to recover the necessary information from disk for C types and as specified by nested pickle, map, and directory types. The runtime system raises an error if it cannot do so.

Default expressions and actual parameters to nested map, pickle, and directory fields may refer to global identifiers, the formal parameters from the containing directory declaration (*e.g.* the actual parameter for `cTab`), or to previously declared fields (*e.g.* the parameter passed to the maps `ctOut` and `ctIn`). Hancock initializes directory fields in order, guaranteeing that earlier fields have well-defined values when later ones are initialized. This property means that the `cTab` field will have the appropriate value before the `ctOut` and `ctIn` maps are initialized. Hancock writes directory fields to disk in the reverse order, which guarantees that all uses of a field will be completed before that field is flushed to disk.

In memory, the programmer manipulates each directory as a pointer to the underlying C `struct`. As an example, the following code:

```
cellTower_d  cellData(: 9 :) = "cellData.current";
cellData->lastUpdated =  "2002.03.04";
```

initializes the directory `cellData` from the UNIX directory `cellData.current`. If the directory exists on disk, then Hancock uses the values stored in that directory in the `cTab` file to initialize the two maps. If the directory does not exist, Hancock creates it (because the user did not specify the `exists` qualifier). By packaging the hash table, the compression table, and the maps in the same directory, we ensure that each time either map is used, it is paired with the same hash table and the same compression table. This property is crucial, as the data in the maps would be meaningless if paired with either the wrong hash table or the wrong compression table. After the declaration, the above code fragment sets the value of the `lastUpdated` field. Before the program terminates, Hancock will flush the new value back to the file `lastUpdated` in the directory `cellData.current`.

3.5 Signature collections

Hancock's original class of applications needs a highly efficient persistent data structure for associating values with keys, so Hancock provides a persistent data type, called maps, to support these applications. We chose to incorporate a specialized data structure instead of a general database interface because our applications have tight space and performance requirements. General databases have a hard time satisfying these requirements because the applications do not display typical access patterns[Belanger et al. 1999; Fisher et al. 2001].

Hancock maps meet the tight performance requirements of our applications 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 is in the map's range of keys, 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.

3.5.1 *Map 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(compTable_p cTab)  {
  key 0..9999999999LL;
  split (10000, 100);
  value profile_t;
  default {{CTD, CTD. CTD, CTD, CTD},
           {0.0, 0.0, 0.0, 0.0, 0.0},
           0.0};
  compress ctSqueeze(:cTab:);
  decompress ctUnsqueeze(:cTab:);
};
```

This declaration defines the map type named `cellTower_m`, which appears twice in the `cellTower_d` directory in Section 3.3. 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`. 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`.

Programmers influence the underlying structure of a map through the `split` specification. The runtime system breaks each key into three pieces: a *block* number, a *stripe* number within that block, and an *entry* number within that stripe. It uses these pieces to index into tables to locate the value associated with each key. Intuitively, blocks serve as the unit of I/O, while stripes serve as the unit of compression. Using the `split` clause, programmers can tune the performance of their applications by controlling the key decomposition for their maps. 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 describes the implementation of maps in detail and discusses efficient key decompositions

for different data access patterns [Fisher et al. 2001]. [5]

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 2, 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*.

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, *i.e.*, 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 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 5.2).

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 contains constant CTD 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.

Hancock allows programmers to specify functions to compress and decompress values before they are stored on disk because programmers can often leverage domain-specific knowledge to produce better compressors than the generic ones that Hancock supplies. The optional compress and decompress clauses name functions that compress (or decompress) a value of the map type. Hancock allows these functions to use variable-width compression schemes because such schemes often yield better compression ratios.

Maps may be parameterized just like directories. Parameters can be used in expressions in the key, split, and (constant) default clauses. Parameters can also be passed as arguments to function defaults and compression functions. The cellTower_m map takes a compression table as a parameter, which is passed as an argument to the compression and decompression functions. Maps can also be parameterized by other maps. Default functions often use such maps as backup sources of data.

When an initializing declaration for a parameterized map does not provide actual parameters, the Hancock runtime system attempts to use information on disk to reconstruct the necessary information. If it cannot do so, it halts the program with a diagnostic error message.

Returning to the example, the C function ctSqueeze has the following prototypes:

```
int ctSqueeze(char set, compTable_p cTab, profile_t *from,
              unsigned char *toSpace, int size);
```

The function takes a compression table (implemented as a pickle type) plus a pointer to a profile_t and returns a collection of bytes (toSpace) and the size of that collection (the return value). The compression table, cTab, is a parameter to the map type. Because programmers are not required to supply parameters in initializing declarations, ctSqueeze needs to know whether the programmer supplied a compression table. Hancock passes this information to the function with a flag (passed as the first parameter to the

[5] 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.

```
void manipulateMap(cellTower_m ct){
  long long mpn, mpn1;
  profile_t oldProfile, newProfile;

  oldProfile = ct<:mpn:>;              /* Read mpn's data */
    ⋮
  newProfile = ...                     /* Compute new value */
  ct<:mpn:> = newProfile;              /* Write mpn's data */

  if ((ct?<:mpn1:> && (ct@<:mpn1:>)    /* Test mpn1  */
    ct\<:mpn1:>                        /* Remove mpn1's data */
}
```

Fig. 3. Map operations.

function) that indicates whether parameters were supplied. Default and compression functions are responsible for taking appropriate action if the flag indicates that no parameters were supplied at initialization time. Such actions might include constructing an appropriate value or returning an error.

3.5.2 *Map operations.* Hancock provides five operations for manipulating individual items in maps: read, write, remove, test key, and test active key. 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 key operator, written `?<: ... :>`, queries a map to determine whether the key is in the map's range. Finally, Hancock's test active key operator, written `@<: ... :>`, queries a map to determine whether the key is active. It is a run-time error to pass an out-of-range key to any of these operations other than test key.

As an example of these constructs, the code in Figure 3 first reads the value for the key `mpn` from map `ct`, writes a different value `newProfile` into map `ct` for the key `mpn`, tests whether the key `mnp1` is in `ct`'s range and is active, and if so, removes that item.

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 Cell Tower application uses map iteration to age each cell tower `count` every day, a process which allows new information to replace old information gradually. The 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. Section 5.2 describes Hancock's operations for manipulating streams.

3.6 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

```
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 pht_p {initPHT => pht_rep => flushPHT};
```

Fig. 4. Pickle type for persistent hash tables.

of Hancock's persistent data system, in particular, with directories, dynamically-specified type parameters, 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.

The programmer uses the `pickle` declaration to specify the name of the pickle type, the type's parameters, 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 4 shows the code for the `pht_p` persistent hash table.

The in-memory representation is the C `struct pht_rep` shown in Figure 4. This `struct` contains an in-memory hash table, an indication of whether the structure is read-only, and an indication whether the hash table has been updated since it was opened. Hancock's runtime system allocates the space to hold this `struct`; values of the pickle are treated as pointers to it. The on-disk representation is not specified directly in the pickle type. Instead, it is encoded in the initialization and write functions specified in the pickle type.

The initialization function fills the in-memory representation from the contents of the file containing the on-disk representation. This function can assume that the status of the file matches any qualifiers specified in the declaration because the runtime system will call the function only if the qualifiers were satisfied. The 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.

Figure 4 shows the prototype for the function `initPHT`. The Hancock runtime system supplies three parameters (`fp`, `data`, `readOnly`) to the initialization function. Parameter `fp` is a file pointer that points to the file containing the on-disk representation of the pickle; the runtime system opens this file before calling the function `initPHT`. If the programmer declared the pickle to be read-only using the `const` qualifier, then the runtime opens the file in read-only mode to ensure the persistent data is preserved unmodified. The runtime waits to close the file until it deallocates the associated pickle, so the initialization function can save the file pointer in the in-memory representation of the pickle. This technique of caching the file pointer allows pickle programmers to implement data structures that are split between memory and disk (*e.g.*, as the runtime system does for the implementation of maps). The `data` parameter is a pointer to the in-memory representation of the pickle.

```
typedef struct { ... } cTab_rep;

int initCTab(char set, int level, Sfio_t *fp,
             cTab_rep *data, char readOnly)
int writeCTab(Sfio_t *fp, cTab_rep *data, char close);

pickle compTable_p(int level)
        {initCTab(:level:) => cTab_rep => writeCTab};
```

(a) Type declarations

```
#define COMPRESSION_LEVEL_DEFAULT     6
int initCTab(char set, int level, Sfio_t *fp,
             cTab_rep *data, char readOnly)
{
    if (set == 0) {
      level = COMPRESSION_LEVEL_DEFAULT;
    }
    /* initialize compression table from the file */
    return HRS_OK;
}
```

(b) Initialization function

Fig. 5. Compression table pickle type definition

The `readOnly` parameter is a flag that indicates whether the pickle was declared with the `const` qualifier. The return value for the function is used to signal an error if necessary. Upon receiving an error signal, the runtime system terminates program execution. The body of the `initPHT` function uses its parameters and the contents of the associated file to initialize the `data` structure.

The write function flushes in-memory data to disk so that this data may be preserved across program executions. To protect persistent data, the runtime system calls this function only when the associated pickle is not declared to be `const` in its initializing declaration. Figure 4 contains the prototype for the write function for our sample pickle. This function takes three parameters. The first, `fp`, contains a file pointer for the on-disk representation. The second, `data`, points to the in-memory representation of the pickle. The third, `close`, indicates whether the function should free any resources allocated by the programmer on behalf of the pickle. The runtime system sets this parameter to true when deallocating the pickle; it sets it to false when flushing data to disk to support persistent copying for pickles.

Our example pickle type, `pht_t`, does not take any parameters, but pickles, like directories and maps, may be parameterized. Figure 5 contains the type definition for a compression table pickle that takes an integer argument (`level`) for specifying how hard the compressor should work to save space. The initialization function, `initCTab`, takes the usual pickle initialization function parameters (`fp`, `data`, and `readOnly`) plus two extra parameters (`set` and `level`) that arise from the `level` type parameter. The `set` parameter indicates whether the parameter was supplied when a given pickle variable was declared. The `level` parameter contains the supplied value if one exists or an undefined

value otherwise. The write function takes only the usual parameters because it was not declared to take any type parameters.

Hancock does not provide any operations for pickles other than the copy operator. Instead, the pickle designer provides operations as functions that take values of the pickle type (*i.e.*, pointers to `pht_rep structs`) as arguments. For example, the following code:

```
int insertPHT(pht_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`.

3.7 Summary

The mechanisms described in this section collectively provide a persistent data system that satisfies the primary goals of efficiency and data safety as well as the auxiliary goals of flexibility, transparency, and ease-of-use. Support for efficiency includes maps, which provide a highly efficient and tunable representation for signature collections, and pickles, which allow users to write custom persistent data structures that can scale to large volumes of data. Many aspects of Hancock's design contribute towards data safety, including the static and dynamic checking provided by initializing declarations and the grouping role of directories. Parameterization and pickles permit customization, allowing programmers flexibility in how they use Hancock's persistent data system. The persistent structures integrate smoothly into the host programming language, where they appear simply as program variables with special types, supporting the transparency goal. This integration is possible because the type declarations serve to isolate details about persistence. Collectively, these mechanisms make it easy to use persistence, which also supports data safety by helping programmers avoid making errors in the first place.

4. APPROXIMATE DATA

The volume of data in daily transaction streams requires analysts to consider carefully how much information to associate persistently with each entity of interest. Larger signatures contain more information, but require more time to compute and more space to store. Often analysts can chose to store approximations that sacrifice precision but retain the key characteristics of the entities in the stream. For example, the original long-distance applications stored very small signatures by quantizing signature values. Examples of this technique include quantizing a floating point number representing the probability that a phone number is behaving like a business into sixteen levels or categorizing the number of daily outbound minutes into one of eight logarithmically spaced usage bins. These approximate representations can be compressed conveniently into a few bits each.

Despite the efficiency of approximate signatures, it is often more convenient and more accurate to manipulate precise representations when processing transaction records. This dichotomy leads to programs that use two different representations for the same conceptual piece of information: an *approximate* form for long-term storage and a precise, *logical*

form for computation (as well as a compressed *physical* form that is written to disk). If not carefully structured, this practice can result in confusing code. For example, the original C signature programs contained routines to approximate and compress each signature before writing it to disk and routines to uncompress and expand it before computing with it. However, we found situations in which the original C code performed computations not only on the logical representations, but also directly on the approximate and on the compressed representations. While the code was very efficient, it was highly unreadable, making it difficult to verify and maintain.

In this section, we describe Hancock's `view` mechanism, which helps programmers manage multiple data representations. With this construct, programmers can specify two views of a single piece of data and the conversion between them. Signature programs can use one view to describe the logical representation of each signature and another to describe the approximate representation.[6]

As an example of views, consider the following declaration that specifies approximate (`bin`) and logical (`minute`) representations of a unit of time:

```
view time(bin, minute) {
  char <=> int;
  bin(m) { return min_to_bin(m); }
  minute(b) { return bin_to_min(b); }
}
```

The line `char <=> int` declares that the `bin` view is represented as a `char` and the `minute` view is represented as an `int`. The `bin` function specifies how to convert from the logical to the approximate representation by computing the bin associated with `m` minutes. Similarly, the `minute` function converts from the approximate to the logical representation by assigning a default number of minutes to the bin `b`. The view operator ($) allows the programmer to translate between these views:

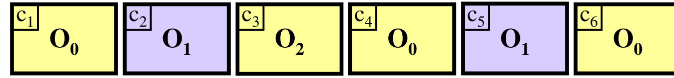
```
bin b = 3;
minute m;
m = b$minute; // Convert bin b to minutes
    ...
b = m$bin;    // Convert minutes m to the
              // corresponding bin number
```

Views allow Hancock programmers to document the representation they are using in a given context. Views also ensure that the definition of how to convert between the representations appears only once in the program. Both of these aspects of views make Hancock programs easier to read and maintain than the corresponding C programs.

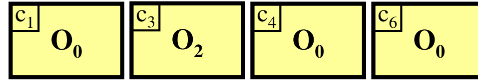
4.1 Related work

Although developed independently, Hancock views closely resemble Wadler's view construct, proposed as a way to integrate functional data types with ML-style pattern-matching [Wadler 1987].

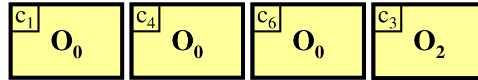
[6]We pushed a third representation, the compressed representation, into the map abstraction through the compression/decompression functions.



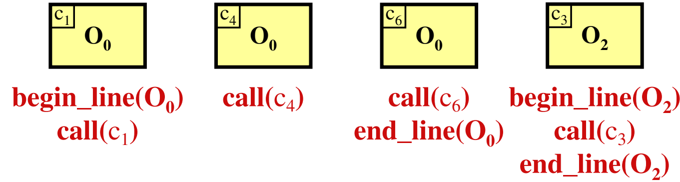
(a) Initial stream of records.



(b) Stream after removing unneeded records.



(c) Stream after filtering and sorting to ensure access locality.



(d) Filtered, sorted stream labelled with events.

Fig. 6. 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 labelled 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.

5. STREAMS

In addition to supporting persistent data types and views, Hancock provides mechanisms for describing and computing with streams of transactions. In this section, we present Hancock's stream type definition, its model of stream events, and its mechanism for consuming streams. Figure 6(a) depicts a sample stream of wireless calls.

The fields in raw transaction records are often encoded and packed to save space. In studying the original signature programs, we noticed that code to decode the representation of stream records was interleaved with signature processing code, which made it difficult to update the programs when the stream representation changed.

In Hancock, we separate the *physical* representation of the records in a data stream from the *logical* (expanded) representation on which we perform computations. This separation 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.

Hancock supports two forms of stream type 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. The binary form is more convenient for binary records, while the general form is more broadly applicable. Either of these forms may be parameterized in the same fashion as Hancock directories, maps, and pickles.

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.

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 specifies how to convert from the physical to the logical representation. This function, whose prototype is:

```
int getvalidWCRbinary(wcrPhy_t *pc, wcrLog_t *c);
```

checks that the record `*pc` is valid and if so, unpacks `*pc` into `*c` and returns true to indicate a successful conversion. Otherwise, `getvalidWCRbinary` simply returns false. The validity check ensures that down-stream code need not worry about malformed records.

In the general case, a stream declaration specifies a function that reads data from a file and returns a logical record. We use the following (parameterized) declaration:

```
stream wireless_s (pht_p pht) {
  getvalidWCR(:pht:) : Sfio_t => wcrLog_t;
 }
```

of a general stream `wireless_s` in our running example. Identifier `getvalidWCR` names a function with prototype:

```
int getvalidWCR(char set, pht_p p, Sfio_t *fp, wcrLog_t *c);
```

This function reads a physical record from the file `*fp`. If the record is valid, it converts the cell tower names into hash codes using the persistent hash table parameter. It then fills in the logical record `*c` appropriately and returns `HRS_STREAM_KEEP_REC`, a constant defined by the runtime system, to indicate it found a valid logical record. Otherwise, it returns `HRS_STREAM_DROP_REC`. This function must read at least one character from the input file to ensure that consuming the stream does not loop infinitely.

Programmers can declare stream variables using standard C syntax (*e.g.*, `wireless_s calls`) or Hancock initializing declarations:

```
wireless_s calls(:pht:) = "call-detail.current"
```

In the latter case, the path name specifies the on-disk location of the physical records that constitute the stream. The location can either be a file or a directory of files, each of which contains physical records. If the programmer omits a parameter from an initializing declaration, the `set` parameter to the stream translation function will be false. In this case, the translation function must take appropriate action, either supplying a default value or returning an error to halt computation.

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.

5.1 Events

Much of the work in computing a signature is done in response to "events" in the input stream. For example, when a program sees a new mobile phone number in an `wireless_s` stream, it might initialize counters to be used for computing the signature for that phone number. The original signature programs contained a hierarchy of events, which included seeing a new area code (`npa_begin`), a new exchange (`nxx_begin`),[7] a new phone number (`line_begin`), an individual call record (`call`), the last record for a phone number (`line_end`), *etc.* Similar event hierarchies may be defined for streams of credit card charges, IP packets, *etc.*

When a call-detail signature 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 the signature values stored for all 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. Analogous actions may be taken when processing other kinds of event hierarchies.

In Hancock, we divide this processing into two pieces: event detection and event response. Event detection includes defining the events of interest in a given stream and specifying how to identify them. Event response indicates what to do when an event is detected. The Hancock compiler generates the control flow that sequences the detection and response code. This control-flow code involves deeply nested loops that have an inverted control structure.[8] Hancock programs, which hide this structure, are much easier to read and maintain than the corresponding C programs, which mix this inverted control-flow with event response code. In the following two subsections, we discuss how to describe and detect events in a stream. Section 5.2 discusses how to respond to detected events.

5.1.1 *Describing events: multi-unions.* To define events in a general fashion, we introduced a new kind of type into Hancock: a *multi-union*. A multi-union declares a set of labels and associated types. 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." When we use multi-unions to describe events, however, we often refer to their labels as "events." 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.

[7] An *exchange* is the first six digits of a ten digit telephone number.
[8] The easiest way to find events is to detect "ending" events before "beginning" events. Loops that have this structure are said to be "inverted."

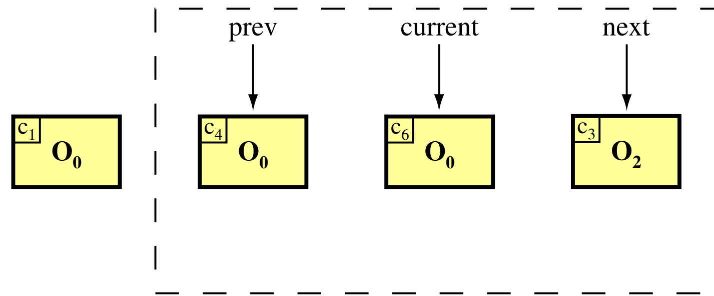


Fig. 7. Stream with window of type*wcrLog_t[3:1]

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 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 (:\:).

5.1.2 *Detecting events.* After describing the events of interest using a multi-union declaration, the programmer must specify how to detect such events by writing an *event-detection* function. Such a function looks at a small portion of a stream and returns a multi-union to describe the events detected in that window.

To describe a small portion of a stream, Hancock provides a *window* type (see Figure 7). Each entry in a window points to a record in the stream. The size of the window determines how many records in the stream can be viewed at once. A window is like an array, but has the added notion of a "current" record. In specifying a window, the programmer indicates both the size of the window and the placement of the 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, *i.e.*, 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. If the window overlaps either the beginning or the end of the stream (or both), the slots with no corresponding record are set to NULL.

An event detection function takes a window onto a stream and returns a multi-union describing the events detected in that window. For example, the Cell Tower application uses the event detection function shown in Figure 8. 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 7, the originDetect function generates the value:

$$\{: \text{call} = c_6,\ \text{line_end} = O_o\ :\}.$$

```
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 ((0 == prev) || (prev->origin != current->origin))
    b = {: line_begin = current->origin :};
  else b = (wline_e){: :};
  if ((0 == next) || (next->origin != current->origin))
    e = {: line_end = current->origin :};
  else e = (wline_e): :;
  return b :+: {: call = *w[1] :} :+: e;
}
```

Fig. 8. An event detection function for the Cell Tower application.

The Cell Tower application uses another 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.

5.2 Consuming a stream

As in the original signature programs, Hancock's computation model is built around the notion of iterating over a sorted stream of transaction records. Appropriately sorting the records groups all the data relevant to one entity into a contiguous segment of the stream and ensures good locality for map references that follow the sorting order. Consequently, each signature program typically makes multiple passes over its data stream. During each such pass, the signature program sorts the stream in a different order and updates a different portion of each entity's signature. We call each pass a *phase*.

We implement phases using Hancock's `iterate` statement. This statement has the following form:

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

The header specifies an 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 explain each of these pieces in turn.

The `over` clause contains a stream-valued expression. The resulting stream must either be a stream that was connected to data on disk previously using an initializing declaration (or `sig_main` parameter, *cf.* Section 6) or a stream that resulted from a map-to-stream expression.

The `filteredby` clause specifies a predicate to remove unneeded records from the stream. For example, a `wireless_s` stream may include land-to-cell calls, which are not used by the outgoing phase of the Cell Tower application. Figure 6(b) shows a sample wireless call stream after filtering. Immediately removing such records improves the efficiency of sorting and simplifies event response code.

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 following clause:

```
sortedby origin, dialed
```

produces a stream sorted primarily by the originating telephone number and secondarily by the dialed phone number. Figure 6(c) shows a sample call stream after the calls have been filtered and sorted by the originating number.

The `withevents` clause specifies an event detection function. As described in Section 5.1, such a function takes a window onto the stream and returns a multi-union describing the events detected in the given window. Figure 6(d) shows a sample wireless call stream labelled with events.

The *event clauses* 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 the 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 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 9 shows the outgoing phase of the Cell Tower application, which processes the calls made *from* mobile phone numbers. The function `doOrigin`, which encapsulates this phase, contains a single `iterate` statement that processes a wireless call-record stream. It uses the predicate function `originCellCall` to remove non-cellular 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 in Figure 9 specify how to respond to the detected events.

5.3 Related work

Many languages support streams. We focus our discussion of related work on systems have stream models similar to Hancock's. Other systems, such as AWK[Aho et al. 1979] and SAX[2002], use event-based models for streams. AWK is a string processing language that is based pattern-action pairs. Patterns can be arbitrary boolean combinations of regular expressions and relational expressions. Although Hancock's event detection functions could be used to implement AWK's patterns, AWK is superior for simple processing of text files. On the other, AWK's string focus makes it less suitable for processing large amounts of binary data.

SAX[2002] is an event-based API for processing XML documents. It converts an XML document into a stream. SAX programmers process this 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 program-

```
void doOrigin(char *streamLoc, cellTower_m m, pht_p pht)
{
  wireless_s calls(:pht:) = 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:>;
        integrate(&op,&p);
        m<:origin:> = op;
      }
  };
}
```

Fig. 9. Outgoing phase for the Cell Tower application.

mers 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 additional programmer support, such as static type checking, beyond the host language. Hancock, on the other hand, provides much richer programmer support, but at the cost of learning (and using) a new language.

Tribeca[Sullivan and Heybey 1998] is a system for monitoring network traffic that is based on streams, but does not use an event processing model. Instead Tribeca 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 separating and recombining operations 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 sub-stream into a single record for a session, and finally 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 elements in a stream.

6. PUTTING IT TOGETHER: SIG_MAIN

To simplify command-line processing and to make it easy to connect persistent data structures to their in-memory variable names, Hancock provides the sig_main function, which supersedes C's main function. This function specifies the entry-point into Hancock programs and provides a simple way for programmers to specify command-line arguments. The Hancock compiler, rather than the programmer, generates code to parse the actual command-line parameters. Parameters to sig_main are annotated with the same information as initializing declarations. In addition, they indicate a character to be used as the command-line switch and an optional default value. For example, the following code:

```
int sig_main(exists const cellTower_d oldCT <d:>,
             new cellTower_d newCT <D:>,
             char * callsLoc <c:>,
             char * date <t:> default ``unspecified'')
{
   newCT :=: oldCT;
   age(newCT->outMap);
   age(newCT->inMap);
   doOrigin(callsLoc,newCT->outMap, newCT->ctHashTable);
   doDialed(callsLoc,newCT->inMap, newCT->ctHashTable);
   newCT->lastUpdated = date;
}
```

declares four command-line arguments. The oldCT parameter is a Cell Tower directory. The syntax (<d:>) after the variable name specifies that this parameter will be supplied as a command-line option using the -d flag. The colon indicates that the flag takes an argument, in this case the name of the UNIX directory that holds the cell tower data. The absence of a colon indicates that the parameter is a boolean flag. The const qualifier indicates the directory is read-only, while the exists annotation requires the directory to exist on disk. The newCT parameter names the Cell Tower directory to hold the result of this program. The -D flag specifies the file name for this directory, and the new qualifier indicates that the directory must not exist previously on disk.

The callsLoc parameter expects the programmer to use the -c flag to pass a string that contains the name of the location of the raw wireless call data. Functions doOrigin and doDialed use this string in their initializing declarations for the wireless stream.

Finally, the date parameter takes a string specifying the date of processing using the -t flag. The default clause specifies a value to use if the programmer omits this flag.

In general, the body of sig_main is a sequence of Hancock and C statements. In the Cell Tower application, sig_main copies the data from oldCT into newCT, ages the signature data in the new directory, invokes the outgoing and incoming phases, and finishes by updating the date of processing in the new directory.

7. 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 multi-level tables [Gupta et al. 1998; Lampson et al. 1999; Huang et al. 1999]. The compiler translates the source-level map operations described in Section 3.5.2 into the corresponding operations in the runtime system. An earlier paper provides a more detailed discussion of Hancock's map implementation [Fisher et al. 2001].

8. EXPERIENCES

In this section, we describe our experiences with Hancock programs in practice and address the question of why we implemented Hancock as a language not a library.

8.1 Hancock in practice

We rewrote the original 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 855 and 866 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[9] 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 per account. 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 applications that associate over 100 bytes with each account. With that level of detail,

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

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.

8.2 Language versus library

One question we are asked often is why we chose to design a language rather than a library. There are two technical reasons for choosing the language option. First, expressing Hancock's event model and the information sharing it provides proved awkward in a call-back[10] framework, the usual technique for implementing such abstractions. Second, by designing a language we could use the language's type system to provide more precise type checking than is provided by C. For example, the natural way to implement maps in C using a library interface would require the programmer to cast between the actual type of a value and `void *`, thereby losing the benefits of type checking. The scale of the data makes the complexity of finding and fixing bugs in signature programs substantial. Therefore, good static error detection is essential.

The more compelling reason to choose a language over a library for us is sociological. The experience of writing a Hancock program is fundamentally different than writing the equivalent program in C. This difference arises in part because Hancock removes issues of scale, leaving programmers free to concentrate on the design of the individual profiles, and in part because Hancock provides a vocabulary tailored to the domain of signature design.

9. SUMMARY

The current design of Hancock reflects what we have learned from four years of working with signature programs. In this section, we summarize our design by tracing its evolution.

Before designing the original version of Hancock[Bonachea et al. 1999], we carefully examined the signature programs that had been hand-written in C. In studying these programs, we observed several problems. First, the programs manipulated their signatures using several different representations, corresponding to what we term the logical, approximate, and physical abstraction levels. Because the logical and approximate levels often share the same C representation, *i.e.*, an `int`, it was often unclear which representation was being manipulated at any given point in the program. Second, the original programs did not separate the implementation of signature collections from their use, making it very difficult to modify that representation. Third, the code to decode the physical representation of a stream was interleaved with code to process the records in that stream, making it difficult to modify the programs when the physical representation of the stream changed. Fourth, the code to process the events in the stream was difficult to decipher and maintain because of the inverted, nested loops that expressed the desired control flow. Finally, the dataflow between the phases was often unclear. Despite the weaknesses of these programs, they had some important strengths: they were efficient and they used an effective representation for their persistent data.

---
[10] A call-back is a call from a function in a library "back" to a function in user code.

We designed Hancock to address these problems while preserving the strengths of the original programs. Views allow a programmer to document the relationship between two representations of a value. The view conversion operator ($) guarantees that programmers switch between representations in a consistent and well-documented way. The map abstraction separated the implementation of signature collections from their use, allowing us to experiment with and eventually improve their representation. The `stream` construct allows Hancock programmers to isolate the details about the physical layout of their transaction data, making it easier to adjust to changes in the physical layout of that data. Hancock's `event` clauses clarify stream processing without losing efficiency. Such clauses have several advantages. First, they have the flavor of function definitions with their attendant modularity advantages, but without their usual cost because the Hancock compiler expands the event definitions in-line with the control-flow code. Second, having the compiler generate the complex control flow removes a significant source of bugs and complexity from Hancock programs. And third, programmers can share information across events easily. Finally, Hancock's `sig_main` mechanism relieves the programmer of the burden of command-line argument parsing, provides a way to connect the in-memory representation of Hancock data types to their on-disk counterparts, and clarifies the dataflow between phases.

Solving these problem led to a clean, easy-to-use language, but our experience with early versions of Hancock led us to conclude that these features were insufficient for our domain. In particular, programmers often needed auxillary persistent data structures, such as hash tables to convert variable-width data to a fixed-width format and compression tables to improve the efficiency of maps. In addition, it was often the case that we needed to define collections of types that differed only in small ways, *e.g.*, four map types that differ only in their range of keys.

To address these problems, we extended Hancock's persistent data system to provide a more comprehensive solution. In particular, we added pickles, directories, initializing declarations (to augment `sig_main`), and parameterized types. In designing this extension, we focused on the goals of efficiency, data safety, flexibility, and transparency. Our design addresses these issues as follows:

Efficiency. Maps provide only the operations necessary for signature collections, an economy that permits an implementation with very little overhead. Pickles allow programmers to load data into memory from disk lazily and in pieces, an approach which scales to large data structures.

Data Safety. Hancock's PDS provides data safety through its static and dynamic type checking mechanisms and through the qualifiers provided by initializing declarations. Directories support data safety as well by allowing the programmer to group tightly coupled data under a single name. Through the use of type parameters, the programmer can make this coupling explicit and guarantee that the necessary connections between coupled persistent types are made automatically when the containing directory is initialized.

Flexibility. Hancock's PDS provides flexibility through a number of different mechanisms. Maps are customizable to a particular application through the `key`, `split`, `default`, and compression clauses. Although the programmer could fall back to standard C and explicit I/O when maps and directories are not appropriate for a given application, pickles allow the programmer the same flexibility while integrating cleanly with the rest of Hancock' persistent data system. Finally, type parameters allow the programmer to de-

fine a general type and specialize it to a particular need at runtime without sacrificing data safety.

Transparency. Once a persistent variable has been initialized in Hancock, a programmer can ignore the fact that the variable is persistent. All details related to its persistence are isolated in the type and variable declarations.

Collectively, these mechanisms make it easy to write efficient and maintainable signature programs and help programmers guard against corrupting valuable data.

10. CONCLUSIONS

Working with transactional data streams is like drinking from the proverbial fire hose: the volume is simply 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 non-commercial 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. *Software Practice and Experience 9,* 4 (April), 267–79.

APPEL, A. W. 1990. A runtime system. *Lisp and Symbolic Computation 4,* 3, 343–380.

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

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 Springer Verlag LNCS 1819 (pp. 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, 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.

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`.

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. In *IEEE Trans Soft Eng Vol 13, No 2*.

FAWCETT, T. AND PROVOST, F. 1997. Adaptive fraud detection. *Data Mining and Knowledge Discovery 1*, 291–316.

FISHER, K., GOODALL, C., HOGSTEDT, K., AND ROGERS, A. 2001. An application-specific database. In *Proceedings of 8th Biennial Workshop on Data Bases and Programming Languages (DBPL '01)*.

GUPTA, P., LIN, S., AND MCKEOWN, M. 1998. Routing lookups in hardware and memory access speeds. In *Proc. 17th Ann. Joint Conf. of the IEEE Computer and Communications Societies*. Vol. 3. 1240–7.

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 *Proc. 18th Ann. Joint Conf. of the IEEE Computer and Communications Societies*. Vol. 3. 1429–36.

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 Transactions on Networking 7,* 3, 324–34.

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.

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.

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. python.sourceforge.net/devel-docs/lib/lib.html.

WADLER, P. 1987. Views: a way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*.

WANG, D. C. 1998. The asdlGen reference manual. See www.cs.princeton.edu/zephyr/ASDL.