# Overview of the STanford Real-time Information Processor (STRIP) *

Brad Adelberg† Ben Kao‡ Hector Garcia-Molina§

**Abstract**

We believe that the greatest growth potential for soft real-time databases is not as isolated monolithic databases but as components in open systems consisting of many heterogeneous databases. In such environments, the flexibility to deal with unpredictable situations and the ability to cooperate with other databases (often non-real-time databases) is just as important as the guarantee of stringent timing constraints. In this paper, we describe a database designed explicitly for heterogeneous environments, the STanford Real-time Information Processor (STRIP). STRIP, which runs on standard Posix Unix, is a soft real-time main memory database with special facilities for importing and exporting data as well as handling derived data. We will describe the architecture of STRIP, its unique features, and its potential uses in overall system architectures.

## 1 Project Goals

The STanford Real-time Information Processor (STRIP) was designed with the following goals:

1. support for transactions and data with soft timing constraints,
2. high performance,
3. high availability,
4. the ability to share data with other components in open systems.

The first goal, support of soft timing constraints, is implemented through two mechanisms: value function scheduling and maximum age constraints. A number of different scheduling algorithms are available, including earliest deadline first, highest value first, highest value density first, and custom scheduling algorithms. To support temporal constraints on the data, users can define *maximum age* requirements for views and for data accesses, where the age of data is defined to be the amount of time since it was last changed. The second goal, high performance, is achieved by using a main memory database. To provide high availability, a STRIP system can be reconfigured on the fly: Changes to user and database code can be made without bringing the server down. As another feature, if extremely high availability is required, STRIP databases can be configured in master/slave pairs. The master executes all of the transactions and ships its log records across to the slave. The mechanism that supports the log transfer is the same one that allows views to be exported to other systems.

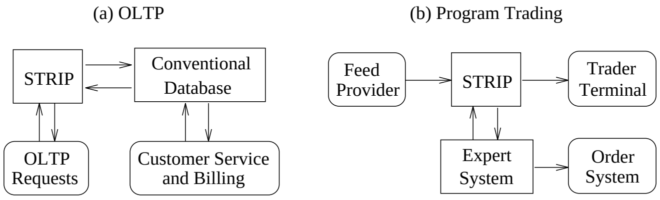


Figure 1: Example applications.

Determining how to meet the fourth goal was one of the main challenges of the project. Our aim was a database that could be incorporated into many different applications, with varying data requirements. To illustrate, consider the applications pictured in Figure 1. The first, Figure 1(a), is indicative of a credit card system. STRIP, due to its high throughput, is used to approve charges at the point of sale, while the conventional database, due to its large storage capacity and sophisticated report generation, is used to store charge records, to generate statements, etc. Of course, the two systems are not entirely separate: they need to share data. STRIP needs to know when accounts have been payed or when cards have been reported stolen and so it needs a materialized view of information from the conventional database. The conventional database needs the authorization records created at the MMDB and needs to know the available balance of the accounts to handle customer inquiries. Furthermore, there are temporal constraints on the data transfer: New status information should be sent to STRIP within a short time after a card is reported stolen.

The second example application, Figure 1(b), is program trading in financial markets. Here STRIP is used by an expert system to store security prices and compute derived data (e.g. composites). In this example, the source of data is not another database but a commercially provided feed of price changes. Similarly, the destination of the exported data is a trader's terminal. Although the type of data being shared and the timing constraints on the propagation vary widely between this and the previous example, STRIP has been designed as a standard component that can be used for any of them.

In the rest of this paper we briefly summarize the overall STRIP architecture, and we highlight some of the research problems that were addressed. Due to space limitations, we do not provide references to prior work; instead, we refer the reader to our full papers that contain extensive references.

* This work was partially supported by the Telecommunications Center at Stanford University, by Hewlett Packard, and by Philips.

† Stanford University Department of Computer Science. e-mail: adelberg@cs.stanford.edu

‡ Hong Kong University Department of Computer Science. e-mail: kao@cs.hku.hk

§ Stanford University Department of Computer Science. e-mail: hector@cs.stanford.edu

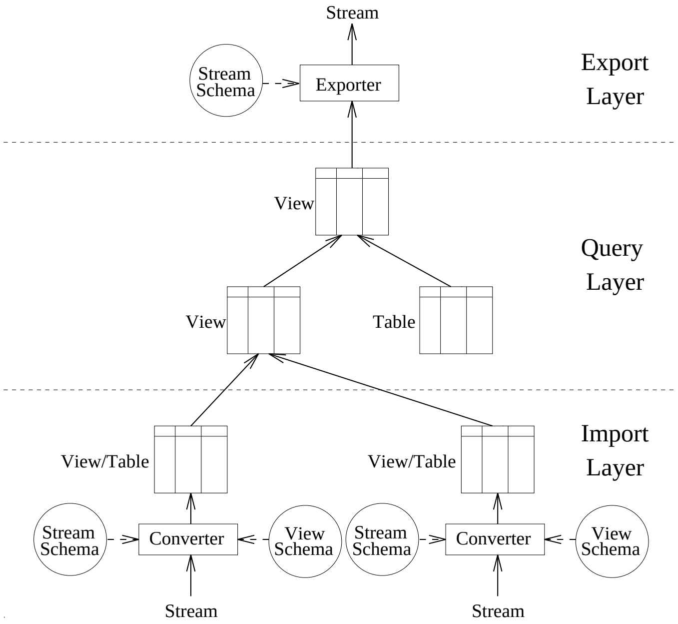


Figure 2: System architecture.

## 2 Data Sharing Facilities

The data sharing architecture of STRIP is shown in Figure 2. The middle layer, the query layer, has the equivalent functionality of a stand-alone database. The additional functionality in the import and export layers is what allows STRIP to be integrated with other systems. STRIP exchanges information with other systems over *streams*. The actual format of the data in the stream is determined by the source and is described in a stream schema record that is provided whenever a new connection is established. The schema describes the attributes of the tuples being sent as well as information about the properties of the stream itself which are described below.

The stream protocol was designed to provide flexible data sharing with other systems. For example, consider a stream being used to support distributed materialized views. Depending on how often the source refreshes the remote views, it may prefer differential or complete refresh. The stream protocol, which can transmit entire relations or sets of deltas (inserts, deletes, and modifies), will support both. In addition to distributed views, streams also support data migration: Tuples can be passed from the source to a remote site. Data migration is different from remote materialization because in the former, ownership (right to update) is transferred. A site receiving a stream of migrated data is actually importing a table and not a view. An example of use data migration is the transfer of authorization records in the credit card application. They are created in STRIP when a charge is approved and then transferred to the conventional database for permanent storage. Using the export facilities described below, the application designer can specify when the records should be transferred, allowing sophisticated migration strategies without custom coding.

### 2.1 Importing

In order to insulate the user from all of the details of streams, the import layer has been added to convert streams of remote data into views. This conversion presents the query layer with a single paradigm for all imported data regardless of its original format. The views defined in this layer can be derived from only one stream (although one stream can generate many views). Views that combine data from different streams must be defined in the query layer. To create an import view, the converter needs both the stream schema, provided by the source, and the view schema, provided by the user. For each attribute in the view, the view schema must specify not only its name and type but also a function to compute its value. The schema can also define real-time constraints on the use and upkeep of the view, such as the maximum age at which its data becomes useless, how important it is to maintain its freshness compared to the other work in the system (described in Section 4.2), etc. It is also possible to define consistency requirements such as: the view will eventually be consistent with the remote source, the changes to the view will be applied in the same order they occurred at the source, etc. Finally, the user can define an import table rather than an import view for cases when data will be migrated as explained above. In this case, if the source has requested 2-safe transfer (involving 2-phase commit), STRIP sends an acknowledgement to the source upon receipt of the data so that the source can delete it if desired.

### 2.2 Exporting

Views are exported (converted into output streams) by the export layer. For each export view, the user must specify a destination(s) and whether the view is to be migrated or just shared. For non-migrated views, the user must also select a refresh method: send the entire view every time, send all of the changed/inserted/deleted tuples, or send only the net effect of the changed/inserted/deleted tuples. Having defined 'what' is to be sent, the user must specify 'when' it is to be sent. The options are: when a tuple is changed, when a tuple becomes older than a predefined age, periodically, or when explicitly flushed by the application. Finally, the user can specify real-time and consistency constraints similar to those defined on the imported views.

## 3 Process Architecture

The process structure used to implement STRIP is shown in Figure 3. All of the data access and modification is done by a single pool of *execution* processes. These processes both apply updates to import views and run transactions. The interface to the process pool is a collection of queues: a transaction queue and one update queue for each import view. The transaction queue contains both application requests and internally triggered transactions. The updates are divided into separate queues so that the system can maintain some views more diligently than others in the case of different timing constraints or different values.

The other processes serve to connect the database to the outside world. The *import* and *export* processes decouple the network interface from the database core and perform the conversion from the internal format to the stream format. The *request* process conveys remote requests to the database and sends the results back to the originator. Local applications can access the queues directly using a client library for greater efficiency.

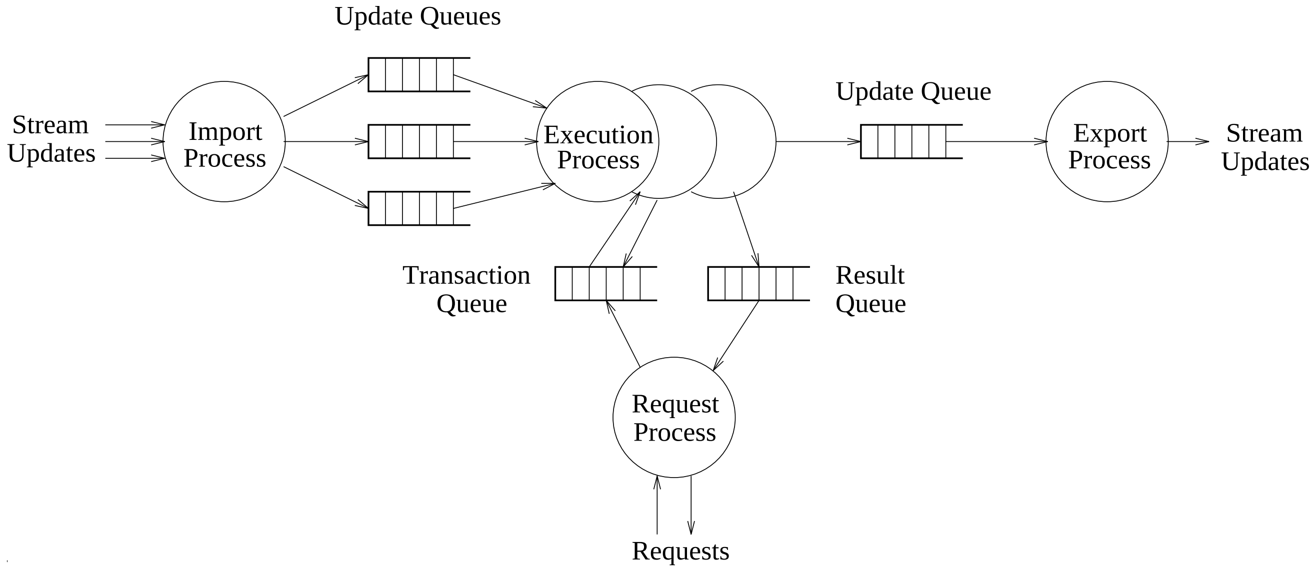


Figure 3: Process architecture.

## 4 Problems Studied

In the course of designing STRIP, we encountered a number of challenging sub-problems. When this happened, we performed detailed simulation studies in order to select the appropriate algorithm or strategy for STRIP. The results of three such studies are described below.

### 4.1 Developing under Unix

Since STRIP is intended for use in open systems, we decided to build STRIP on top of a standard operating system, Posix Unix, instead of a specialized real-time OS. Since standard operating systems do not provide real-time scheduling, this approach is not acceptable for hard real-time systems. However, it is acceptable in a soft real-time system like STRIP, as long as we somehow make the system adhere as closely as possible to the timing constraints of data and transactions.

One possibility is to write a threads package on top of Unix and then code any desired scheduling algorithm into it. This approach, however, requires a lot of code and is not portable across different Unix vendors. Therefore, we chose to use the built-in scheduler but to try to set the process priorities to emulate real-time scheduling algorithms. For instance, to emulate earliest deadline first (EDF) scheduling, a new process is assigned a priority greater than those of processes with later deadlines but less than those of processes with earlier deadlines. There are two complications to the assignment strategy:

- the limited number of priority levels - Most systems provide between 40 and 128 priority levels, This means that when a new task arrives, there may not be a free priority level to give it. If the priority of a running process cannot be changed, or is expensive to change (due to system call overhead), this problem is exacerbated: there could be free priority levels but not between the two processes that have the closest deadlines above and below that of the new process. This means that collisions can occur with far fewer processes than the number of total priority levels.
- the disruption of multi-level feedback - The standard Unix scheduler adjusts the priority of processes as they are running. Processes that hog the CPU have their priorities temporarily lowered to allow other processes to run. While this is important in a time-shared environment, it is undesirable in a real-time environment where some tasks should be allowed to run to completion before others are allowed to run. Posix Unix provides a new class of processes whose priorities are not adjusted by the scheduler.

In [AGMK94], we developed three strategies for priority assignment to emulate EDF and least slack first scheduling. Through simulation, we compared the performance of our emulation strategies to EDF and LSF. The effects of the number of available priority levels as well as the effects of multi-level feedback were studied. The results showed that the emulation algorithms are comparable in performance to the real-time algorithms (e.g., the number of missed deadlines is almost the same). In some cases, the emulation algorithms even outperform the conventional ones. This occurs because when the system is overloaded, EDF wastes system resources scheduling transactions with nearly expired deadlines that have no hope of finishing on time. In the same situation, our emulation algorithms sacrifice the oldest transactions to give resources to transactions that can still feasibly finish.

### 4.2 Update Scheduling

In a stand alone RTDB, the only source of work is the user transactions which are typically scheduled based on deadlines or value functions. In a system that imports or exports data, however, there are additional sources of work. If the volume of importing/exporting is relatively low, it can be given higher priority than user transactions without greatly affecting response time. In applications with a heavy import/export load, however, new scheduling algorithms must be designed to balance the two classes of work. For example, in the program trading application described in Section 1, rates as high as 500 updates per second have been reported just for the security price feed alone. If all of these updates are performed before user transactions, the missed deadline percentage of the system will increase.

Thus the basic tradeoff is between transaction response time and data freshness. How freshness is defined can vary by application. In cases where the value stored in the database is a sample of a continuously changing real world variable, such as the temperature of a solution in a factory control setting, the data starts becoming stale immediately. In cases where the real world variable changes at discrete points in time, such as the price of traded securities, the data is fresh until another trade occurs. The behavior of different applications in the presence of stale data can vary as well: less critical applications may choose to use whatever is available while other applications may rather choose to abort the transaction. In [AGMK95] we study, through simulation, how four different algorithms to schedule both applying updates to imported views and running user transactions perform under these different assumptions about data freshness. We also consider other factors such as the update and transaction arrival rates, the costs of queueing and applying update, and the access patterns of the transactions. The results show that the choice of algorithm and system properties profoundly affect system performance. Hence, STRIP offers a variety of choices for update installation, to give good performance in all scenarios.

### 4.3 Derived Data

The program trading example from Section 1 requires STRIP to maintain substantial amounts of derived data. Let us consider it in more detail. For simplicity, assume that the only instruments we wish to arbitrage among are stocks, options, and index futures. The feed provider pictured in Figure 1(b) must supply STRIP with the current prices of all such instruments. From the reported price data the following derived data must be computed:

**computed index prices** - For each index on which a futures contract is traded, its price based on the underlying stocks (a weighted sum) must be computed.

**theoretical option prices** - For each listed option, a theoretical price must be computed. The pricing models are very complicated but depend mainly on the price of the underlying stock, the variance of the price of the underlying stock, the risk free interest rate, and static parameters of the options contract itself.

The expert system can then look for discrepancies between calculated and market prices of securities and, if it finds a large enough difference, execute trades.

Such a system has not only the import load (from the price feed) and the user transaction load (from the expert system) described in the previous section but also a new source of work: recomputing derived data. This makes the scheduling problem even more difficult. Recomputation can actually be more expensive than applying updates for the following reasons:

- Recomputing a derived item may be an expensive operation. To compute the theoretical option price, for example, requires computing the cumulative distribution of the standard normal function and the natural log function.

- Recomputations can have high fan-in and fan-out. A derived data item may depend on a large numbers of base data items i.e., high *fan-in*. For example, the S&P 500 index is derived from a set of 500 stocks. When any one of these base data items changes, the derived data has to be updated, so the recomputation is triggered very frequently. Similarly, a base data item may be "popular" in the sense that it is used to derive a number of derived data items, i.e., high *fan-out*. For example, Intel's stock price is used to calculate 30 options and a number of composite indices.

In many applications, including program trading, we can use the *locality* of the updates to reduce the recomputation workload. Update locality means that when a base item is updated, it is very likely that the same item or a *related* one will be updated soon thereafter. For example, a stock price update indicates that there is an interest in its trading. The same stock is therefore likely to be the subject of further trading activities and have its price changed again. Update locality implies that recomputations for derived data occur in bursts. Recomputing the affected derived data on every single update is probably very wasteful because the same derived data will be recomputed very soon, often before any application transaction has a chance to read the derived data. Instead of recomputing immediately, a better strategy might be to defer a recomputation by a certain amount of time and coalesce the same recomputation requests into a single computation. In [AKGM96], we developed a variant of this strategy called *forced delay*. We then studied, through simulation, how it compares to other recomputation strategies, such as immediate and lazy recomputation, in terms of the response time of user transactions and the timeliness of the derived data. We also examined how the properties of the derivation functions affect the relative performance of all of the strategies. The simulation results showed that forced delay greatly reduced the recomputation cost while only modestly diminished data timeliness across a wide range of parameter values. Given the demonstrated desirability of the forced delay strategy, we proposed an an extension to the standard event-condition-action rule model to elegantly and efficiently support it.

## 5 Project Status

The first version of STRIP was completed in 1994 and contained the components shown in Figure 3 except for the import and export processes. It also supported very limited triggering. We are currently coding version 2, scheduled for completion by February 1996, which contains a powerful rule system and integrated support for import/export. We plan to test the new system in a full program trading application. For more information on STRIP, and for more detailed references, see the papers listed below.

## References

[AGMK94] B. Adelberg, H. Garcia-Molina, and B. Kao. Emulating soft real-time scheduling using traditional operating system schedulers. In *IEEE Real-Time Systems Symposium*, 1994.

[AGMK95] B. Adelberg, H. Garcia-Molina, and B. Kao. Applying update streams in a soft real-time database system. In *ACM SIGMOD Proceedings*, 1995.

[AKGM96] B. Adelberg, B. Kao, and H. Garcia-Molina. Database support for efficiently maintaining derived data. In *EDBT Proceedings*, 1996.