A Stream Library using Erlang Binaries

Jay Nelson
DuoMark International, Inc.
http://www.duomark.com/

Abstract

An implementation of a Stream Library for `erlang` is described which uses Built-In Functions (BIFs) to speed access. The approach uses binaries to represent and process stream data in high volume, high performance applications. The library is intended to assist developers dealing with communication protocols, purely textual content, formatted data records and the routing of streamed data. The new BIFs are shown to improve performance as much as 250 times over native `erlang` functions. The reduction in memory usage caused by the BIFs also allows successful processing in situations that crashed the runtime as application functions.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages

General Terms Design, Languages

Keywords Erlang, binary, BIF, stream

1. Introduction

Functional programming languages support data types such as integers, symbols, lists and structures which may be efficiently allocated, contain elements that are easily shared and can be automatically reclaimed. `Erlang` follows this approach with the addition of native binary data due to its telecommunication roots[6]. Binary data is appropriate for streams that are used in communication protocols, network packets and bulk data transfers such as when reading an entire file into memory. The novel approach implemented in `erlang` allows large blocks of data to coexist with the more typically fragmented memory structures of functional languages. This in turn provides an opportunity for computational efficiency natively embedded in a high-level language rather than resorting to C or other languages when developing applications.

This paper introduces the memory and behavior characteristics of lists, tuples and binaries in `erlang`, then continues with a description of the Bit syntax and standard methods of using binaries to deal with streamed data. Next it introduces BIF functions that are shown to be much faster than using the traditional Bit syntax to manipulate binary data. Performance measurements of the library functions are given in Appendix A.

Erlang '05 September 25, 2005, Tallinn, Estonia

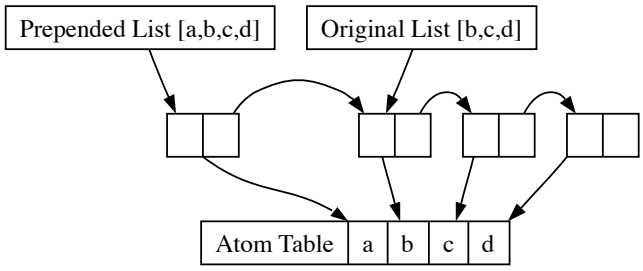

Figure 1. Prepending to a list.

2. Lists, Tuples and Binaries

One of the most important characteristics of the `erlang` language is that variables are "write once". They can be initialized, but if an alternate value is desired, a new variable must be allocated and a new initial value must be provided. This is a fundamental feature of the language because it is the key to simple concurrency – no process or access locks are necessary if all data is read only. The cost of this benefit is the overhead of copying data rather than modifying it in place.

2.1 List Representation

`Erlang` provides a variety of high-level data structures to offset the cost of the "write once" approach. Lists are the most commonly used because they are flexible yet perform reasonably fast. Lists offer a way to chain together other data elements, so that a single copy of an element may participate in more than one data structure. The benefits of read only data can be leveraged to reduce the amount of copying when reusing an element in other lists. Lists may be of arbitrary length and can easily be modified to reorder, eliminate or insert data elements. The elements of a list need not be related by type. Most imperative languages do not offer lists as a fundamental data type, but often include libraries which implement lists.

The memory layout of a list consists of a header block which identifies the type as a list and the location of the first element. Each element consists of a data reference and a pointer to the location of the next element. The header block is 4 bytes, while each list cell occupies 8 bytes. References to data are compatible with header blocks, so that any `erlang` term may be stored as the data element of a list node. The nodes are allocated dynamically, allowing a large list to be stored without requiring a contiguous block of memory. Also, the data elements may be referenced by other data structures resulting in a net savings of memory space.

2.2 Tuple Representation

Tuples are designed for fast access to individual elements. A tuple is identified by its header which contains the size of the tuple (up

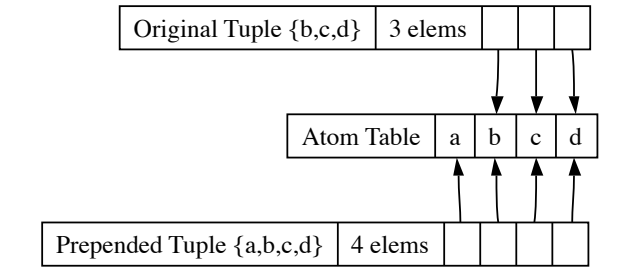

Figure 2. Prepending to a tuple.

to 26-bits), followed by an array of 4-byte element references. Any `erlang` datatype may be an element of a tuple. The structure of a tuple means that a contiguous block of memory is required.

Tuples are intended to serve the purpose that arrays serve in most imperative languages, with the added twist that they may hold any data structure as an element and the individual elements need not be of the same type. The significant differences between a list and a tuple relate to flexibility and speed. Lists require linear time to traverse, an action that is necessary to determine their length or to access an element by position. Any element of a tuple can be accessed in constant time and the length of the tuple can be determined without visiting any of its elements. On the other hand, adding, removing or changing elements in a tuple requires copying the entire tuple whereas with lists the entire sequence only need be copied if adding elements to the end.

Figure 2 shows the scenario of adding a single element to the front of a tuple for comparison with lists as shown in Figure 1. The structural approach to allow faster access results in a copy operation which a list avoids. The structure also requires that the full memory size of the tuple is allocated as a single block. This limits the practical size of a tuple that can be used, however, each element of the tuple is allocated separately. The entire tuple may occupy much more space than the top level vector of pointers.

2.3 Binary Representation

Binaries are used to represent a stream of bits. All binaries must be a multiple of 8 bits in size. A binary starts with a header that identifies the type and location of the data. There are three kinds of binaries: 1) *heap binaries*; 2) *reference counted* or *REFC* binaries; and 3) *sub-binaries*. The `erlang` programmer need not worry about the different types as the run time system hides this complexity.

Heap binaries are designed for efficient handling of small chunks of data. In R10B the maximum size of a heap binary is 64 bytes. A heap binary can be allocated on the stack or heap and consists of a single contiguous structure containing the header tag (4 bytes), the size of the binary (4 bytes), a pointer to the data and an array of 8-bit bytes representing the binary. There is only a 12 byte penalty for storing data as a heap binary. The binary must be copied on transmission to other processes if the node is not configured to share its heap across processes.

Reference counted (REFC) binaries are used when the size exceeds the maximum for a heap binary. A REFC binary consists of a header, a size, a pointer to the next REFC binary in off heap memory, and a pointer to the actual bytes. REFC binaries are automatically created by the runtime system when needed. The savings come if the binary is communicated to another process. In that operation, only the header of the binary is copied to the other process' heap memory since the pointer refers to a shared memory area used by all processes on a single `erlang` node.

Sub-binaries are fragments of larger binaries. They can be constructed from either heap binaries or REFC binaries. A sub-binary consists of a header identifying its type, a size field, an offset into the original binary, and a reference to the original binary, making a total of 16 bytes. Sub-binaries are an extremely useful and efficient representation if the code is breaking data into components that do not need any transformation. They are particularly useful in the world of write-once data since the underlying representation of the original binary cannot be modified. The run time system automatically recognizes when sub-binaries are most efficient and constructs them accordingly.

Whereas tuples and lists are implemented using pointers to other data elements, binaries are raw chunks of memory. Any modification to a binary requires the header information and the entire block of memory to be copied, instead of just the top level pointers as in the case with lists and tuples. Also, the entire chunk of allocated memory cannot be reclaimed until every reference to any portion of it is dropped, unlike both lists and tuples which can recycle the memory of elements that are no longer referenced.

Binaries were originally introduced for efficient storage, loading and transmission of `erlang` code. BIFs were required to access or manipulate them. The options were limited to `list_to_binary`, `binary_to_list`, `binary_to_term`, `term_to_binary`, `concat_binary`, `is_binary`, and `split_binary`. With these functions, splitting raw data or a stream into constituent elements was a tedious process.

2.4 Binary Bit Syntax

The need to handle telecom protocols proved that manipulation of binaries would be important as a language extension. This led to the adoption of the Bit syntax and subsequent optimization using the HiPE compiler[4].

Binaries are indicated by a leading `<<` and trailing `>>` in source code using the Bit syntax. The elements between these tokens make up the binary data. Each element in a binary expression is termed a *segment*. The following expression indicates a binary constructed from two elements that are already bound to values:

```
NewBinary = << Segment1, Segment2 >>
```

A binary expression may appear on the left side of an expression when matching to bind its values or testing already bound values; it may appear on the right side of an expression when its values are already bound. Each segment of a binary is assumed to be an integer unless specified otherwise with additional attributes. Segments may be integers, floats or binaries. The full syntax for a segment is:

```
Value:Size/Type-Signedness-Endianism-unit:Unit
```

This full expression allows for the indication of the effective size of the segment in units, the type of data, whether it is signed or unsigned, whether bits should be interpreted as high byte first (`big`) or low byte first (`little`), and the size of a single unit (specified as `unit:8` to indicate an 8-bit basis). The size of a segment in bits is computed as Size multiplied by Unit. Integers and floats default as unit:1 with a Size of 8 and 64 respectively, while binaries default to unit:8 and a Size that subsumes the rest of the binary. Floats may only be of Size 32 or 64, while integers must be a multiple of 8 and binaries can be any cardinal number of bytes. The total sum of all segments must be a multiple of 8 bits. For further details of the syntax and usage of Binary patterns, consult the `erlang` documentation[3].

Listing 1 is an example of deconstructing a binary using the Bit syntax. It shows a recursive function that extracts variable length binary records from a large binary that represents a database. The first byte indicates the length of each record.

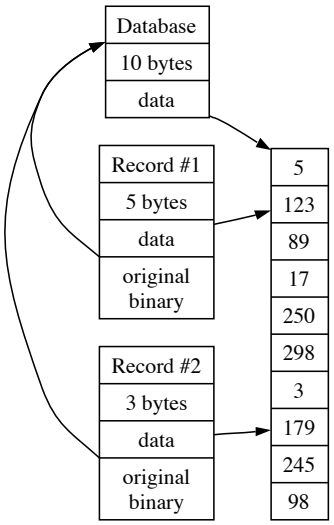

Figure 3. Extracting records from a binary using sub-binaries.

```
get_records(Bin) -> get_records(Bin, []).

get_records(<<>>, AllRecs) ->
   lists:reverse(AllRecs);
get_records(<<Size, ThisRec:Size/binary, More/binary>>,
            AllRecs) ->
   get_records(More, [ThisRec | AllRecs]).
```

Listing 1. Recursively deconstructing a binary.

In this example, we can see how a binary data element can be consumed a piece at a time. The process is very efficient because the original large binary is not modified in any way. The binary matching expressions create new sub-binaries that reference particular segments of the original, using a list to build up the set of sub-binaries. Figure 3 shows the relationship of the extracted records to the original database.

3. Stream Operations

`Erlang` makes it possible to create and manage thousands of concurrent processes. This ability leads to alternative techniques for designing applications[5] which may rely heavily on streaming data from one process to another or to and from external sources. A simple method for quickly and efficiently transforming stream data is essential for such solutions to be competitive with traditional application architectures. External sources of data include socket and port applications, disk files and packets involved in communication protocols. Data from these sources may be received as a continuous or pulsed stream of bytes, fixed length data structures or variable length data structures. In all three cases, component processes may have to deal with a large block of raw binary data, break it into constituent pieces, transform and then reassemble or route for outbound delivery.

Many situations may be described in terms of stream operations. The primary categorization considered here is in terms of the number of streams involved. Single streams are the basis of transformational data, multiple streams converging allows the consolidation of data, single stream splitting supports the differentiation of data and multiplexed inbound and outbound streams apply to complex situations with commingled data or tangled command and control communications.

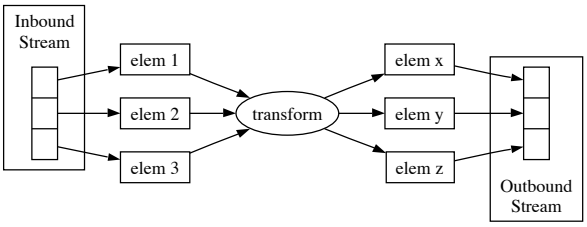

Figure 4. Canonical single stream transformation.

3.1 Single Stream Manipulation

The single stream scenario occurs when receiving data and passing it on, or when reading a file and interpreting or converting its contents. The data may be written back out to a socket or file, or it may be used in an internal format that is more convenient for computation. The operations performed on a single stream can be characterized by the pattern depicted in Figure 4. The inbound stream is split into elements, each of which can pass through a computational transformation, followed by reassembly and transmission as an outbound stream.

3.1.1 Transform / translate elements

Transformation of a stream occurs by breaking it into constituent elements, and then converting them to another format. The new elements are forwarded as a new stream or are used in an internal data structure. This is the standard approach when reading a file, whether it contains fixed-length records, variable length records as with comma-delimited value format, marked up data such as in XML or HTML, program source code, or even free text that is parsed using natural language methods. Different algorithms are used in each of these cases to determine the constituent elements, but once broken up the elements can be manipulated individually before being emitted, performing tasks such as downcasing, converting to another encoding format, transforming graphical coordinates or attributes, or other translations.

3.1.2 Compress / expand stream

Video, audio and general compression applications may use algorithms that build on the stream nature of their content. Codecs are based on compression / decompression algorithms. In this case the splitting of the data stream is more complicated because there may be interleaved data, but the basic approach is similar. Compression consists of transforming the elements by recognizing patterns and encoding them with fewer bits, while expansion reverses the process using tables and other replacement mechanisms to reinstate the original sequence of bits. Several processes may be pipelined to get from the original data to the desired final data, making several transformations in between, but using this approach can take advantage of multiple processors.

3.1.3 Reorder sequence

Applications which store data for future reference often need to sort a file of records. When dealing with peripheral input such as keyboard, mouse, and serial devices, it may be necessary to manage a merged stream of concurrent events which must be prioritized based on timestamps or datatypes. Multi-user applications require maintaining multiple chains of interleaved transactions when there is a single channel of user data. Internet music and video applications often have multiple channels of data which must be synchronized. They are delivered in a single stream, but must be differentiated to reproduce the original media accurately. In all these

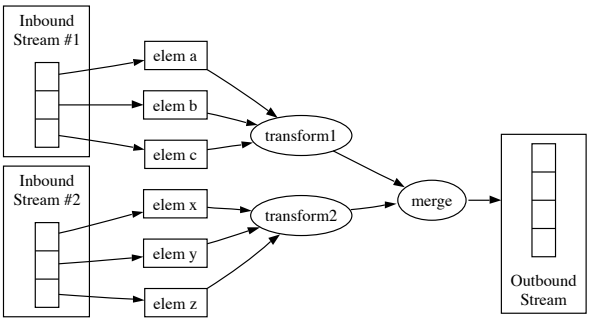

Figure 5. Canonical merging of streams.

cases, the elements of a stream are extracted, rearranged and then reassembled.

3.1.4 Filter

When handling large datasets or large streams, only a portion of the available data is generally required for the task at hand. In these situations, the elements are extracted and each is considered for elimination. Only those that survive the selection criteria are passed through to the outbound stream. Complex database queries are the most common example of this behavior, but data extraction from HTML or XML sources is becoming more prevalent. Merged transaction streams may need to be reduced based on transaction type; messaging and alert systems would overwhelm administrators if thresholding of critical events were not employed.

3.1.5 Combined transformations

All of the above techniques may be combined in arbitrary ways to produce complex transformations on the stream. A particular series of steps may be implemented in separate processes with intermediate streams between the original and the resulting stream. The architectural decision to subdivide the problem will be driven by timing constraints, completeness of data, efficiency of computation, or other considerations. An example of this form of computation is exhibited by a multipass compiler which generates parse trees, intermediate code, optimization graphs, object code and final executables. Each step contains enough computational complexity to warrant separation of concerns, while at the same time differentiating the problem in a way that would benefit from a pipelined approach.

3.2 Multiple Streams Merging

Multiple streams occur in situations with asynchronous or independent events generated by more than one process or data source. This may include multiple internet clients connected to a single server, multiple peripherals such as mouse, keyboard and scanner, multiple databases when matching and merging, data sources being formatted with multiple templates or other situations that involve coordinating or correlating data from more than one source.

Figure 5 displays the data flow involved in multi-stream merging. The input streams must be broken into elements that may be interleaved. The extraction of elements can include all the transformation examples described in the single stream explanation above. The transformed elements are then merged to construct the final output stream. Using individual processes and a network of interactions allows the format and sequence of the output stream to be easily altered or adapted for new situations.

3.2.1 Merge

Merging occurs when two or more devices, such as a keyboard and mouse, are generating independent events on the same channel. It also applies when multiple users are accessing the same server and a single log file or transaction history is desired, or the individuals are contributing to a single data repository. Integration of the transactions or stream elements generally depends on the contents of the elements in comparison to each other, producing a chronological stream, an element ordered stream or a constraint based ordering of elements. Any number of streams can be merged in this manner if their elements can be transformed to compatible data types. Adding one additional stream with control information allows yet another independent source determine the ordering or transformation of elements prior to production of the final output stream.

3.2.2 Composition

Functional languages have their roots in the mathematical combinations of function arguments or even functions themselves. Sensor systems with multiple external sensors can provide real time data streams which must be combined mathematically to produce an accurate data model of the physical world. The merge capability can be implemented as a function which takes one or more arguments from each of the synchronized streams to produce a real time output stream of the consolidated sensor information. Tracking systems, secure perimeter violation detectors, real time targeting and other monitoring situations are ideal candidates for this approach.

3.2.3 Control flow / synchronization

In the description of Merge operations, multiple streams were interleaved based on the data contained within the elements from each stream. In a control flow or synchronization architecture, the ordering of elements is determined either by embedded control information or by a separate stream containing control information. The Synchronized Multimedia Integration Language (SMIL) protocol[1] exhibits embedded control by defining commands that may occur within a stream which identify that elements may occur in parallel, sequentially or exclusively among other synchronization choices. A bandwidth reservation or prioritization scheme may be implemented using an extra control stream for providing command and control indicating packets in the outbound stream to be interleaved.

3.3 Single Stream Splitting

The splitting of a single stream occurs on the receiving end of multiplexed data which is communicated over a single channel for encryption or bandwidth efficiency reasons. It is also useful when a database contains several scenarios or types of data that must be differentiated and distributed independently, or a single data stream supplies information for more than one database or storage location. The canonical approach is the exact mirror of Figure 5 where outbound and inbound are swapped, ”merge’ is replaced by ”split” and the arrowheads point from right to left.

3.3.1 Route / dispatch

In routing applications, several types of data or several instances of the same data type are destined for a different location. Often the addressing information is included in the stream, as in the case with delivery of email[2]. The elements of the stream are differentiated, then the addressing information is consulted and the discrete elements are routed to the requested destinations. This situation also arises when a single internet based server is used to service multiple types of requests. The transaction stream of requests is directed at a single server, but can be desegregated behind the server in the middleware so that the individual requests

can be routed to different applications. Communication packets are distributed in a similar manner by phone and internet network switches.

3.3.2 Prioritize

Consolidated streams may cause high priority data packets to be delayed behind lower priority requests. As a piece of the stream is received, a prioritization architecture would disassemble the stream into elements, verify the priority of each of the elements and deliver them according to priority. Lower priority elements would wait until no further higher priority elements have been received on the stream. Situations where real time data is interspersed with non-real time data requires a prioritization mechanism or bandwidth reservation scheme to ensure continuous retransmission.

3.4 Multiplexing

A complex network may have a central switching point (peer-to-peer networks should be considered as single stream communications once two peers have established a connection) which handles data routing and transformation. These situations generally combine all the techniques seen in the previous three scenarios. Data streams are generated by multiple sources, some of which may include control streams, and they converge at the central switching server. Its job is to coalesce the streamed elements, ordering and routing them to one or more destinations, maintaining throughput and consistency. One example is the simultaneous coordination of multiple input devices and multiple output devices or display windows. In this scenario, the appearance of concurrency is normally achieved by rapid sequential output in multiple streams, but the use of multiple processes or even multiple processors allows for true concurrency when the federation of processes represents a multiplexing architecture.

Some form of control and synchronization is generally necessary to maintain data independence and responsiveness of the overall system. In a fully networked environment, switching systems may be implemented where every participant has the ability to address a message to every other participant. It is the responsibility of the switching mechanism to maintain the multiple connectivity channels, to route all messages efficiently in the light of competing interests and to manage the allowed capabilities for each participant, enforcing security constraints, terms of service and quality considerations. Internet services that allow community interaction, such as multiplayer game sites with multiple simultaneous games or chat servers with multiple chat rooms, can be architected as a multiplexing application.

4. Stream BIFs

The key elements of all the architectures described above are: 1) splitting; 2) transforming; 3) filtering; 4) composing; 5) reordering; 6) routing; and 7) merging. Any stream application needs efficient implementations of these elements. As described earlier, lists support an efficient means for dealing with dynamically expanding and contracting data sets, whereas tuples are suited towards collections of data elements that are primarily accessed in a read only manner via indexing. Binaries provide a more compact representation than either lists or tuples, but at the expense of great cost for even a single byte modification. Tuples are the most limited in terms of the number and capability of BIF functions, binaries are next due to the high flexibility of the Bit syntax in spite of the lack of binary BIFs, while lists have the largest variety of BIFs available as well as a strong pattern syntax for manipulating elements and comprehensions for the list as a whole. The `lists` package in `stdlib` provides several alternatives for rearranging, substituting, deleting and otherwise altering a list.

Stream based applications need to be very efficient in both memory and time. Binaries offer the promise of high efficiency in memory usage, not only because of their compact representation but because they can be broken into sub-binaries without the penalty of copying data. Since REFC binaries are stored in memory that is shared by all processes on a single node, they have the added benefit of reducing message transmission overhead.

The following sections address a subset of the stream operations outlined above, with a description of a binary based implementation provided by the author for each of the BIFs. In general, copying of binary data is avoided through the use of sub-binaries which overlay the original binary. This works even in cases where the passed in binary stream itself was created as a sub-binary. These BIFs are extremely efficient when the data is merely carved up and passed on because neither block memory allocation nor copying of memory blocks is necessary. When data is transformed, the least amount of copying is performed to maintain high-speed and efficient memory usage. Performance comparisons are provided in Appendix A.

The set of functions implemented and benchmarked in this study are:

stream_xlate(Data, Map)
stream_xlate(Data, Map, Filter)
stream_extract(Data, [Pos1, Pos2, ...], Size)
stream_extract(Data, [{Pos1, Size1},{Pos2, Size2}, ...])
stream_extract_nth(Data, RecSize, ExtractSize)
stream_split_rll(Data, RecLengthSize)

These functions are used to split an input stream into constituent elements. They each apply to different common situations or formats. The main goal was to provide a very fast mechanism for absorbing large input streams and turning them into collections of elements without running out of memory, since tests with normal `erlang` functions encountered memory consumption problems.

4.1 stream_xlate

ASCII streams are commonly used to communicate textual information. They can also be generated by reading a file or retrieving HTML files. Direct, byte by byte translation is used when converting text to lowercase or uppercase. Graphical data may also be represented as a stream of bytes where each represents pixels and colors. Translation via filters and colorspace transformations involve similar conversions. A general approach to this problem would be to apply a function every time a series of bits must be converted, however, standard `erlang` functions may not be called directly from a BIF.

The function `stream_xlate/2` takes a binary data block and a binary representing a 256-byte lookup table. The use of a binary lookup table was chosen in lieu of a function call for each byte to translate, and to provide a speed advantage by using direct C-based array indexing to perform character conversions. The BIF directly allocates a new binary of the same size as the original. Each original byte is used as an index into the lookup table and the corresponding element is written to the newly allocated binary which is subsequently returned. The speed of this BIF was increased by a factor of two by unrolling the loop across the original binary and performing it ten bytes at a time in line before looping. This function is as efficient as possible with memory usage because it does not create any intermediate data structures, nor does it generate anything for the garbage collector to recycle. The memory efficiency is dramatically demonstrated in the benchmark, allowing the runtime to handle a binary that is between 20 and 200 times larger than the largest allowed with a similar approach using the Bit syntax.

The function `stream_xlate/3` extends the interface to include a 256-byte binary filter table. The table serves the purpose of a filter

function, only allowing the translation of those bytes which pass the filter. The BIF allocates a new binary and visits each byte in the original as before, except that any byte that returns zero when used to index the filter binary is copied as is to the new binary. All other bytes are translated via the map as before. A single map can be set up once, and multiple filters may be used to apply only portions of the map to particular streams. Again, the performance increase and memory reduction are quite dramatic, allowing the solution to problems that would normally crash the runtime.

4.2 stream_extract

When a control stream accompanies a data stream, the location of important data may be communicated independently from the data itself. In these cases, a list of the relevant locations and sizes of key elements can be compiled prior to extraction. Similarly, a user may request specific elements from a database or a graphical system may attempt to transform, translate or clip particular segments of the display data stream. Knowing the position and size ahead of time allows the excision of specific portions of the data and subsequent efficiency in handling the data subsets rather than the entire stream.

The function `stream_extract/3` supports the extraction of a series of fields, all of which are the same length. The most common situation is a database of records with a single element such as a transaction date to be matched or transaction amount to be accumulated. It applies equally well to graphical data when a rectangular region is to be treated without affecting the rest of the data stream. This BIF determines the length of the list of desired extract positions and then allocates a block of memory to hold all the sub-binaries that will be created. For each element of the positions list it then fills out a sub-binary with the correct size starting in the position specified. The list of sub-binaries is constructed and linked so that they do not need to be reversed on return.

The function `stream_extract/2` accepts a binary representing the database of records and a list of position and size pairs. The difference with this BIF is that the extracted data segments may be of differing sizes, whereas in the previous BIF all were the same size. The same approach is used, precomputing the number of sub-binaries to allocate and then linking them in the proper order.

4.3 stream_extract_nth

Fixed length records and fields are common in binary data formats. Packet sizes are predictable and file sectors can be optimized when data is broken into fixed size pieces. When a stream of fixed length records is received, it can be broken up using simple size calculations for extraction of particular fields. To deal with an entire database, the individual records are split from the stream and distinct fields from the records. While less flexible, fixed format records can outperform variable length records because the location of a particular record or field may be computed ahead of time. The drawback is that the full space must be allocated even if the fields do not contain enough data to fill the predetermined size.

The function `stream_extract_nth/3` takes a binary representing the entire database or set of records, a record size and a field size to be extracted. A new list is allocated of the correct size by dividing the size of the binary based on the record size. The BIF then iterates over the original binary creating fixed size sub-binaries from each record. The sub-binaries are stored in a preallocated list, walking from the front of the list to the back. The use of sub-binaries prevents any copying of the original data, so the BIF ensures minimum memory usage and minimum CPU effort.

The extraction always starts at the first position at the beginning of the binary. If the desired extract is not at the first position, pass in a sub-binary that starts at the first byte of the extract using the Bit syntax to do so. To extract entire records, make the size of the record and the size of the extract match. This single BIF can then extract full records, multiple adjacent fields, or a single field from each of a concatenated set of fixed length records.

4.4 stream_split_rll

Communications protocols are often designed to deal with bandwidth constrained situations. A common technique is to use an encoding mechanism which identifies the length of the next packet, followed by the corresponding number of bytes. This reduces the storage space over a fixed record size protocol. Handling these types of protocols needs to be efficient because the protocol is specifically designed to overcome a resource constraint.

The function `stream_split_rll/2` attempts to automatically split a stream of length encoded records. It takes a binary representing the database of records or stream of packets and a parameter indicating the size of the record length field in bytes. The record length is assumed to be the first field in the binary and to always occupy the number of bytes indicated by the caller. First the BIF determines the number of records in the stream and whether the encoding works out to the last byte, returning an error if it does not. Next it allocates a block of sub-binaries to hold the number of records discovered. Finally it iterates over the binary again, constructing sub-binaries which match up to the records without the size header, since binaries contain their size already. A matching merge function is needed to reconstruct the stream from sub-binaries since the length of each needs to be extracted from the sub-binary header and inserted before each record.

5. Conclusion

`Erlang`'s facilities support a stream-based architectural approach to constructing applications, however, the existing ability to split binaries comes with memory and speed costs. Once a binary is split, the elements can be efficiently used natively in data structures and functions. In most cases, the `list_to_binary/1` BIF or the ability to write nested lists of binaries directly to sockets, ports and files provides enough functionality to merge streams for output. The BIFs discovered in this study prove to be a good starting point for a stream library addition to the runtime system. Further work is needed to produce a larger set of useful BIFs based on experience with a variety of input formats. In addition, a convenience library is needed for standard formats and protocols such as comma-delimited values, HTML and XML formats, or standard communication protocols used for chat, multimedia and telephony applications. This library can most likely be implemented using native `erlang` functions which call on stream splitting and merging BIFs. A growing library with community contribution could produce a very robust toolset for implementing streaming applications.

Experiments with full applications are needed to determine if the stream based approach is competitive with traditional database applications, graphics systems, communication protocols and internet based services. The memory and speed measures demonstrated here offer promise in producing viable alternatives to traditional applications. The hope is that stream based architectures lead to simpler and more accurate implementations because of their approach to deconstructing applications.

Acknowledgments

Special thanks to Kostis Sagonas and the rest of the `erlang` review committee for their excellent recommendations and guidance in organizing and presenting the views of this paper. All credit for any remaining opinions, mistakes, omissions or other errors is mine alone.

References

[1] D. Bulterman, et al. Synchronized Multimedia Integration Language (SMIL 2.1) *W3C Synchronized Multimedia Website* Available at http://www.w3.org/TR/2005/CR-SMIL2-20050513/.

[2] D. Crocker. RFC 822 - Standard for the format of ARPA Internet text messages *Internet RFC/STD/FYI/BCP Archives*, Aug. 1982. Available at http://www.faqs.org/rfcs/rfc822.html.

[3] Ericsson AB. Erlang/OTP R10B Documentation. Available at http://www.erlang.org/doc.html.

[4] P. Gustafsson and K. Sagonas. Native code compilation of Erlang's bit syntax. In *Proceedings of the ACM SIGPLAN Erlang Workshop*, Oct 2002. Available at http://www.erlang.se/workshop/2002/.

[5] J. Nelson. Structured Programming Using Processes. In *Proceedings of the 3rd ACM SIGPLAN Erlang Workshop*, Sept 2004. Available at http://www.erlang.se/workshop/2004/.

[6] C. Wikstrm and T. Rogvall. Protocol programming in Erlang using binaries. In *Proceedings of the Fifth International Erlang/OTP User Conference*, Oct. 1999. Available at http://www.erlang.se/euc/99/.

A. Performance Results

All the BIFs provided in the Streams Library described above could be implemented directly in `erlang` using the binary Bit syntax and list operators, however, using the BIFs resulted in a 3 to 250-fold speed up. The performance numbers were the fastest results in milliseconds as measured by `timer:tc(Module, Function, Args)` in an `erlang` shell. Timings were taken with Beam compiled code and HiPE compiled code where possible. The test system was an Apple iMac G5 1.8 GHz with 1.25GB DDR SDRAM, running Mac OS X v10.3.9 and OTP R10B-5 compiled natively.

The code used in the performance tests is available from the author's website at `http://www.duomark.com/erlang/`.

A.1 stream_xlate

The `stream_xlate` benchmark shown in Table 1 consisted of converting a 500.000 byte binary containing ASCII alphabetic characters to lowercase. The baseline approach was to convert the binary to a list, use a tail recursive function to build a new list, reverse it and then concatenate back to a binary. The Bit syntax approach was similar to the baseline approach except that each element was extracted an integer at a time from the binary using a matching pattern. The translation with filter BIF only converted characters between `$A` and `$Z`, copying as is when not matching the filter. The translation without filtering used a map of all characters with the non-uppercase characters mapped to themselves.

Larger binaries were tried in an attempt to see where problems would crop up. Both the `binary_to_list` approach and the Bit syntax approach were successful with a 5M byte binary, but crashed the runtime with a 50M byte binary. The BIF succeeded with 1.05B bytes in 104.5 seconds, but failed with 1.1B bytes giving a message that memory could not be allocated from the heap.

A.2 stream_extract/3

The `stream_extract/3` benchmark shown in Table 2 consisted of extracting 1000 random records from a 500.000 byte binary containing records of either 20 bytes or 5000 bytes. The baseline approach was to use Bit syntax to extract the records, collecting them in a list that is reversed on return. The slow approach was to convert the entire binary to a list and use the `lists:sublist/2` BIF to extract records followed by reversing the collected binaries. In the tuple approach, the binary was first split into a tuple of binary records of fixed length, and then the `element/2` BIF was used to extract the list of records which was reversed on return.

Record size did not seem to impact performance, except for the tuple approach. The speed difference in the tuple case can be attributed to the overhead of subdividing the binary before performing the record extractions, although there may also be an indexing efficiency boundary crossed when comparing the access to a tuple of 100 elements versus a tuple of 25,000 elements. The difference between using Bit syntax and the new BIF is primarily caused by the construction and reversal of a list in the Bit syntax case. Using the HiPE compiler caused the system to crash with a 500,000 byte binary, however, successful runs were possible with a 500 byte binary consisting of 25 records of 20 bytes each. The BIF proved faster even with Bit syntax compiled with HiPE, although a larger example would be needed for a conclusive measure of performance.

A.3 stream_extract/2

The `stream_extract/2` benchmark shown in Table 3 consisted of extracting random blocks of data from a 500.000 byte binary. Both the position and size of each block were random, with the size being from 1 to 20 bytes in the first case and from 1 to 5000 bytes in the second case. The same techniques were used as in `stream_extract/3` except that the tuple approach could no longer apply as the record lengths are not predetermined, but instead vary with each selected element.

A.4 stream_extract_nth

The `stream_extract_nth/3` benchmark shown in Table 4 consisted of extracting a fixed size block of data from each record in a 500.000 byte binary. The size of each record was 20 and 5000 in the two tests, with a field of size 7 and 1700 extracted from each respectively. The binary_to_list case was the naive approach of converting the entire binary to a list and then using `lists:sublist/3` to extract the field and `lists:nthtail/2` to remove the record from the front of the list before recursing for the next record. The fields were collected in a list that was reversed on return. The baseline test consisted of converting the binary to a list of binaries matching the record size first, followed by Bit syntax to extract the beginning field in the record. The Bit syntax approach avoided the initial splitting of the binary and just used an ever increasing skip field at the beginning of the match to get to the next record in the sequence. In both of these cases, a list of fields was accumulated and reversed on return. The Bit syntax approach is very similar to the BIF, since both use sub-binaries to represent the result, however, the Bit syntax needs to accumulate and reverse a list whereas the BIF preallocates and links the list in the correct order from the start. This BIF offered the least speed up when extract length of each element was small, but performance benefits increased significantly with longer extracts.

A.5 stream_split_rll

The `stream_split_rll/2` benchmark shown in Table 5 consisted of extracting variable sized records from a block of 30.000 records. The set of records was generated with a random size of between 1 and 255 bytes each. The same seed was used for all tests, resulting in a total binary size of the concatenated record set of 3.850.182 bytes. In the first test, the entire binary was converted to a list, followed by using `lists:sublist/3` and `lists:nthtail/2` to extract the record and skip to the next record respectively, accumulating records in a list and then reversing it. The splitting Bit syntax approach consisted of a recursive loop that extracted the size and record from the front of the binary and then continued with the remainder of the binary, reversing the collected list of binary records. The whole Bit syntax approach recursively skipped over the part of the binary that was already extracted, and pulled out the record while incrementing the size of the portion already seen, reversing the accumulated list to return the result. In this case HiPE compiling worked and managed to increase the performance of the Bit syntax, however, the BIF saved the effort of constructing a list and reversing it with a resultant increase in speed.

Approach	Beam	Speed Up	HiPE	Speed Up
binary_to_list	414.000	1.0x	345.000	1.2x
Bit syntax tail recursion	435.000	0.9x	372.000	1.1x
stream_xlate(Bin, Map, Filter)	5.620	74x	5.640	73x
stream_xlate(Bin, Map)	1.650	251x	1.650	251x

Table 1. stream_xlate performance

Approach	Beam (20)	Speed Up	Beam (5000)	Speed Up
binary_to_list	19.000.000	N/A	19.500.000	N/A
Bit syntax	404	1.0x	420	1.0x
Tuple	9.500	0.4x	210	1.9x
stream_extract/3	56	7.2x	57	7.1x

Table 2. stream_extract/3 performance

Approach	Beam (20)	Speed Up	Beam (5000)	Speed Up
binary_to_list	19.360.000	N/A	19.960.000	N/A
Bit syntax	310	1.0x	307	1.0x
stream_extract/2	80	3.9x	81	3.8x

Table 3. stream_extract/2 performance

Approach	Beam (20)	Speed Up	Beam (5000)	Speed Up
binary_to_list	180.300	N/A	174.800	N/A
List of binary records	15.300	1.0x	78	1.0x
Bit syntax	9.950	1.5x	68	1.1x
stream_extract_nth	9.790	1.6x	30	2.6x

Table 4. stream_extract_nth/3 performance

Approach	Beam	Speed Up	HiPE	Speed Up
binary_to_list	2.276.000	N/A	2.251.000	N/A
Bit syntax splitting	18.145	1.0x	15.789	1.1x
Bit syntax whole	14.893	1.2x	14.258	1.3x
stream_split_rll	62.72	2.9x	6.183	2.9x

Table 5. stream_split_rll/2 performance