A Study of Erlang ETS Table Implementations and Performance

Scott Lystig Fritchie
Snookles Music Consulting
Minneapolis, Minnesota, USA
slfritchie@snookles.com

ABSTRACT

The viability of implementing an in-memory database, Erlang ETS, using a relatively-new data structure, called a Judy array, was studied by comparing the performance of ETS tables based on four data structures: AVL balanced binary trees, B-trees, resizable linear hash tables, and Judy arrays. The benchmarks used workloads of sequentially- and randomly-ordered keys at table populations from 700 keys to 54 million keys.

Benchmark results show that ETS table insertion, lookup, and update operations on Judy-based tables are significantly faster than all other table types for tables that exceed CPU data cache size (70,000 keys or more). The relative speed of Judy-based tables improves as table populations grow to 54 million keys and memory usage approaches 3GB. Term deletion and table traversal operations by Judy-based tables are slower than the linear hash table-based type, but the additional cost of the deletion operation is smaller than the combined savings of the other operations.

Resizing a hash table to 2^{32} buckets, managed by a Judy array, creates the most consistent performance improvements and uses only about 6% more memory than a regular hash table. Other applications could benefit substantially by this application of Judy arrays.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Data Types and Structures*; E.1 [**Data**]: Data Structures—*Trees*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Performance evaluation (efficiency and effectiveness)*

General Terms

AVL tree, B-tree, Erlang, hash table, in-memory database, Judy array

1. INTRODUCTION

The Erlang Open Telecom Platform (Erlang/OTP) has an efficient in-memory database known as ETS tables (Erlang term storage) for storing and retrieving Erlang data terms. As RAM prices continue to fall and as the gap between CPU instruction execution latency and RAM access latency widens, it becomes increasingly worthwhile to examine the performance of large ETS tables populated with millions of terms. This research compares the efficiency of the current ETS table implementations with several new implementations: one based on an in-memory B-tree and three based on a relatively new data structure called a Judy array [3].

The Judy array turns out to be an excellent data structure for building in-memory datastores that exceed CPU cache size. A performance analysis of seven small benchmark programs shows that an ETS table implemented with a Judy array usually runs faster than tables implemented with a resizable linear hash table and much faster than tables implemented with AVL trees or B-trees. In cases where Judy-based tables are slower, the combined speed advantage of the faster operations outweighs the penalty of the slower ones.

The primary audience for this research is Erlang developers, most of whom use ETS tables in their applications directly or indirectly via Mnesia [11], an important distributed database application written in Erlang. ETS tables are the bedrock on which Mnesia is built.

A secondary audience is the much larger community of C and C++ developers. Many algorithms that they use daily rely on hash tables for in-memory datastores of one kind or another. Most of those developers assume that their only options for optimizing the datastore portion of their applications are either (a) to tune the hash function or (b) to adjust the size of the hash table. This paper proposes another option: (c) to use a Judy array to create a really big hash table, 2^{32} hash buckets, to reduce the time spent searching and managing collision lists.

This paper is structured as follows. Newcomers to Erlang are given a brief introduction to Erlang in Section 2 to aid them in understanding some of the Erlang jargon and syntax that Erlang programmers take for granted. Section 3 presents a summary of existing ETS table types and their implementations. Then the focus of attention shifts to a new data structure, the Judy array. Section 4 introduces

```
-module(test).
-export([square/1]).

% This is a comment.

square(X) -> mult(X, X).

mult(X, Y) -> X * Y.
```

Figure 1: A simple Erlang source module

the reader to the Judy array and what it looks like from a user's point of view. Section 5 describes how Judy arrays are used to implement three new ETS table types. The benchmark programs using all of the ETS table types are discussed next. Section 6 explains several of the design decisions made while creating the benchmark programs. Section 7 analyzes the results of seven ETS benchmark programs. Section 8 presents a small survey of related work. The paper ends with Section 9 naming areas for future research and Section 10 presenting the conclusion.

2. ERLANG INTRODUCTION

This section provides a minimal Erlang primer so that Erlang neophytes can understand the syntax used in the later sections of this paper. For greater detail, see the original reference book on the Erlang language [1].

2.1 Erlang data types

Erlang terms can be divided into two general categories, simple and complex. Three simple term types are used in this paper:

- **Numbers** Numbers may be integers or floating point numbers. Integers may grow beyond native CPU word length to any size (i.e., "bignum" support). Syntax examples: `-4`, `6.02e23`, `3141592653589`.
- **Atoms** As in LISP, atoms are constants with human-friendly names. Erlang atoms must begin with a lower-case letter or must be enclosed in single quotes. Syntax examples: `atom1`, `'ATOM2'`, `'$foo'`.
- **Binaries** The binary data type represents a sequence of bytes stored contiguously within the Erlang virtual machine. Syntax examples: `<<115,99,111,116,116>>`, `<<"scott">>`. The latter is syntactic sugar for the former; both specify the ASCII character codes for the string "scott".

The two complex term types used in this paper are lists and tuples. An Erlang list, like a LISP list, may be of arbitrary length, and its elements may be any data type, including other lists or tuples. Erlang tuples are similar to lists but are fixed length. A tuple's constant length allows $O(1)$ access to any element within the tuple. Example syntax of a three-element list and a tuple are `[1,2,3]` and `{1,2,3}`, respectively. The syntax `"scott"` is syntactic sugar for a list containing the ASCII character codes for the string "scott" (i.e., `[115,99,111,116,116]`).

2.2 Erlang syntax and virtual machine

Figure 1 contains the code for an Erlang source "module" called `test`. The notation `F/N` denotes a function `F` that has N arguments, for example, `square/1`. Within a module, functions may be called by their simple name, for example, `mult(X, X)`. Outside of a module, however, only functions named in the `-export` attribute's list are callable. Furthermore, they must be called by a fully-qualified name using the syntax `module:func(`$\mathcal{A}$`)`, where $\mathcal{A}$ represents zero or more arguments, for example, `test:square(5)`. Specifying the module name `erlang` is optional when calling an Erlang built-in function (BIF), for example, `date()`.

Erlang programs may be compiled into abstract byte code or native executable code. Both types of code are executed by a host operating system process that implements the Erlang virtual machine. Like the Java virtual machine, the Erlang virtual machine provides services such as consistent access to host operating system resources, memory management and garbage collection services, and exception handling facilities.

3. ETS TABLES

ETS is an acronym for Erlang Term Storage. ETS permits Erlang programs to store large amounts of data in memory with $O(\log N)$ or $O(1)$ access time for tables with sorted and unsorted keys, respectively. This section presents an overview of what ETS tables are, then briefly explains the behavior and underlying data structures of existing ETS table types as well as the new ETS table types developed for comparison purposes.

3.1 ETS table overview

Conceptually, an ETS table is a key-value database. In many ways, an ETS table is analogous to associative arrays found in many other languages, such as Perl's hash and Python's dictionary types. Familiar operations such as key insertion, query, and deletion are supported. Table traversal operations such as "get first item" and "get next item" are also available. However, ETS tables also support additional features such as pattern-matching queries, for example, match all tuples where the first element is an integer less than 5 and the third element is the atom `louise`.

Unlike many other databases, ETS imposes only two constraints on the types of data that it stores:

1. All terms stored in an ETS table must be tuples.
2. A term's key position is defined at table creation time. The default is the first element, but the key may be configured to be any element within the tuple.

These two rules allow the programmer a great deal of flexibility. Any Erlang term, be it a simple number or a very large list or a deeply-nested tuple, may be used as the key for an ETS-stored tuple. Furthermore, there are no restrictions on how many elements the tuple may have (without violating rule #2) or on the data type of each tuple element.

The type of an ETS table is defined when the table is instantiated. The virtual machine may have over a thousand active ETS tables of different types at one time, and the compile-time limit may be overridden by an environment variable when the virtual machine is started.

For performance reasons, ETS is primarily implemented as BIFs inside the Erlang virtual machine. All BIFs and their supporting functions, together with the rest of the virtual machine, are written in C.

3.2 Ordered ETS tables

The current release of Erlang, version R9B, provides one type of ordered ETS table: the `ordered_set` type. The semantics of an `ordered_set` ETS table require that traversal of the table using functions such as "get first key" and "get next key" must return keys in sorted order. Only one tuple with any key K may be stored in the table at any time. If a tuple with key K is inserted into an `ordered_set` table, and a tuple with that key is already present in the table, the old tuple will be replaced by the new tuple.

The sorting order of an `ordered_set` table's keys is maintained by a standard AVL balanced binary tree. The AVL tree data structure dictates that operations on the tree take $O(\log N)$ time to complete, where N is the number of tuples stored in the table.

Because an ETS table key may be an arbitrary Erlang term, a sorting order is defined in order to compare terms of different types. For example, the value of `42 >= "mark"` is false. The sorting order is as follows: numbers < atoms < tuples < the empty list `[]` < non-empty lists < binaries. This rule is applied recursively to the elements of lists and tuples in order to break a tie.

3.3 Unordered ETS tables

Erlang provides three types of unordered ETS tables: `set`, `bag`, and `duplicate_bag`. A table of type `set`, like the `ordered_set` type, may store only a single tuple using any key K. A `bag` type table may store multiple tuples using K, but the value of each tuple must be different. A `duplicate_bag` type table may use K to store multiple copies of the exact same tuple.

The unordered ETS table types are implemented using a linear hash bucket array. The hash function folds all of the parts of the key term into an unsigned long integer. The hash bucket array is automatically resized as hash bucket populations rise above or fall below a compile-time constant, `CHAIN_LEN`. The default value of `CHAIN_LEN` is six items.

3.4 New ETS table types

To supplement the built-in ETS table types, the author added four new table types to the Erlang virtual machine. The first type is the `btree` type. The `btree` ETS table type is implemented using an in-memory B-tree structure.

A B-tree is a $2M$-ary tree where all paths from the root node to a leaf node are the same length. In addition, a B-tree limits the number of items stored in any node (except the root) to be between M and $2M$ items. The performance results in Section 7 demonstrate B-tree run-time behavior with $M = 4$.

B-trees share three characteristics with AVL trees. First, key sorting order is preserved by both tree types. Second, insertion and deletion operations may affect multiple nodes as the tree is rebalanced. Third, operations on the tree take $O(\log N)$ time to complete.

The other research ETS table types are based on Judy arrays: the `judysl`, `judyesl`, and `judyeh` types. Their implementation is discussed in detail in Section 5.

4. THE JUDY ARRAY

The Judy array[1] was invented by Doug Baskins while working at Hewlett-Packard. This data structure is relatively new and has not been mentioned in any publications that Baskins or this author is aware of. The Judy array source code has been released to the Open Source community under the GNU Lesser General Public License and is available at this time at [3]. Documents describing the implementation of Judy arrays can be found online at [2] and [13].

The Judy array's inventor claims that it can operate well on big- and little-endian CPUs, 64-bit and 32-bit CPUs, with small or exceptionally large data populations, and with sparse or dense populations without external tuning parameters in a memory-efficient manner that, in most cases, is as fast or faster than traditional hashing techniques and much faster than tree-based algorithms. Fortunately, Judy array source code is available with an Open Source license [5] for independent testing.

4.1 Judy1 and JudyL arrays

From an application programming interface point of view, a Judy array is simply a dynamically-sized array. The two principal varieties of Judy arrays, called Judy1 and JudyL, use a processor-dependent word, 32 bits or 64 bits, as the array index. (To simplify descriptions, this paper will assume that the size of a C `unsigned long` integer is 32 bits.) Each value stored in a Judy1 array is a single bit. Each value stored in a JudyL array is a word. The sorting order of indices in the array is preserved.

Basic Judy array operations include the following: insert index I into the array, delete index I, find index I, find the first/last index in the array, and find the previous/next index in the array that precedes/follows index I. Other operations include finding the previous/next unused index preceding/following index I, counting the indices stored in the array between index I and I', and finding the N^{th} index stored in the array.

Judy arrays do not require explicit initialization. There are no tuning parameters: there is no need to specify how many indices will eventually be stored in the array, what the index range will eventually be, or if the index population will be dense or sparse. There is no need to specify a custom hashing function or index comparison function. Judy arrays use aggressive compression techniques to try to minimize memory consumption.

From an implementation point of view, Judy arrays are tries (also called digital search trees). A trie is a search tree

[1]The array is named after the inventor's sister.

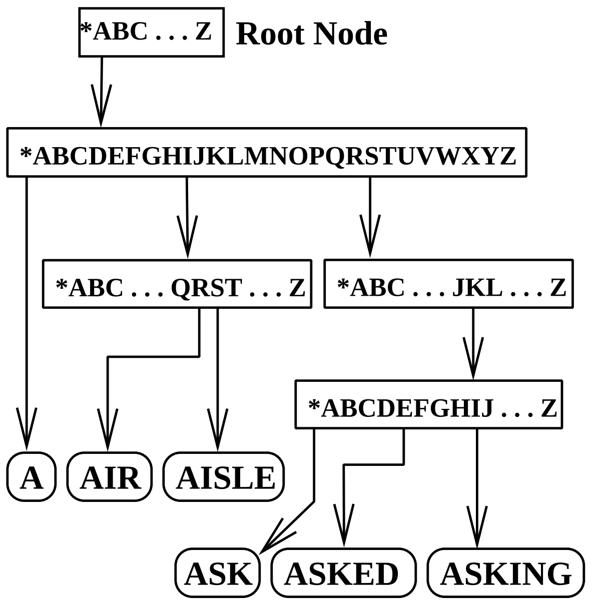

Figure 2: A trie containing six English words

where portions of the index key are stored in different nodes throughout the tree. As the search tree is traversed from root to leaf, the index key is reconstructed from data in each visited node. See Figure 2 for a diagram of a 27-ary trie that stores six English words.

By design, tries are not height-balanced trees. Their strength lies in predictable height for any given search key.

Tries can be very efficient time-wise: search time is dependent upon the length of the search key, not upon the number of keys stored in the trie. However, naïve trie implementations are very inefficient space-wise, except at very high population densities. The trie in Figure 2 uses only 10 of the 135 total pointers inside the five trie nodes. The trie would have even more nodes if it were not using a common space-saving technique that stores unique word suffixes outside of the trie (e.g., the letters "LE" in "AISLE").

Judy arrays are implemented as a logical 256-ary trie: one byte's worth of key per level. A naïve 256-ary trie implementation would waste enormous amounts of space at all but the highest population densities. Judy uses node compression techniques that may store multiple bytes of the key at any particular level of the trie. Judy selects the number of key bytes at a particular level by the population density in that portion of the sub-trie. According to [2], approximately 25 major data structures and a similar number of minor data structures are used to implement Judy's node compression. This is in stark contrast to the one or occasionally two data structures typically used to implement most traditional trees.

While the Judy array attempts to save memory by using dozens of data structures for node compression, it also attempts to minimize execution time by optimizing the layout of those data structures to avoid CPU cache line fill operations. CPU cache line fill operations can be very expensive

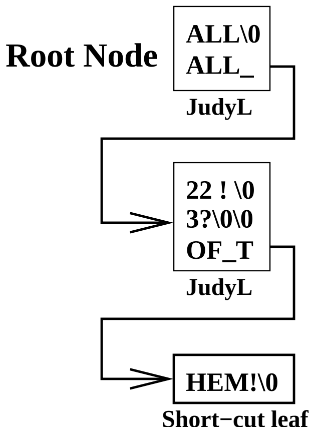

Figure 3: A JudySL array containing four strings

time-wise. Main memory access time is 2–3 orders of magnitude slower than CPU register access time. If all data is in registers or fast local cache, a CPU can execute dozens to many hundreds of instructions in the same amount of time that it can wait for a single cache line fill operation to finish.

A Judy array with a population of 1–31 keys is stored in a single-level trie: all keys and their values are stored in the root node. This may seem like a counter-intuitive speed optimization, but sequentially searching 31 keys in the root Judy trie node incurs a lower worst-case CPU cache line fill rate than a traditional search tree's worst-case.

Once a Judy array's population exceeds 31 keys, the trie grows beyond one level. Each level of the trie may store an additional 1–3 bytes of the key. Not all search paths through the trie may be the same length: key population may be dense in one part of the sub-trie and sparse in another. See [13] for a detailed explanation of the complexity that erupts once a Judy tree grows beyond one level.

4.2 The JudySL array

The Judy source distribution contains a third data structure library: the JudySL array. Like a JudyL array, a JudySL's value is a word. However, the key for a JudySL array is a NUL-terminated (ASCII NUL character) string. Figure 3 contains a diagram of a simple JudySL array that stores the strings "ALL", "ALL_22!", "ALL_3?", and "ALL_OF_THEM!".

JudySL is implemented as a trie layered on top of JudyL arrays. Each four bytes of the key string are used as the key for a JudyL array. The value of that JudyL array is either (a) a pointer to a JudyL array that stores the next four bytes of the index string or (b) a pointer to a "short-cut leaf" that stores the remaining bytes of the key in a single contiguous buffer.

A JudySL tree inherits all of JudyL's space-efficiency and execution speed traits. JudySL has two other desirable traits. First, like JudyL, it preserves the lexigraphic sorting order of the keys that it stores. Second, it often requires less memory to store strings in a JudySL array than required to store the strings themselves, due to the trie's common prefix sharing.

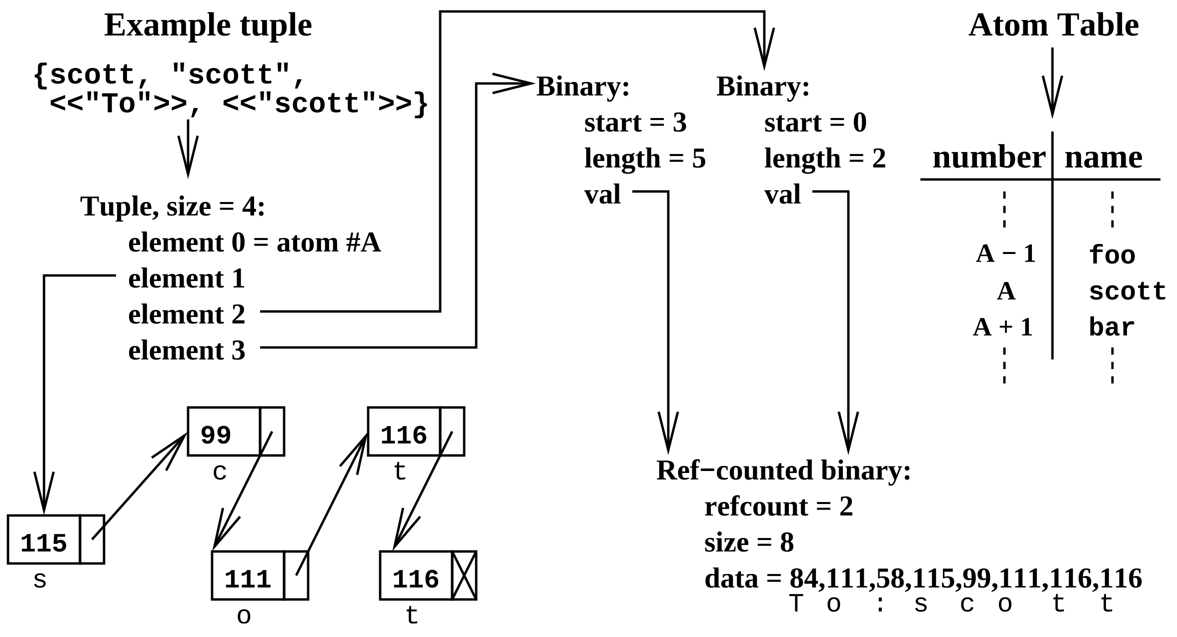

Figure 4: A diagram of internal Erlang virtual machine term storage data structures

5. JUDY-IMPLEMENTED ETS TABLES

JudyL and JudySL arrays use keys that are fundamentally different from the Erlang virtual machine's internal representation of term data. This section explains why the Erlang virtual machine creates the "contiguous key problem" and then discusses three techniques used to solve that problem.

5.1 The contiguous key problem

Most associative array implementations expect to use a single contiguous region of memory for each key that it stores. All three Judy array types are no different: they use single machine words or contiguous byte strings for their keys. Unfortunately, the Erlang virtual machine does not store complex terms contiguously in memory.

As mentioned in Section 3.1, the key used by an ETS table may be an arbitrary Erlang term ranging from a simple number to a list or tuple containing several different types of terms. For an example of the latter, see Figure 4.

Figure 4 contains a simplified diagram of the internal data structures used to represent the Erlang term {`scott, "scott", <<"To">>, <<"scott">>`} within the Erlang virtual machine. A tuple's data structure contains the number of elements in the tuple and an array of tagged pointers to each element term.

The first element is an atom. The second element is a list containing five ASCII character codes. Each code is stored in a cons cell together with a pointer to the next cons cell in the list. An empty cons cell is appended to the end of the list. The third and fourth elements are binaries containing the ASCII strings `To` and `scott`, respectively. Their actual binary data is stored in a reference-counted binary structure that contains the eight ASCII codes for the string `To:scott`.

5.2 The judysl table type

Any attempt to use an Erlang term as a key for a JudySL array must first convert the key term into a NUL-terminated string. The `judysl` table implementation performs this conversion in a three-step process:

1. The key is serialized using the same C function that implements the BIF `term_to_binary/1`. This function is the same one that is used to serialize an Erlang term before writing it to disk or transmitting it across a network.
2. The result of step #1 is run through a conversion function that replaces all ASCII NUL values with non-zero values. The function converts every 7 bits of step #1 data into an 8-bit value where the most significant bit is always 1. The leftover bit is prepended to the next byte, and the loop is repeated.
3. A trailing ASCII NUL is appended to the end of the string.

Unfortunately, the sort order of a list of Erlang terms [A, B, ...] is not the same as the lexigraphic sort order of [`term_to_binary(`A`)`, `term_to_binary(`B`)`, ...]. Therefore, this JudySL-based technique cannot be used to implement an ETS ordered table type.

5.3 The judyesl table type

The author created an array library called JudyESL to remove JudySL's requirement of a NUL byte at the end of the key string. JudyESL's API is very similar to JudySL's API. The only difference is that JudyESL functions have an additional function argument to pass in the length of the key string and to return the key length (as calculated by

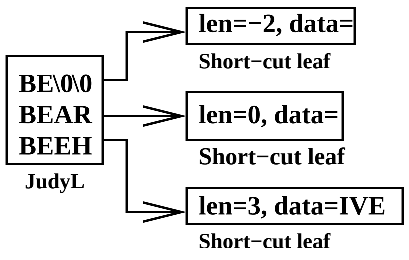

Figure 5: A JudyESL array containing three strings

the traversal-related functions). The same basic trie structure, using JudyL arrays as 4-byte building blocks and using short-cut leaf nodes, is used by JudySL and JudyESL to store the serialized form of the key term.

In order to provide end-of-string information, all leaf nodes in a JudyESL trie are short-cut leaf structures. The short-cut leaf node provides unambiguous information for determining the exact length of the index string. For strings 4 bytes or shorter, the short-cut leaf node may describe a trailing string length that is zero or even negative. See Figure 5 for a diagram of a simple JudyESL array that stores the strings "BE", "BEAR", and "BEEHIVE".

The current JudyESL implementation does not have short-cut leaf nodes complex enough to store the strings S and S' simultaneously, where S is single NUL byte and S' is two NUL bytes. This limitation does not affect JudyESL's use for `judyesl` type tables: the term serialization process cannot create two strings S_1 and S_2 where S_1 is a complete prefix of S_2.

5.4 The judyeh table type

The Judy documentation suggests replacing a traditional hash table with a JudyL array. Such a substitution frees the programmer from having to define the size of the hash table statically or having to implement a hash table that shrinks and grows as populations fluctuate. As an added benefit, the combination of a good hash function and a JudyL "hash table" with 2^{32} buckets will create very few hash collisions, even with tables populated by millions of elements. On 64-bit machines, a good hash function can ensure that collisions will almost never happen, even with table populations in the billions.

Several attempts were made to implement the JudyEH library used by the `judyeh` ETS table type before an acceptable solution was found. The first attempt used a data structure similar to JudySL's short-cut leaf node. When storing an Erlang tuple T with a key K, the JudyL hash array is indexed by HASH(K). The hash array's value in the no-collision case would point to a short-cut leaf node structure that contains a pointer to T. In the event of a collision, the hash array's value would point to a JudyL array that stores the collision list. The second JudyEH implementation eliminated the short-cut leaf node structure and always used two levels of JudyL arrays: the top-level hash array and the bottom-level collision arrays. However, both implementations had the same problem: memory bloat. The culprit was `malloc()`'s overhead when allocating millions of small (4–8 byte) short-cut leaf nodes or Judy arrays. The third attempt managed short-cut leaf node structures with a custom memory manager: memory overhead was close to ideal, but the runtime performance was awful.

The best solution was to avoid solving the problem of collisions at all. In the no-collision case, the JudyL hash array's value is the Erlang term pointer T. In the collision case, the hash array's value is a magic number, `0xffffffff`, which is easily-discernible from a valid Erlang term pointer. The presence of this magic number means that an alternate data structure must be consulted to find the real answer. In the end, the alternate structure used is the two-level JudyL technique from implementation number two. The run-time speed is acceptably fast, and the additional memory and time overhead is negligible because the incidence of collisions in the top-level JudyL hash array is so rare: the collision rate is less than 0.2% for 7 million objects hashed across 2^{32} buckets.

6. EXPERIMENT DESIGN

The benchmark programs described in Section 7 were designed to emphasize, as much as possible, the runtime differences in ETS table implementations. One way to accentuate those differences is to minimize the amount of non-ETS activity within the Erlang virtual machine. As the results in Section 7 show, minimizing non-ETS activity was not necessary to determine if any differences between ETS table types exist at all. This section discusses five design choices made to limit the virtual machine's intrusiveness and to reduce variability in benchmark execution times.

6.1 Minimize memory copying overhead

In the current Erlang virtual machine implementation, ETS tables have their own memory management system that is independent from the rest of the virtual machine. When a tuple T with key K is stored in an ETS table, a complete copy of T is made in ETS-managed memory. As program execution continues, the garbage collector may reclaim T when it is no longer referenced.[2] When K's value is retrieved from the table, a complete copy of T is made in garbage-collected memory.

To minimize term-copying overhead, a simple one-element tuple should be used. A one-element tuple would be sufficient for six of the seven benchmark programs. However, one benchmark program exercises the commonly-used function `ets:update_counter/3`, which efficiently and atomically performs a table lookup, increments the value of one of the stored tuple's elements, and stores the new tuple back in the table. Therefore all of the benchmarks operate on two-element tuples: the first element is the integer key, and the second element is the integer counter value.

6.2 Minimize the time spent comparing terms

As the Erlang virtual machine traverses the AVL tree or B-tree of an ordered ETS table or traverses a collision list in an unordered ETS table, it calls the C function `cmp()` to compare Erlang terms. As Figure 4 shows, `cmp()` may have

[2]An analogous situation would be when T is serialized and then written to disk: the local file system is also a datastore that is independent of the Erlang virtual machine.

Term	Bytes
`1`	3
`123456789`	6
`6.02e23`	33
`scott`	9
`"scott"`	9
`<<"scott">>`	11
`{1, "scott"}`	13
`[1,2,[[3,4],5],[[6]]]`	34
Return value of `make_ref()`	33
Return value of `self()`	27

Table 1: Size (in bytes) of various terms as serialized by `term_to_binary/1`

to examine many areas of memory to calculate its result. To minimize each benchmark's term comparison overhead, the key term should be as quick to compare as possible.

Erlang atoms are the quickest to test for equality, but they are slow to test for relative magnitude: the atom table must be consulted to retrieve each atom's ASCII name. Furthermore, constructing and storing several million atoms would take a lot of time and space. Numbers are equally quick to test for both equality and relative magnitude. As a further efficiency measure, the virtual machine's term pointer tagging scheme can "stuff" small integers (less than $2^{27}-1$) into the object pointer itself, avoiding the overhead of allocating a separate object to store the integer.

6.3 Minimize memory overhead from JudySL and JudyESL arrays

The performance of `judysl` and `judyesl` table types are affected to a greater degree by key choice than are the other table types. Unlike the other table types, two complete copies of the key term are made. One copy is inside the JudySL or JudyESL array, and the other is stored by the ETS table itself.

Table 1 shows the serialized sizes of several Erlang terms. The use of complex tuples or lists, or use of larger basic term types, can create a large serialized term string which, in turn, creates a long key string for a JudySL or JudyESL array. It is difficult to predict how key choice will affect how much memory will actually be consumed by a `judysl` or `judyesl` table or what the impact on actual execution time will be.

6.4 Minimize arithmetic and GC overhead

Erlang's bignum support sometimes comes at an unwanted cost. Many pseudo-random number generators rely on fixed integer sizes to keep a tight lid on computational cost and space. The Erlang virtual machine will always use bignums to preserve all of the overflow bits that fixed-size arithmetic in C will discard.

One technique that could hide the computational cost of bignum arithmetic is to generate a long sequence of pseudo-random numbers before starting the stopwatch on a benchmark. However, that method may cause unexpected garbage collection activity while the benchmark is running. Furthermore, the term storing the precomputed random numbers will consume physical RAM that may prevent the creation of a 54 million key ETS table.

The random number generation technique used by the benchmark programs does not precalculate values. Instead, the generator function is given the current number in the sequence, N_i, and returns the next number in the sequence, N_{i+1}. It uses one multiplication and one division (remainder) operation. Its disadvantage is that the operands are usually bignums and thus add more overhead to the benchmark, but it is very cheap memory-wise.

6.5 Do not change the hash function

All of the unsorted ETS table types use the same C function, `make_hash2()`, to create a 32-bit unsigned long value for each key. The `judyeh` table implementation could probably run faster by recognizing simple key terms, such as small integers, and using their value directly as the hash value. However, the intent of the benchmarking tests is to measure the effect of changing only the hash table implementation itself.

7. PERFORMANCE RESULTS

This section presents the results of seven small benchmark programs designed to demonstrate wall-clock execution time and memory consumption properties of all ETS table types under a variety of workloads and table populations.

Of all the standard Erlang/OTP ETS table types, the fastest overall performer is the `set` type. Therefore, any new table type should consistently match or beat the `set` type to be considered a viable addition to ETS. Accordingly, most of the graphs below are presented with execution times relative to `set`'s runtime (i.e., `set`'s runtime is always 1.0). Smaller relative runtimes are better; runtimes below 1.0 indicate performance better than `set`'s.

Each benchmark program was run using an assortment of table populations in order to show how table performance changes as the tables grow larger than CPU caches can store. Each iteration was repeated enough times to be confident that the measured runtimes were accurate to three significant figures.

The discussion in this section proceeds as follows: description of the experiment platform, execution time results, memory consumption results, and a final analysis of all test results.

7.1 Experiment platform

The test platform was a rack-mounted dual CPU Intel Xeon system running at 2.0GHz. The machine contained 4GB of PC2100 (266MHz) DDR RAM with a Linux 2.4.18 kernel from the RedHat 8.0 Linux distribution. All tests were run within the amount of RAM available and did not trigger any virtual memory-related disk activity. The machine was otherwise idle while all tests were run. The version of Erlang used was the Erlang/OTP R9B-1 release with Pthreads disabled at compile time. It was linked with GNU libc version 2.2.93.

The Intel Xeon processor is a member of the Pentium 4 processor family. According to [7], Xeon processors have a

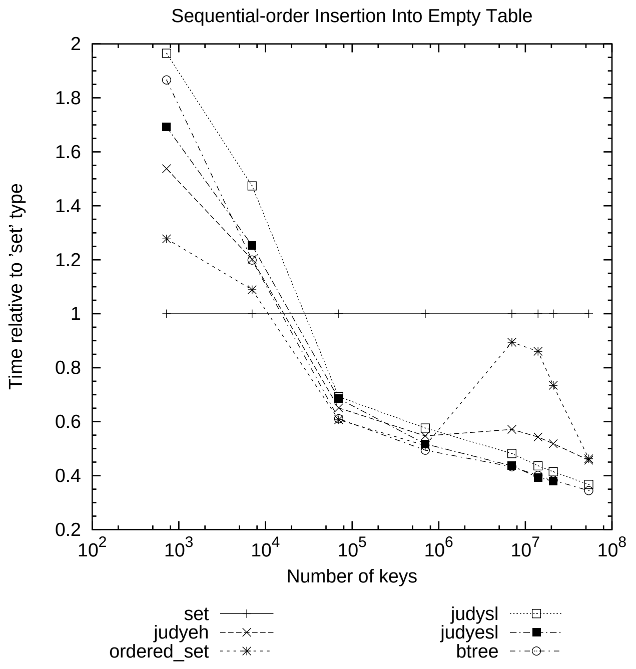

Figure 6: Average benchmark runtimes for sequential-order insertion into an empty table, relative to the set table type

16-word (64 byte) cache line size. This cache line size is a departure from earlier IA-32 processors, which have 8-word (32 byte) cache lines. Judy's internal data structures were laid out with the assumption that the cache line size is 16 words.

Each CPU has a L1 data cache of 8KB and a unified L2 cache of 512KB. An `ordered_set` ETS table with 7,000 two-element tuples, in the form described below, occupies approximately 205KB. The amount of memory used varies slightly for the other data structures studied; in general, an entire 7,000 key table fits within the CPU's L2 cache, but a 70,000 key table does not.

The following number of keys used for each test: 700; 7,000; 70,000; 700,000; 7 million; 14 million; 21 million; and 54 million. The Linux kernel imposed a 3GB limit on the size of any individual process. Unfortunately, inserting 54 million two-element tuples into a `judyesl` table caused the process to try to grow above 3GB, terminating the virtual machine. Therefore, all of the graphs are missing a 54 million key data point for the `judyesl` table type.

7.2 Times for sequential & random insertion

The average runtimes of the sequential and random insertion tests are shown in Figure 6 and Figure 7, respectively. Each test starts with an empty table, then inserts the tuples $\{K, 1\}$, where K is a value from 0 to X. In both cases, the `set` table type is fastest when the entire table size is smaller than the L2 cache size. Once the table size exceeds L2 cache size, it is slower than all three Judy-based table types.

The AVL tree and B-tree used by the `ordered_set` and `btree` tables, respectively, preserve the Erlang key sorting

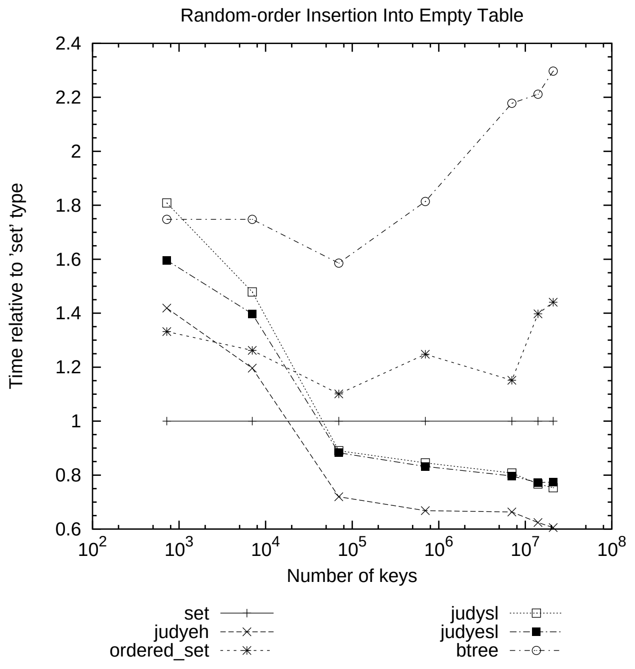

Figure 7: Average benchmark runtimes for random-order insertion into an empty table, relative to the set table type

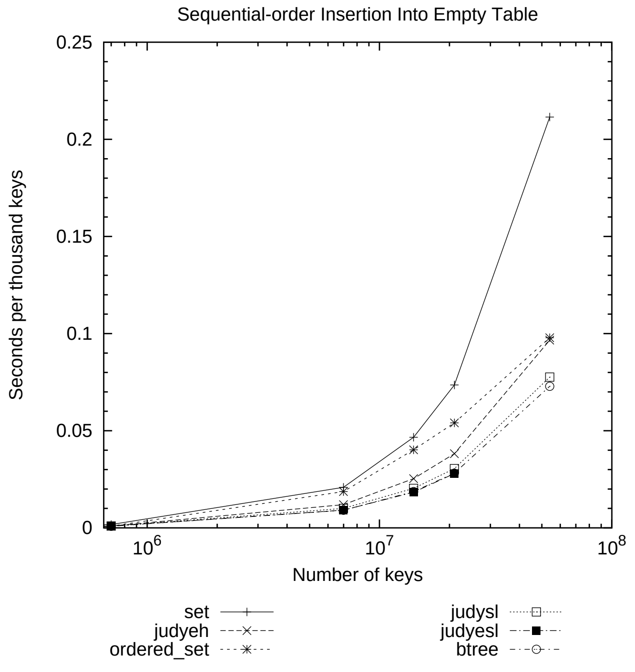

Figure 8: Average benchmark runtimes for sequential-order insertion into an empty table, per thousand keys

order. Similarly, the JudySL and JudyESL arrays used by the `judysl` and `judyesl` table types, respectively, preserves the lexical sort order of the serialized byte string created from the key. All four structures take advantage of excellent locality of reference conditions during the sequential insertion test. Accordingly, they are much faster than `set` at 70,000 keys and beyond.

The sequential insertion test results graph in Figure 6 provides the first hint of a pattern that appears in the `ordered_set` table type in several of the tests. At 7 million keys, `ordered_set` suddenly performs worse relative to all types except `set`. The cause of this pattern is not clear. If it were a sudden change in `set`'s time, then similar jumps should appear in all of the table types. The cause does not appear to be related to AVL rebalancing activity because the same pattern also appears in read-only operations (e.g., the `ordered_set` value at 7 million items in Figures 9 and 10).

The random insertion test takes away most of the locality of reference advantage of the sequential insertion test. All three Judy-based table types suffer slower runtimes than the sequential insertion test, but they remain consistently faster than the `set` type at 70,000 keys and beyond.

A reasonable person might argue that the insertion tests unfairly disadvantage the `set` table type. In the 7 million key case, the calls that the `set` table makes to the C function `grow()` triggers the resizing the `set` table's hash table 3,905 times. Profiling with gprof [6] shows that `grow()` accounts for approximately 12% of execution time at 7 million keys. Figure 8 shows that `set`'s overhead grows at a far faster pace than the overhead of any other table type. However, even if the hash table could avoid calling `grow()` by having all required buckets available at table creation time, the `judyeh` table would still perform better than `set` at 70,000 keys or more.

It is worth noting that the `judyeh` table type performs slightly better in the sequential insertion test than it does in the random insertion test. The average relative sequential insertion time at 21 million keys was 45.7%, and the average relative random insertion time at 54 million keys was 60.5%. If the hash function had created a truly random distribution of values, the two sets of results should be identical or nearly so. The `make_hash2()` function does not have a hash collision rate much higher than the ratio of keys to hash buckets, so hash collisions cannot explain the difference in results. However, visual inspection of the sequence of hash values from the sequential insertion test reveals an oscillating pattern of large and small hash values. The CPU's data caches are likely taking advantage of the pattern to lower the sequential test's runtime.

7.3 Times for sequential & random lookups

The average benchmark runtimes for the sequential-order and random-order lookup tests are shown in Figure 9 and Figure 10, respectively. Both tests were performed on tables first created by the sequential insertion test described in Section 7.2; term insertion time was not measured by the benchmarks. Both use the `ets:lookup/2` function to retrieve each stored tuple.

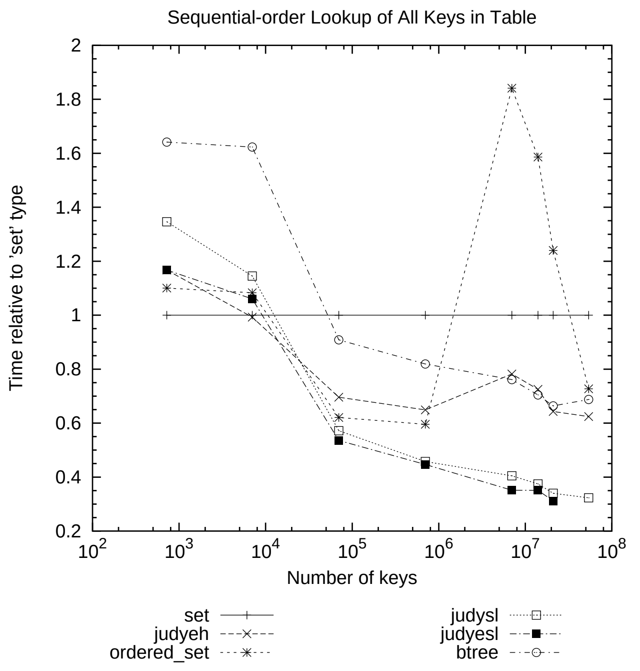

Figure 9: Average benchmark runtimes for sequential-order lookup, relative to the `set` table type

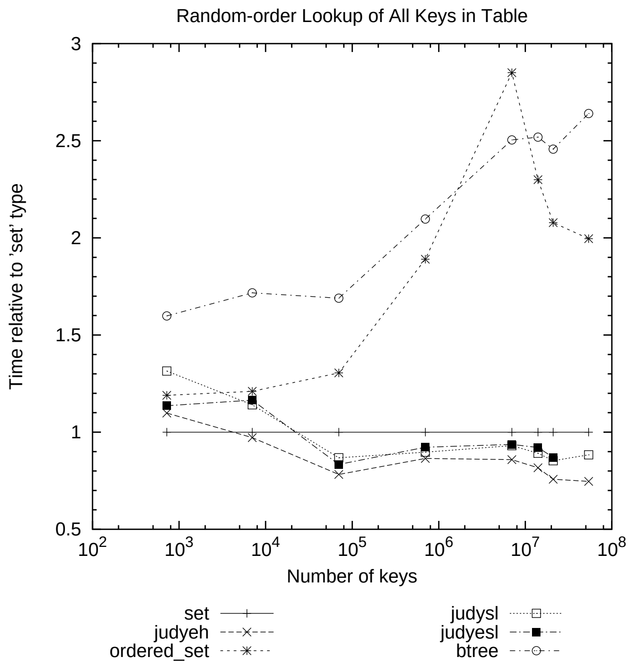

Figure 10: Average benchmark runtimes for random-order lookup, relative to the `set` table type

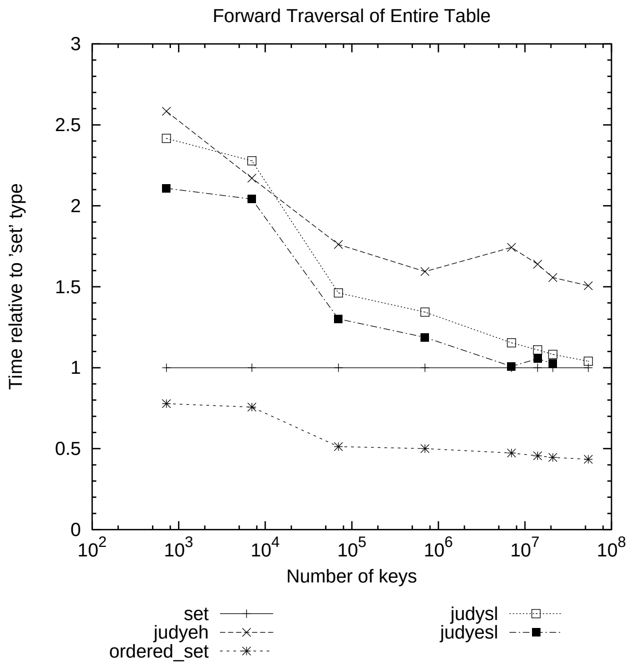

Figure 11: Average benchmark runtimes for forward table traversal, relative to the `set` table type

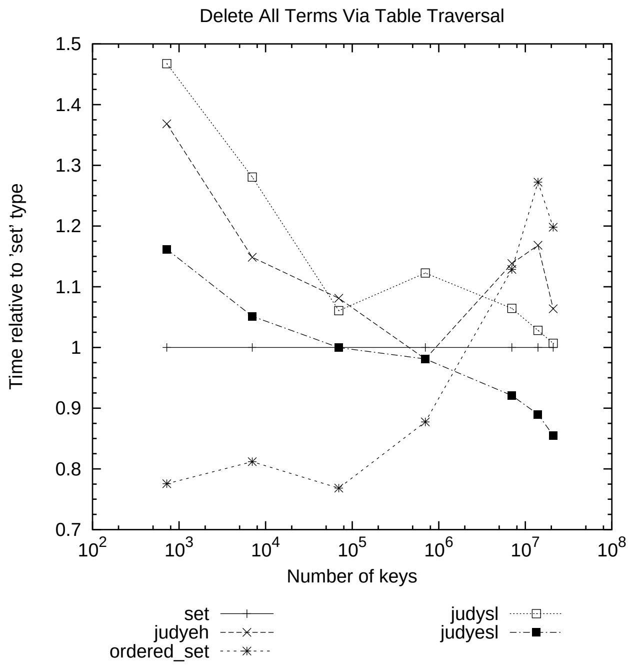

Figure 12: Average benchmark runtimes for key deletion, relative to the `set` table type

As with the insertion tests, the lookup tests show a sharp change in runtimes relative to `set` when the table population grows from 7,000 to 70,000 keys. Recall that the 7,000 tuple ETS table can fit entirely within L2 cache but a 70,000 tuple ETS table cannot.

The performance of all Judy-based table types in the sequential lookup test is good when compared to the `set` table type. The locality of reference advantage given by looking up sequential keys results in remarkable runtimes for the `judysl` and `judyesl` table types: 32.3% at 54 million keys and 31.2% at 21 million keys, respectively. The same locality of reference conditions do not appear to help the `ordered_set` and `btree` types to nearly the same degree. The sequential lookup test's access pattern appears ideally-suited for JudySL and JudyESL arrays.

The random-order lookup test's poor locality of reference conditions disadvantage the `ordered_set` and `btree` table types. However, the `judysl` and `judyesl` types perform surprisingly well: at their largest table size, they run at average relative times of 86.8% and 88.3%, respectively. The fastest random-order lookup relative time belongs to the `judyeh` type: 74.7% at 54 million keys.

7.4 Times for forward traversal, counter updates, & key deletion operations

All tests in this section were performed on tables first created by the sequential insertion test described in Section 7.2; term insertion time was not measured by the benchmarks. Test results for the `btree` table type are unavailable due to `btree`'s incomplete implementation at the time of writing.

The table traversal test uses the `ets:first/1` function to find the first term in the table, then repeatedly uses `ets:next/2` to retrieve all other terms from the table. The average runtimes of the table traversal test are shown in Figure 11.

The traversal test clearly favors the `ordered_set` table type. Its worst average time is 77.8% at 700 keys, and its best average time is 43.5% at 54 million keys. The Judy-based table types should also be able to take advantage of similar locality of reference conditions as they do in the sequential insertion and sequential lookup tests. However, they do not appear to do so. The `judysl` and `judyesl` types never run in less average time than the `set` table type. The `judyeh` table time fares even worse: 258% at 700 keys and 151% at 54 million keys.

The poor table traversal results for the Judy-based table types are puzzling. The author does not have a ready explanation. Profiling does not identify an obvious culprit. The extra overhead may be due to the ETS table interface's need to traverse the underlying hash table JudyL trie at least twice from root to leaf.

The random-order counter update test uses the `ets:update_counter/3` function to increment the value of the second element of each tuple stored in the table. A graph depicting the results of this test is not included here: the results were identical to the random-order lookup results shown in Figure 10. The counter update test uses the same virtual machine C functions that the lookup test uses to find the correct tuple, and the additional work required to store a new tuple with the new counter value is the same for all table types.

The term deletion test results are shown in Figure 12. The test traverses the table to identify the keys in the table and

Table type	**Memory used by 70K keys**	**Memory used by 21M keys**
`btree`	10.4MB	1,055MB
`judyeh`	10.4MB	1,036MB
`judysl`	10.4MB	1,033MB
`judyesl`	11.3MB	1,324MB
`ordered_set`	10.7MB	1,129MB
`set`	10.2MB	980MB

Table 2: Memory used by various ETS table types

then to delete each of those keys with the `ets:delete/2` function.

Analysis of the deletion test's results are complicated by the table traversal used by the test. However, it is noteworthy that the performances of the Judy-based tables are significantly better in the term deletion test than they are in the table traversal test, despite the fact that both tests share an `ets:next/2`-based table traversal. For example, at 21 million keys the `judyeh` table type's average runtime in the table traversal test is 155%, but it is only 106% in the term deletion test.

7.5 Memory consumption

Any analysis of faster or slower benchmark times must also be accompanied by an analysis of the amount of memory used by those benchmark programs. Table 2 shows the amount of memory used by each table type to store 21 million tuples from the sequential insertion test described in Section 7.2.

The amount of memory used by the `judyeh`, `judysl`, and `btree` tables all fall in between the amount used by the `set` and `ordered_set` types. The `judyeh` and `judysl` tables use only about 6% more memory than `set` at 21 million keys. The most memory-hungry type, `judyesl`, uses 11% and 35% more memory than `set` at 70 thousand and 21 million keys, respectively. The extra memory is consumed by the additional "short-cut" leaf node structures required by the `judyesl` trie and by `malloc()`'s overhead of managing all of those small leaf node allocations.

7.6 Performance summary

For unsorted tables with populations of 70,000 keys or more, performance improvement by using the `judyeh` table type instead of `set` is easily measurable and significant. This improvement can be seen with keys in sequential order and random order during operations involving table insertion, lookup, and counter updates. The deletion operation is not as fast as `set`, but deletion's extra cost is smaller than the benefit of insertion, lookup, and update operations combined.

Furthermore, the additional RAM required by `judyeh` tables is quite modest. The `judyeh` table type requires only about 6% more than `set` tables, which is smaller than the additional 15% required for the same data in an `ordered_set` table.

The only operation this research examines which is significantly worse for `judyeh` tables than `set` tables is table traversal using the `ets:next/2` function. Optimization of the JudyL library itself and/or ETS-specific changes to the JudyL library may be required. Note that a similar double-traversal is required by `ets:delete/2`.

In all tests, Judy-based tables are slower than `set` table populations that fit inside L2 cache. For machines with L2 cache sizes of 256KB or 512KB, the test results suggest that Judy-based tables are not recommended for unsorted tables with key populations under 7,000. For unsorted tables from 7,000 to 70,000 keys, the `set` table type is probably still the best overall choice.

The `judysl` and `judyesl` table types have a memory consumption handicap of storing the key term of each tuple twice: once within ETS-managed memory and once within the JudySL or JudyESL trie. Despite this handicap, both types of tables typically perform well compared to the `set` type. Both types are especially fast when the operations are performed with keys in sequential order, much faster even than the `judyeh` type.

The `btree` table type was included in the performance analysis primarily to provide additional data points regarding the cost of maintaining key sort order. The `btree` and `ordered_set` random-order test results confirm that there is a significant cost to maintain key sort order. It is noteworthy that the `btree` table does not show the same peculiar behavior pattern around the 7 million key table size that `ordered_set` does.

The performances of the `judysl` and `judyesl` table types demonstrate that the cost of maintaining key sort order can be substantially lower than an AVL tree's or B-tree's cost. If the key term type were restricted so that a serialization technique could be developed to maintain the sort order of the original unserialized terms, a JudySL- or JudyESL-based table type should provide superior performance to the unsorted `set` table type while still maintaining key order.

8. RELATED WORK

ETS was first mentioned in an unpublished technical report [12] about the distributed database that became known as Mnesia [11]. Subsequent publications about Mnesia make indirect references to ETS tables but only mention their linear hashing implementation.

A performance analysis [9] of the HiPE native code compiler for Erlang states that BIF execution was the major execution time bottleneck for the AXD/SCCT benchmark. SCCT is the time-critical portion of the code used by the Ericsson AXD 301 ATM switch [4] that is responsible for connection setup and teardown. A study [8] of SCCT, running on an UltraSPARC platform and an earlier version of the Erlang virtual machine, show that SCCT spent 32% of total CPU cycle time executing BIFs. Approximately 64% of that time was spent executing ETS-related code, or 18% of the total execution time. The analysis gives little information about the ETS table types used or ETS table populations, but both papers demonstrate that an industrial-strength Erlang application like SCCT might see significant performance improvement if a Judy-based ETS table were available.

The HiPE research group has done a lot of memory management work in the Erlang virtual machine using a "shared heap" architecture [10]. Most or all of the copying of tuples into and out of ETS tables could, in theory, be eliminated by putting ETS tuples into a shared heap. For the small two-element tuples used by the benchmark programs described in Section 7, profiling shows that the overhead of copying tuples into and out of ETS-managed memory is only about 1.5% of total runtime. Bigger and/or more complicated tuples can raise that figure significantly.

The author is peripherally aware of efforts that make Judy trees available to the Ruby and Perl user communities. However, those efforts appear to use each language's foreign language interface to make Judy arrays accessible by scripts written in those languages; they do not (yet) attempt to replace any of the data structures within the Perl or Ruby interpreters themselves.

9. FUTURE WORK

The data presented in this paper shows that replacing traditional data structures with Judy arrays can provide substantial performance improvement to a real-world application. The work also prompts many ideas for further development and research.

1. Optimize the implementations of the Judy-based `ets:next/2` and `ets:delete/2` functions. A special Judy library tailored for ETS's needs may make those operations consistently faster than their `set` type equivalents. This tuning may also be necessary for good performance by other ETS functions such as `ets:match/2` and `ets:foldl/3`.

2. Extend the analysis to 64-bit CPUs. The Erlang virtual machine can only address 4GB of memory, so many additional changes need to be made to the virtual machine before it will be possible to create an ETS table larger than 4GB. Pointer sizes of 64 bits would significantly simplify the `judyeh` table type's current collision handling scheme for cases where a poor hash function were to create collision rates higher than one in a billion.

3. Use CPU hardware performance counters to measure CPU stalls while operating on large Judy- and non-Judy-based ETS tables. Baskins suggests that the interactions between very large in-memory datastores and CPU TLB caches is not well-understood and requires further research [personal correspondence].

4. Work with Ericsson to see if a Judy-based ETS table type is worth including in a future Erlang/OTP release.

10. CONCLUSION

Judy arrays are indeed an excellent data structure for building unordered ETS tables. Although Judy-based tables, when compared to the baseline `set` table type, do not do well at small table sizes (i.e., those that fit completely within CPU secondary cache: less than about 10,000 small tuples), they do perform well at table sizes of 70,000 keys up to 54 million keys. Specifically, the `judyeh` type performs significantly faster than `set` for insertion, lookup, and update operations: up to 54% faster with sequentially-ordered keys and up to 39% faster with randomly-ordered keys. The `judyesl` performs up to 69% faster than `set` at lookups with sequentially-ordered keys.

The Judy-based tables do not perform as well as `set` in the table traversal and term deletion benchmarks. However, the cost of the term deletion operation is more than offset by the benefit gained by the insertion, lookup, and update operation improvements. Additional work is required to improve table traversal performance.

The performance gains of the `judysl` and `judyeh` tables are much larger than the additional memory cost. Both types require only about 6% more memory than `set` for the same table population.

Assuming that cost of traversing a Judy-based hash table can be lowered, the author believes that many applications, including the Erlang virtual machine, could run faster by replacing traditional linear hash tables with Judy arrays. This finding is significant because optimization of such hash tables has typically been limited to either (a) optimizing the hash function or (b) adjusting the hash table size. The data from this research shows that execution time can be reduced without optimizing the hash function and by letting the JudyL library manage the hash table size.

11. ACKNOWLEDGMENTS

A thousand thanks to all my reviewers: Francesco Cesarini, Nick Christenson, Michael Coyle, Philip Guenther, Olaf Hall-Holt, Mark Henning, and Ulf Wiger. I owe y'all a lot of beer.

Doug Baskins, Judy's inventor, lent a lot of advice and support during the course of my research. It is his general idea to replace existing hash tables with JudyL arrays. It is great fun to see a big, real-world application get a performance boost out of a relatively minor code change. Thank you, Hewlett-Packard, for allowing the Judy source code out into the world with an LGPL license.

Jason DeStefano and Ralph Jenson at FireVue permitted me to borrow several days of Xeon CPU cycles. The performance analysis would have been much shakier without their assistance. Thanks also to Mark Henning and to my wife, Louise Lystig Fritchie, for lending CPU cycles on their RISC machines.

May Erlang users everywhere forgive my unforgivably terse Erlang overview in Section 2.

Thanks again to Louise for her red editor's pen and for simply putting up with me while I worked on this project.

12. REFERENCES

[1] J. Armstrong, R. Virding, C. Wikström, and M. Williams. Concurrent Programming in ERLANG. Prentice Hall Europe, Essex, England, 1996. Chapters 1–9 available at `http://www.erlang.org/doc.html`.

[2] D. Baskins. A 10-Minute Description of How Judy Arrays Work and Why They Are So Fast. Available at `http://judy.sourceforge.net/downloads/-10minutes.htm` as of July 2003.

[3] D. Baskins. Judy functions — C libraries for creating and accessing dynamic arrays. Available at `http://judy.sourceforge.net/` as of July 2003.

[4] S. Blau, J. Rooth, J. Axell, F. Hellstrand, M. Buhrgard, T. Westin, and G. Wicklund. AXD 301: A new generation ATM switching system. Computer Networks (Amsterdam, Netherlands: 1999), 31(6), 1999.

[5] C. DiBona, S. Ockman, and M. Stone, editors. Open Sources: Voices of the Open Source Revolution. O'Reilly & Associates, Sebastopol, California, 1999. Appendix B.

[6] S. Graham, P. Kessler, and M. McKusick. gprof: a Call Graph Execution Profiler. In SIGPLAN Symposium on Compiler Construction, 1982.

[7] Intel Corporation. IA-32 Intel Architecture Optimization Reference Manual. Intel Corporation, Santa Clara, California, 2003.

[8] E. Johansson, S.-O. Nyström, T. Lindgren, and C. Jonsson. Evaluation of HiPE, an Erlang Native Code Compiler. Technical Report 99/03, Uppsala University ASTEC, October 1999.

[9] E. Johansson, M. Pettersson, and K. Sagonas. A high performance Erlang system. In Principles and Practice of Declarative Programming, 2000.

[10] E. Johansson, K. Sagonas, and J. Wilhelmsson. Heap Architectures for Concurrent Languages using Message Passing. In Proceedings of the ASM SIGPLAN International Symposium on Memory Management (ISMM'02). ACM Press, June 2002.

[11] H. Mattsson, H. Nilsson, and C. Wikström. Mnesia — A Distributed Robust DBMS for Telecommunications Applications. In Practical Aspects of Declarative Languages (PADL '99), volume 1551 of Lecture Notes in Computer Science. Springer, 1998.

[12] H. Nilsson, T. Törnquist, and C. Wikström. Amnesia, a distributed telecommunications DBMS. Technical report, Ericsson Computer Science Laboratory, October 1995.

[13] A. Silverstein and D. Baskins. Judy IV Shop Manual. Available at `http://judy.sourceforge.net/-application/shop_interm.pdf` as of July 2003.