The great (internal) Var reform of 2007

Miguel Sofer
Universidad Torcuato Di Tella

September 15, 2007

Abstract

The Var struct used as internal representation for Tcl's variables currently contains six pointers and 2 integers, or 32 bytes on a 32-bit platform. For variables in hashtables, be they namespace variables or array elements, a hash entry structure consuming a further minimum of 24 bytes (but typically 28) is also maintained.

These requirements reflect a history of the structure, and are far from optimal. We will explain the requirements that have to be satisfied, how they gave rise to this structure, and a way to thin them down considerably: compiled variables are reduced to 8 bytes (75% reduction), hashtable variables to a grand total of 24 bytes (60% reduction)[1]. Further performance advantages of the new implementation will also be described.

1 Introduction

Variables are among the most-often accessed structs in Tcl[2]. Their impact on Tcl's performance is undeniable[3], both in terms of runtime and memory footprint. The current Var struct[4] in Tcl, defined in tclInt.h and reproduced in Figure 1, is not optimal: it is too large and its cache-friendliness is further handicapped by an unlucky layout, cache-unfriendly for every r/w operation.

This paper describes the motivation and design decisions in the new variable code in Tcl8.5, implemented in Patch #1750051. Section 2 describes the definition of variables in current Tcl, section 3 the newly defined structs in 8.5. We describe the advantages of the new code in section 4, and its disadvantages in section 5. Section 6 describes related work that will not appear in Tcl8.5 and remains to be done in the future.

[1] A partial implementation is already committed to HEAD, a full implementation may or may not appear in Tcl8.6

[2] Commands may profit from a similar reform in the future

[3] tbd: precise measurements

[4] from Tcl8.0 up to Tcl8.5a6

2 The Var struct in Tcl8.x (x<5)

The Var struct currently requires 32 bytes[5] and accomodates in the single struct both storage types for variables: compiled locals and variables held in a hashtable (either namespace variables or array elements). The necessity of the different fields can be classified as follows

1. *flags, value*: the really necessary fields for normal read/write operation. The first maintains the current state and type of the variable, the second its current value.

2. *name*, *nsPtr*, *hPtr*: necessary so that compiled variables (*name*) and hashtable variables (*nsPtr*, *hPtr*) can find out their names. Knowing the name is required for error messages and trace processing. Note that at most two of these fields are non-NULL for any variable.

3. *refCount*, *hPtr*: needed for lifetime management of the struct, which is only really necessary for hashtable variables.

4. *searchPtr*, *tracePtr*: necessary to store searches and traces currently defined on this variable.

Furthermore, hashtable variables require the maintenance of a Tcl_HashEntry struct that is allocated separatedly and requires either 24 bytes (for variable names of up to three characters), 28 bytes (between 4 and 7 characters), 32 bytes (between 8 and 11 characters), and so on.

The claim of suboptimality is based on the following observations:

- the fields *name*, *nsPtr*, *hPtr*, *refCount*, *tracePtr*[6] and *searchPtr* are rarely accessed, only *flags* and *value* are needed for normal r/w operation

[5] all calculations done for 32-bit platforms as illustration

[6] *tracePtr* is actually tested against NULL at each r/w operation, but this can be avoided

```
typedef struct Var {
    union {
        Tcl_Obj *objPtr;        /* The variable's object value. Used for
                                 * scalar variables and array elements. */
        Tcl_HashTable *tablePtr;/* For array variables, this points to
                                 * information about the hash table used to
                                 * implement the associative array. Points to
                                 * ckalloc-ed data. */
        struct Var *linkPtr;    /* If this is a global variable being referred
                                 * to in a procedure, or a variable created by
                                 * "upvar", this field points to the
                                 * referenced variable's Var struct. */
    } value;
    char *name;                 /* NULL if the variable is in a hashtable,
                                 * otherwise points to the variable's name. It
                                 * is used, e.g., by TclLookupVar and "info
                                 * locals". The storage for the characters of
                                 * the name is not owned by the Var and must
                                 * not be freed when freeing the Var. */
    Namespace *nsPtr;          /* Points to the namespace that contains this
                                * variable or NULL if the variable is a local
                                * variable in a Tcl procedure. */
    Tcl_HashEntry *hPtr;       /* If variable is in a hashtable, either the
                                * hash table entry that refers to this
                                * variable or NULL if the variable has been
                                * detached from its hash table (e.g. an array
                                * is deleted, but some of its elements are
                                * still referred to in upvars). NULL if the
                                * variable is not in a hashtable. This is
                                * used to delete an variable from its
                                * hashtable if it is no longer needed. */
    int refCount;              /* Counts number of active uses of this
                                * variable, not including its entry in the
                                * call frame or the hash table: 1 for each
                                * additional variable whose linkPtr points
                                * here, 1 for each nested trace active on
                                * variable, and 1 if the variable is a
                                * namespace variable. This record can't be
                                * deleted until refCount becomes 0. */
    VarTrace *tracePtr;        /* First in list of all traces set for this
                                * variable. */
    ArraySearch *searchPtr;    /* First in list of all searches active for
                                * this variable, or NULL if none. */
    int flags;                 /* Miscellaneous bits of information about
                                * variable. See below for definitions. */
} Var;
```

Figure 1: The Var struct in Tcl8.x (x<5)

- the fields *tracePtr* and *searchPtr* are NULL most or all of the time for most variables.
- the normal r/w operations[7] access the fields *flags, tracePtr* and *value* (in that order): first the end of the struct, then the beginning.
- creating a new variable in a hashtable requires two separate calls to malloc() - one for the Var, one for the Tcl_HashEntry. As these two structs (can) have the same lifetime and are in 1-1 relationship they could be allocated together, reducing the necessary calls to malloc/free by 50% on variable creation and destruction.

3 The Var structs in Tcl8.5

The layout in memory and access modes for variables has been completely redesigned in Tcl8.5[8], with significant reductions in the required memory and better memory access patterns.

In order to do this, two different structs have been designed for variables: Var (Figure 2) for compiled local variables, and VarInHash (Figure 3) for variables kept in hashtables. The most frequent operations, reading and writing, are impervious to the difference as they only access the Var part; the difference is only relevant for operations related to the variable's lifetime management: creating a link to the variable and unsetting it.

The details are described in this section.

3.1 Removing *tracePtr* and *searchPtr*

As observed previously, the fields *tracePtr* and *searchPtr* are NULL most or all of the time for most variables. The first one is accessed at each variable r/w operation in order to determine if the variable is traced, so that the correct r/w procedure can be used. That is: the fact that these fields are NULL or not is part of the state of the variable. By defining new flag bits to record the complete state of the variable, the linked lists currently held at *tracePtr* and *searchPtr* can be moved elsewhere: only if the corresponding bits are set will their values be accessed.

Two special hash tables (hanging from the Interp struct) have been designed to hold these linked lists. The trace and search code has been modified to maintain the new flag bits.

As an added advantage, the state of the variable can now distinguish the type of trace. This means for instance that reading a variable that carries a trace on write can proceed at full speed, without traversing the list of traces to (fail to) find a possible read trace.

3.2 Compiled local variables: most fields removed

Each time a proc is invoked, an array of variables is allocated on the Tcl stack to hold the body's local variables. These variables are normally accessed by indexing into this array, much faster than an access by name[9].

The lifetime management of the required memory is fairly simple: it is reserved when the proc is invoked and returned when the proc returns. Compiled local variables have no use for the *refCount* and *hPtr* fields, they are gone.

The name of a local variable is unqualified, the *nsPtr* field is not needed. Furthermore, the *name* does not change at each invocation, and it can safely be held in either the Proc or ByteCode structs[10].

Losing this field also simplifies the initialisation of local variables during a proc's invocation. The new flag bits and semantics were designed so that a local variable has to be initialised to {0,NULL}[11], which can be done by a fast memset.

As a result compiled variables only use 8 bytes, as seen in Figure 2, for a 75% size reduction.

3.3 Variables in hashtables: thinned down and consolidated with the entry, Tcl_Obj keys

Namespace variables require knowledge of their namespace in order to reconstruct their fully qualified name. But they have a pointer *hPtr* to their hash table entry, which in turn has a pointer *tablePtr* to the namespace's hash table. We have chosen to store a pointer to the namespace right after the hash table[12], so that every variable can find its namespace without needing to store *nsPtr* in the struct. As *name* was always NULL for these variables, it is gone too.

The Var structure is now allocated together with its corresponding Tcl_HashEntry, which requires that their lifetimes be tied together.

[7] the modes of access in decreasing order of frequency are: read, write, create, destroy, create a link to it.

[8] after Tcl8.5a6

[9] this is among the main performance wins of bytecompiling

[10] in the current implementation it is held in both; this choice was made in order to minimise the changes that might impact extensions.

[11] the proc arguments obviously require a different initialisation. Furthermore, variable resolvers as defined e.g. by incrTcl still require special var-by-var processing

[12] a new TclVarHashTable struct has been defined with two fields: a Tcl_HashTable and a Namespace*

```
typedef struct Var {
    int flags;                  /* Miscellaneous bits of information about
                                 * variable. See below for definitions. */
    union {
        Tcl_Obj *objPtr;        /* The variable's object value. Used for
                                 * scalar variables and array elements. */
        TclVarHashTable *tablePtr;/* For array variables, this points to
                                 * information about the hash table used to
                                 * implement the associative array. Points to
                                 * ckalloc-ed data. */
        struct Var *linkPtr;    /* If this is a global variable being referred
                                 * to in a procedure, or a variable created by
                                 * "upvar", this field points to the
                                 * referenced variable's Var struct. */
    } value;
} Var;
```

Figure 2: The Var struct in Tcl8.5

```
typedef struct VarInHash {
    Var var;
    int refCount;         /* Counts number of active uses of this
                           * variable: 1 for the entry in the hash
                           * table, 1 for each additional variable whose
                           * linkPtr points here, 1 for each nested
                           * trace active on variable, and 1 if the
                           * variable is a namespace variable. This
                           * record can't be deleted until refCount
                           * becomes 0. */
    Tcl_HashEntry entry;  /* The hash table entry that refers to this
                           * variable. This is used to find the name of
                           * the variable and to delete it from its
                           * hashtable if it is no longer needed. It
                           * also holds the variable's name. */
} VarInHash;
```

Figure 3: VarInHash struct

The hash tables for variables now use Tcl_Obj keys, as opposed to the previous string keys[13], insuring that the size of the VarInHash struct does not depend on the length of the variable's name[14].

3.4 Simplified flag semantics

The reform provided the oportunity to simplify the flag semantics by removing some previously allowed possibilites:

- VAR_SCALAR removed: scalar is the default state of a variable, signaled by the absence of array or link bits. The previous scheme allowed for a variable to be neither scalar nor array nor link.
- VAR_UNDEFINED removed: a variable is undefined precisely when its value is NULL. The previous scheme allowed a non-NULL value (garbage)

4 What has been won

The variable reform is a big change, slightly traumatic (see next section). The reasons that make it worthwhile in the author's view include:

4.1 Reduced memory consumption for variables

Assuming variable names between 4 and 7 characters, 24 bytes per variable are saved:

	Bytes		
Type	Tcl8.4	Tcl8.5	Reduction
Local	32	8	24 (75%)
Namespace	60	36	24 (40%)
Array elem.	60	36	24 (40%)

The savings increase for longer variable (or array element) names.

4.2 Cache friendliness

- normal r/w access addresses the first two fields in the struct, in order
- the table of compiled locals is 75% smaller, reducing the data cache pressure for the bytecode engine
- joint allocation of variables and their hash table entries allows to eliminate one level of indirection in the variable's access: instead of following the entry's clientData (which hold a pointer to the Var), the Var pointer is computed from the entry's using a known offset.

4.3 Reduced impact of traces

Up to Tcl8.4 access to a traced variable was always slower: even if the current access mode was not itself traced, this could only be discovered by traversing the linked list of traces. The new code proceeds at full speed when there is no trace relevant for the current access mode

4.4 Faster creation and destruction of variables

Typically a single call to malloc() on creation, and a single call to free() on destruction

- Reuse the Tcl_Obj in the creation request as the hash table key: increase its reference count instead of allocating a new string
- Decrease the name's reference count on destroying the variable; if the name is shared, no additional calls to free()

4.5 Faster access to variables[15]

The Tcl_Obj keys allow for faster lookup: shared literals increase the probability of a very cheap test in the matching case, the fact that the string length is stored allows for faster failure in many cases. Better possibilities for caching of resolved variable names.

5 TANSTAAFL

5.1 Added complication in trace code

The trace code has to maintain the trace-related bits in the Var's flags, while previously it would just add/remove items from the front of the trace linked list

5.2 Added complication of variable code

More code is dependent on the flag values. For example, compiled local variables do not have a refCount field. All the code that manages the reference count of variables has

[13] the variable access code is as of this writing not yet fully optimised for this change

[14] further advantages are described below

[15] not yet fully optimised as of today

to check the storage class of the variable (a special flag bit) to determine if a r/w of the reference count is necessary.

5.3 A new hashtable type

The hash table used to store variables is defined via a new tclVarHashKeyType. However, the current implementation uses the standard Tcl hash tables with a custom key type.

5.4 Slower trace invocation

The invocation of variable traces is somewhat slower, as it involves a new search in a hash table. This is deemed to be more than compensated with the faster access to variables when the access mode itself is not traced.

5.5 Breaking "rogue" extensions

Extensions that include tclInt.h and interact directly with the core's variables, variable hash tables or bytecodes are broken. The code of incrTcl, XOTcl and tbcload has been adapted to the new core; it is not known if more extensions are impacted.

As a general rule, it is relatively straightforward to adapt an impacted extension to restore source compatibility.

Binary compatibility in the sense that "a previously compiled extension runs in a current core" is essentially impossible. It is possible[16] to create "binary compatible sources", in the sense that a newly compiled extension can run on both a Tcl8.4 and Tcl8.5 core. This has been done for the three extensions mentioned above.

It is to be stressed that normal extensions that only include tcl.h suffer no effects, and that most extensions that do use tclInt.h (including Tk!) are also immune.

6 Remaining (somewhat) related rewrites

There are other related modifications that may (or may not) occur in the future - either Tcl8.6 or Tcl9. Among them

6.1 Optimisation of variable lookup and caching of variable names

Variable lookup and the caching of variable names has not yet been optimised for the reformed code[17]. This optimisation will occur before the Tcl8.5 release. It is expected to provide significant speedups.

6.2 Specialized hash tables for variables

The reform described here uses Tcl's standard hash tables. These are versatile and performant, but impose some penalties on variables that could be avoided with specialized hash tables:

- the entries are "too big" for our purposes, three more fields could be eliminated for a further 33% reduction in the VarInHash size (Figure 4), from 36 to 24 bytes.
- the hash table code owes its versatility to its generous usage of indirect calls; coding specifically for variable hash tables would allow faster access

6.3 commandReform

The second most critical struct in Tcl is the command. The core has a sophisticated mechanism for caching command names, and trying to avoid renewed lookups[18]. However, lookups by name are still frequent.

A reform of the command lookup code similar to the one described in this paper will be tested. The memory footprint of commands likely being much smaller than that of variables, and command creation/destruction being rarer than the analogon for variables, the payoff in terms of memory management is unlikely to exist.

On the other hand, especially if a reform manages to also reduce the cost of verification of a cached pointer's validity, the pay off in terms of increased command dispatch performance could be sizable.

6.4 objReform

I lied: the most time critical struct in Tcl is not Var, it is Tcl_Obj. Modifying Tcl_Obj handling is however very difficult without breaking every extension out there. Some experimental attempts that could pay off handsomely (but may be infeasible in Tcl8.x) include:

- Improving the alignment of Tcl_Objs, using the padding to store small strings within the Tcl_Obj struct itself (see Patch 1772004). This reduces indirection as well as calls to malloc()/free()
- Usage of tagged pointers to Tcl_Objs within the bytecode engine to store "small" integers to reduce indirections

[16] with some nasty hacks

[17] and is clearly suboptimal for some access patterns, see Bug 1793601

[18] already improved in Tcl8.5 to reduce sharing of command name literals in different namespaces

```
typedef struct VarInHash {
    Var var;
    int refCount;               /* Counts number of active uses of this
                                 * variable: 1 for the entry in the hash
                                 * table, 1 for each additional variable whose
                                 * linkPtr points here, 1 for each nested
                                 * trace active on variable, and 1 if the
                                 * variable is a namespace variable. This
                                 * record can't be deleted until refCount
                                 * becomes 0. */
    TclVarHashTable *tablePtr;  /* Pointer to the table containing this
                                 * variable */
    Tcl_Obj *keyPtr;            /* Pointer to the object containing the
                                 * name of this variable. */
    struct Var *nextPtr;        /* Pointer to the next variable in this
                                 * same bucket (only necessary for certain
                                 * types of hashtables).
} VarInHash;
```

Figure 4: Future VarInHash struct

7 Conclusion

Internally there are two different kinds of Tcl variables: compiled locals which reside in an array, and the rest which live in hash tables. Tcl8 defined a common struct to describe both, with some fields that are only useful for one kind and are wasted in the other. Additionally, part of the variable's state is kept in extra fields that are NULL for most variables most of the time.

Different specialised structs for each kind, and a redesign that insures that the state is fully described by the flag values, permitted a very significant reduction in the memory footprint of variables. There is no space/time tradeoff involved as there are also speed gains (not of the same magnitude).

This major change respects all public interfaces; however, a few extensions that use internal apis lost both binary and source compatibility and required an adaptation.

A similar revision of other important structures and access patterns in Tcl will be explored. The expectation is that there is a possible payoff in terms of speed, but no major impact on the memory footprint is to be expected.