NRE: the non-recursive engine in Tcl8.6

Miguel Sofer

October 16, 2008

Abstract

Tcl8.6 has a completely redesigned execution model that allows among others arbitrary deep recursion in constant C-stack space. The design is derived from the ideas already present in mod-8-3-branch, implemented at a different level and much generalized. It is completely transparent to extensions, users of the standard Tcl APIs keep perfect compatibility.

We describe the basic design of NRE, with special emphasis on the trampolines that manage the execution flow and the structures that are kept in the C stack and heap during evaluation/execution. Extensions can take advantage of stackless execution using the new evaluation APIs.

New exciting possibilities will just be hinted at, a separate paper describes a couple of them in deeper detail.

1 Introduction

Usual Tcl scripts can cause deep recursive calls to the evaluation engine, as witnessed by the choice maximal nesting level of 1000 as default. In order to insure that the C stack does not blow up, Tcl has attempted to estimate the C-stack depth and error out before the stack is overrun and the operating systems shuts down the program. The mechanisms for stack checking are notoriously fragile and non-portable, in contrast with the rest of Tcl's core. Additional problems include that some platforms have either relatively small C stacks by default (BSDs), are very stack-hungry (Windows debug builds) or are even severely stack constrained (small machines like routers running Cisco's IOS).

A special Tcl build (maintained as mod-8-3-branch) was developed specifically to address IOS's constraints (among other goals); Andreas Kupries reported on that effort in the "Tcl on small machines" section of the Tcl2002 Conference, in a paper titled *Experiences with Modularizing the Tcl Core for Better Portability*. The early and very limited version of the NRE engine I contributed to that effort helped obtain a notable reduction in the stack consumption of usual Tcl scripts. The present version offers essentially[1] unlimited recursion depth: the limit is now set by the size of the heap instead of the C stack.

Tcl's stackful execution, which essentially follows the C model, also poses a barrier for lightweight cooperative multitasking independent of OS threading support. Tcl's outstanding event loop does provide facilities for cooperative multitasking, but requires an unfamiliar coding style and saving state in global variables. NRE enables a simpler approach via *coroutines*, which together with the event loop provide an extremely general and flexible approach to cooperative multitasking. NRE's liberation of the C stack also allows new constructs like *tailcalls*, which are considered to be fundamental for functional programming. These new constructs are the subject of another paper, we just mention them here as additional advantages brought by the NRE engine.

NRE was engineered to be completely backwards compatible, both in the source and binary sense[2]: no script or C extension should need adapting to the new engine. There is a C api that extensions can use to be able to exploit NRE: the traditional Tcl_Eval* APIs maintain the standard execution model, and new

[1] really unlimited depth is offered by using the new *tailcall* command

[2] Compatibility is guaranteed for extensions that do not include tclInt.h. Most extensions that do include it will also be unaffected, but the guarantee is not valid for them as some structs did suffer changes.

Tcl_NREval* APIs are provided for extensions that want to profit from stackless execution. This decision permits a gradual migration to the new execution model, or none at all.

A further goal is to remain fully portable, that is, not to resort to special features that might be provided by specific compilers or platforms. NRE is coded in standard C, it does not touch the C stack at all: it just avoids keeping so much information in it, moving data from the stack to a special Tcl maintained stack which is allocated on the heap.

NRE is somewhat related to *Stackless Python*[3]. I confess to having read some documentation on stackless years ago, but I have not looked at the code nor studied the documents in any depth. That means that I do not really know how close or remote the familiarity between stackless and NRE might be.

As to the (not really good) name: NRE stands for Non-Recursive Engine. The main design goal is to avoid recursive calls to TclExecuteByteCode(), the main component of Tcl's engine.

2 The NRE execution model: introductory example

Let us start with an example, consider the script:

```
proc foo args {...; moo $a $b; ...}
proc moo args {...; puts hello; ...}
foo 1 2 3
```

Up to Tcl8.5 the C stack while puts is running looks essentially as follows:

1. Tcl_EvalObjv # running "foo 1 2 3"
2. TclObjInterpProc
3. TclExecuteByteCode # executing foo's body
4. Tcl_EvalObjv # running "moo $a $b"
5. TclObjInterpProc
6. TclExecuteByteCode # executing moo's body
7. Tcl_EvalObjv # running TclPutsObjCmd
8. Tcl_PutsObjCmd

[3]http://www.stackless.com/

With NRE foo is completely gone from the stack, which now looks like

1. Tcl_EvalObjv # running "foo 1 2 3"
2. TclNRRunCallbacks # run callbacks registered by TclNRInterpProc
3. TclExecuteByteCode # executing moo's body
4. TclNRRunCallbacks # invoking TclPutsObjCmd
5. Tcl_PutsObjCmd

Remark that foo's body does not seem to be present in the C stack. The way this happens is as follows[4]

- Tcl_EvalObjv (level 1) is called with "foo 1 2 3"
- TclNRInterpProc registers callbacks for postprocessing and cleanup, then registers a callback to run foo's body bytecodes and returns
- Tcl_EvalObjv calls TclNRRunCallbacks
- TclNRRunCallbacks calls TEBC with foo's body
- TEBC calls TclNREvalObjv with "moo $a $b"
- TclNREvalObjv does some checks, adds callbacks for postprocessing the proc's return, adds a callback to invoke TclNRInterpProc and returns immediately
- TEBC calls TclNRRunCallbacks
- TclNRRunCallBacks invokes TclNRInterpProc, which registers a callback to run moo's bytecodes
- TclNRRunCallbacks notices that the (new!) top callback demands executing a bytecode, and that it was itself called by TEBC: it returns to TEBC to let it handle the bytecodes
- TEBC saves its state in order to resume executing foo's body later on, and starts execution of moo's body

[4]This is the pre-optimisation model; it is almost certain that some peepholing shortcuts will be taken in the optimised release version

- TEBC calls TclNREvalObjv with "puts hello", which registers a callback to run TclPutsObjCmd
- TEBC calls TclNRRunCallbacks
- TclNRRunCallbacks invokes TclPutsObjCmd

After that what will happen is

- Tcl_PutsObjCmd returns normally, without registering a new callback
- TclNRRunCallbacks runs the callbacks registered by TclNREvalObjv at puts' invocation and returns
- TEBC notes that no new bytecode evaluation has been registered and proceeds with the execution of moo's body
- When moo's body returns, TEBC processes moo's return, notes that this was a nested call, restores the state of foo's execution and continues executing foo's body

In the pre-NRE model, a proc's implementation TclObjInterpProc() does some preparation (check arguments, push a CallFrame, compiles the body if necessary, initializes arguments and local variables) and then calls TEBC with the body's bytecodes. On TEBC's return it processes the result and pops the CallFrame before itself returning.

The new TclNRInterpProc() does the same preparations, registers a callback to process the result and pop the CallFrame, registers a callback to execute the body's bytecodes and returns immediately[5]

That is: in the standard execution model the proc's implementation runs the bytecodes, whereas in the new model it registers a request that its caller run them and returns. In this manner the C call stack has been freed.

The structures that are maintained in the C-stack are a sort of two-level semi-permeable trampoline. The main player is TclNRRunCallbacks: it will run all callbacks until the top callback coincides with what it found at the top when it was first called. It functions as a trampoline in the sense that it will also run callbacks that were registered by commands it called itself. It is semi-permeable in the sense that it may allow some callbacks to go through: when TclNRRunCallbacks was itself called from TEBC, and when the top callback is a request for bytecode evaluation[6], it will just return and let TEBC handle the callback.

Thus, if all commands were NRE-enabled, Tcl's C stack when executing a bytecode will just contain Tcl_EvalObjv, TclNRRunCallbacks and TEBC. If some commands are not NRE-enabled, the C stack will contain a sequence of (Command implementation, Tcl_EvalObjv, TclNRRunCallbacks, TEBC) triplets - one for each non-NRE enabled command in the call sequence.

3 The NRE API: adapting to stackless execution

A command that does not evaluate a script or command (like *puts* in the previous example) needs no adaptation whatsoever: the NRE engine will process it normally. A command that calls one of the standard Tcl_Eval* functions is not NRE-enabled: it occupies the C stack until the requested evaluation returns. Should the evaluation require executing a bytecode, a new TEBC will be instantiated to run it.

In order to exploit NRE, that is, to be able to request an evaluation to be done by its caller, a command has to use the new Tcl_NREval* functions instead. The API provided for extension writers includes the functions:

- *Tcl_NREvalObj(interp, objPtr, flags)*
- *Tcl_NREvalObjv(interp, objc, objv, flags)*
- *Tcl_NRCmdSwap(interp, cmd, objc, objv, flags)*
- *Tcl_NRAddCallback(interp, postProcPtr, data0, data1, data2, data3)*[7]
- *Tcl_NRCreateCommand(interp, cmdName, objProc, nreProc, clientData, deleteProc)*[8]

[5]All values necessary for postprocessing (for instance the name of the proc) are passed to the callback via arguments to Tcl_NRAddCallback, see below.

[6]or other special instructions to TEBC (like yield and tailcall, described in the companion paper)

[7]The rationale for providing four ClientData field is given below

[8]The need for a new function instead of hijacking Tcl_CreateObjCommand is explained below

- *Tcl_NRCallObjProc(interp, nreProc, clientData, objc, objv)*

An NRE-enabled command requires two implementations: the regular objProc and a new nreProc. The last utility function permits a simple construction of the objProc given an nreProc implementation.

Consider a command that has a structure sketch as in *Standard Command*. The corresponding NRE-enabled command can be coded as sketched in *NRE Command.*

It is apparent that NRE-enabling a command is akin to splitting it in pre- and post-processing parts, and insuring that the two parts communicate via the callback arguments.

4 NRE-adapted commands in the core (current status)

Most commands in the core are adapted to NRE, that is, they execute without occupying the C stack[9]. The list includes:

- all procs and lambdas (arguments to *apply*)
- imported commands
- same-interp aliases
- commands created via *namespace ensemble*
- scripts evaluated by *eval*, *uplevel*, *namespace eval*, *if*, *for*, *foreach*
- TclOO object commands

The main remaining exceptions are command and variable traces, *source*, ...

5 The NRE execution model: some implementation details and technical choices

5.1 Oh so many stacks

Apart from the now diminished C stack, Tcl maintains state in three preexisting and two new stacks:

[9]These commands are also interruptible by *yield*, see the companion paper

1. the evaluation stack (actually implemented as a stack of stacks, but that is neither here nor there). The evaluation stack is used by the bytecode engine to hold the state of execution of a bytecode. The evaluation stack is attached to an execution environment, which is normally in a one-to-one relation with an interp. NRE exploits the difference, and allows more than one execution environment per interp (used for coroutines)
2. the stack of CallFrames, one per interp. A new CallFrame is pushed for each proc or lambda body, as well as for scripts run by *namespace eval*. Used for command and variable resolution, the stack is exposed by *info level*
3. the stack of CmdFrames, one per interp. A new CmdFrame is pushed by each command that is evaluated. The stack is exposed by *info frame*.
4. the new stack of callbacks, one per execution environment.
5. a new stack of BottomData, maintained by TEBC. A new BottomData is pushed by each ByteCode, it is used to store information that permits executing different ByteCodes in the same instance of TEBC. Physically the BottomData is maintained within the evaluation stack; logically each TEBC instance has its own stack.

TEBC reorganization will certainly lead to moving some of the unfrequently used automatic variables to the BottomData, and possibly to a cleaner conceptual merging of the evaluation stack and the stack of BottomData.

5.2 TEBC magic

The old NRE engine in mod-8-3-branch featured a clever TEBC, able to recognize procs (and only procs) and execute them in place. The new NRE relies on cleverness elsewhere, TEBC remains a simple minded worker. It just learned one interesting new trick, and a couple of minor ones.

The big new trick is that TEBC is now capable of storing enough data about the state of a bytecode's execution to enable suspending that execution and resuming it at a later time. That data is stored in the new BottomData struct. When a proc calls another proc, TEBC

```
int
MyCmdObjProc(
    ClientData clientData,
    Tcl_Interp *interp,
    int objc,                /* Number of arguments. */
    Tcl_Obj *const objv[])   /* Argument objects. */
{
    <preparation>
    result = Tcl_EvalObjEx(interp, objPtr, flags);
    <postprocessing>
    <cleanup>
    return result;
}
Tcl_CreateObjCommand(interp, name, MyCmdObjProc, clientData, deleteProc);
```

Standard Command

```
typedef struct TEOV_callback {
    Tcl_NRPostProc *procPtr;
    ClientData data[4];
    struct TEOV_callback *nextPtr;
} TEOV_callback;
```

TEOV_callback struct

saves the caller's state and starts executing the callee. When the callee returns, TEBC restores the caller to its previous state and resumes its execution. TEBC itself only returns when its stack of BottomData is empty.

5.3 struct TEOV_callback

The main new struct defined by NRE is the TEOV_callback (see figure). The choice of an array of four ClientData is a bit unusual, maybe even mysterious. NRE churns callbacks at an insane rate, so that the allocation and deallocation of these structs has to be extremely fast. The choice was made to use (subvert?) the fastest allocator available in the Tcl core: the Tcl_Obj allocator. A Tcl_Obj struct gives us enough space for four pointers, so that is a maximum. It turns out that in most cases a callback does not need any more data than that; restricting the interface to the standard single ClientData would force a separate allocation/freeing of a struct in all cases where more than one datum is needed.

5.4 struct Command and Tcl_NRCreateCommand

The Command struct was grown by one field to accomodate the new nreProc pointer. The first implementation of NRE just stored the nreProc in the objProc's slot, as the core's code needn't really make a difference. But this arrangement was found to break at least one common extension using an idiom which seems to be frequent: find a command's implementation via Tcl_GetCommandInfo, and then invoke the objProc directly without passing through the core's Tcl_Eval* API's. This usage breaks miserably if it happens to pick an nreProc, as it expects the command to have run when the objProc returns[10]. The choice was made to insure compatibility: every command needs to have a functioning objProc that can be called directly by a function that is ignorant about NRE. The compromise also means that Tcl_SetCommandInfo clears the nreProc - whenever this interface is used, we are back to pre-NRE functionality.

Longer term it would probably be better to deprecate invoking *objProc directly: extensions should use one of the provided evaluation functions. At that time we can slim down Command to have a single implementation, and provide APIs to allow extenders to invoke it as a string, obj or nre procedure at their choice.

[10]Remember that an nreProc doesn't run anything, it just registers callbacks for TclNRRunCallbacks

```
int
MyCmdNreProc(
    ClientData clientData,
    Tcl_Interp *interp,
    int objc,                /* Number of arguments. */
    Tcl_Obj *const objv[])   /* Argument objects. */
{
    <preparation>
    Tcl_NRAddCallback(interp, MyPostProc, data0, data1, NULL, NULL);
    return Tcl_NREvalObj(interp, objPtr, flags);
}

int
MyPostProc(
    ClientData data[],
    Tcl_Interp *interp,
    int result)
{
     Tcl_Obj  *fooPtr = data[0]; /* data0 retrieved */
     MyStruct *mooPtr = data[1]; /* data1 retrieved */
    <postprocessing>
    <cleanup>
    return result;
}

int
MyCmdObjProc(
    ClientData clientData,
    Tcl_Interp *interp,
    int objc,               /* Number of arguments. */
   Tcl_Obj *const objv[])   /* Argument objects. */
{
    return Tcl_NRCallObjProc(interp, MyCmdNreProc, clientData, objc, objv);
}

Tcl_NRCreateCommand(interp, name, MyCmdObjProc, MyCmdNreProc, clientData, deleteProc);
```

NRE Command

6 Remarks on performance and the current state of the implementation

- The current implementation (Tcl8.6a3) has not been optimized, the NRE runtime penalty may be noticeable. The initial implementation has focused on correctness and simplicity of the code, avoiding duplication even when it could have provided some speedup. Benchmarks of an earlier implementation actually showed a very slight speedup, but the implementation got a bit more involved since.
- The beta phase will see a large investment in optimization. Some obvious opportunities are available: allowing TEBC to avoid some extra calls by peepholing, inlining some frequent callbacks in TclNRRunCallbacks, merging some frequent callbacks into one, maybe streamlining the memory churn of callbacks.
- The current state of TEBC is even messier than usual. Some code reorganization will definitely happen.
- Debugging the NRE core is certainly no fun, as stack traces as provided by eg gdb are not too informative: one has to dig into the Tcl maintained stacks to determine the state of the computation. Tools to assist in debugging will be developed during the beta phase, if only to preserve what little remains of this author's sanity.

7 New possibilities and outlook

NRE opens the door to interesting new possibilities, enabled by doing surgery on the callback stack[11]:

- two of them (coroutines and tailcalls) are included in Tcl8.6a3 and described in the companion paper.
- the possibility of registering new callbacks from a callback for instance suggests the possibility of programming in continuation passing style.
- a modification of the management of CallFrames in memory would enable closures and full one-shot continuations, slightly more general than the currently provided coroutines

[11]Editing the callback stack is an endeavor that requires a lot of care, especially in the absence of reasonable debugging tools.

8 Conclusion

Tcl's new NRE evaluation model significantly reduces Tcl's C-stack footprint, enabling it to run on limited hardware and providing essentially unlimited recursion depth. It also opens the way to significant new capabilities.

The burden it places on extension writers that want to exploit the new capabilities is extremely light, and no burden at all on extensions that do not evaluate Tcl commands or that do not care about the new NRE features.

The implementation in Tcl8.6a3 should be ready in terms of features, but not yet in terms of code cleanliness and performance.