The TclQuadcode Compiler

Donal K. Fellows and Kevin B. Kenny

Abstract

This paper presents work in progress on compilation of Tcl to native code via a novel intermediate language, quadcode, and LLVM IR. It discusses some of the details of how we analyse Tcl in order to make useful type assertions, the strategy for issuing IR and native code, and presents some of the early performance results, which are believed to be of great interest.

Overall Picture

The Lehenbauer Challenge was set at the Tcl Conference in Chicago in 2012 . It actually consists of two challenges: a challenge to double the speed of Tcl, and a challenge to multiply Tcl's speed by 10.

Doubling Tcl's speed is not trivial, as it requires greatly improving the speed of key areas such as I/O (e.g., by reducing the number of times a buffer gets copied). Moreover, we have a bytecode engine that is clearly in a local optimum: most small changes to it make it slower, and Tcl 8.6.4's bytecode now covers almost all operations that it makes sense to have in an inner loop. Possible improvements would be to optimize the generated bytecode at a higher level than the current peephole system, so allowing detection of cases where a reference does not need to be shared and a copy can be avoided. While this would not improve the best Tcl code, it is likely to have quite a strong effect on code out there "in the wild".

The ten-times speedup is not in this category at all. Going by past history of participation in wider cross-language performance measurement challenges, we know that ×10 acceleration takes us into the same performance category as C and C++, especially for numerical code. Those languages get their speed from generating native code and applying careful optimization to the code they produce, and this is an area where a significant amount of effort has been devoted over decades. Making Tcl work in this performance domain is deeply non-trivial; the current bytecode engine simply is not up to the task.

But that isn't to say that it cannot be done. The real performance boost of the `Tcl_Obj` value system (one of the two key performance accelerators of the Tcl 8 series of releases) depends on the fact that values tend to be used as if they are of a particular type, i.e., the value "`1`" tends to be used as an integer, or at least as a number. This means that if we could examine the code and determine what this implicit type is, we might be able to generate efficient native machine code from Tcl. This would be ideal, as it would allow us to avoid requiring people to put extensive type annotations on their code in order to take advantage of the performance gains: while experience suggests that many would do it, it would significantly reduce the rate of adoption, and much code would never make the changes at all.

There is another fly in the ointment when it comes to generating native code. There are many platforms out there, each with it's own nuances. If we wish to generate native code, it would help tremendously to go to an intermediate format accepted by some other system so that we can leverage both their optimization code and their native code issuer. There are a number of such platforms out there [GESA+, GouGou, HHCM, KWMR+, MOSK+], but one of the more highly regarded ones is LLVM [LatAdv], the Low Level Virtual Machine, as that is used as the back end of one of the better C and C++ compilers, and can already generate executable code in memory so that we do not need to rely on temporary files for code issuing and linking.

The code associated with this paper is currently hosted in fossil on chiselapp in the *tclquadcode* repository[1].

Type Analysis of Tcl using Quadcode

The first step in the process of compiling Tcl is to convert it into conventional bytecode. This allows us to avoid having to understand all the nuances of Tcl's many commands, and leverages our existing bytecode compilation system. We see below a typical Tcl procedure (plus a couple of lines at the start that we use to introduce an ad hoc type assertion), and a fragment of the bytecode that it produces.

Figure 1: The conversion of a Tcl procedure to Tcl bytecode

[1] http://chiselapp.com/user/kbk/repository/tclquadcode/

We then translate the bytecode into a new abstract bytecode language that we call quadcode. Quadcode is named for the fact that it was originally written with exactly four list items per instruction, though that restriction has subsequently been relaxed. However, it is still highly stylized; the first word is always the opcode name, the second word is always the target (a variable or a location to jump to, when defined at all) and the third and subsequent words are arguments (variables or literals).

This simple quadcode is obtained by direct translation of Tcl's bytecode; every bytecode instruction becomes one or more quadcode instructions, ordinary variables translate in the most obvious fashion, and stack locations become temporary variables. We see below the initial translation of the bytecode to quadcode.

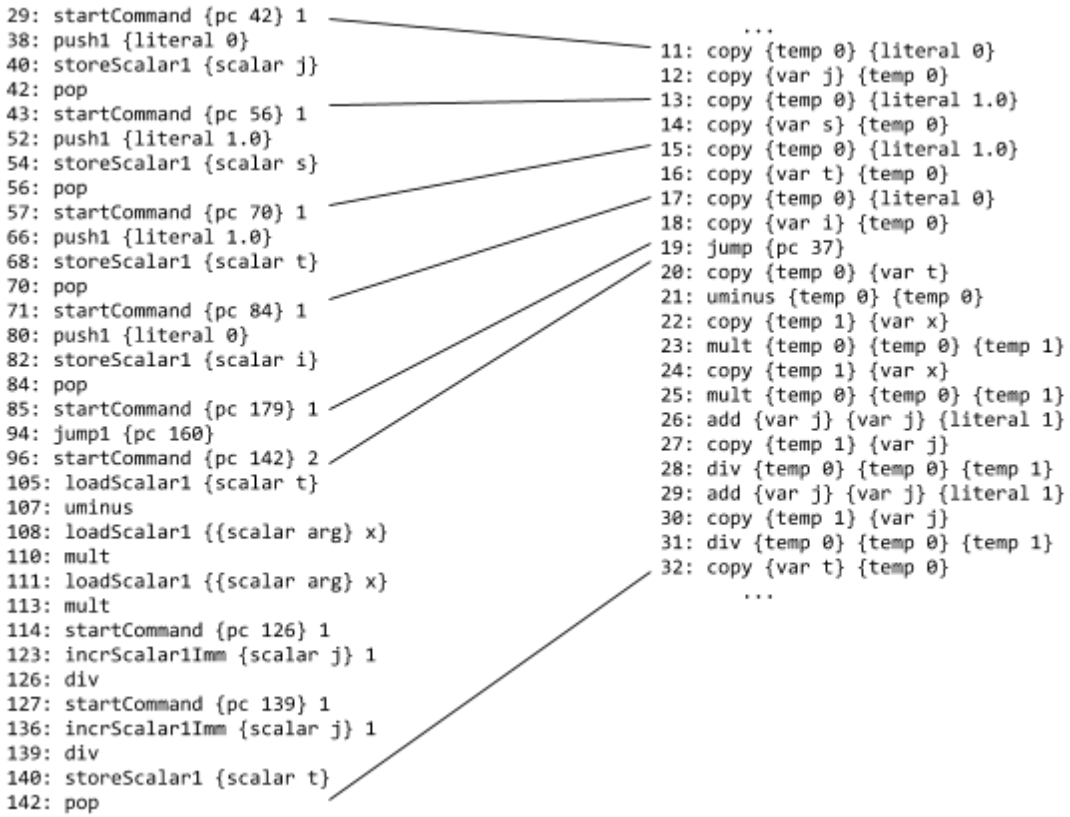

Figure 2: The conversion of Tcl bytecode to quadcode

The quadcode generated in this process is quite verbose, and working with it is expensive in both space and time. (Moreover, reading it to debug the compiler is taxing.) To speed up the rest of the compiler (and more important, to preserve the developers' sanity), several 'clean up' optimizations are performed at this phase.

> **Copy propagation.** If any operand is always the result of a copy (such as the occurrence of `{var j}` at quadcode index 12 above), it is replaced with the source of the copy (e.g., `{temp 0}`). This process may make the copy into an unused variable.

Dead code elimination. If any result (such as `{temp 0}` at quadcode index 11 above) is unused, the corresponding quadcode instruction may be eliminated. When this happens, all jumps need to be adjusted. This process may, in turn, mean that further code is unused or introduce further opportunities for copy propagation. These two steps are carried on in alternation until they detect no further opportunities to improve the code.

Jump tidying. Jumps to unconditional jumps, are replaced with jumps to the targets, and jumps to the immediately following instruction are removed. Following this adjustment, jumps are rewritten so that a conditional jump never jumps to a place in the code where multiple paths of the control flow join (by introducing an additional unconditional jump). This last adjustment simplifies inserting quadcode where needed, since it ensures that there is always a place to do so that will not interfere with other flows of control.

These optimizations produce considerably simpler quadcode. Shown below is the translation of the original procedure to the cleaned-up code.

```
proc cos {x {n 16}} {
    set x [expr {double($x)}]
    set n [expr {int($n)}]
    set j 0
    set s 1.0
    set t 1.0
    set i 0
    while {[incr i] < $n} {
        set t [expr {
            -$t*$x*$x / [incr j] / [incr j]
        }]
        set s [expr {$s + $t}]
    }
    return $s
}
```

```
0: param {var x} {arg 0}
1: param {var n} {arg 1}
2: invoke {var x} {literal tcl::mathfunc::double} {var x}
3: invoke {var n} {literal tcl::mathfunc::int} {var n}
4: copy {var j} {literal 0}
5: copy {var s} {literal 1.0}
6: copy {var t} {literal 1.0}
7: copy {var i} {literal 0}
8: jump {pc 18}
9: uminus {temp 0} {var t}
10: mult {temp 0} {temp 0} {var x}
11: mult {temp 0} {temp 0} {var x}
12: add {var j} {var j} {literal 1}
13: div {temp 0} {temp 0} {var j}
14: add {var j} {var j} {literal 1}
15: div {temp 0} {temp 0} {var j}
16: copy {var t} {temp 0}
17: add {var s} {var s} {temp 0}
18: add {var i} {var i} {literal 1}
19: lt {temp 0} {var i} {var n}
20: jumpTrue {pc 9} {temp 0}
21: return {} {var s}
```

Figure 3: Comparing original Tcl code with cleaned quadcode

Conversion to SSA

Reasoning about types and determining the lifetimes of variables requires that at each point a variable is used, it is possible to determine all the assignments to the value that could reach that point. Modern compilers, including ours, generally do that by rewriting the program into a form called Static Single Assignment (SSA) [CFRW+]. In this form, any variable in the program is assigned to only once, so it is possible immediately to see where the assignment took place. If a value is assigned in multiple places, each assignment gets its own name. When two or more assignments to a value reach a use of the value, a pseudo-function called ϕ is introduced at the place where the flows of control converge, and its result becomes the new value that flows into the use. In other words, an operation like

$$v_3 := \phi(v_1, s_1; v_2, s_2)$$

means "v_3 gets the value of v_1 if control got here from s_1, or the value of v_2 if control got here from s_2." This notation frequently enables simple one-pass algorithms to do the work that would otherwise have required complicated iterative data-flow analysis.

Converting the quadcode into SSA form, we get:

```
0: param {var x 0} {arg 0}
1: param {var n 1} {arg 1}
2: invoke {var x 2} {literal tcl::mathfunc::double} {var x 0}
3: invoke {var n 3} {literal tcl::mathfunc::int} {var n 1}
4: copy {var j 4} {literal 0}
5: copy {var s 5} {literal 1.0}
6: copy {var t 6} {literal 1.0}
7: copy {var i 7} {literal 0}
8: jump {pc 18}
9: uminus {temp 0 9} {var t 21}
10: mult {temp 0 10} {temp 0 9} {var x 2}
11: mult {temp 0 11} {temp 0 10} {var x 2}
12: add {var j 12} {var j 19} {literal 1}
13: div {temp 0 13} {temp 0 11} {var j 12}
14: add {var j 14} {var j 12} {literal 1}
15: div {temp 0 15} {temp 0 13} {var j 14}
16: copy {var t 16} {temp 0 15}
17: add {var s 17} {var s 20} {temp 0 15}
18: confluence
19: phi {var j 19} {var j 4} {pc 8} {var j 14} {pc 17}
20: phi {var s 20} {var s 5} {pc 8} {var s 17} {pc 17}
21: phi {var t 21} {var t 6} {pc 8} {var t 16} {pc 17}
22: phi {var i 22} {var i 7} {pc 8} {var i 23} {pc 17}
23: add {var i 23} {var i 22} {literal 1}
24: lt {temp 0 24} {var i 23} {var n 3}
25: jumpTrue {pc 9} {temp 0 24}
26: return {} {var s 20}
```

In effect, what has happened is that the block of code from lines 18–22 has been added, describing what happens inside the loop: the four loop variables `i`, `j`, `s`, and `t` may originate either in the block of code before the loop or from assignments within the loop.

Lifetime Analysis

The machine code translation gains part of its speed from doing explicit memory management, freeing objects when they are no longer required, rather than performing reference counting the way that the Tcl interpreter does. From the SSA form, it is reasonably simple [BHDG+] to determine the points at which a value is required at a given quadcode instruction but not at the following one, and free the value explicitly. Reference counting is not required, since there is only, ever, a single reference to a value. The translator inserts '`free`' quadcodes when this occurs.

The loop body (instructions 9–17 above) is a fairly good example of this translation. When expanded for memory management, that block of instructions looks like the following:

```
11: free {} {temp 0 37}
12: uminus {temp 0 12} {var t 33}
13: free {} {var t 33}
```

```
14: mult {temp 0 14} {temp 0 12} {var x 2}
15: free {} {temp 0 12}
16: mult {temp 0 16} {temp 0 14} {var x 2}
17: free {} {temp 0 14}
18: add {var j 18} {var j 31} {literal 1}
19: free {} {var j 31}
20: div {temp 0 20} {temp 0 16} {var j 18}
21: free {} {temp 0 16}
22: add {var j 22} {var j 18} {literal 1}
23: free {} {var j 18}
24: div {temp 0 24} {temp 0 20} {var j 22}
25: free {} {temp 0 20}
26: copy {var t 26} {temp 0 24}
27: add {var s 27} {var s 32} {temp 0 24}
28: free {} {temp 0 24}
29: free {} {var s 32}
```

This stanza begins with freeing the temporary that informed the '`jumpTrue`' instruction that entered the loop. Virtually every time a value is used, it is the last use of the value, and a '`free`' follows immediately. Note how the SSA form means that loop variables at the bottom of the loop are named differently from the ones at the top: '`j := j + 1`' results in creating a new instance of `j` and freeing the old one.

Type Reasoning

Finally, the translator must label all values with their types. This labeling is done by walking the dependency graph of the quadcode instructions. Types of literals are assumed optimistically (if a literal looks like an integer, a double, and so on, it is one). Types of operations are determined by the types of their arguments; for example, an add of an integer and a double is guaranteed to produce a double.

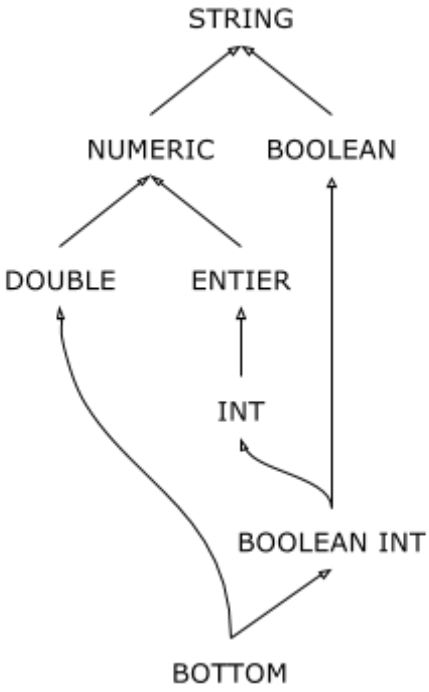

Figure 4: The inherent type logic of Tcl's built-in bytecode operations.

Circular dependencies always flow through ϕ operations and represent the natural loops in the program. Loops are solved from the inside out (this happens by identifying strongly connected components of the dependency graph), and are also handled optimistically, by having the rule at a ϕ operation be, "if the two operands are of a given type, assume that the result is of the simplest common type (integer → numeric, integer → string, etc.)." The implications of the assumption are traced. If the result is inconsistent, then the assumption is repeated with the simplest type that makes it consistent, until the process converges.

If the result of a ϕ operation is different from any of its operands, explicit '`convertToType`' instructions are inserted into the quadcode to inform the machine code translator that an operand must be widened at that point.

The result of all this is that a procedure is reduced to a block of quadcode, with ϕ operations describing the data flow, and with all values annotated with their types: the type table from the '`cos`' procedure is shown below. LLVM code translation can then begin.

```
Data types inferred:
var x 0: DOUBLE
var x 2: DOUBLE
var n 1: INT
var n 4: INT
var j 6: INT BOOLEAN
var j 10: INT
var i 9: INT BOOLEAN
var i 12: INT
var t 8: DOUBLE
var j 35: INT
var j 22: INT
var j 26: INT
var t 37: DOUBLE
temp 0 16: DOUBLE
       ...
temp 0 41: INT BOOLEAN
Return type inferred: DOUBLE
```

Compilation of Quadcode to Machine Code using LLVM

Once we have typed quadcode, it is possible to generate efficient machine code from it. The strategy we use for doing this is translation into LLVM Intermediate Representation (IR), as this allows us to use the optimizers and code generators developed by groups with far more expertise in this area. We also had the good fortune of being able to use an existing Tcl package, *llvmtcl* by Jos Decoster[2], to get started.

IR defines programs to be split up into modules, which are containers of functions and global variables (much as code units in C and C++ work). Each function is in turn split up into basic blocks, one of which is designated the entry block, the first block executed when the function is called. Each basic block starts with a sequence of ϕ-nodes (an empty sequence in the case of the entry block) and ends with a jump of some kind (always to the start of a basic block) or a

[2] https://github.com/jdc8/llvmtcl

return. Other kinds of instructions — loads, stores, arithmetic, function calls, etc. — comes in-between. LLVM does not have an explicit stack or registers; instead, instructions yield an immutable value (where appropriate) and that value is used as an operand in other instructions. The actual allocation of variables and stack space as required is done during target platform code generation. Assignable variables are done using the abstract address of a memory location that can hold the type, and an explicit load from that memory location when the value is desired, so making explicit any distinctions between the variable and its value at a particular time.

As can be seen, there are many similarities between IR and quadcode, particularly in how functions are regarded as being a collection of basic blocks, and how results of operations are immutable values. There are some substantial differences as well. The actual operations are very different, as might be expected (quadcode is still a kind of abstract Tcl bytecode, IR is more of an abstract assembly language), and quadcode arranges basic blocks into a predefined sequence, whereas IR does not. IR also has a substantially different type system, being based on integers of arbitrary fixed widths, structures, vectors and pointers; it is very obviously a system designed to support implementing C. In addition, the way that types are expressed within the instructions is quite different, as quadcode has them implicitly attached to values (with some exceptions) whereas IR explicitly attaches them to instructions. Because of these differences, the translation of quadcode to IR is non-trivial and *definitely* a one-way transformation.

The strategy we use to perform the translation is to turn each quadcode instruction into a short sequence of IR instructions. In the simplest and most common cases, the code issued instantiates the arguments to the instruction (reading from variables or generating constants as necessary) and generates a call to an internal library of functions that implements the semantics of the operation, before finally storing the result back into another variable. Some instructions are special in this regard, notably returns and jumps (which do not produce results and terminate their basic block) and ϕ-nodes (which *cannot* take arguments as they must come first in a basic block).

What happens in practice can be seen with the conversion of the loop body example from the previous section. It turns out that the floating point operations can usually be issued directly inline, but the integer operations cannot; they are variable width operations that need to do a considerable amount of processing in the case where the width changes.

```
%tmp.0 = fsub double -0.000000e+00, %phi_t
%tmp.03 = fmul double %tmp.0, %x1
%tmp.04 = fmul double %tmp.03, %x1
%j = call %INT @tcl.add(%INT %phi_j, %INT { i32 1, i32 undef, i64 1 })
%2 = call i64 @tcl.int.64(%INT %j)
%cast = sitofp i64 %2 to double
%tmp.05 = fdiv double %tmp.04, %cast
%j6 = call %INT @tcl.add(%INT %j, %INT { i32 1, i32 undef, i64 1 })
%3 = call i64 @tcl.int.64(%INT %j6)
%cast7 = sitofp i64 %3 to double
%tmp.08 = fdiv double %tmp.05, %cast7
```

```
%s = fadd double %phi_s, %tmp.08
```

Some quadcode instructions may generate failures, i.e., Tcl exceptions of various kinds (usually errors). These are distinguished by having an α-FAIL type, which is modeled using a variation on the Haskell Maybe monad. Where an instruction produces such a value, we also pass in an extra hidden argument that is the location where the Tcl result code should be written. The resulting α-FAIL is still checked for whether a failure occurred (`TCL_OK` is not used as signaling value) but it does capture what type of exception was encountered.

The other major wrinkle is that a substantial proportion of quadcode instructions are variadic, e.g., for list creation or dictionary entry reading, whereas IR instructions are almost all not. (There are some exceptions to this rule, but they are either for special operations or do not usually generate particularly efficient code, in the case of variadic function calls.) Because of this, for variadic instructions we pack the variadic part into an array and pass that array (plus its length) to the implementation function. This is practical because we completely understand just what the nature of the variable number of arguments is.

The functions in the internal library are all marked as being suitable for mandatory inlining; this means that the LLVM optimizer will insert them into the generated IR. It allows us to inject splits into the IR's basic blocks without changing the level that the compiler reasons about them (critical for correct processing of ϕ-nodes) and also to exclude execution paths that we can prove cannot happen. This greatly improves the quality of the generated code, allowing the optimization out of many basic operations such as `Tcl_Obj` reference count management (helped by the fact that we also correctly annotate the description of the Tcl API in LLVM with information about whether functions modify their arguments, what is an allocation, etc.)

The optimization step allows us to convert the `cos` loop body into this, which is close to as efficient as theoretically possible:

```
%6 = fmul double %phi_t64, %x
%7 = fmul double %6, %x
%tmp.04 = fsub double -0.000000e+00, %7
%8 = extractvalue %INT %phi_j62, 0
%9 = icmp eq i32 %8, 0
%10 = extractvalue %INT %phi_j62, 1
%11 = sext i32 %10 to i64
%12 = extractvalue %INT %phi_j62, 2
%x.6425.i43 = select i1 %9, i64 %11, i64 %12
%z.643.i44 = add i64 %x.6425.i43, 1
%cast = sitofp i64 %z.643.i44 to double
%tmp.05 = fdiv double %tmp.04, %cast
%z.643.i = add i64 %x.6425.i43, 2
%13 = insertvalue %INT { i32 1, i32 undef, i64 undef }, i64 %z.643.i, 2
%cast7 = sitofp i64 %z.643.i to double
%tmp.08 = fdiv double %tmp.05, %cast7
%s = fadd double %phi_s63, %tmp.08
```

However, the really large issue is that quadcode instructions need to understand the implicit type system of Tcl, and that does not map simply onto the types of LLVM IR simply.

Types and Their Mappings

Each type in quadcode is mapped to a specific type in LLVM IR. The simplest example is the DOUBLE type, which is mapped to an IEEE double. The INT type is rather more complex, as it needs to be able to be expanded from one type to another. Because of this, it is mapped to a triple of a discriminator flag and a pair of fields for holding the value, one a 32-bit integer and the other a 64-bit integer (the expansion to bignums is obvious, but not currently done due to some Tcl API limitations). The other major general type, STRING, which also encompasses both lists and dictionaries at present, is mapped to a `Tcl_Obj` reference that is guaranteed to be non-NULL.

Many operations support various options for implementation. For example, the `add` quadcode can be applied to either a pair of INTs, a pair of DOUBLEs, or a mix of an INT and a DOUBLE (with the INT being promoted to a DOUBLE prior to the addition). Similarly, the strlen quadcode is designed to take a STRING but needs to accept any other type. The code for all of these possible implementations would be rather long and likely to be error-prone; for that reason, we have a mechanism for automatically applying the type-lifting operation when issuing instructions.

The lifting code leverages the fact that we are using a TclOO class to manage the instruction issuing. We encode the types of the arguments in the name of the method (e.g., an add of an integer and a float becomes `add(INT,DOUBLE)` under the scheme) and detect where we do not have an implementation through TclOO's unknown method handler mechanism. Where such a method is found to be absent, we generate it automatically by applying type-widening fragments to the arguments before calling an existing method (e.g., `add(DOUBLE,DOUBLE)`) to perform the core of the operation, generating a new method on the fly that supports the operation that we really want. The mechanism for this uses a list of possible type-widenings and tries to find the minimal widening (i.e., minimum extra code added) that will generate a type-correct method. This allows us to handle the generation of a whole family of methods for doing, say, string-length on all supported values while not having to write very much code.

The final major type to mention is the FAIL type (the key to how we handle exceptions), which is an addition to the other types in that it wraps them up: it's always an INT FAIL or a DOUBLE FAIL or a STRING FAIL or one of the other basic types with FAIL applied to it. The FAIL usually manifests itself as a wrapping structure, with one field being a bit saying whether the value is present in the other field, or whether it is a failure (with the error code stored in an auxiliary variable and any error message in the Tcl interpreter's result, as normal). There are two exceptions to this: VOID FAIL and STRING FAIL. With STRING FAIL, the type is actually mapped to a `Tcl_Obj*` as with a STRING, except that the pointer

may be NULL. With VOID FAIL (only used as the result type of a procedure that has no non-error exit paths) the FAIL is just a single bit.

One significant wrinkle is where a quadcode instruction is variadic, since we avoid using variadic implementation functions. In this case, we pack the variadic parts as a sequence of `Tcl_Obj` references into an array and pass that array (plus its length) into the implementation function; the type widening is handled in a special non-branching code generation sequence that is implemented in a separate method of the instruction issuer, `buildPath` (the name reflects its original use in generating the dictionary instructions), that does the same type widening that would be used elsewhere and which produces a path structure. The methods of the instruction issuer that consume the path structure delegate the reading of the values to a separate simple method, `ExtractPath`, that picks the values out of the path and stores them in Tcl variables in the caller; those are then used in the call to the implementation function, which can have a single simple type.

Exception Handling

As you have seen, the FAIL type modifier on an instruction result indicates that that instruction may (and sometimes may always) produce an error. The initial code issuing then adds a separate check for whether the error occurred, jumping to the relevant exception target if that occurs. Every procedure that can have an exception reach the top level includes a global exception-handling basic block, which is used as a target when no other target is described in the exception contexts of the original bytecode.

However, there are also two key hidden variables that have an impact on this scheme: the interpreter and the result code (used typically for values other than `TCL_OK`). The interpreter is used to provide a route to report error messages, and is actually a global variable to the whole LLVM IR module. This is a reasonable thing to do because we never compile code that is shared between interpreters: it is always intended as a replacement for specific procedures in a specific interpreter. Because the value is a module-global, the interpreter is just used as required by the implementation functions, and is never explicitly referred to in the arguments.

The result code is specific to the current function, but is never mentioned explicitly in the quadcode as something other than a value that magically is produced by the `returnCode` instruction. In practice, the result code is stored in a piece of memory allocated at the start of the function, and a pointer to that memory is passed to the implementation functions as a “hidden” argument (i.e., not explicitly mentioned in the type-tuple in the method name) for each quadcode instruction that can produce an error so that they can set the value; the `returnCode` instruction just reads the current value of the variable. The implementation functions are all inlined and the LLVM optimizer prioritizes the promotion of variables in memory to local values, this actually gives efficient code for the management of the shadow result code.

It should be noted that take substantial care to avoid handling the `TCL_BREAK` and `TCL_CONTINUE` exceptions via the exception code system. Tcl already tries to avoid generating them in Tcl 8.6's bytecode compiler, using general jump instructions where we can, and we optimize this further by detecting cases that have slipped through that initial net and producing the jump sequence anyway during quadcode generation. This means that the actual instruction sequence issued for handling exceptions is relatively slow; as is normal with Tcl, we do not attempt to make the error case optimal.

Low-Level Code Organization

The Tcl code of the IR issuer can be divided into a few pieces. At the top level is the main driver, which presents an interface to the user and coordinates with the quadcode generator before calling the IR code issuer. The next level down is the code issuer, which is in four parts:

1. Generation of the library of implementation functions.
2. Generation of the declarations of the functions that will come from the bytecode (done as a separate stage to allow inter-procedural calls).
3. Generation of the definitions of those functions, each via issuing an entry sequence, a translation of all the quadcode instructions, and a final closing of all the unfinalized basic blocks.
4. Generation of the "thunks" which adapt the functions listed above to be Tcl commands. For example, where a function takes a DOUBLE and produces an INT, there will be an initial check of the number of arguments, then `Tcl_GetDoubleFromObj` will be called to extract the value to pass in, the call to the implementation will be done, then the resulting INT will be converted to a `Tcl_Obj*` using `Tcl_NewIntObj` or `Tcl_NewWideIntObj`, and then stored as the result with `Tcl_SetObjResult`.

Below this level are the basic organization levels that implement the type mapping, know how to translate constants, issue LLVM IR instructions and manage the interface to the LLVM module, functions and basic blocks. A substantial part of what the low level code does is act as a sanity checker, e.g., ensuring that non-trivial LLVM type constraints are maintained so that rather than causing a process abort (by far the most common failure mode when working with LLVM) coding errors generate a Tcl error that can be debugged in the usual manner.

Performance Measurements

The performance gains expected from the compilation to machine code are varied. For heavily-numeric code, the gains will come from being able to avoid the costs of getting argument values out of `Tcl_Objs`, determining the exact operation to apply, and putting the value back into a `Tcl_Obj` (including avoiding a call into the value allocation subsystem, though that should be relatively quick). For true string-based code (e.g., where a lot of strings are being concatenated into a longer string) the gains would be expected to be relatively

modest, as the implementation functions lean on the Tcl API and that is efficiently implemented. List-heavy and dictionary-heavy code may be between these extremes, as it will gain from improved reference management (such as effectively being able to avoid copies) but will still be drawing on Tcl's API.

Performance figures are computed on an idle 2.7GHz Intel Core i7. Performance ratios are expected to be constant on all similar platforms. The build was done using LLVM 3.6; performance characteristics are very similar when using other versions of LLVM (we currently support 3.5, 3.6 and 3.7; 3.4 lacks a few critical features in assertion support). We used timing runs with 1359 iterations (using more iterations was not found to yield more meaningful results) and reran the timing runs 4 times each and took the minimum execution time, so as to minimize OS-induced jitter; the execution scripts used with time are objects that are not used for anything else (i.e., they are explicitly de-duplicated) and a dummy timing run is used initially to avoid the initial bytecode compilation cost of the timing script itself. Acceleration factors are the ratio of the difference in time taken to the time taken after acceleration; an acceleration of 100% is therefore a halving of the time to execute an operation.

Sample Procedure	**Time** (µs)		**Acceleration** (%)
	Uncompiled	**Compiled**	
`fib`	12.15	0.4758	2452.90
`cos`	6.277	0.3936	1494.91
`replacing`	1.233	0.8792	40.24
`listjoin`	2.300	0.6946	231.08
`wordcounter`	18.67	5.660	229.86
`errortester`	13.73	4.999	174.54

Table 1: Summary of measured timings and acceleration factors.

Numeric Performance

It is in the area of numeric code that the most impressive improvements to performance are seen. For example, this procedure (which we use as a test of combined integer performance):

```
proc fib {n} {
    set n [expr {int($n)}]
    if {$n < 1} {
        return 0
    }
    set a 0
    set b 1
    for {set i 1} {$i < $n} {incr i} {
        set c [expr {$a + $b}]
        set a $b
        set b $c
```

```
    }
    return $b
}
```

Goes from taking 12.15µs to compute '`fib 85`' to 0.4758µs, an acceleration of 2452.90%, i.e., over 25 times faster!

Floating-point performance is also substantially improved. This procedure is our main test of that (though it also uses some integers):

```
proc cos {x {n 16}} {
    set x [expr {double($x)}]
    set n [expr {int($n)}]
    set j 0
    set s 1.0
    set t 1.0
    set i 0
    while {[incr i] < $n} {
        set t [expr {-$t*$x*$x / [incr j] / [incr j]}]
        set s [expr {$s + $t}]
    }
    return $s
}
```

It goes from taking 6.277µs to compute '`cos 1.2`' to 0.3936µs, an acceleration of 1494.91%, i.e., nearly 16 times faster, and only marginally slower than the cosine implementation in the standard C math library.

The main thing to note about the code above is that we are using the `int()` and `double()` functions to act as input type coercions. These are the only ways in which we are not writing ordinary Tcl code.

String Performance

The difference in string performance is nothing like as impressive, largely because Tcl's string performance is actually already pretty good (provided care is taken with reference count management). For example:

```
proc replacing {from to} {
    set s abcdefghijklmnopqrstuvwxyz
    set from [string first $from $s]
    set to [string last $to $s]
    return [string replace $s $from $to \
            [string cat > [string range $s $from $to] <]]
}
```

This is accelerated from taking 1.233µs to run '`replacing e k`' to taking 0.8792µs, which is a modest 40.24% faster. Longer string-based code might be a bit better, but this is the sort of performance improvement that is expected, as the limiting factors are often to do with memory management and the cost of doing the copies.

List and Dictionary Performance

Performance is substantially better with lists and dictionaries, so much so that they can be used instead of some other built-in operations. For example, this code:

```
proc listjoin {list} {
    set result ""
    set sep ""
    for {set i 0} {$i < [llength $list]} {incr i} {
        append result $sep [lindex $list $i]
        set sep ","
    }
    return $result
}
```

Goes from taking 2.300µs to compute '`listjoin {a b c d e f g h}`', to taking 0.6946µs, which is 231.08% faster. For comparison, replacing the body with '`join $list ","`' creates a procedure that takes 0.8326µs.

Similarly, we can use dictionaries to simulate arrays (provided we don't need to support traces, which we currently cannot handle anyway). This allows this code:

```
proc wordcounter {words} {
    foreach word $words {
        incr count($word)
        set done($word) 0
    }
    lmap word $words {
        if {$done($word)} {
            continue
        }
        set done($word) 1
        list $word $count($word)
    }
}
```

To go from taking 18.67µs on a short sample text to taking 5.660µs, which is 229.86% faster.

Error Handling Performance

This is not a case that has been optimized for, but there are some reasonable performance improvements possible anyway. In particular, LLVM's optimizer makes short work of most of the complexity of the try command, so with this procedure:

```
proc errortester {x} {
    set msg ok
    try {
        if {[string length $x] == 3} {
            error $x$x$x
        }
    } on error msg {
```

```
            error "error occurred: $msg"
        }
        return $msg
    }
```

The time to do '`catch {errortester abc}`' goes from 13.73µs to 4.999µs, which is 174.54% faster, and to do '`catch {errortest4 qwerty}`' (which doesn't exercise the error path) goes from 0.5955µs to 0.2335µs, which is 155.02% faster. At least some of that comes from being able to disentangle the knot of error handling paths and construct simpler code paths.

Future Directions

Some bytecodes remain to be implemented, or have *extremely* sketchy implementations at the moment. For example, the current implementations of the bytecodes that support the `unset` command just overwrite the variable with a constant, and the bytecodes that support `info exists` do not convert at all (despite the fact that it should be trivial to make them work). There are also a substantial number of opcodes that provide various forms of introspection of the interpreter that we do not currently support, such as getting the current namespace or the current TclOO context object. We also currently do not support bignums due to it being non-trivial to pick up the correct C `#defines` in order to access them correctly, and certain functions not being exposed via the Tcl stub API. These are not fundamentally difficult problems to overcome.

However, the main challenges are more substantial.

We currently do not do a good job at all at handling the invocation of other commands except for selected white-listed math functions. This is bad enough when it comes to the handling of simple commands that are not bytecode compiled (e.g., `split`, `join`) where we could easily do better than we do now, but gets rapidly more complex when dealing with commands that may deal with variables (e.g., `regexp`, `regsub`) as the compilation process does not produce a conventional Tcl stack frame.

Where we invoke commands that are actually procedures that are being compiled at the same time, we should be able to avoid going via the Tcl command dispatch engine and use a direct call of the code that we generate. However, doing this may substantially change the type signature that we are working with; we must tackle inter-procedural analysis to solve this.

A substantial fraction of Tcl procedures involve access to variables that are not defined in the local scope; if you look at an average procedure, there's a fairly high chance that at least one of the commands `global`, `variable` and `upvar` will be used, or that a fully-qualified variable name will be present. We cannot currently handle these at all, yet we must if we are to bring the benefits described in this paper more widely to the average Tcl programmer.

At a higher level, the challenges are more to do with how to extend beyond procedures to handling TclOO methods (we do not even know whether this is easy or difficult yet) and leveraging up to the package level.

Another challenge is how to integrate this with normal workflow. The current compiler is rather slow — especially the critical optimization phase — so there's a need for either allowing code to be compiled in parallel with using it, or for there to be an option to perform the compilation as a separate out-of-band phase. There's also potential for caching the generated code for future reuse, which has its own set of complexities (e.g., whether to share between users, how to ensure that the caches are correct, whether to redistribute).

Furthermore, we currently assume that code is specific to a particular interpreter; being able to share the code inside the process would help make doing efficient threaded code in Tcl much better, but would require substantial changes to how we currently generate code.

References

[BHDG+] Boissinot, B., Hack, S. Dupont de Dinechin, B., Grund, D. & Rastello, F. Fast Liveness Checking for SSA-form Programs. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '08)* (pp. 35-44) Boston, Mass., USA: ACM.

[CFRW+] Cytron, R., Ferrante, J., and Rosen, B.K., Wegman, M.N. & Zadeck, F.K. (1991) Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 451-490.

[GESA+] Gal, A., Eich, B., Shaver, M., Anderson, D., Mandelin, D., Haghighat, M. R., ... & Franz, M. (2009, June). Trace-based just-in-time type specialization for dynamic languages. In *ACM Sigplan Notices* (Vol. 44, No. 6, pp. 465-478). ACM.

[GouGou] Gough, J. J., & Gough, K. J. (2001). *Compiling for the .Net Common Language Runtime*. Prentice Hall PTR.

[HHCM] Ha, J., Haghighat, M. R., Cong, S., & McKinley, K. S. (2009). A concurrent trace-based just-in-time compiler for single-threaded JavaScript. *Proc. PESPMA*.

[KWMR+] Kotzmann, T., Wimmer, C., Mössenböck, H., Rodriguez, T., Russell, K., & Cox, D. (2008). Design of the Java HotSpot™ client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization (TACO)*, *5*(1), 7.

[LatAdv] Lattner, C., & Adve, V. (2004, March). LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on* (pp. 75-86). IEEE.

[MOSK+] Matsuoka, S., Ogawa, H., Shimura, K., Kimura, Y., Hotta, K., & Takagi, H. (1998, October). OpenJIT-a reflective Java JIT compiler. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java* (Vol. 92).