7 Availability

The ICEM CFD compiler will be eventually available as a commercial product, together with other modules including a 3-D viewer widget [9], and a form creation package. A beta release of the compiler is currently available by anonymous ftp from `ftp.netcom.com`, in `pub/ic/icemcfd/tclc`. Please send any comments and suggestions to the authors.

References

[1] John K. Ousterhout. “Tcl and the Tk Toolkit”. 1994, Addison-Wesley.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. “Compilers: Principles, Techniques, and Tools”. 1988, Addison-Wesley.

[3] Adam Sah. “An Efficient Implementation of the Tcl Language”. Master’s Report. UC Berkeley Technical Report #UCB-CSD-94-812.

[4] Adam Sah, Jon Blow, and Brian Dennis. “An Introduction to the Rush Language” Proc. Tcl’94 Workshop. New Orleans, LA. June, 1994.

[5] Michael J. McLennan. “[incr tcl] -- Object-Oriented Programming in TCL”. Proceedings of the Tcl/Tk ’93 Workshop, page 31--38.

[6] “Comparisons of Tcl with other systems”. `http://icemcfd.com/tcl/comparison.html`.

[7] “html2mif converter”. Anonymous ftp from `icemcfd.com:pub/html2mif.tar.gz`

[8] Thomas A. Phelps. “Two Years with TkMan: Lessons and Innovations”. Proc. Tcl’95 Workshop. Toronto, ON July 1995

[9] Wayne Christopher. “A 3D Viewer Widget for Tk” Proc. Tcl’94 Workshop. New Orleans, LA. June, 1994.

```
        t_4.allValue.integer = (i->v.a.integer +
            x->v.a.integer);
        TCL_SETONLYVALID(&t_4.typeAttr,
            TYPE_TEMP | TYPE_INTEGER);
        Tcl_OnlyValid(x, TYPE_INTEGER |
            TYPE_UNKNOWN);
        x->v.a.integer = t_4.allValue.integer;
        Tcl_InvokeTraces(interp, x, 0, “set”,
            W_TRACES);
        z = LookupVar(interp, “z”, NULL, 0,
            “to->C”, 1, &tap, &td);
        VAR_DELETENOTTYPE(z,
             TYPE_DOUBLE);
        Tcl_InvokeTraces(interp, i, 0, “get”,
            R_TRACES);
        Tcl_ConvertFromUnknownToDouble(interp,
             &i->v.a, &i->va.t);
        t_4.allValue.doub = (i->v.a.integer * 0.001);
        TCL_SETONLYVALID(&t_4.typeAttr,
            TYPE_TEMP | TYPE_DOUBLE);
        t_5.allValue.doub = cos(t_4.allValue.doub);
        Tcl_OnlyValid(z, TYPE_DOUBLE |
            TYPE_UNKNOWN);
        z->v.a.doub = t_5.allValue.doub;
        Tcl_InvokeTraces(interp, z, 0, “set”, 32);
        Tcl_OnlyValid(i, TYPE_INTEGER |
            TYPE_UNKNOWN);

        /* Reincrement */

        i->v.a.integer += 1;
    }
    tReturn_i = TCL_OK;
    Tcl_SetResult(interp, ““‘, TCL_STATIC);
tExit:
    <Clean up>
    return tReturn_i;
}
```

If we call the procedure `a` with the number 10000, the Tcl script takes 16.90 seconds whereas the compiled code takes only 1.04 seconds. This shows a speed-up factor of 16.25.

A more realistic example is an HTML to MIF converter. The converter needs to read text from a HTML file, parse it using `regexp`, and emit the appropriate MIF formatting commands. An example HTML file took 100 seconds using the interpreted script. The compiled code completed the same file in 16 seconds. This gives a speedup factor of 6.25.

6 Future Work

The first version of the compiler is only a starting point. We expect that future versions will expand on our base of work and continue to improve both the performance of the compiled code and the tuned Tcl interpreter that goes with it.

Our first target is to more fully support list commands. We have begun this work by adding a list structure that is similar to that used in TC. We implemented the list in order to compile `foreach` commands. The next step is to modify the list commands to use the internal list representation.

We also anticipate incorporating dynamic loading into the infrastructure. This will avoid the step of manually linking the application after the Tcl to C code is compiled.

Our initial timing tests have shown that variable declarations are extremely important. This will allow the compiler to reduce the amount of emitted code and to avoid costly conversions from strings. The number of lines of emitted code will drop by about a factor of seven if the user declares the type of each variable. Additionally, as discussed above, if we know that the variable cannot be traced or that there will be no reference via upvar, additional optimizations can be made. We also plan to implement the typed argument and return value API’s discussed above. We expect to improve the performance of the compiled code by at least another factor of four by making these changes.

Incr Tcl [5] uses methods in classes, rather than procs. There is no reason why we cannot compile these methods in a similar fashion to how procs are handled. We also plan to support namespaces when they are added to the core or otherwise become widely used.

Eventually we also expect to do byte-compilation. We will soon move to a formal parse tree representation of the Tcl script in order to carry out further analysis of the script. The byte-compilation can be implemented as a different back-end to this parse tree representation. Although code compiled to C will almost surely run faster, the convenience of on-the-fly byte compilation will be very important for many applications.

literal character `"`, followed by the string `defg` followed by another literal quotation mark character. Here, the quotation mark is not being treated as a special character. So, the second Tcl word is in reality two words: `abc"de` and `fg"`. Similarly the third Tcl word is not as expected. It is actually 4 words for the same reason as given above: `abc{d`, `e`, `f`, and `g}`.

However, the following Tcl strings are not legal:

```
“gfed”cba
“gf ed”cba
{g f e d}abc
```

The existence of characters following the quotation mark and right brace is the error in these examples. Notice that only the order of characters is reversed between the the legal examples and the illegal examples. One would expect that either both forms are legal or both are illegal. Of course, any desired result can be obtained by liberal use of backslashes, but this is not something that most users find pleasant.

5 Current Status

The compiler has passed all of the Tcl regression tests, plus additional ones that include expressions with variables. The tests all give the expected results with the exception of some error message formats. We have also tested the compiler on two tests: a HTML to MIF (FrameMaker Interchange Format) converter, and an in-house customer database management tool. The two scripts combined total about 1.3K lines of Tcl code. The compiled code produced the same results as the Tcl scripts in both cases.

Additionally, we have compiled and tested two large commercial applications in the area of computational fluid dynamics. These applications include between 15K and 20K lines of Tcl code each, and there has been a noticable performance improvement, although it is hard to quantify this improvement — most operations involve operations in both Tcl and C, in addition to graphics operations.

Users can expect a very large speed-up for numerically intensive operations. Consider the following example:

```
proc a {n} {
        set x 0
        for {set i 0} {$i <= $n} {incr i} {
                set x [expr $i + $x]
                set z [expr cos($i * .001)]
        }
}
```

Excluding the code used to set up and cleanup the proc environment, the code used for error handling, and most of the polymorphic variable code, the code emitted by the compiler can be summarized as follows:

```
int t_comp_a(ClientData t_cld, Tcl_Interp *interp,
        int t_argc, char **t_argv){
    <Variable declarations + initialization>

    n = LookupVar(interp, “n”, NULL, 0, “to->C”,
        1, &tap, &td);
    n->v.a.string = t_argv[1];

    x = LookupVar(interp, “x”, NULL, 0, “to->C”,
         1, &tap, &td);
    VAR_DELETENOTTYPE(x,
        TYPE_INTEGER);
    Tcl_OnlyValid(x, TYPE_INTEGER |
        TYPE_UNKNOWN);
    x->v.a.integer = 0;
    Tcl_InvokeTraces(interp, x, 0, “set”,
         WRITE_TRACES);
    i = LookupVar(interp, “i”, NULL, 0, “to->C”,
         1, &tap, &td);
    Tcl_OnlyValid(i, TYPE_INTEGER |
        TYPE_UNKNOWN);
    i->v.a.integer = 0;
    Tcl_InvokeTraces(interp, i, 0, “set”,
         W_TRACES);
    while(1) {

        /* Test Expression */

        Tcl_InvokeTraces(interp, i, 0, “get”,
            R_TRACES);
        Tcl_InvokeTraces(interp, n, 0, “get”,
            R_TRACES);
        t_1.allValue.integer = (i->v.a.integer
            <= n->v.a.integer);
        TCL_SETONLYVALID(&t_1.typeAttr,
             TYPE_TEMP | TYPE_INTEGER);
        t_3.allValue.integer = t_1.allValue.integer != 0;
        if (!t_3.allValue.integer) break;

        /* Body */

        Tcl_InvokeTraces(interp, i, 0, “get”,
            R_TRACES);
        Tcl_InvokeTraces(interp, x, 0, “get”,
            R_TRACES);
        tReturn_i = Tcl_ConvertUnknownToNumeri-
c(interp, &x->v.a, &x->va.t);
```

Tcl variables — before each use the compiled code must check the current type and value and be prepared to handle all possibilities. This slows down the code substantially and contributes to its size.

Our proposed solution is to introduce variable type declarations. These would allow the compiler to bypass all type checks and do much more agressive optimization.

There are two things we need to be able to declare. The first is the type of a variable: if we know that a given variable is always an integer, for example, we do not have to include the code that is run if it is of a different type. The second is whether it can be modified by `uplevel`, `upvar`, or `trace`. If we can declare a local variable to be invisible to these commands, then we are free to implement it as a C variable and bypass the Tcl variable handling facilities.

The current version of the compiler doesn't provide a variable declaration facility but we plan to do this in the future. A possible syntax is:

```
      declare variable [ int | double
| string | list | notrace | noupvar |
nounset | notcl ]* varname ...
```

The `notcl` attribute is a shorthand for `notrace noupvar nounset`: we expect that all of these attributes will be required simultaneously for the compiler to perform good optimization. This also allows the possibility of other kinds of compile-time declararations, such as “proc”, and additional types, such as float, long, handle, etc. Declarations should be usable both with local variables and with procedure parameters, although it is not clear how this would work with global or upvar'ed variables. Any comments or suggestions on this facility would be welcome.

4.6 New Command API

Variable types can be very useful for optimizing individual routines, but full advantage can only be taken of types if it is possible to pass non-string values to commands implemented by C code. In general, a command invocation will have actual arguments with a certain set of types, and the command invocation will have formal arguments with a potentially different set of types. We must be prepared to pass the types through without modification when they match, and convert them when they don't. To support this we needed to implement the following things:

1. A new form of `Tcl_CreateCommand`, which allows the specification of argument and return types.

2. A new calling convention for type-aware commands.

3. Enhancements to `Tcl_Eval` (or in our version, `Tcl_Eval_Args`) to handle the parameter type matching case.

All this must be done in such a way as to preserve backward compatibility with naive (i.e., string-only) commands.

We have not yet implemented a type-aware command API, but plan to in a future version of the compiler. A similar modification was also performed as part of the TC compiler, and should probably be taken as a starting point for future development.

4.7 Syntax Problems

We ran into the following syntax problems during the implementation of the compiler. Many are not specific to compilation, and some have been pointed out before by other users.

The quoting rules for variables are rather irregular. There are four forms of quoting in Tcl: backslash, quotation marks, braces, and, for variables, parentheses. The problem is that the substitution rules interfere with certain kinds of variable names. One that we encountered in the Tcl tests is `a(())`. The variable `a` is an array with an element named “`()`”. However, the notation `$a(())` does not do what is expected: the subscript terminates at the first closing parenthesis. Of course, the array element can be accessed with the `set` command, but the failure of `$a(())` should be considered a misfeature.

The Tcl interpreter promotes all strings to numbers if it can inside of `expr` even before it knows how the number may be used. Mostly, this is correct and will not cause a problem. However, in the case of string comparisons, there is a descrepancy. Consider the Tcl statement

set c [expr “0x12” > “0y”].

This comparison is should be false. But, 0x12 is converted to decimal 18 which is greater than “0y” so the interpreted result is 1.

Finally, the rules for quotation marks and braces are also irregular. The following Tcl strings are legal:

abc”defg”
abc”de fg”
abc{d e f g}

The first Tcl word is the string `"abc"` followed by the ner:

```
<initialization code>
while (1) {
        <convert i to numeric type if it is not already >
        if (run_time_type(i) == integer)
                test = (i.integer < 3);
        else if (run_time_type(i) == double)
                test = (i.double < 3.0);
        else if (run_time_type(i) == string)
                test = (strcmp(i.string, “3”) < 0);
        if (!test) break;
        body statements ...
        <the same test code as above, to check the
        value of i depending on the type>
}
```

4.3 Catch, Eval, and Uplevel

There are three Tcl commands that imply that the arguments are to be reparsed because of delayed evaluation. They are `eval`, `catch` and `uplevel`. In all these cases, the only feasible option is to evaluate and concatenate the arguments and then pass them to the appropriate internal Tcl routine.

However, in many cases, the user knows a priori that the arguments to these routines are already broken up correctly into words. In the case of `eval`, one would simply omit the `Tcl_Eval` call and invoke the command directly. However, there is no way in vanilla Tcl to call either `catch` or `uplevel` without reparsing the arguments. We have provided two new calls for this purpose: `catch_c` and `uplevel_c`. The compiler handles these calls by statically analyzing the command line and generating code to assemble the arguments at run time, without calling `Tcl_Eval`. This issue came up when preparing the Tcl tests, but should be useful in other cases also.

4.4 The Return Statement and the Result String

Perhaps the most troublesome feature currently in the Tcl language for an optimizing compiler is the “implicit return”. If a procedure does not include a return statement, then its return value is defined to be the result produced by the last proc or command that was executed in that procedure. This is not a problem if these result values are always produced anyway, but the compiler can determine in many cases that it does not need to maintain the result. For example, in this case:

```
        proc foo {} { set xx [expr 1 + 2] }
```

we would prefer not to explicitly set the `interp->result` string, because the type of the variable `xx` is integer, and there is no need to convert it into a string. However, because of the semantics of Tcl, the return value of `foo` must be 3, even though the user may not use it at all.

In most cases, we can simply make sure that the `result` string is set by the last statement in a proc body, which is not a large performance problem for multi-line procs. However, the last statement of a proc may be an `if` or `switch` control statement. The body may be or may not be executed. Determining the possible set of “last executed statements” of a proc body can be rather tricky at compile time, and the need to be conservative here generally leads to too much code being generated and executed.

A related issue is the `result` field in the `Tcl_Interp` structure. Because the C API to Tcl variables is well-defined and procedural, we can easily modify the variable handling code to support optional types and also maintain backward compatibility with naive code that expects variables to be always string-valued. However, the `result` string in the interpreter has no such API, and is fully exposed to the world as a string. This means that at any time that user’s code might examine the result, we must ensure that it is correctly updated, and that any intermediate integer, double, or list values have been converted into a string. Needless to say, this is both a performance problem and additional complexity for the compiler.

We suggest two incompatible changes to Tcl to fix these problems. First, procedures that don’t include an explicit `return` statement should have a return value of the empty string. It can easily be argued that the use of implicit return values is poor style. Second, the result string in the `Tcl_Interp` structure should be hidden, and made accessible only via a C API. This would allow us to maintain non-string result values until a conversion to a string is explicitly requested. We feel that the second change would be relatively painless, since as soon as old code is recompiled with the new `Tcl_Interp` definition, the compiler will point out all the lines that need to be changed.

4.5 Variable Declarations

In general, a Tcl variable can be modified at any time. Called procedures and commands can use `uplevel` or `upvar`, and traces can be applied to a variable that changes its behavior unpredictably. This makes it almost impossible to do compile-time type determination for aid in debugging. However, the Tcl variable names may be reserved words in C, or may contain illegal C identifier characters like quotation marks and parentheses. Because of this, Tcl variables that are C keywords are given the prefix `__temp_c_` and all illegal variable characters are translated to underscores. This is probably not relevant for most users since these variables are all local to the generated C functions.

Also, function names must be produced for the generated code. These are visible to the user, but rather than trying to guarantee any particular translation scheme, the main compiler command returns these names together with the original proc names, so that the user can be sure to correctly initialize the compiled code.

3.4 The Main Compiler Command and Driver Scripts

The basic compilation command is `tclc_compile`. It takes a Tcl script as input, and produces C code that implements the script. Currently it is possible to compile both procs and statements, but the code for statements will not be enclosed in a C function — this is currently useful for debugging but should not be used otherwise.

Options to the `tclc_compile` command include the following:

`-verbose`	Print the names of procs as they are compiled.
`-only_procs`	Ignore all statements other than procs.
`-file` *fid*	Write the output to the Tcl file handle *fid*, rather than returning it as a result.

The return value of `tclc_compile` is a list of triples, one for each proc compiled. The first element is the proc name, the second is the C code that declares the command function, and the third is the `Tcl_CreateCommand` line to actually define it for Tcl. The results are provided in this form because future versions of the compiler may require different function declarations and different versions of the "create command" call.

Since this interface is rather low-level, we have provided a few sample scripts that take files with Tcl code, and produce .c files, including an initialization function to set up all the commands. It is easy for users to customize these scripts for their own applications.

4 Difficulties in Compiling Tcl and Suggested Language Changes

We have found a number of places in the Tcl language definition and C API that presented special difficulties for compilation. Some of these have been pointed out before, but others were unexpected.

4.1 Variable types

In general, static analysis cannot be used to determine the types of variables. Consider the following code fragment, for example:

```
set a 1
a_proc $a
set c $a
...
proc a_proc {val} {
        uplevel 1 {set a “The value of a is $val”}
}
```

The compiled code has the following features.

1. Since constants have a known type, the first set statement directly assigns the value and type fields of the variable `a`.

2. In the proc invocation, we are able to pre-parse the command line and call the function implementing `a_proc` directly, using `Tcl_Eval_Args`.

3. After the proc invocation, we cannot assume anything about the value and type of `a`. This variable and any others may have been modified during the call, and in fact in this example `a` was modified. Because of this, to do the second `set` statement we have to check all possible types.

4.2 Control statements

We recognize the Tcl control statements `for`, `while`, `if`, `switch`, and `foreach`, and generate code specially for them. However, there is a potential problem relating to type overloading in the loop commands for and while. It is possible that the test expression type changes in the body of the loop. Therefore, it is necessary to convert the test expression to a numeric type if one can and otherwise use string comparision, every time we go through the loop. Consider the following code fragment:

for {set i 0} {$i < 3} {incr i} { *statement* ... }

This loop command is translated in the following manspeedup of between 5 and 10 times for non-trivial computationally-intensive applications.

A secondary goal was to provide something that is simple and convenient to use. The compiler is able to handle individual Tcl commands one at a time or in a sequence, but the easiest way to use it is to compile proc definitions, and then ensure that the Tcl files containing these definitions are not loaded when the executable that includes the generated C code is run.

Other objectives include efficiency of compilation, both in terms of compilation speed and size of generated code. These are areas we are currently working on.

3 The Compiler — Design and Implementation

The formal definition of the Tcl language can be summerized as

*accept: tcl_command tcl_string**

where *tcl_command* is a non-terminal that must derive to a valid Tcl command name and *tcl_string* is a non-terminal that ultimately must derive to a string. The syntax of arithmetic expressions is regular, so we were able to use Yacc to parse this portion of Tcl. However, since there are no reserved words in Tcl, and because of the way strings are put together with embedded commands and variable expansions, the definition of a Yacc parser and a Lex analyser proved infeasible for the rest of the language.

Instead, we wrote a customized state machine lexigraphic analyzer based on the Tcl interpreter. We parse the language with a recursive-descent parser. Currently, we avoid explicitly producing a parse tree by using a syntax-directed translation scheme [2]. We will build a parse tree in the future to allow us to carry out further syntactic and semantic analysis to produce more efficient C code.

Commands and words in the Tcl command line are first recognized. A compilation routine is associated with each compilable built-in command by adding some information to the Tcl command table. This compilation routine is called with all of the recognized command line words.

Generally, C++ nodes representing the command and each command line argument are constructed. The translation of the command is a synthesis of all of the command arguments along with a set of attributes that we associate with each node. The attributes include type, error status, return status, etc.

Finally, once all of the Tcl commands are parsed, a string representing the entire translation is returned to the caller via the Tcl interpreter `result` variable. If the user requests that the output be written to a file, a set of records representing the commands compiled is returned to the user.

3.1 Changes to libtcl.a to support variable types

One of the major speed improvements that we have obtained from Tcl is because of the introduction of runtime variable typing. We expanded the `Tcl_Var` structure to include additional fields for integer, double, and list representations, in addition to the string representation. The normal `Tcl_SetVar` and `Tcl_GetVar` routines operate on the string representations, but additional accessor methods have been defined that work with the other representations. The code now maintains flags that specify which of the representations are valid: when one representation is modified, all of the others are invalidated. If a currently invalid representation is requested, it is re-created from the valid ones. (E.g., if a string representation is needed and only the integer representation is valid, then the conversion is done using `sprintf`.) A similar scheme was described for the TC compiler.

This enhancement is downward compatible, since the previous Tcl accessor routines continue to work as before, and the new API is available not only to the compiler but also to any user code that wishes to take advantage of it.

3.2 Addition of Tcl_Eval_Args

There is only one way in standard Tcl to invoke a Tcl command: `Tcl_Eval`. This requires that the command name and all its arguments be flattened out into a string, and then later re-parsed. In many cases, both in the compiler and in other applications, the arguments are already known to the calling code, and the flattening and reparsing of the command is unnecessary. We have added a new routine, `Tcl_Eval_Args`, that takes an argc/argv pair, and calls the command directly. This routine is to `Tcl_Eval` as the UNIX `execv()` is to `system().`

3.3 Name Space Management

When possible, the Tcl variable names used by the user are the ones produced for the C output. This is done to

A Tcl to C Compiler

Forest R. Rouse
ICEM CFD Engineering and the
University of California at Davis
rouse@icemcfd.com

Wayne Christopher
ICEM CFD Engineering
wayne@icemcfd.com

Abstract

A Tcl compiler is described, which accepts Tcl code and generates C code as ouput. Each Tcl proc is converted into a C function that can be called as a Tcl command. This code is then compiled and linked with a special `libtcl.a` to create an executable. This compiler has passed all the Tcl tests, and has provided a speedup for realistic computationally-intensive applications between a factor of 5 and 10, and for simpler benchmarks up to a factor of 20. In this paper, we will discuss the compiler and the issues involved with compiling Tcl code, including suggestions for future changes to the language.

1 Introduction

The Tcl language was originally designed to be an easy-to-use command language that could be embedded in existing programs that needed a command line interface. Later, with the introduction of Tk, it evolved into a GUI scripting language. For both of these applications, speed is not critical —the runtime of the existing program, in the first case, and overhead of the X window system, in the second, were expected to greatly exceed the execution cost of interpreting Tcl scripts.

However, in the past several years, many Tcl users have developed large applications that do significant computation at the Tcl level. Examples we are familiar with include an HTML to MIF converter written entirely in Tcl [7], and early versions of the TkMan system [8]. Tcl has proven to be rather slow for such applications. The usual solution has been to convert selected procs into commands implemented in C, which are either loaded together with the Tcl interpreter or run as separate programs. For example, the computationally intensive parts of TkMan were later rewritten as a C program because Tcl wasn't fast enough.

This situation has led many Tcl users to suggest the implementation of a Tcl compiler. Comparisons with other interpreted systems that include a compiler, either a full compiler into C or machine code, or a byte-compiler that produces code for a Tcl virtual machine, indicate that a speed improvement of an order of magnitude should be possible [6]. The TC system by Adam Sah [3] supports this conclusion.

This paper describes a first implementation of such a compiler. The ICEM CFD Tcl compiler accepts Tcl proc definitions, and generates C code that implements commands that do the same thing as the procs. Facilities like dynamic loading and byte compilation should be able to further streamline this process.

We first describe the goals of this project, and present an overview of the design and implementation of the compiler. We will then discuss the areas of Tcl that make compilation difficult and suggest changes to the language and the C API that would make it more amenable to compilation. Finally we will outline future work in the area and discuss the availability of the compiler.

2 Goals

Our major goal in the implementation of the compiler was to accept the Tcl language as it is currently defined, and to produce code that is compatible with the existing C API for Tcl. Clearly, if we were able to make certain changes to the language, as was done with Rush [4], we would be able to greatly simplify our task, but this does not adequately help users who cannot rewrite their existing applications. Also, the Tcl language as it exists has become widely used, and every day it becomes more difficult to change the language in an incompatible way. Because of this, we made sure that the compiler passed all of the Tcl tests before the first beta release.

Another important objective was to produce C code that is significantly faster than the Tcl code it replaces. There are a number of factors that make this difficult, as discussed below, but nevertheless we are able to achieve a

The following paper was originally presented at the
Third Annual Tcl/Tk Workshop
sponsored by Unisys, Inc. and USENIX Association
Toronto, Ontario, Canada, July 1995.

A Tcl to C Compiler

Forest R. Rouse and Wayne Christopher
ICEM CFD Engineering and the
University of California at Davis