CLIFF: C LANGUAGE INTERFACE FOR THE FUNCTIONAL SIMULATOR

Kurt Baudendistel

AT&T Bell Laboratories
Murray Hill, NJ 07974
baud@research.att.com

ABSTRACT

Linkable simulators for Programmable Digital Signal Processors allow the development of heterogeneous executables that mix object code executing directly on a host workstation with object code executing indirectly on a target device via a software simulator or hardware emulator running on the host. However, these linkable simulators typically require extensive development of interface code to create such heterogeneous executables. This new tool allows interface code to be generated automatically, allowing the creation of heterogeneous executables with only minor modifications to original C language source code.

Since the required modifications to the original source code are minor and independent of the execution domain, a new methodology of porting DSP applications from host workstations to PDSPs can be considered, where functions are moved one at a time from the host to the target execution domain.

1. INTRODUCTION

Modern, high level computer languages make the task of porting software from one platform to another quite manageable, given that both platforms provide equivalent languages, arithmetic capabilities, library support, software development tools, and and run-time environments. This is not the case, however, when developing production-quality software for a DSP application targeting a Programmable Digital Signal Processor (PDSP), precisely because the host and target platforms are quite dissimilar.

Traditionally, C language host code simulating an entire DSP application and PDSP assembly language target code realizing that same application are separately created and compiled to produce two homogeneous executables.[1] From the shell prompt of a host workstation, the host code is then directly executed, while a software simulator or hardware emulator is invoked to execute the target code.[2] The ultimate goal of such software development is to produce equivalent executables in the two domains, but this is a difficult and labor intensive process.

Recent advances in PDSP simulators, however, allow the creation of heterogeneous executables. "Linkable" or "object code" versions of such simulators have become available which allow target code to be invoked and executed from within host code [1]. This allows the mixing of host and target code in a single, conceptual executable with object code "executed" by a processor of the appropriate type—the result is heterogeneous execution of the DSP application. This capability is similar to the remote procedure call capabilities of the UNIX operating system, which provide a standardized method for realizing heterogeneous execution in a workstation environment.

Like remote procedure calls, however, linkable PDSP simulators are difficult to use from the point of view that significant attention must be paid to the interface between host and target code [1]. A typical application requires extensive recoding at the point where a transition is to be made between host and target execution, and this can become especially onerous if the point of transition is to be changed relatively often during the code development process. For this reason, linkable simulators are most often used not as stand alone tools by end users, but rather they are used by designers of systems such as Ptolemy or COSSAP to incorporate heterogeneous execution capabilities into these DSP design environments [2, 3]. While the use of heterogeneous execution within such environments can alleviate many of the problems associated with code development, it provides no relief outside of the environment.

The process of generating the required interface code can be automated, however, by using a compiler. The C Language Interface For the Functional simulator (CLIFF), uses a compiler to convert original C language application code, with minor modifications, into the C language interface code required to drive a linkable simulator. Furthermore, since only minor modifications to original C language source code are required to utilize CLIFF, a new methodology of porting DSP applications from host workstations to PDSPs can be used. Host and target domains are considered to exist, and individual functions and external variables are assigned to one of the two domains on an item-by-item basis at compile time. C language functions can then be migrated one at a time from the host domain to the target domain, with (1) a C compiler or assembly language programmer providing the target object code, (2) a linkable simulator providing the execution environment for that code, and (3) CLIFF automatically generating the interface code required by the linkable simulator. When hand-coded, assembly language target code is to replace original C language source code, the results of these two versions of a

[1] *Compile* here is taken to mean conversion of source code, both high level or assembly language, into object code.

[2] The term *simulator* will be used henceforward to refer to both software simulators and hardware emulators or accelerators. While the distinction has significant practical implications in terms of processing speed and communications costs, it has no bearing on this discussion regarding interface code generation. The tools and methodologies presented here can be applied to both hardware and software realizations with equal success.

0-7803-2431-5/95 $4.00

function can be directly compared to verify the correctness of the new code. Such incremental cross-compilation using CLIFF provides a secure path for code migration from host to target, allowing verification at all steps along the way.

2. COMPILATION

CLIFF allows the programmer to control the domain in which object code is to be executed, without changes to the C language source code of the application, given that the coding conventions presented in the next section are followed. Alternate compiling and linking steps are all that are required to use CLIFF to change the domain in which a function is executed. The granularity of this control is the same as that used for other purposes in the C language—the file.

Consider the C compilation command

```
$ cc a.c b.c c.c                    (1)
```

which compiles the three source files and links the resulting object files to produce the executable `a.out`. Other command line options can of course be added to alter this compilation process.

Conceptually, the CLIFF compilation command

```
$ cliff -host a.c -transition b.c -target c.c  (2)
```

is equivalent, except that it produces a heterogeneous executable where execution is divided between the host and target domains. Practically, this command is actually much more complex:

1. Compile `b.c` into interface code known as *stubs* into `stub-b.c`. Note that the original external data and function definitions, not declarations of same, are compiled into stubs.
2. Compile `a.c` and `stub-b.c` into host object code and link to produce a host executable `a.out`.
3. Compile `b.c` and `c.c` into target object code and link to produce a target executable `a.out-cliff`.[3]

Any command line options or files appropriate for `cc` are appropriate for `cliff`—they are simply routed to one of the 3 compilations steps listed above. Of course, assembly language files can be specified on the command line as well, as long as they are routed appropriately to the host or target. Additionally, several `cliff`-specific options are available to perform optional bookkeeping and debugging functions.

`cliff` allows the programmer to route command line files and options, and thus partition the functions contained in the source files, using switches to control the *compilation state* of command line processing: `-host`, `-transition`, or `-target`.[4] The compilation state in force when the file name is encountered on the command line determines the type of the functions contained in that file:

Host functions can be called only from within other host functions and execute on the host.

Transition functions can be called from within any function but execute on the target.

Target functions can be called from within transition or target functions and execute on the target.

External data is also partitioned according to the compilation state. *Host-, transition-,* and *target-external data* can be accessed from host, any, and transition or target functions, respectively. In (2), `a.c` is considered to contain host functions and data, `b.c` transition functions and data, and `c.c` target functions and data.

While this partitioning is static with respect to a given compilation, it is not static between compilations. For example, the compilation expressed by (1) effectively places all functions and external data in the host domain. Specification of the partition on the command line allows any partition to be produced with no modifications whatsoever to the original C language source code.

3. TRANSITION CODING CONVENTIONS

The *transition* is the point at which control passes between the host and target domains—the call and return of a transition function. While flow control is obviously vital, a transition is only of real value if information can be communicated across the transition. When a transition is made with CLIFF, the state of the target machine, registers and memory, is set up as needed, the target function is executed and ultimately returns, and the state is copied back to the host registers and memory as needed.

Because of the basic nature of the transition as an interface between two, separate machines, each with associated register sets and memory spaces, only a restricted subset of the data types of the C language can be communicated through the transition. Furthermore, the mechanism for this communication is not exactly like that of the C language.

The most basic requirement is that parameters must be passed by value. For most quantities, this is not a problem, since this is the usual parameter passing mechanism of the C language. Arrays, however, present a problem that is discussed in detail below.

The second important requirement is that the data values in the host and target memory spaces be *structurally equivalent.* That is, the bit patterns in the host and target memory spaces must be identical even though the word sizes of the two machines may be different. If this is not the case, *translation* would be required to convert from one form to the other. One form of translation is performed automatically by CLIFF for the DSP1610—character strings are converted from the host format, a null-terminated sequence of characters represented by 8-bit bytes, to the target format, a 0-terminated sequence of characters represented by 16-bit words. This translation feature is provided so that debugging information can be passed into and printed from within target code. Other translation rules could be devised, but any fixed set would ultimately fall far short of the capabilities desired by users, and a general translation mechanism is beyond the syntactic scope of the C language. No facilities for translation in the transition are provided by CLIFF, except for character strings.

Because of these two requirements, only a restricted subset of C language data types can be passed through the transition:

[3] This executable is not useful in isolation—functions contained in it are to be invoked from the host executable `a.out` via calls to the linkable simulator contained in the stubs.

[4] Actually, the `-transition` compilation state is equivalent to parallel specification of `-stub` and `-target` compilation. Separate specification is required when hand-coded assembly language is to be used.

- *Constant-sized* integral scalars, here types `short` and `long`, can be passed as parameters, return values, or external variables. Constant-sized here means that the `sizeof()` the data type is the same in the host and target domains.[5]
- IEEE single-precision floating-point values, here type `float`, can be passed as parameters, return values, or external variables.
- Structures consisting of a single-size scalar at the atomic level, `short` or `long/float`, can be passed as parameters, return values, or external variables.[6]
- Arrays of any other legal quantities can be passed as parameters or external variables. Multidimensional arrays are allowed.
- Read-only strings, here type `const char *`, can be passed as parameters.

Non-constant-sized integral scalars, here types `char` and `int`, other non-integral scalars, here type `double`, pointers, and structures other than those noted above cannot be communicated through the transition because they are translational entities.

Arrays present a special challenge for CLIFF because, in the C language, arrays are passed as parameters to functions by reference—a pointer to the first element is simply provided. The dual memory spaces of the host and target require, however, that the arrays be passed through the transition by value. Note that a function definition with an array as a parameter can take any one of three equivalent forms in ANSI C:

```
void moe   (short *i)     { ... }
void larry (short i[])    { ... }
void curly (short i[10]) { ... }
```

In all cases, the parameter `i` is implemented within the function as a pointer. The definition of the function `curly`, however, uses what is termed here a *sized-reference parameter*. While the array dimension 10 is ignored by the C language compiler, it is used by CLIFF to determine that the reference parameter `i` points to an array of 10 elements, each of type `short`. This allows arrays to be passed through the transition by value, since the ultimate size of the entity to which the reference refers is given.

Transition functions with reference (pointer) parameters are not legal with CLIFF, but transition functions with sized-reference parameters are allowed. Sized-reference parameters effectively transform the call-by-reference mechanism used by the C language for arrays into the call-and-return-by-value mechanism used by CLIFF. This mechanism is described as "call and return by value" because the array is copied from the host memory space to the target memory space at the invocation of the transition function, and it is copied back from the target space to the host space when the function returns. If the array is declared `const`, however, no return-copy takes place, and the mechanism can be described by the more familiar term "call by value."

[5] It is assumed here that the host compiler uses 8-bit `chars`, 16-bit `shorts`, and 32-bit `ints` and `longs`, while the target compiler uses 16-bit `chars`, `shorts`, and `ints`, and 32-bit `longs`. With different assumptions, a different set of integral types would be constant-sized and hence legal in the transition.

[6] A single-size at the atomic level is required because alignment differences in the host and target can make other structures translational.

For simple calling interfaces, call-by-reference and call-and-return-by-value are equivalent, but it is important for the programmer to understand the subtle difference so that *aliasing*, the use of different names for the same data, does not cause problems. Aliasing can cause incorrect results in CLIFF simulations if

- A transition function that takes two non-`const` sized-reference parameters as arguments is invoked with the same array for both arguments, or
- A transition function that takes a non-`const` sized-reference parameter as an argument is invoked with a transition-external array as that argument.

Three other potential pitfalls of sized-reference parameters are the following:

1. Scalar arguments to be passed by reference must be defined using a sized-reference parameter rather than a pointer:

   ```
   bob (short white[1]) { ... }
   ```

2. Arguments that are pointers to arrays to be accessed at an offset from the given pointer value can be handled simply by increasing the array size for positive offsets:

   ```
   this (short works[OFFSET+SIZE]) { ... }
   ```

   Negative offsets cannot, however, be accommodated.

3. Array arguments with a length that is to be determined at run-time can be most easily handled using the variable-length array capabilities of `gcc` [5]. The result, however, is non-ANSI compliant C source code. If this is a significant concern, other mechanisms, such as `const` declarations with arrays padded out to their maximum length or preprocessor macros, can be used to produce ANSI compliant code.

Even with all of these caveats, however, sized-reference parameters are quite easy to use in practice since aliasing and the other special cases are not particularly common occurrences in DSP applications.

4. TARGET CODE INTERFACE

The conventions used in receiving data via the transition in the target domain are controlled by the type of *frame* in use for the system. Any kind of frame could be defined, but two of special interest that are described here are the *C frame*, which realizes the calling conventions of the target C compiler, and the *register frame*, which gives the programmer full control over the transition. Note that these frames are mutually exclusive compiler options, and one must be used for an entire CLIFF simulation system.

4.1. The C Frame

The C frame realizes the calling conventions of the GNU C Compiler `gcc` in the target domain. As such, it is suitable for use when the run-time environment of this compiler is in use—stack structure, calling conventions, *etc.* In this case, no additional code must be provided by the programmer to effect the transition, since (1) transition-external definitions allocate storage for external variables and (2) the stack itself is used to allocate storage as needed for sized-reference parameters that must be passed and returned by value.

4.2. The Register Frame

The register frame, however, assumes nothing about the structure of the run-time environment of the target code, even regarding the existence of a run-time stack. This capability is quite powerful, in that any coding or calling conventions can be accommodated, but it also places an additional burden on the programmer to specify variable names appropriately and to provide any required memory buffers.

The `asm` construct of the GNU C Compiler can be used to set assembly language names to something other than their default values:

```
short a;            /* default name: _a */
short a asm ("a"); /* explicit name: a */
```

While this construct may be of some use with the C frame, for the register frame the assembly language name of an external variable determines how that variable is passed through the transition:

```
short x;             /* in Y memory at _x */
short x asm ("z");   /* in Y memory at z  */
short x asm ("Z");   /* in X memory at Z  */
short x asm ("a1"); /* in register a1    */
```

The default mechanism is to place the external in memory at the named location, but if the name used is that of a register of the target device, then that value is loaded to and restored from the target register instead (a1 is a register of the DSP1610).

The C name of a function parameter determines how that parameter is passed to and returned from the target domain:

- Parameters with common names are stored in memory that must be allocated with the name *_name*.
- Non-sized-reference parameters with register names have the named register set to the parameter value upon function entry.
- Sized-reference parameters with register names must have storage allocated with the name *_function-name_register-name*. On function entry, memory is set to the values of the parameter, and the register is set to the address in memory at which the values have been stored.

The register from which a function return value is taken, other than the default (a0 for the DSP1610), can be specified by giving an assembly language name for the function of the form *function-name$register-name*.

The memory space used in the transition is usually the Y memory (data) space. If the assembly language name is given with no lowercase letters, however, this quantity is placed in the X memory (code) space rather than in the Y memory (data) space. This convention is used for both external data and parameters, including sized-reference parameters with register names. Thus, a common notation with the register frame for the DSP1610 will be the use of the parameter name PT for sized-reference parameters, indicating that the data should be stored in the X memory (code) space at the address *_function-name_*PT and that the pt register is to be loaded with this address. If this code space is ROM, the const qualifier would also be appropriate.

5. SUMMARY

CLIFF provides an automatic mechanism for generating the interface code needed to exploit the capabilities of a PDSP simulator to produce a heterogeneous executable from original C language source code. Because the partitioning of functions and external data between host and target domains takes place via `cliff` command line options, source code can be easily and readily moved between domains, allowing incremental cross-compilation to be used as part of the process of porting code from the host to the target.

CLIFF has been implemented using an SGI Indigo as the host and the AT&T DSP1610 as the target, and it has been in use for the past year at Murray Hill. Two versions of CLIFF have been implemented, one based on the `lcc` compiler and one based upon the GNU C compiler `gcc` [4, 5]. While both provide baseline ANSI C compatibility, the GNU C compiler also provides several important language features over and above those of ANSI C, most notably variable length array notation and C++ compilation capabilities.

Porting the current DSP1610 target to another host platform would be a simple task, given that gcc and the DSP1610 linkable simulator have already been ported to that platform [1, 5]. Developing CLIFF for alternate PDSP targets would be more challenging, but straightforward, given an equivalent linkable simulator and GNU C compiler for the new target. Developing CLIFF to interface with a hardware emulator or accelerator would require simply exchanging the linkable simulator interface for an equivalent linkable emulator or accelerator interface.

The C++ compilation capabilities of the GNU C compiler are of special interest for heterogeneous compilation. One of the significant problems associated with PDSP code development is that the transition between host and target domains with the C language requires structurally equivalent parameters and external data—the bit patterns in memory must be exactly the same in the two domains. Allowing alternate representations in the two domains and providing for automatic translation between them will create the much more powerful and flexible simulation capabilities needed to deal with problems such as those presented by fixed-point arithmetic. Translation strategies could be embedded in CLIFF, but such fixed translation strategies are doomed to failure because of the wide variety of data formats that will certainly be required. Future development of CLIFF will concentrate on using C++ to allow user-specified parameter translation in the transition.

6. REFERENCES

[1] *DSP1600-LFS User Guide*, AT&T Microelectronics, 1994.

[2] Lee, Edward A., *et al.*, *An Overview of the Ptolemy Project*, University of California at Berkeley, 1994.

[3] *COSSAP User's Manual*, Cadis, GmbH, 1993.

[4] Fraser, Chrisopher W., and Hanson, David R., *A Code Generation Interface for ANSI C*, Technical Report CS-TR-270-90, Princeton University, 1992.

[5] Stallman, Richard M., *Using and Porting GNU CC*, Free Software Foundation, Inc., 1993.