GANDF:Status and Design

Richard L. Ford

Open Software Foundation Research Institute

April 7, 1993

GANDF is an experimental ANDF installer component, based on the GCC back-end technology. Details of the design of GANDF and its current status are presented.

1. Introduction

ANDF is an architecture- and language- neutral distribution format being developed by OSF and other collaborators around the world. It is based on the TDF technology provided by the Defence Research Agency (DRA) of the UK Ministry of Defense.

GANDF is an experimental ANDF translator being implemented at the Open Software Foundation (OSF) Research Institute (RI), based on the back-end technology of the Gnu C Compiler(gcc), produced by the Free Software Foundation, along with some support routines from the DRA technology.

The primary objective of GANDF is to show that ANDF installers can be successfully built by interfacing ANDF to existing compiler back ends. Thus system vendors can make use of ANDF while still preserving their compiler investments. Secondary objectives are to increase the availability of ANDF installers and to help educate others in the technology needed to write ANDF tools.

An ANDF installer is considered “successful” if it produces code that is correct and that has efficiency that is close (within 5%) to that produced by the best non-ANDF technology. The installers that DRA has developed (targetted to the Intel 386, Motorola 68k, MIPS, and SPARC) achieve this level of efficiency (for selected benchmarks) when used in conjunction with the DRA C producer. One question to be answered by the GANDF experiment is whether similar efficiency can be achieved by an installer that does not have detailed knowledge of the strategy of the ANDF producer. In other words, does the ANDF itself carry enough information for generation of efficient code, or must one also know something about the patterns of ANDF that the producer will produce?

More specifically, we would like to see whether an ANDF installer based on a particular back-end technology (e.g. the GCC back end) can achieve parity with a C compiler based on that same back end. Similar questions should be investigated for other languages (e.g. Fortran, Cobol) when producers for those languages become available.

The reader is referred to the paper “GANDF: A GCC-based ANDF Translator” for further background information on GANDF.

2. Status

GANDF is still under development. Currently it supports 75 of the 87 ANDF expression operators. This subset is sufficient to support ANSI C. Implementation of this subset was completed recently and there still are a few bugs left. Here is a summary of work still remaining to complete the GANDF project:

- Implement Pointers to bitstrings
- Support Debugging (-g)
- Implement local label constructs
- Support error treatments
- Implement long_jump.
- Implement “visible” attribute of variables, environments, and variable access using environments
- Implement local allocation constructs
- Implement “round_as_state” argument of the round construct
- Our GANDF work thus far revealed one feature of ANDF that could not be implemented while conforming to the C ABIs on the platforms that GANDF is presently targeted to. That is the feature of ANDF which allowed a single

procedure to return more than one type of result, depending on which return statement was executed. This feature, which is not needed to implement C or any other language we know of, has been removed from the latest ANDF draft (not yet published).

- Recognize when a procedure has a variable number of arguments. Currently we always assume this is the case, but that assumption requires a more costly entry and exit sequence than is necessary in the common case.
- Put out GCC loop notes. Loop notes are marks in GCC’s RTL code that let it know where the beginning and ending of the loop are. The thing that prevents doing this from being straightforward is that the not all of the code in the body of an ANDF repeat operator is necessarily really in the loop. For example, the body of a repeat construct could include a branch of a labelled construct which does some calculation and then exits the loop. This problems should probably be solved by breaking the ANDF code into basic blocks first and detecting the presence of loops. Currently, because of the absence of these marks, GANDF is not doing any loop optimizations.
- We can’t think of any real reasons why GANDF code should be any worse than GCC code. After the obvious impediments to good code (mentioned above) are removed, if GANDF still produces poorer code than GCC, then we will have to study why. One possibility is that there are “implicit contracts” between the GCC front end and back end, or between the DRA producers and installers.An example of this would be an installer which only optimized certain ANDF patterns and an associated producer which made sure to produce those patterns. If the installer then was used to install ANDF from another producer which did not take pains to produce those patterns, the code produced might be poor. Alternately, if an installer did not know what special patters were being produced by an producer, it might not be able to optimize as well as one that did. One solution to this is to make the implicit contracts more explicit. That is what things like GCC’s loop notes are. Perhaps ANDF will also need some way of adding annotations (the token mechanism would be one possibility). Or maybe ANDF might need to be enhanced in some other ways to avoid the performance penalty.

GANDF has been configured and tested on these seven platforms:

- HP PA-RISC 1.1/HPUX
- IBM RS-6000/AIX
- DEC ALPHA/OSF1
- Intel 386/OSF1

- Intel 386/SCO 3.2.1
- MIPS/Ultrix
- SPARC/SUNOS

As an example of how easy it is to configure GANDF for a new target (that is supported by GCC), Andy Johnson got GANDF working on three of the above platforms in a single week.

GANDF Source Size Statistics

To give some idea of the size of the components of GANDF, here are the number of lines of source code, including comments, for them:

- GCC component: 432k lines, of which 285k lines are target independent code and 147k lines are target dependent (machine descriptions or configuration files). This includes only the part of GCC that GANDF is actually using, and does not include files automatically generated from the machine descriptions.
- DRA component: 24k lines. This is just the part used by GANDF.
- OSF component: 6k lines. When GANDF is completed this is likely to be more like 8k lines.

A complete installer will also require a token binder. The current token binder, tld, is about 7k lines of source.

Effort Expended So Far

The GANDF project was started August 18, 1992 so it has been about 7 months since it was started. However, GANDF has not been a full-time task during that time period (my guess is about half time). The following figures are an estimate. More precise figures will be available at a later time. About 600 total hours have been spent. Estimated percentages of time spent are as follows:

- 30% studying GCC, validating GCC revisions by bootstrapping it on various platforms, building GAS and GDB.
- 10% studying DRA installers

- 5% building the DRA tools on various platforms, including creating the target-dependent token definition libraries
- 5% studying the instruction set architectures of target machines.
- 50% actual time spent designing, coding and testing the OSF component of GANDF.

As can be seen, only about 300 hours of time was spent on the actual designing and coding of the OSF part of GANDF. Much of the time was spent getting familiar with the target back-end technology. If someone already familiar with a compiler back-end technology were to take GANDF and adapt it to their back end, it should take less than this amount of time, at least to get to an equivalent stage of development (GANDF is not finished yet).

GANDF Test Results

GANDF was tested using these test programs:

- As a preliminary to more complete testing, I used the Plum-Hall Sampler tests. This is a collection of 79 simple tests of a variety of ANSI C features. I used passing of the sampler as a prerequisite to attempting to run the following tests.
- Plum-Hall Version 3 ANSI C conformance suite. We did not run the library tests because many of the systems do not support the full ANSI C library.
- Benchmarks: We use the integer SPEC benchmarks (gcc, espresso, li, and eqntott) and the dhrystone benchmark.
- The XFIG Drawing Program
- The Informix WINGZ spreadsheet.

One point that should be made regarding these results is that all of the versions of GCC that have been used are advanced experimental versions, not production versions. Thus it should come as no surprise that these versions are not completely stable.

The ports to the Alpha and MIPS platforms are still in a preliminary stage and although they are passing a significant number of individual test points, they have not yet been able to execute these tests without failing some test points.

The problems with the Alpha port are due to the immaturity of the GCC port to the Alpha. We have used two versions of GCC for the Alpha port. Using GCC 2.3.1 the Alpha was able to pass 71 of the 79 Plum Hall Sampler tests. One problem with GCC 2.3.1 was that its method of handling variable numbers of arguments was not consistent with the native compiler, so calls to library routines with va_list arguments (like vsprintf) do not work. I then upgraded to GCC 2.3.3+-alpha, a version of GCC 2.3.3 which has additional fixes for the alpha. This version of GCC does handle variable numbers of arguments in a compatible way. Unfortunately, this version of GCC requires Version 1.2 of the Alpha OSF/1 system software which we have not received. Thus this port is currently on hold.

We haven't really investigated the problems on the MIPS yet, but it appears that it is probably a simple configuration problem, because some of the tests are giving unaligned address traps.

The following test results apply to the remaining platforms (other than the Alpha and the MIPS).

GANDF has passed the Plum-Hall Version 3 ANSI C conformance suite, without optimization, on all of remaining platforms. When optimization is enabled the remaining platforms all still pass the Plum Hall tests except for the RS/6000 which fails one of the expression tests (this is not due to a problem in the ANDF to GCC translation, but the exposing of a latent GCC back-end bug).

The Benchmark programs have successfully run on all of the platforms that can run the Plum Hall tests successfully, except that on the RS/6000 a few workarounds are necessary to get the gcc benchmark to run, and on the SPARC even with the workarounds the gcc benchmark is not giving correct results. On the platforms where the Benchmarks ran successfully with the optimizer enabled, the GANDF code is running about 20% slower than the GCC code (using the same GCC on which GANDF is based). We believe that the reason that GANDF code is slower than GCC is largely due to the factors alluded to above, namely always assuming a variable number of arguments, and the lack of loop notes. However, we will have to wait until we have solved those problems before we can say for sure.

The XFIG Drawing program has successfully executed, without optimizations, on all of the platforms except the Alpha (which we did not attempt). When the optimizer is turned on, only the SPARC and Intel 386/OSF1 platforms are still able to successfully execute XFIG.

The WINGZ spreadsheet is only executing successfully on the two i386 platforms. On the others WINGZ is able to start up (usually), but does not execute correctly. Clearly GANDF still has a few bugs that need fixing. However, we do not know whether these bugs are in GANDF proper (i.e. the OSF component), or in either the DRA or GCC components. Since our WINGZ source license is only on a single platform, we cannot try building it with GCC on the other platforms.

3. DESIGN

Overall Structure

GANDF reads the ANDF form of a program (or program fragment) and produces the assembly language form of that program. GANDF is comprised of the DRA component, the GCC component, and the OSF component. GANDF does its work in the following (logical) phases. We mark each phase with the component responsible for that phase.

- Top level control and option processing (GCC)
- While Decoding ANDF, also do constant folding, expansion of tokens, expansion of conditional compilation constructs (e.g. exp_cond), and other optimizations to be done while reading the ANDF. The result of this phase is an internal representation of the ANDF for the program. (DRA)
- Translate the internal form of ANDF into the GCC abstract syntax tree form. (OSF)
- Translate the GCC abstract syntax tree form. into the GCC RTL (register transfer list) form. (GCC)
- Optimize the GCC RTL form, then convert it into assembly language. (GCC)

There currently is no publicly available documentation of the DRA component, but it is mostly straightforward once one understands the internal representation it uses for ANDF. DRA does have documents describing their installers. The DRA component of GANDF is a target-independent subset of the DRA code for their installers.

The GCC component uses all of the parts of the GNU C compiler except the C lexer and parser. The internals of GCC are described in the documentation the accompanies GCC. GANDF is currently using variants of GCC, version 2.3.3 [we are using variants in order to get some of the latest work done by the University of Utah for the PA-RISC, and by Richard Kenner of NYU for the DEC Alpha].

This rest of this section will concentrate on the design of the OSF component.

ANDF Internal Representation

GANDF uses the internal representation for ANDF that is produced by the DRA component. This mostly consists of expression nodes (the C typedef “exp” is a pointer to such a node), but there are also declaration nodes (typedef “dec” points to such) which are only used to provide extra information (e.g. external name, if any) for top-level expressions. There is a declaration node for each top-level tag that is either declared or defined.

The ANDF declaration nodes of the top level tags form a linked list whose head is pointed to by variable “top_def”. Each such declaration has a pointer to an exp node which forms the root of the exp tree representing the definition or declaration. Thus one traverses the ANDF internal representation by traversing the list of declaration nodes and for each calling a recursive routine to walk the corresponding ANDF expression tree.

Each ANDF exp node has a code (called a “tag” internally, and not to be confused with ANDF TAGs) that is used to tell what ANDF operator it represents. Most fields of an exp node are accessed via a C macro. For example, name(e) will get the code (tag) telling what kind of expression node e is. There is a mostly one-to-one correspondence between these codes and the ANDF expression constructs. However, some codes represent more than one kind of ANDF construct. For example, the ident_tag code is used to represent the variable and the identify constructs, as well as the formal parameters of a procedure. Such an exp node has an “is_var” bit which will be true for variable constructs or parameters, and an “is_param” bit which will be true for formal parameters. On the other hand, there are some codes that represent some special cases of a more general ANDF operator. Handling these as special cases makes it easier to produce good code. For example, reff_tag is a special case of the add_ptr construct when the offset argument is a constant.

“Son” and “brother” fields of exp nodes are used to form exp trees. son(f) is the first son of exp node f, bro(son(f)) is f’s second son, and so on. There also is a “last” field. last(s) is true if s is the last son of its father. In that case, bro(s) points back up to the father rather than really being a brother field (this had me really confused until I figured it out). Sharing the same field for both brother and father helps to keep the representation more compact.

In some cases there are nodes, e, such that name(e) is 0, rather than having a normal code. These are used to represent certain ANDF EXP lists, for example, the “statements” argument of the sequence construct is such a node and its children are the individual statements.

There is one case where the son field does not really represent a son, namely for the obtain_tag construct (internal code of name_tag). In this case son(e) points to the exp representing the introduction of the tag. Thus when recursively traversing an ANDF exp tree one must be careful not to recurse on the operand of the obtain_tag construct.

Not all ANDF operands are represented using pointers to other exp nodes. Some operands are guaranteed to be constants at install time so they are represented as integers within the exp node. In fact, most of the fields of exp nodes are declared as unions so that they can hold either pointers of some sort (exp or dec or to a character string or to diagnostic information), or integers or floating point numbers. Macros are used to make accessing a field as a particular type convenient.

One important field in each exp node tells its shape. Actually ANDF shapes are also represented by exp nodes, but they have their own independent set of “shape codes” so one must know when looking at an exp node whether it represents a shape or a “normal” expression. In the external ANDF form most EXP nodes do not have their shape specified explicitly. The shape field in the internal form is produced as part of the “static semantic analysis” that is done by the DRA component code as the ANDF is being decoded. The shape of an exp node is deduced from the shape of its operands and, in some cases, from an explicit shape operand that tells what shape it should have (e.g. the change_variety construct).

GCC Abstract Tree Representation

The output of the OSF component of GANDF is the GCC abstract tree representation (also called expression trees, or just trees for short). Alternately, GANDF could have chosen to produce the GCC RTL representation directly. However, going to the TREEs has the advantage that it is target independent (mostly) and simpler than the RTL. For full details of the GCC TREE representation, the sources of GCC should be consulted, in particular the files tree.def and tree.h. I will here just give a brief description of its main features. The GCC representation actually includes more than tree nodes (e.g. the binding level stacks), but I will ignore most of those other details in this paper.

The tree structure is a collection of tree nodes which are linked together by means of pointer fields in the nodes. Each node is a variant structure. Sometimes the fields of the node structures have more than one use, depending on the setting of other fields. Generally C macros are used to access information in the tree nodes, and if a component of a node is used in more than one way, there will usually be a distinct C macro for each. For example TREE_CODE(e) is the expression used to reference the field which tells which kind of tree node is pointed to by e. There are 122 different tree codes in GCC 2.3.3 The tree codes are classified into one of 11 tree code types. A character is used to denote each tree code type. I’ll now describe the tree code types and list their tree codes. In most cases the meaning of a tree code is evident from its name.

- “d”, declarations: FUNCTION_DECL, LABEL_DECL, CONST_DECL,TYPE_DECL, VAR_DECL, PARM_DECL,RESULT_DECL, FIELD_DECL. The declaration nodes are used not only to represent the information in a declaration, but also to represent a reference to the declared item. Thus the operands of the PLUS_EXPR node for “I+J” would be pointers to the VAR_DECL nodes representing the declarations of I and J.
- “t”, types: VOID_TYPE, INTEGER_TYPE, REAL_TYPE, COMPLEX_TYPE, ENUMERAL_TYPE, BOOLEAN_TYPE, CHAR_TYPE, POINTER_TYPE, OFFSET_TYPE, REFERENCE_TYPE, METHOD_TYPE, FILE_TYPE, ARRAY_TYPE, SET_TYPE, STRING_TYPE, RECORD_TYPE, UNION_TYPE, FUNCTION_TYPE, LANG_TYPE.

- “c”, constants: INTEGER_CST, REAL_CST, COMPLEX_CST, STRING_CST.
- “r”, memory references: COMPONENT_REF, BIT_FIELD_REF, INDIRECT_REF, OFFSET_REF, BUFFER_REF, ARRAY_REF.
- “1”, unary expression operators: FIX_TRUNC_EXPR, FIX_CEIL_EXPR, FIX_FLOOR_EXPR, FIX_ROUND_EXPR, FLOAT_EXPR, NEGATE_EXPR, ABS_EXPR, FFS_EXPR, BIT_NOT_EXPR, CARD_EXPR, CONVERT_EXPR, NOP_EXPR, NON_LVALUE_EXPR, CONJ_EXPR, REALPART_EXPR, IMAGPART_EXPR
- “2”, binary expression operators:PLUS_EXPR, MINUS_EXPR, MULT_EXPR, TRUNC_DIV_EXPR, CEIL_DIV_EXPR, FLOOR_DIV_EXPR, ROUND_DIV_EXPR, TRUNC_MOD_EXPR, CEIL_MOD_EXPR, FLOOR_MOD_EXPR, ROUND_MOD_EXPR, RDIV_EXPR, EXACT_DIV_EXPR, EXPON_EXPR, MIN_EXPR, MAX_EXPR, LSHIFT_EXPR, RSHIFT_EXPR, LROTATE_EXPR, RROTATE_EXPR, BIT_IOR_EXPR, BIT_XOR_EXPR, BIT_AND_EXPR, BIT_ANDTC_EXPR, TRUTH_AND_EXPR, TRUTH_OR_EXPR, IN_EXPR, RANGE_EXPR, COMPLEX_EXPR
- “<” comparison operators: LT_EXPR, LE_EXPR, GT_EXPR, GE_EXPR, EQ_EXPR, NE_EXPR, SET_LE_EXPR
- “e”, other expressions: CONSTRUCTOR, COMPOUND_EXPR, MODIFY_EXPR, INIT_EXPR, TARGET_EXPR, COND_EXPR, BIND_EXPR, CALL_EXPR, METHOD_CALL_EXPR, WITH_CLEANUP_EXPR, TRUTH_ANDIF_EXPR, TRUTH_ORIF_EXPR, TRUTH_NOT_EXPR, SAVE_EXPR, RTL_EXPR, ADDR_EXPR, REFERENCE_EXPR, ENTRY_VALUE_EXPR, PREDECREMENT_EXPR, PREINCREMENT_EXPR, POSTDECREMENT_EXPR, POSTINCREMENT_EXPR
- “s”, expressions with side effects: LABEL_EXPR, GOTO_EXPR, RETURN_EXPR, EXIT_EXPR, LOOP_EXPR
- “b”, blocks: BLOCK
- “x”, codes fitting no category: ERROR_MARK, IDENTIFIER_NODE, OP_IDENTIFIER, TREE_LIST, TREE_VEC

OSF Component Phases

The OSF component itself operators in three phases:

- An information collection pass. This pass is not yet implemented.
- An edit-declare pass. This pass would do ANDF to ANDF transformations (it doesn't do any of these yet) and makes GCC tree declarations for the top level tags (i.e. variables and procedures) so that they will be declared when needed in the third phase.
- A translation phase which does that actual translation from ANDF to GCC trees.

Currently the edit-declare phase and the translate phase are both implemented by a single procedure, andf_to_tree, which analyzes an ANDF expression and returns a representation of it in terms of GCC trees. During the edit-declare phase, andf_to_tree does not recurse and ignores variable initializations, thus it only does declaration processing. Then during the translation phase it fully recurses, thus producing the GCC tree representations of the executable code and variable initializations.

Translation of Shapes

A basic part of translating an ANDF expression is finding the GCC data type that corresponds to the expression's ANDF shape. This translation is performed by routine andf_shape_to_tree. Every ANDF exp node (whether a shape or "normal") has two fields that can be used to save information about the translation of the node. Thus a node only needs to be translated once. If it is referenced several times, its first translation will be reused on subsequent times. For shapes, the saved translation information is a pointer to the GCC tree for the type representing the shape.

Here are the GCC types used to represent ANDF shapes:

- bottom and top are represented by void_type_node, corresponding to the C type void.
- GCC has signed and unsigned integer types with sizes powers of 2 up to the largest supported size. GCC supports long long, so on 32 bit platforms it supports up to 64 bit integers and on 64 bit platforms, up to 128 bit integers. The ANDF integer variety shapes map to the smallest of these that will hold its range of values.
- GCC supports float, double and long double types (the latter two are not necessarily distinct). These are used to represent the ANDF floating varieties.

- ANDF bit fields usually cannot be represented as a simple GCC type (see explanation later), but under some circumstances they are represented using GCC structures.
- The ANDF proc shape is represented using a function type returning void without any information about it arguments. This is necessary because ANDF does not provide any of that information with its proc shape. Later, variables of shape proc (corresponding to C pointers to functions) must be cast to function types with the correct return type and arguments, in order to be processed correctly by the GCC back end.
- All ANDF pointer shapes(except pointers to bitstrings which are not yet implemented) are represented as pointers to char. This is necessary so that pointer arithmetic will be done correctly, since the ANDF offsets already take into account the size of the items that the pointer points to.
- Offsets are represented as signed integers of the same size as pointers. It is necessary that they be signed since ANDF offsets can be negative. Currently bit-resolution offsets are not implemented (except for constant bit offsets which are handled specially, see below), only byte-resolution offsets. One issue that will need to be resolved for bit-resolution offsets is whether to use double-sized offsets for them or just to use the same size. In the former case, bit resolution offsets could cover the whole address space, whereas in the latter case only 1/16 of the space could be covered (1/2 because they are signed and 1/8 because 3 bits are needed to give bit within the byte).
- In ANDF, sizes are represented by offsets. However, internally in the DRA representation there is a separate “size” shape. I don’t recall where it came from, but I represent it using an unsigned integer represented in a native “word”.
- ANDF has a “compound” shape which is used to represent fixed-sized aggregates such as structures and unions. However, unlike languages like C, ANDF does not tell what fields are in the “compound”, only its size and by implication its alignment. Since we don’t have information about the components, one might think that these should be represented as arrays of bytes. However, owing to its C heritage, GCC does not treat arrays quite as first class citizens. For example, it does not allow arrays as function return types. For this reason, we represent compound shapes using C structures which have a single component which is a byte array of the appropriate size. You might think that there would be difficulty later when it is necessary to select a component from such a structure (these arise from explicit pointer arithmetic in ANDF). How that problem is solved is discussed later.

- The ANDF NOF shapes are not represented as arrays, as one might expect, because of GCC’s deficiencies in handling arrays (as mentioned above). Instead structures are used for these also. An additional problem I encountered when I initially tried to use arrays was that the DRA code was digesting the NOF shapes so much (it reduces an NOF to its alignment and size) that is was not possible to come up with appropriate array types. This is a problem that would not have arisen if an internal representation for shapes had been chosen that more closely mirrored the original ANDF.
- The local label shape is not currently supported. However, GCC allows label variables, so when these are supported I will use whatever type GCC uses for its label variables.

Translation of ANDF expressions

The andf_to_tree function is the main recursive procedure for translating ANDF exp nodes into GCC trees. Actually, andf_to_tree’s return type, gandf_trans, is more complex than just a GCC tree node. The reason for this is that not all ANDF nodes can efficiently be represented as simple GCC trees. gandf_trans is a typedef that is a pointer to a gandf_trans_node structure. These structures are dynamically allocated, as needed, to represent the results of translating an ANDF expression. The gandf_trans type has the following purposes:

- In the simplest case, its base_tree component will point to the GCC tree representing the translation of the expression.
- For pointer (address) expressions, it can hold the base address, run-time offsets (with a constant factor) and a constant bit offset as separate components of the translated value. This allows constant parts of addresses to be combined at compile time instead of at run-time.
- To handle offset expressions efficiently. As with pointer expression, offset expressions can be handled more efficiently by constant on non-constant components to the offsets.
- To handle bitstring results, which also need to have information about the wide of the bitstring and whether it is signed or not. Also ANDF bitstrings can result either from a memory reference or by converting an integer expression to a bitstring. In both of these cases, the GCC code cannot be produced until it is seen what context the bitstring will be used in. Thus the

> gandf_trans result for a bitstring just captures the information about how the bitstring was created and it is then processed further when the bitstring is used.

Time does not permit giving the complete details of the translation of the ANDF operators, at this time, so the remainder of this section will just touch on the high-lights.

The array andf_tree_code is used to map ANDF codes to GCC tree codes for the 38 ANDF operators which have a straightforward translation into GCC codes (this are mostly unary and binary expressions).

A utility routine, andf_proc_to_tree, is used to set up a GCC function definition tree.

ANDF uses explicit pointer arithmetic to get the address of array elements and structure components, then uses the contents construct to reference the data item at that address. When such a pointer value is at an variable offset from the base address, GANDF uses explicit pointer arithmetic and the INDIRECT_REF (like C's * operator) operator to reference the data. When the offset relative to the base address is constant (such as would be the case for a component of a C structure), GANDF uses a COMPONENT_REF. One problem that had to be faced here was whether it was going to be necessary to create dummy structure types with components at the appropriate offsets in order to make use of the COMPONENT_REF operator. As it turned out, we do need to have a field declared with the appropriate type and offset, but it is not necessary that it be declared in the structure type you are accessing. Thus GANDF uses a hash table to keep track of fields that it has declared with each type and bit offset (and width, if it was a bitstring). Another place where there might have been a problem was in initializing such structures. As it turned out, even then it was not necessary to create full structures with fields at the proper offsets. GCC has a facility where initial values of structure can be labeled with the component that they are to initialize. GANDF was able to make constructors with such explicit labels. In that case, GCC does not bother to check that the fields being selected are actually part of the structure being initialized. However, it might be that some other compiler back ends will be more rigid than GCC. In that case, it might be necessary to do a more thorough analysis of ANDF memory reference patterns, in order to try to recover some of the data structure information that is lost in ANDF.

As alluded to above, pointer, offset, and bitstring constructs do not return a single GCC tree result, but rather a structure used to accumulate (at install time) such expressions. When it is necessary to reduce such a result to a single GCC tree expression, the trans_tval routine is used to combine the various components into a single tree.

For the conditional construct, the result is either first expression, if it terminates normally, or the alt expression otherwise. When the shape being returned is TOP (i.e. actually no meaningful result), this can be implemented using flow of control operators. However, when a meaningful result is being returned, it is necessary that a temporary variable of the appropriate shape be created and passed to the subsidiary expressions. When andf_to_tree is called with such a suggested target, it assigns the result of translating the expression into the target. Similar techniques are used to translate the repeat, labelled, and sequence constructs when they return a meaningful result.

The work that andf_to_tree does is factored into two parts. First there is a switch statement that is used to preprocess the arguments to the expression according to one of these schemes:

- All of the arguments are translated (by recursive calls to andf_to_tree) and the results of their translation saved in handy local variables (t1, t2, etc.) while the untranslated arguments are saved in local variables (e1, e2, etc.). This scheme e is used for things like binary operators which always must have their arguments evaluated before they can be evaluated.
- The untranslated arguments are saved in local variables (e1, e2, etc.), but the arguments are not translated yet, because they must be translated in a specific order. This scheme is used for flow of control constructs like sequence, conditional, and repeat, as well as some other others that need special processing.
- The obtain_tag construct is handled specially. I mentioned before that each exp node could save up to two translated results. Ordinary expressions save only one, but declarative constructs like identify and variable need to save two results: the tree node of the declaration of the tag that they introduced, and the result of translating the body of the construct. In the case of obtain_tag, its operand is a pointer to the variable or identify construct that introduced the tag being referenced. This scheme will fetch the declaration translation into local variable t1.

After the above switch statement has done some standard processing of the operands of the expression, there is another switch statement which completes the processing for each expression.

The operations on pointer, offset, or bitstring operands are characterized by sometimes lengthy case analysis on the translations of their operands (e.g. does operand 1 have a constant bit offset, does operand 2 have a run-time offset, etc.)

For a number of ANDF constructs that return aggregate results (e.g. concat_nof), the GCC CONSTRUCTOR operator was indispensable. This operator allows one to build a structure value in which the components have the specified expressions as values. Any back end wishing to translate ANDF would do well to make sure it has such a powerful building block to use.

One unexpected thing I discovered was that a number of the operations which are binary in the external ANDF are extended in the internal representation to be n-ary. My initial implementation of these only looked at the first two operands. In the cases where the DRA decoding code had made some of these n-ary, I was losing part of the computation. Of course it was relatively easy to handle these as n-ary once I knew that it was necessary.

DRA suggests that the make_value construct not result in any code. However, I found difficulty in doing this because the make_value was used in contexts where some value for it was needed (e.g. it was the definition of an identify whose tag is referenced). Perhaps a better analysis of the ANDF could eliminate such references to undefined values, but for now I’m supplying a zero value for make_value.

For the identify construct, I use GCC’s save_expr tree code. This operation is such that even if there are multiple reference to the tree node, the expression will only be evaluated once. Subsequent references will get the value from the first evaluation (which will have been saved somewhere).

4. Conclusions

The GANDF experiment is not yet complete, but our results so far are encouraging. Final conclusions must await the completion of the experiment, but we believe the results so far show that interfacing ANDF to an existing compiler back end is practical and relatively easy to do, and that the result will probably produce code with quality comparable to that of a native compiler based on the same back end. In addition, for a retargetable back end like GCC, the additional effort to get additional ANDF installer targets is quite small.

For further information please contact:

Richard Ford
richford@osf.org
(617) 621-7392

Copyright 1993 by Open Software Foundation, Inc.

All Rights Reserved

Permission to reproduce this document without fee is hereby granted, provided that the copyright notice and this permission notice appear in all copies or derivative works. OSF MAKES NO WARRRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. OSF shall not be liable for errors contained herein or for any direct or indirect, incidental, special or consequential damages in connection with the furnishing, performance, or use of this material.