ANDF: Finally an UNCOL after 30 Years

Jack Davidson, Manuel E. Benitez, et al

Computer Science Report No. TR-91-05
March 6, 1991

ANDF: Finally an UNCOL after 30 Years (Extended Abstract)

Manuel E. Benitez†, Paul Chan‡, Jack W. Davidson†, Anne M. Holler†,
Sue Meloy‡, Vatsa Santhanam‡

‡Hewlett-Packard Company
California Language Laboratory
19447 Pruneridge Avenue
Cupertino, California 95014

†University of Virginia
Department of Computer Science
Charlottesville, VA 22903

Abstract

In the late 1950's it was proposed that a Universal Computer Oriented Language (UNCOL) be developed to facilitate the development of language processors for various architectures. While an UNCOL was never realized, the use of some type of intermediate language for supporting the construction of compilers has found widespread use. Popular examples include P-code which is used to support Pascal, U-code, a descendant of P-code, which has been used to support several languages, OCODE which was used as the intermediate language for BCPL, and EM-1 which is used in the Amsterdam Compiler Kit and also supports several languages. These are only a few of the more well-known and widely used intermediate languages. This paper describes an intermediate language developed in response to the Open Software Foundation's request for the development of an Architecture Neutral Distribution Format (ANDF). The intermediate language, called HPcode-Plus, permits the distribution of a single version of an application that, without modification, will run on any hardware platform. The intermediate language and the accompanying translators demonstrate that an UNCOL is now technologically feasible. Clearly, if accepted in the marketplace, such an intermediate language will have tremendous benefits for end-users.

1. Introduction

The acronym UNCOL (Universal Computer Oriented Language) is well-known to the compiler construction community[AHO86, FISC88, TREM85]. In the late 1950's, UNCOL was proposed as a way to reduce the effort to construct compilers for new languages and new architectures[STEE61, STRO59]. The classic argument was that if there were M languages and N machines, M×N compilers would be required to make each language available on all the machines. The creators of the UNCOL concept noted that only M+N translators would be required if a language could be constructed to serve as a bridge between the languages and the architecture. For each language, a source-language-to-UNCOL translator would be constructed. To implement the language on any machine would simply require the construction of an UNCOL-to-machine-language translator.

Conceptually the approach is quite appealing. In addition to reducing the cost of developing compilers, programs written in a language where a source-language-to-UNCOL translator exists could be immediately moved to any machine for which there was also an UNCOL-to-machine-language translator. This would, of course, include the translators themselves.

Unfortunately, despite the benefits, UNCOL was never realized. There were a number of technological problems that could not be overcome. A major problem was that the UNCOL process could not produce executable

code that was as fast as the executable code produced by a compiler that was designed specifically for the target architecture. Consequently, applications produced using UNCOL translators would run much slower than those produced using a conventional compiler. The primary reason for this loss of performance was existing code generation and optimization technologies were not able to efficiently map a language-independent, architecture-independent intermediate language onto the range of architectures available.

Another problem was the inability to design an intermediate language and construct the accompanying source-language-to-UNCOL translators that avoided assumptions about the target architecture. Typical source-language translators are written with knowledge of various key characteristics of the target architecture. For example, most source-language translators or front ends are written with knowledge of the sizes and alignment requirements of the basic data types supported by the target architecture. Such information permits the front end to compute sizes of records and structures, determine offsets of variables, properly initialize locations in memory, and in some cases decide the most appropriate operations to use. In order to minimize the effort to move these translators to different architectures, such information is usually isolated and parameterized so that it is easy to change. Nonetheless, this information as well as other information about the target architecture is used and its ramifications appear in the intermediate language the front end produces.

This paper describes the design of an intermediate language and accompanying translators that addresses the problem UNCOL attempted to solve 30 years ago. The intermediate language was developed in response to the Open Software Foundation's Request for Technology to produce an Architecture Neutral Distribution Format (ANDF)[OSF90]. The intermediate language, called HPcode-Plus, contains no architecture dependencies. Application programs compiled into HPcode-Plus can be moved to any architecture that has a HPcode-Plus-to-machine-code translator. To demonstrate the feasibility of the process, we have constructed a front end that translates ANSI C programs to HPcode-Plus as well as translators for three machines (Motorola 68020, Hewlett-Packard PA-RISC, and Intel 80386/80387) that translate HPcode-Plus programs to machine code. Many applications totaling over 500,000 lines of code have been compiled and the resulting HPcode-Plus files have been moved to and installed on the three machines. The paper focuses on the features of the intermediate language that allow architecture dependencies to be avoided.

2. ANDF Rationale, Terminology and Requirements

2.1 Rationale

The widespread availability of applications for personal computers has been enabled because there is only one target architecture, the Intel 80X86, and only one operating system, MS-DOS. Thus software vendors can develop an application for one platform and yet be assured of a large market for their application. Furthermore, the vendor only needs to distribute one type of executable. While the workstation market has standardized on UNIX† there is not one common hardware platform. Thus, a primary goal of ANDF is to bring the advantages of having one target architecture to vendors of applications for workstations.

2.2 Terminology

Figure 1 contains a diagram of the major ANDF components. The Producer is essentially a compiler front end

† We will ignore the fact that all Unixes are not the same.

that translates the source code to ANDF. For each target machine, an Installer is responsible for translating the ANDF files into target machine code and producing an executable version of the application. The installer is similar to a compiler back end.

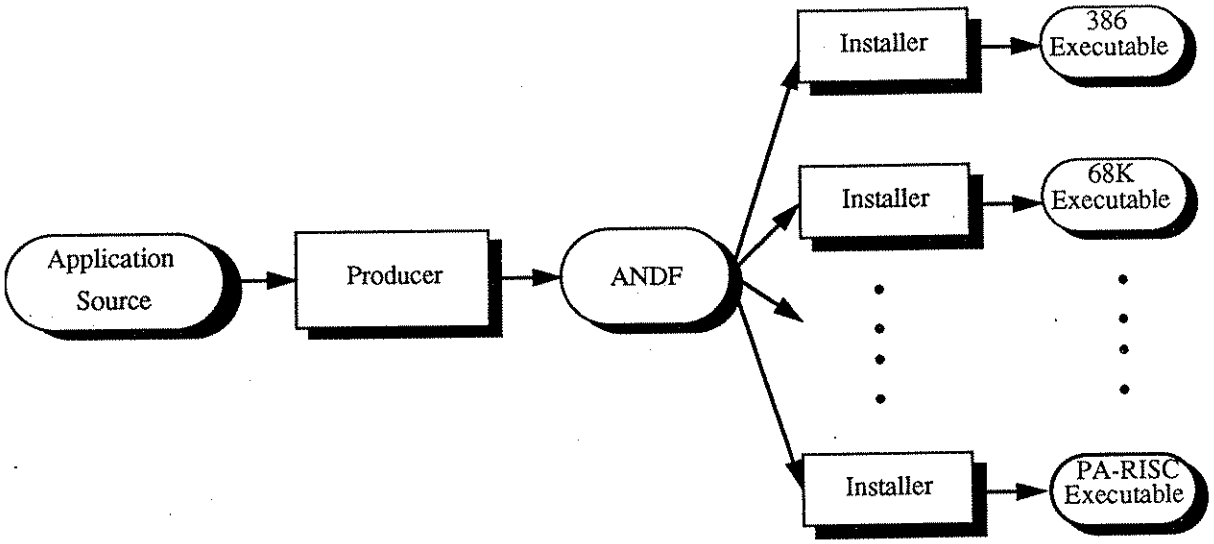

Figure 1. Major ANDF Technology Components

2.3 Requirements

There are a number of requirements that must be met if a successful ANDF is to be realized. Some of these requirements are:

- Architecture neutrality—ANDF must not contain any hidden assumptions about the target architecture.
- Excellent run-time performance—an application produced using ANDF technologies must perform as well as an application produced using the native compiler of the development platform.
- Programming language neutrality—ANDF will initially support ANSI C, but must be capable of being extended to support other languages.
- Consistent application behavior—applications produced using ANDF technologies must exhibit the same behavior on all hardware platforms.
- Reasonable installation times—installing an application should take no more than twice the time it would take to install source code.
- Small file sizes—the distributed files should be reasonably compact, ideally no bigger than an executable.
- Debug support—the ANDF must support the a variety of symbolic debuggers.
- Protection of proprietary information—the ANDF must protect a developer's intellectual property.

Of these requirements, there are two that standout as keys to the technological success of an ANDF†. First, an ANDF must truly be architecture neutral. Independent software vendors must be assured that their applications will exhibit consistent behavior on all platforms. Second, applications must not suffer a performance penalty because they were distributed in ANDF format. The following sections describe how these goals were achieved.

† This paper will ignore the myriad marketing issues that will affect whether ANDF becomes widely accepted and used.

3. HPcode-Plus

3.1 Overview

HPcode-Plus is derived from HPcode, the compiler intermediate language used by Hewlett-Packard in many of its compilers for its 32-bit RISC and 16-bit stack-based architectures. HPcode is descended from the U-code developed at the University of San Diego and Stanford [CHAN90].

HPcode-Plus is essentially an assembly language for a virtual stack machine. The virtual machine provides the following abstractions:

- an expression stack for carrying out most computations.
- a read-only memory area for storing instructions and constants.
- a static storage area for global and local variables.
- an implicit memory stack for pushing and popping stack frames on procedure calls and returns.

Currently there are 67 HPcode-Plus operators. These are sufficient to support ANSI C (including the standard header files). It is expected that a handful of additional operators and data types will be added to support other languages such as Fortran, C++, and Ada. Indeed, HPcode supports Fortran, Ada, Pascal, Cobol, and C++. For the most part, HPcode-Plus resembles the instruction set of a typical stack-based machine. Operators such as add and subtract take their operands from the expression stack and push the result back onto the stack. HPcode-Plus, however, includes a data type system and several special operators that allow programs that are translated to HPcode-Plus to not contain any architecture dependencies. These special features are described in the following sections.

3.2 Data Types

Data objects are declared using the HPcode-Plus SYM operator. Using the SYM instruction, producers may declare data objects of simple or aggregate HPcode-Plus data types. The actual storage allocation is deferred to the installer. HPcode-Plus has the following predefined data types. For the integral data types, a minimum range is defined.

TYPE_BOOLEAN—TRUE/FALSE or LOGICAL
TYPE_UNS_CHAR—character, not sign extended when converted to TYPE_INT ($0 \le$ range ≤ 255)
TYPE_SCHAR—character, sign extended when converted to TYPE_INT ($-127 \le$ range ≤ 127)
TYPE_CHAR—character, installer determines whether signed or unsigned
TYPE_SHORTINT—signed short integer ($-32767 \le$ range ≤ 32767)
TYPE_INT—signed integer ($-32767 \le$ range ≤ 32767)
TYPE_LONGINT—signed long integer ($-2147483647 \le$ range ≤ 2147483647)
TYPE_UNS_SHORTINT—unsigned short integer ($0 \le$ range ≤ 65535)
TYPE_UNS_INT—unsigned integer ($0 \le$ range ≤ 65535)
TYPE_UNS_LONGINT—unsigned long integer ($0 \le$ range ≤ 4294967295)
TYPE_REAL—float real
TYPE_DOUBLE—double real
TYPE_LONGREAL—long double real
TYPE_ANY_PTR—address for unknown type data object
TYPE_VOID—void type for function

In general, no producer may make any assumptions about the format of the data type on the target machine. A

producer can make use of the minimum ranges of values representable by each predefined data type. It is the responsibility of the installer to use the appropriate representation for the target machine. Furthermore it is the responsibility of the installer to insure that the allocated data satisfies the hardware's alignment requirements.

All other data types are constructed using the following special constructor data types:

KIND_POINTER—define pointer type
KIND_ENUM—define enumeration type
KIND_MEMBER—define members of enumeration type
KIND_ARRAY—define array type
KIND_STRUCT—define structure type
KIND_UNION—define union type
KIND_FIELD—define structure or union field
KIND_FUNC_PTR—define pointer to function type
KIND_MODIFIER—define additional attributes such as volatile or const

For example, a pointer to an integer would be defined as:

```
SYM symid KIND_POINTER static type [name]
```

where *symid* is the identifying tag for the new data type, *static is* a flag that indicates whether the symid is freed when the enclosing function is terminated, and *type* is the identifying tag for the type that the pointer points to. The optional *name* of the pointer type is specified when debugging support is requested. When debugging information is not requested, user variable and type names are removed in order to protect proprietary information.

It is the responsibility of the HPcode-Plus installers to map the predefined and synthesized HPcode-Plus data types into the appropriate machine data types. For example, TYPE_INT can correspond to a 16-bit data type on one machine and a 32-bit data type on another. It is also the responsibility of the installers to use a storage allocation scheme that obeys any target architectural requirements such as the alignment of data items in memory. Typically an installer would use an allocation scheme that is consistent with the native compiler. This allows a program generated by an installer to invoke natively compiled code such as library routines.

Consider the following typdef and structure declarations:

```
typedef struct s2 {
    int k;
    double l;
    };
struct s2 var;
```

Typical front ends compute the offset of the fields and emit the appropriate constant offsets in the intermediate language. On a machine with four-byte integers and eight-byte doubles that requires all data items to be aligned on four-byte boundaries, the computed offset of `var.l` would be four. On a machine that requires data items to be aligned on boundaries that correspond to their size, the computed offset of `var.l` would be eight. In order to remain architecture neutral, the computation of the offsets of the fields and the size of the structure must be deferred to install time.

3.3 Special Operations

Just as the producer must defer storage allocation to the installer, it must also defer to the installer certain type conversion decisions as well. This has two effects on the design of HPcode-Plus. First, because maintaining type

information must be the responsibility of the installer, HPcode-Plus operators do not require a type specifier. The data type of an item on the expression stack is initially determined by the instruction that created that item. Subsequent operations on the stack item are tracked by the installer and the type information for the item is updated accordingly. The second effect was that several new operators were required to support ANSI C conversion rules. These operators are described below.

3.3.1 ACVT

This operation performs an arithmetic conversion as defined by ANSI C. It directs the installer to check the data types of the two operands on top of the expression stack and to emit code to perform the necessary type conversions on the operands to prepare them for an arithmetic operation. This operation is required because the producer may not be able to determine the type of data object in an expression: 1) where an operand's type after integral promotion cannot be determined by the producer (see the following example), 2) where an operand is a constant loaded by CLDC, 3) where the type of an operand is defined by a system standard include file, 4) inside a macro body where the operands are macro arguments, and 5) where an operand is the result of a previous ACVT or UCVT instruction.

Consider the following example:

```
unsigned short int i;
short int k;
int j;

j = j + i;
j = j + k;
```

this must be translated as:

```
LOD <symid of j>          ; load j
LOD <symid of i>          ; load i
ACVT                      ; convert operands according to integral promotion rules
ADD                       ; add converted operands
CVT TYPE_INT              ; convert result to integer
STR <symid of j>          ; store result
LOD <symid of j>          ; load j
LOD <symid of k>          ; load k
CVT TYPE_INT              ; convert k to integer
ADD                       ; add
STR <symid of j>          ; store result
```

Note that `i` may be converted to a TYPE_INT or TYPE_UNS_INT depending on whether TYPE_INT can represent all the possible values of TYPE_UNS_SHORTINT according to ANSI C rules. ACVT checks the types of the two operands on the expression stack and performs the necessary conversions required by the language definition. In this case, on a machine where a value of TYPE_UNS_SHORTINT cannot be represented by TYPE_INT, ACVT will promote `i` to TYPE_UNS_INT and `j` to TYPE_UNS_INT. For the expression 'j+k', the producer can determine that k should be converted to TYPE_INT, and thus an explicit conversion to integer is generated.

3.3.2 UCVT *parcvt*

This operation takes one argument *parcvt* which indicates whether to apply ANSI C integral promotion rules to the operand, or whether to apply ANSI C default argument promotion rules to the operand.When *parcvt* is 0, the operation performs integral promotion of the value on top of the stack. The result type depends on the data type of the operand and its range relative to that of TYPE_INT. The following table describes the conversions possible.

Original Type	Range	Result Type
TYPE_UNS_CHAR,TYPE_SCHAR, TYPE_CHAR, TYPE_SHORTINT, TYPE_UNS_SHORTINT, or a KIND_ENUM, or a bit-field KIND_FIELD, or a KIND_MODIFIER whose base-type is one of the aforementioned data types.	TYPE_INT object can represent all numeric values of the original type.	TYPE_INT
	TYPE_INT object cannot represent all numeric values of the original type.	TYPE_UNS_INT
Other data types		Same as the Original Type

When *parcvt* is 1, UCVT performs default argument promotion on the operand. The semantics of the operation is identical to the case when *parcvt* is 0 when the data type of the operand is anything other than TYPE_FLOAT. When *parcvt* is 1 and the data type of the operand is TYPE_FLOAT, the operand is converted to TYPE_DOUBLE.

To illustrate the use of the operation, consider the following C code:

```
unsigned short i;
unsigned short k;
int j;

foo(i);
j = ~k;
```

This example yields the following HPcode-Plus instructions:

```
LOD <symid of i>          ; load i
UCVT 1                    ; apply default argument promotion rules
PAR                       ; pass parameter to function
CUP <symid of foo>        ; call foo
LOD <symid of k>          ; load k
UCVT 0                    ; apply integral promotion rules to k
NOT                       ; apply bitwise not
CVT TYPE_INT              ; convert to integer
STR <symid of j>          ; store value into j
```

Note that `i` may be converted to a TYPE_INT or TYPE_UNS_INT depending on whether TYPE_INT can represent all possible values of TYPE_UNS_SHORTINT. Similarly, `k` may be converted to TYPE_INT or TYPE_UNS_INT.

3.3.3 CLDC *flag constant*

This instruction pushes a simple integer constant onto the expression stack. The type of the constant is determined by the installer based on the *flag* parameter. As per the ANSI C specification, the type of an integer constant is the first type from the following corresponding list in which its value can be represented:

Flag	Type of Constant	List of Types (ordered by priority)
0	Unsuffixed decimal	TYPE_INT, TYPE_LONGINT, TYPE_UNS_LONGINT
1	Unsuffixed octal or hexadecimal	TYPE_INT, TYPE_UNS_INT, TYPE_LONGINT, TYPE_UNS_LONGINT
2	Suffixed by u or U	TYPE_UNS_INT, TYPE_UNS_LONGINT
3	Suffixed by l or L	TYPE_LONGINT, TYPE_UNS_LONGINT
4	Suffixed by u or U and l or L	TYPE_UNS_LONGINT

Note that since the result type is unknown to the producer, the result cannot be used immediately for an arithmetic operation. The ACVT or UCVT instruction should be used if an arithmetic operation is to be performed. The following

example illustrates the necessity and use of the CLDC operation.

```
int j;

... j + 40000 ...
... j + 32000 ...
```

would result in the following HPcode-Plus operations:

```
LOD <symid of j>          ; load j
CLDC 0 40000              ; load constant and convert according to ANSI C rules
ACVT                      ; do arithmetic conversion
ADD                       ; add the operands
LOD <symid of j>          ; load j
LDC TYPE_INT 32000        ; load constant 3200
ADD
```

Note that since 32000 is within the minimum range of TYPE_INT, CLDC and ACVT are not necessary. In the example, the 40000 may have a type of TYPE_INT, TYPE_LONGINT, or TYPE_UNS_LONGINT depending on which one on the target machine can first represent the value 40000. This will be determined by the installer. On a 16-bit machine on which TYPE_LONG_INT is the minimum type to represent 40000, the installer will convert the constant to TYPE_LONG_INT. ACVT will promote `j` to the same type before doing the addition.

3.4 Header Files

Supporting the standard header files required by ANSI C and maintaining architecture neutrality proved quite a challenge. Support for the standard headers is achieved by providing HPcode-Plus versions of the headers for both the producer and installers. All machine-specific characteristics are removed from the producer versions of the headers; predefined symids are provided in order for the producer to reference these symbols. The definitions of these symids will be present in the installer versions of the headers. The installer version of the header files can then, if desired, be implemented using HPcode-Plus that contains machine-dependencies. HPcode-Plus includes a powerful macro facility that permits the macros and conditional compilation often found in the standard header files to be reproduced.

Of course it is necessary for the producer to have access to the machine-dependent symbols typically found in the header files in order to process the source code. Pragmas are used to communicate this information to the producer. These pragmas indicate the syntactic category and predefined symid of each symbol, and may include information about the type. The producer must not generate SYM declarations for these machine-dependent symbols; they will be provided in the installer versions of the header files. The producer should simply use the symid provided by the pragma wherever the symbol is referenced.

The following example using `isalnum` illustrates how standard library functions are handled. While ANSI C specifies `isalnum` be a library function, implementations are free to provide macro definitions for efficiency.

Producer's ctype.h (abbreviated):

```
#pragma _OSF_ANDF_FUNC _OSF_ANDF_isalnum -2
extern int isalnum(int)
extern int _OSF_ANDF_isalnum(int);
#define isalnum(__c) _OSF_ANDF_isalnum((__c))
```

User program:

```
#include <ctype.h>
main()
{
  int i = isalnum('a');
}
```

Producer output:

```
OPTN HEADER_FILE "ctype.h"
SYM 257 KIND_FUNCTION TYPE_INT 0 0 "main"
ENT 257
SYM 258 KIND_DVAR TYPE_INT 0 "i"
MST -2                                  ; call isalnum as a function
LDC TYPE_INT "a"                        ; character to test
PAR
CUP -2
STR 258                                 ; store result in i
END 257
SYM 257 KIND_END.
```

For a machine that implements `isalnum` as a macro, the installer implementation of ctype.h is (abbreviated):

```
SYM -275 KIND_POINTER 0 TYPE_UNS_CHAR
SYM -285 KIND_SVAR 0 -275 1 "__CTYPE"     ; declare the mask array
SYM -2 KIND_MACRO
    MINST LOD -285                        ; load pointer to ctype table
    MINST %1                              ; load character to look up
                                          ; installer will have passed
                                          ; (LDC TYPE_INT "a") as the parameter to
                                          ; the macro
    MINST IXE TYPE_UNS_CHAR               ; index into CTYPE table
    MINST ILOD                            ; load the mask at __CTYPE["a"]
    MINST AND                             ; do the test
SYM -2 KIND_END
```

The producer's ctype.h includes a pragma that defines the mapping between the prefixed name and the predefined symid for the symbol. It also includes a normal declaration of the function using the standard name. This is required in order to reference the real function if the user undefines the macro definition or the address of the function is taken. The header also includes a normal declaration of the function using the prefixed name as a special symbol. This declaration provides the parameter and return type information for that symbol. The last line of the producer's version of ctype is a define that maps the standard name to the prefixed name. This must be a function-like macro so that object-like references in the user's code will not be substituted.

When an installer is mapping the HPcode-Plus program to a particular architecture, the call to `isalnum` will be expanded into the code supplied by the macro supplied with installer for that machine. Thus, the resulting code is able to take advantage of any implementation tricks available on the target machine. Standard symbol definitions (such as errno), standard expression definitions (such as DBL_MAX), standard structures (such as div_t found in stdlib.h), and standard types (such as size_t found in stddef.h) are handled similarly to the standard library functions, but some details differ.

3.5 Miscellaneous

In order to avoid architecture dependencies, an HPcode-Plus file is simply a stream of ASCII characters. Instruction opcodes are represented by their numeric values expressed as ASCII characters followed by zero or more

operands. Instruction parameters are separated by white space. Instruction parameters may be integers, quoted strings, floating-point numbers expressed in ANSI C syntax, or by a '%' or '#' followed by an integer representing a macro argument or symid. Every instruction is terminated by an unquoted ASCII new-line character.

4. Results and Summary

To demonstrate the feasibility of our proposed ANDF, the Open Software Foundation required that we construct a Producer that accepts source code written in ANSI C and emits the proposed ANDF as well as Installers for three machines. The three machines should represent a spectrum of current architectures. At our discretion, one machine was to be considered the 'reference' platform. This distinction meant that the implementation of the installer could not be a 'throw-away', rather it had to be near production quality. The other installers were necessary to demonstrate that the proposed ANDF was indeed architecture neutral, thus they could be throw-away implementations. We chose the Intel 80386, the Motorola 68030, and the PA-RISC, with the Motorola 68030 (HP9000 Series 300) being our reference platform. We had approximately four months (June 1990 through September 1990) to complete the construction of these components.

In this abstract we do not have space to describe the interesting details of the implementation of each of these components. However, we describe briefly the results of our efforts and whether they satisfied the requirements outlined in Section 2.3.

Recall that there were two key requirements: architecture neutrality and generation of code that compares favorably to that generated by the native C compiler used for development. Architecture neutrality has been demonstrated by taking many applications compiling them to HPcode-Plus and installing them on the three machines. For example, the producer, *afccom*, and all three installers have been 'produced' on one machine and installed on a different one. The same is true of ANSIfied versions of the C component of the SPEC benchmark suite.

To assay the quality of code generated by the producer and an installer, a set of well-known benchmark programs (including the ANSIfied C component of the SPEC benchmark suite) were used to compare the code generated by the native C compiler and the producer/installer pair for the reference hardware platform. For both compilers, all optimizations (pedal to the metal) were enabled. Averaging over all the benchmark programs, we determined that producer/installer produced code that ran 5% slower than the code produced by the native compilers. Given the short time spent on the implementations, we were encouraged by this result and feel that it proves that high-quality code can be produced from a general, architecture neutral, intermediate language. Indeed, after some preliminary examinations of the generated code, we feel that we can reverse the comparison, and make the producer/installer generated code be, on average, 5% faster than that produced by the native compiler.

Have we demonstrated the feasibility of an UNCOL? We believe so. Will such technology find widespread use? That is more difficult to devine. Certainly there are benefits to both software developers and end-users. Software developers can be assured of a much larger market for their software. Furthermore, development, distribution, and maintenance costs should be lower because with ANDF only one version of an application needs to be developed and maintained and it will run on any hardware platform that has an installer. End-users no longer will be tied to one type of architecture. Applications will be available for a range of architectures. Because of large sales volumes the price of applications should drop. There are problems, however. Determining the correct distribution media, determining how to handle CPU class-based licensing, and how to isolate problems (where's the bug—in the producer or in the installer?) are just a few of the problems that must be resolved.

5. References

[AHO86] Aho, A. V., Sethi, R., and Ullman, J. D., *Compilers Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.

[CHAN90] Chan, P. and Santhanam, V., The Evolution of the U-code Compiler Intermediate Language, *Proceedings of the Summer Usenix Conference.*

[FISC88] Fischer, C. N. and LeBlanc, R. J., *Crafting a Compiler*, The Benjamin Cummings Publishing Co., Menlo Park, CA, 1988.

[OSF90] Architecture Neutral Distribution Format: A White Paper, Open Software Foundation, Cambridge, MA., November 1990.

[STEE61] Steel, T. B., A First Version of Uncol, *Proceedings of the Western Joint Computer Conference*, May 1961, 371-378.

[STRO59] Strong, J., Wegstein, J., Tritter, A., Olsztyn, J., Mock, O., and Steel, T., The Problem of programming communication with changing machines: a proposed solution, *Communications of the ACM* **1**, 8 (August), 12-18.

[TREM85] Tremblay, J. and Sorenson, P. G., *The Theory and Practice of Compiler Writing*, McGraw-Hill Book Company, New York, NY, 1985.