An Apply Compiler for the CAAPP

Michael Scudder
Computer and Information Science Department
University of Massachusetts [1]
Amherst, MA 01003
Phone: (413)545-1519
EMail: scudder@cs.umass.edu

18 July 1990

[1]This work was supported in part by the Defense Advanced Research Projects Agency under contracts DACA76-86-C-0015, and DACA76-89-C-0016, monitored by the U.S. Army Engineer Topographic Laboratory; by the Air Force Office of Scientific Research, under contract F49620-86-C-0041; and by a Coordinated Experimental Research grant from the National Science Foundation (DCR 8500332).

Contents

1 What is Apply

Apply is an application specific programming language for image processing. It is a functional language which describes the operation to be applied to an image by describing the local computation required to produce a single pixel.

1.1 The History of Apply

Apply was developed at Carnegie Mellon University. Apply was originally developed as a tool for writing image processing programs in the C/UNIX based standard vision programming environment at Carnegie Mellon. It now runs on UNIX systems, Warp systems, the Hughes HBA system, the IUA, and others. Apply was developed by Leonard G. C. Hamey, Jon A. Webb, and I-Chen Wu with contributions by Steve Shafer and others [Hamey, Webb, & Wu 1987]. Based on experience from the 2nd Darpa Image Understanding benchmark exercise [Webb & MacPherson 1989], Apply was extended. The extensions make the new version suitable for implementing all the low-level parts of the benchmark. The new version will be called the raster-order Apply and the previous version, original Apply.

A new back end was added to the Apply compiler by Mike Scudder during 1989 as a master's project. This back end produces code for the CAAPP level of the IUA architecture. This implementation does not include the latest extensions to Apply.

1.2 Fundamental Concepts

Apply is intended to be an application specific but machine independent programming language. It provides a way to specify a pixel in an output image or images based on a window around the corresponding pixel in the input image or images. It thus provides for parallelism based on input partitioning.

Each operation is written as a procedure for a single pixel position. The Apply compiler translates this into a program which executes the procedure over the entire set of pixels. In the original Apply no constraint on the order in which the pixels are processed is allowed by the language, allowing the compiler complete freedom in how to partition the pixels among processors.

1.3 The Language Specification

The language is described in [Hamey, Webb, & Wu 1987]. The information in this document supersedes, for the UMass implementation, the information in [Hamey, Webb, & Wu 1987].

1.3.1 Syntax

The syntax of Apply corresponds to the syntax of a subset of Ada, with extensions to the procedure call syntax. This syntax is described in appendix A. Note that the type REAL, as found in this appendix, should be used instead of the type FLOAT found in [Hamey, Webb, & Wu 1987].

Apply is by no means an Ada subset, having keywords not in Ada and different semantics. Nor will it evolve into an Ada subset.

The acceptable syntax is modified by the m4 option, since if the m4 option is used, the *Unix System V m4* macro preprocessor is called and processes the file according to its syntax rules before the Apply compiler's syntax analysis occurs. The option can be specified in the Apply code with a comment of the form:

```
-- Options: -m4
```

as the first line of the file.

1.3.2 Programming Model

When using the Apply language, the programmer writes a procedure which defines the operation to be applied to the current pixel location. The procedure conforms to the following programming model:

1. It accepts a rectangular window from each input image.

2. It performs arbitrary computations without the possibility of side-effects.

3. It returns a pixel value for each output image.

1.3.3 Semantics

Each procedure has a parameter list containing parameters of any of the following types: *in, out,* or *constant.* Input parameters are either scalar variables or two-dimensional arrays. A scalar input variable represents the value of an input image at the current processing coordinates. A two-dimensional array input variable represents a window of an input image. Element (0,0) of the array corresponds to the current processing coordinates. These parameters have an associated border value which is used to fill in windows which extend outside an input image. This border value defaults to zero if not specified.

The diagram shows the input window $(-1..1, -3..0)$ around pixel $223, 2$. The top entry in each cell is the index within the procedure's window; the bottom entry is the index within the image array. The symbol B stands for the specified border value. The dots stand for pixels beyond the procedure's window and outside the image array.

·	·	$221, 0$	$221, 1$	$221, 2$	$221, 3$
·	$-1, -3$ B	$-1, -2$ $222, 0$	$-1, -1$ $222, 1$	$-1, 0$ $222, 2$	$222, 3$
·	$0, -3$ B	$0, -2$ $223, 0$	$0, -1$ $223, 1$	$\mathbf{0, 0}$ $223, 2$	$223, 3$
·	$1, -3$ B	$1, -2$ $224, 0$	$1, -1$ $224, 1$	$1, 0$ $224, 2$	$224, 3$
·	·	$225, 0$	$225, 1$	$225, 2$	$225, 3$

An Example Input Window

Output parameters are scalar variables. *Each output variable represents the pixel value of an output image.* The final value of an output variable is stored in the output image at the current processing coordinates.

Constant parameters may be scalars or arrays with an arbitrary number of dimensions. They represent precomputed constants, such as a convolution mask, made available for use by the procedure.

Each procedure may also have local variables which may be scalars or arrays. Each processing location has, conceptually, its own set of these variables.

However, variables used as FOR loop indices must have the same value at each processing location and are actually stored only once. Direct assignment to these variables is not allowed. Note that Apply FOR loop indices must be declared at the beginning of a procedure and have scope global to the whole procedure, unlike FOR loop indices in Ada.

Apply does not provide subprocedures or user defined functions. Apply does not provide complex data types other than the windows described above and staticly defined arrays.

The reserved variables ROW and COL are defined to contain the image coordinates of the current processing location. This is useful for algorithms which are dependent in a limited way on the image coordinates. The coordinates start at (0,0), which corresponds to the upper left hand corner of the image array.

Apply does not allow assignment of real expressions to fixed variables, or assignment of fixed expressions to real variables. Apply automatically converts expressions of mixed fixed and real types to real expressions.

BYTE refers to 8 bit values, INTEGER to 16 bit values, and REAL to 32 bit IEEE floating point numbers.

Variable names are alpha-numeric strings of arbitrary length, commencing with an alphabetic character. Case is not significant, except in the optional preprocessing stage.

1.4 Features lacking in the implementation of Apply at UMass

These features should be included in future revisions of the implementation.

- Images larger than the IUA CAAPP array are not supported.
- Image reduction and magnification are not supported. Thus the SAMPLE keyword (used to reduce images) is not supported; and arrays as output parameters (used to magnify images) are not allowed.
- The unsigned integer type is not truly supported, although variables may be declared as unsigned integers. The implementation does all fixed point calculations using signed integers.
- Variables stored in the CAAPP array, that is all variables except constant procedure parameters and for-loop indices, cannot be used in array subscript expressions.

2 The Compiler

The compiler translates programs written in Apply to C code with calls to the run-time library. The run-time system was written in C extended with a preprocessor and a library of macros. The preprocessor converts assembly language for the CAAPP to calls in C which broadcast the assembled instructions to the CAAPP array. The library of macros consist of C subroutines which each make a series of broadcasts to the CAAPP for a specific purpose. The compiled Apply program is compiled again by the C compiler and linked with the preprocessed and compiled run-time library, the macro library, and the simulator. The Apply procedure then appears as if it were another macro provided with the simulator.

The compiler runs under Unix (tm) [Sun Unix 1986], making use of Lisp [Sun Lisp 1988], C [Kernighan & Ritchie 1978], and various Unix (tm) utilities [Sun Unix Utilities 1986]. It produces output for the IUA simulator [Burrill 1990].

The compiler consists of a front-end and a back-end. They each consist of multiple phases. A main program sets up the file input/output and calls each of the phases in turn. The front-end consists of the phases: option processing, macro preprocessing, syntax analysis, parsing, and semantic checking. The back-end consists of two phases: parser tree massaging and code generation.

The main program is written in Lisp. The option processing phase is written in LEX and YACC. It makes a call to the Unix utility M4 to do its macro preprocessing. The syntax analysis phase is written in LEX. The parser is written in YACC. The semantic checking phase is written in Lisp. The back-end, both the parser tree massaging phase and the code generating phase, is written in Lisp.

The original compiler was well-written. The front-end remains untouched and it was only necessary to modify the main procedure when adding a new back-end.

The front end produces a Lisp tree structure containing lists of statements and a symbol table for each Apply procedure. The back-end converts this tree structure into a series of statements in the target language. The compiler received from CMU contained several back-ends. These include back-ends for several varieties of the Warp computer and a back-end which produced normal sequential C-code. The I/O package for C was written for the computing environment at CMU. This back-end is not useful at UMass without creating a new I/O package written in C.

All the back-ends were removed and a new back-end was created for the CAAPP. The general form of the existing back-ends was followed as far as practical. However, all of the existing back-ends compiled to intermediate languages which were of a higher level than the CAAPP assembler even when augmented by the run-time library that was created. The existing back-ends either produced C or a version of W2. Both of these languages allow infix expressions with overloaded operators, and manage storage allocation for named variables. The part of the compiler that produced code which executes entirely within the ACU is directly modeled on the back-end of the C compiler. The part of the compiler which produces code which executes calls to the run-time library needs to handle storage allocation, provide the temporaries used during expression evaluation, and choose the correct non-overloaded operator for each combination of generic arithmetic operator and operand type in an expression. This back-end also had to be much more explicit in dividing tasks between a SISD (Single Instruction stream Single Data stream) controller and an attached processor array than the older back-ends. So the new CAAPP back-end is far more detailed than the older back-ends.

The compiler is not an optimizing compiler. There are more complicated methods of producing code which could be used to provide perhaps an order of magnitude speedup.

The UMass Apply implementation needs further testing.

3 How to use Apply at UMass

3.1 The Standard Method

- A directory is created for working with Apply in a user's account.

  You must have an account established on the SUN. After you login you should invoke 'sunview' and enter or create a command window. Then you should create a subdirectory. The examples assume you choose the subdirectory "apply".

- The "/usr/apply/make_apply_environment" shell procedure is invoked to copy over the files needed from the Apply account.

  The Apply run time library, the Apply header files, and the Apply specific makefile are found in the 'apply' account on the SUN workstation CAX. To set up your subdirectory to access these files, type the command 'sh /usr/apply/make_apply_environment' from the command window while in the subdirectory you will be using for Apply programming. This will copy over the files 'apply.h', 'apply.f', 'apply_macros.c', and 'makefile' to your subdirectory.

- An Apply procedure is written using a text editor.

  You should create an Apply source file in this directory, or copy an existing Apply source file to this directory.

- The Apply procedure name and the number of parameters it requires are passed to the Apply specific makefile. If more than four parameters are required, or if any of the parameters are floating point constants or are constant arrays, you must customize the apply_macros.c file and makefile file as described in section 3.2.

To compile and link an Apply procedure which requires 3 parameters, use a line like:

make APP=my_apply_proc N=3

This will compile the apply procedure and the apply_macros.c file and link them both to the simulator. The apply_macros.c file will include a call to my_apply_proc() which can be invoked from Forth.

Generally, a useful Apply procedure needs at least 2 parameters.

- The makefile compiles the Apply procedure into a C module.
- The makefile compiles the C module.
- The makefile compiles the C to Forth interface module called 'apply_macros.c'.
- The makefile links these modules with the simulator to create a custom simulator.
- The makefile invokes the simulator.

  The make operation produces in your account an executable file named "umiua_apply_my_apply_proc". It then invokes this file, which brings up a simulator.

- The user sets up, using Forth macros, the parameters needed by the procedure.

- The user invokes, using the Forth macro ACU2, the procedure, as in:

  `Parm1 Parm2 0 0 0 ACU2 2drop`

  (The use of the ACU2 macro is described in section 3.3).

The procedure may be debugged by using a standard C debugging tool on the SUN.

3.2 Customizing apply_macros.c and apply_make

Eventually you will want to have a library of apply procedures available for an application. To do this, or any thing else beyond passing a single procedure 0 to 4 parameters, you will need to customize the makefile and the macros file. This should not be difficult, using the examples of 'makefile' and 'apply_macros.c'.

3.3 Running the apply code on the simulator

The file "apply.f" includes Forth words apply_init (which it invokes automatically to initialize PE memory and read in PE indices). It also defines several other Forth words useful for setting up, in PE on-chip memory, parameters for passing to an Apply procedure. It can be inserted into another Forth file or used stand-alone as in

apply_umiua_my_apply_proc -f apply.f

Which would invoke the simulator with the Apply procedure linked in. Then you would set up the parameters for the procedure, and invoke the procedure with a line like:

gray-image gradient-image 0 0 0 ACU2 2drop where gray-image and gradient-image correspond to parameters 1 and 2 respectively.

The Forth macro ACU2 accepts 5 arguments, and returns 2 words, on the Forth stack. The first four arguments are passed through to the C function invoked by ACU2. Unused arguments should be zero. The fifth argument is the index of the C function. When the standard method for using Apply described in this section is used, the index of the C function produced by the Apply compiler will always be zero. The ACU2 macro always returns two words of status which are not used by Apply.

The Apply procedure will redisplay the simulator windows after each line of Apply code is executed.

3.4 A complete example

Appendix B contains a complete example of the steps required to compile and execute an Apply procedure to smooth an image. The file '/usr/apply/smooth.app' exists on the SUN, and the appendix can be used as a tutorial by following the steps mentioned using your own account.

4 Debugging Apply Procedures

The Apply compiler will catch syntax and some semantic errors. No array bounds checking is provided in the language or by the C compiler. The Apply compiler should not produce any programs which the C compiler will reject. Apply will also report some run-time warnings and errors.

The SUN tools dbx and dbxtool can be used with Apply and the simulator.

They take up a lot of memory, so that the number of windows available may be restricted when using them. In order to use them invoke the simulator you create with a line like:

dbxtool apply_umiua_my_apply_proc -f apply.f

Breakpoints and the other features of these debugger tools can be used normally. Note, however, that the simulator features are not available while debugging an Apply procedure because the simulator treats the entire Apply procedure call as a single atomic action. That is, you cannot interact with the simulator window while stopped and using the debugger within an Apply procedure. However, it is possible to update the display of the simulator to reflect the current state of the simulator, as you step through the Apply procedure, by typing in:

call rd()

Therefore, when debugging an Apply Procedure, it is important to set up the simulator to display useful information before invoking the Apply procedure.

Please send SCUDDER mail when you find bugs in the Apply compiler or run time system.

5 Interfacing with an Apply procedure

Apply procedures can be called by other C procedures or by Forth procedures through the use of the ACU2 word.

5.1 Calling Apply procedures from C

Apply procedures are translated into C functions which can be called in the normal way. Parameters to Apply procedures are integers, floating point values, or addresses. The addresses may be addresses in PE on-chip memory, Backing Store memory, or C memory. The PE on-chip memory and the Backing Store memory addresses are represented as C integers, while the C memory addresses are represented as C pointers.

5.2 Calling Apply procedures from Forth

The ACU2 Forth word causes the C module apply_macros.c to be called. The ACU2 provides a macro identifier and 4 arguments. The standard makefile passes up to these four arguments to the Apply procedure supplied as a define to the standard Apply makefile. By modifying apply_macros.c other methods of passing parameters can be implemented. In order to pass more than 4 parameters to an Apply procedure from Forth, multiple ACU2 entries should be created to support the call, all but one of which simply copy parameters into temporary variables.

5.3 Apply Memory Usage and Conventions

The Apply compiler and run time system were designed to interface with standard PE on-chip memory and backing store memory allocation practices. Standard procedures can be used to protect memory which needs to be preserved across calls to Apply procedures from the Apply procedures.

5.3.1 PE On-Chip Memory Usage

Apply image parameters must begin on byte boundaries and may not overlap page boundaries. They cannot be in page 2 of on-chip memory.

Images in on-chip memory can be passed to Apply procedures. The images should be allocated with a call to PemAlloc before invoking the Apply procedure. The Apply procedure will copy these images to its backing store working area and then release the on-chip memory. The images are not guaranteed to be preserved.

Images can also be returned to on-chip memory from Apply procedures. The Apply procedure will use PemAlloc to allocate the memory locations passed to the Apply procedure as destinations for result images.

Any memory allocated using PemAlloc before calling the Apply procedure, which is not used for a parameter to the Apply procedure, is preserved across the call to the Apply procedure.

The Apply run time system requires up to 34 bits of page 2 on-chip memory, and two 32 bit chunks and a 77 bit chunk in pages 1 and 2. These three chunks are not allocated until after the memory used by on-chip input images is freed. If this memory is not available, the Apply run time system will signal an error. If an Apply procedure is called with a parameter in backing store, a staging area equal in size to the largest such parameter must also be available.

5.3.2 Backing Store Memory Usage

Apply image parameters with values above 32 (on page 2 and up) refer to locations in backing store memory. No standard method of allocating backing store memory has been established. The parameters must not be stored in the Apply run time system's working area in the backing store. This working area starts at page 0 of the backing store and extends upward. It stores the input parameter windows, the local variables (including temporaries for the output parameters), and the expression stack. Apply procedures do not affect backing store memory outside of their working area except to write out result parameters to specified locations. By default the working area includes the first 32 bytes (2 pages) of backing store. The upper limit of the backing store can be changed by a call of the Forth macro reserve-bsm-above.

5.3.3 C Memory Usage

Apply procedures use relatively little C memory unless they are passed large arrays as CONST parameters.

References

[Burrill 1990] James Burrill. The Image Understanding Architecture Simulator. *To Appear.*

[Hamey, Webb, & Wu 1987] Leonard G. C. Hamey, Jon A. Webb, and I-Chen Wu. Low-Level Vision on Warp and the Apply Programming Model. *Carnegie-Mellon University technical report* CMU-RI-TR-87-17, 1987.

[Kernighan & Ritchie 1978] Brian W. Kernighan and Dennis M. Ritchie. The C programming Language *Prentice-Hall Software*

Series, 1978.

[Sun Lisp 1988] Sun Microsystems. Sun Common Lisp 3.0 User's Guide. *Sun Microsystems Part No.* 800-3046-10, 1988

[Sun Unix 1986] Sun Microsystems. Getting Started *with* UNIX: Beginner's Guide. *Sun Microsystems Part No.* 800-1284-03, 1986.

[Sun Unix Utilities 1986] Sun Microsystems. Programming Utilities for the Sun Workstation. *Sun Microsystems Part No.* 800-1301-03, 1986.

[Webb & MacPherson 1989] Jon A. Webb and Mike B. MacPherson The Second DARPA Image Understanding Benchmark on WARP and extending Apply to Include Global Operations. *Proceedings of the May 1989 Image Understanding Workshop*, pp. 597–616, 1989.

[Weems *et al.* 1987] Charles C. Weems, Steven P. Levitan, Allen R. Hanson, Edward M. Riseman, J. Gregory Nash, and David B. Shu. The Image Understanding Architecture. *University of Massachusetts COINS Technical Report* 87-76, 1987.

A An Apply Grammar

B An Example: Smoothing an Image

B.1 The Apply Code

```
procedure smooth( inimg: in array (-1..1, -1..1)
                         of byte border 128,
                         outimg: out byte)
is
  sum, i, j: integer;
begin
sum := 0;
  for i in -1..1 loop
      for j in -1..1 loop
          sum := sum + inimg(i,j);
      end loop;
  end loop;
  outimg := (sum + 4) / 9; -- 4 is added to round result
end smooth;
```

B.2 In the Sunview Command Window

```
mkdir apply

cd apply

sh /usr/apply/make_apply_environment

cp /usr/apply/user/smooth.app smooth.app

ls

make APP=smooth N=2
```

B.3 In the Simulator

```
SETUP-APPLY-TEST
```

click on the button labeled "grey" to see the original image.

```
0 15 0 0 0 ACU2 2DROP
```

You will see various Apply temporary images go by.

Change the address in the grey scale image pop-up control panel to 112 (which corresponds to byte 14, a copy of the input image) and click on one of the buttons labeled 'update display' to see the original image.

Then change the address to 120 (which corresponds to byte 15, the second parameter given and the output image) and click on one of the buttons labeled 'update display' to see the smoothed image.

BYE

C Apply Statistics

The WEBB Library, created from the Spider library at CMU, was used to gather statistics on the UMass Apply compiler. The Apply portion of the WEBB Library consisted of 115 files, 105 of which proved to be compilable at UMass. The remaining 10 involve image expansion or reduction, or try to use a variable allocated to the attached processor array to compute an array index.

The size statistics were computed by the Unix utilities wc and ls. The timing statistics use the Unix csh time command to time a csh script making all 105 files (or 105 repeats of the trivial file). Each such csh script was executed 3 times on an otherwise idle Sun-4 running at 16MHz with 16Mb of memory, and the results averaged.

C.1 Apply Language Statistics

Measure	Value
Keywords	25
Types of Statements	5
Operators	19
Primitive Data Types	7
Data Type Constructors	4
Types of Parameters	6

C.2 Apply Compiler Size

Item	Size
Compiler Lisp Source	98Kb
Compiler Lisp Executable	11.1Mb
Compiler Lex & YACC Source	27Kb
Compiler Lex & YACC Executables	115Kb
Run-time Library C Source	56Kb
Run-time Library Object	54Kb
Total Source, Objects & Executables	11.4Mb

C.3 Apply Compilation Measurements

C.3.1 Measurement Set Size

Size	Value	Units
Webb Apply Library Source Files	115	Files
UMass Apply Library Source Files	105	Files
UMass Apply Library Source Size	4478	loc
UMass Apply Library Compiled to C	19896	loc
UMass Apply Library Object	1.6	Mb

C.3.2 Compilation Time

Time	Value	Units
Total Library Apply Compile Time	571	sec
Total Library C Compile Time	366	sec
Total Library Processing Time	937	sec

C.3.3 Compilation Overhead

Overhead	Value	Units
Trivial Module Apply Compile Time	3.3	sec
Trivial Module C Compile Time	2.4	sec
Trivial Module Processing Time	5.7	sec

C.3.4 Incremental Compilation Speed

Speed	Value	Units
Webb Library Apply Compile Speed	23	loc/sec
Webb Library C Compile Speed	38	loc/sec
Webb Library Processing Speed	14	loc/sec

C.3.5 Overall Compilation Speed

Speed	Value	Units
Webb Library Apply Compile Speed	8	loc/sec
Webb Library C Compile Speed	12	loc/sec
Webb Library Processing Speed	5	loc/sec