DYNAMIC EXPRESSION EVALUATOR FOR REAL TIME APPLICATIONS*

Daniel Weaver, William Rathbun, and Creve Maples
Univ. of Calif., Lawrence Berkeley Lab., Berkeley, 94720

ABSTRACT

A dynamic expression evaluator developed for on-line data analysis gives experimenters at LBL's 88-Inch Cyclotron and SuperHILAC both flexibility and high throughput. It uses standard FORTRAN rules for expressions, including operator precedence, and allows built-in functions such as sine and cosine. Evaluation techniques utilizing a combination of FORTRAN and Assembly, and XPL (a dialect of PL/1) have been compared. An XPL evaluator was selected because of development time and evaluator speed and can evaluate expressions at approximately 50 usec per operand on a ModComp Classic 7870 with 20 usec overhead per expression.

INTRODUCTION

The dynamic expression evaluator was designed to enhance online data analysis at the LBL SuperHILAC (Heavy Ion Linear Accelerator) or 88" Cyclotron. Data collection for these accelerators is handled by a package called MINUS3¹, developed by Creve Maples. This package, to which the expression evaluator was added, runs on ModComp IVs and on the ModComp Classic 7870, the machines used for primary data collection at the SuperHILAC and the 88" Cyclotron. Expressions may be evaluated with integer or floating point arithmetic.

A dynamic expression evaluator consists of two parts. The first part, the translator, takes the text of the expression and converts it to an interpretive language. This phase is executed only once for each new expression. The second part of the process, the runtime evaluator, uses the interpretive code to evaluate the expression. This operation occurs once for each new piece of data for each expression. Since data is produced in large quantities at research accelerators, the second phase is used many more times than the first. If the evaluator is to be fast, the intermediate code must be designed to minimize the evaluation time needed for each use. Definition of the code is therefore of key importance. In addition the runtime evaluator should be written in the most efficient language available. The language in which the translator is written, however, need not affect the efficiency of the evaluator since identical interpretive code can be produced by translators written in different languages. A significant factor in selecting a language for a translator is the amount of time needed to write the program.

HISTORY

The MINUS3 data acquisition package provides online display and analysis of real time events. The analysis capability was designed primarily for speed in calculation, and uses a set of nine standard equations built into the code. An experimenter may select which of the nine expressions to use, which parameters are fed into the expression, and the values for constants. Two primary techniques are used to increase speed. First, the limited precision of the data (12 bits) permits, in many cases, the calculations to be performed utilizing 32 bit integer arithmetic without loss of accuracy. The second method also capitalizes on this limited precision by recognizing that an expression with one input variable has a fixed number of results (4096) and all possible results may be stored in a table. Table look-up of results is significantly

faster than actual calculation for each use. Although the nine equations were selected to give a wide variety of computation, any limitation in the number of expressions reduces the scope of calculation available to an experimenter. It became clear that a more flexible method was needed, one that each experimenter could tailor to his or her needs. The method decided on was the addition of an evaluator which would permit user selection of expressions, and dynamic change of expressions during experimentation.

As a first requirement, the evaluator had to be as fast as possible within system constraints. Its second requirement was to increase flexibility of computation. Third, it had to be relatively easy for new users to learn to use--and to use at 3 a.m. when they've been running for two days straight. Finally, it had to interface with MINUS3, to receive input and display results.

Two different translators using different parsing techniques and three different runtime evaluators were implemented. The first translator was written in FORTRAN with a runtime evaluator also written in FORTRAN. After preliminary timing tests, the runtime evaluator was rewritten in Assembly, producing an increase in throughput of about 30%. The original FORTRAN version of the evaluator did not fully meet the design criteria. While useful, it suffered because operator precedence had to be explicitly stated through the use of parentheses. For example, with no parentheses, $1+2*3$ would be calculated as $(1+2)*3$, since all expressions were evaluated from left to right. Standard operator precedence would have performed the multiply first. Two solutions presented themselves to solve the problem of operator precedence. The first was to redesign the existing evaluator to include operator hierarchy. The second was to redevelop the evaluator using the XPL language and its automatic compiler writing system. This second approach resulted in a second translator written in XPL with a runtime evaluator also written in XPL.

XPL EXPRESSION EVALUATOR DESIGN

XPL was originally developed as a compiler writing language by W. M. McKeeman, who introduced this dialect of PL/1 in his book *A Compiler Generator*². The XPL language provides powerful character string and bit manipulators, fixed and floating point arithmetic, and structured program flow. XPL was designed to be used in table-driven syntax-directed translators such as the XPL compiler itself, and the expression evaluator was built around just such a design.

The XPL system, which includes the Compiler, Skeleton, and Backus-Naur Form (BNF) Grammar Analyzer, provides the user with a quick and efficient method for defining a language. A BNF description of a grammar may be input to the Analyzer which will in turn output syntax tables. The syntax tables, used with the Skeleton, form a syntax checker for the BNF grammar. This system served as a framework upon which the translator portion of the evaluator could be built quickly and easily.

*This work supported by the Director, Office of Energy Res., Div. Nucl. Physics, Office High Energy and Nucl. Phys. and Nucl. Sci. of the Basic Energy Sci. Program, U.S. Dept. of Energy, Cont. No. W-7405-ENG-48.

BNF is a metalanguage designed to formalize language definition. The metasymbol ::= is used to mean "is defined as." The brackets < > indicate a symbol that is defined in terms of other symbols. The BNF grammar defined for use in the expression evaluator was as follows:

```

<GOAL> ::= <EXPRESSION>
<EXPRESSION> ::= <TERM>
<EXPRESSION> ::= <EXPRESSION> + <TERM>
<EXPRESSION> ::= <EXPRESSION> - <TERM>
<EXPRESSION> ::= * <TERM>
<EXPRESSION> ::= / <TERM>
<TERM> ::= <PRIMARY>
<TERM> ::= <TERM> * <PRIMARY>
<TERM> ::= <TERM> / <PRIMARY>
<PRIMARY> ::= <OPERAND>
<PRIMARY> ::= <OPERAND> ** <OPERAND>
<OPERAND> ::= ( <EXPRESSION> )
<OPERAND> ::= <NUMBER>
<OPERAND> ::= <VARIABLE>
<VARIABLE> ::= <IDENTIFIER>
<VARIABLE> ::= <FUNCTION HEAD> <EXPRESSION> )
<FUNCTION HEAD> ::= <IDENTIFIER> (

```

The symbols <IDENTIFIER> and <NUMBER> are defined within the translator itself.

This grammar was used by the analyzer to create tables in the form of XPL statements which describe all permutations of expressions that can be entered by experimenters, for syntax and operator precedence. After combining the output of the Grammar Analyzer to the Skeleton, code was added to save the interpreted expressions, perform error handling, and interface to MINUS3. The result is the translator program, CRUNCH. When fed an expression, CRUNCH converts it into interpretive code by comparing it against the syntax tables created by the Analyzer. The interpretive code is generated in the form of reverse-Polish notation, i.e. expressions are represented in the code with operands followed by operations. The interpretive code is passed to the evaluator at runtime.

The runtime evaluator is stack oriented. Each operand is placed on the stack, then an operation is performed on the two elements on the top of the stack (one if unary minus or functions) and the result is placed back on the stack.

Example:

Consider the expression $P1 + P2 * 3 - 2$

The first operation that must be performed is the multiply followed by the add and then the subtract. In reverse-Polish the expression would be

P1 P2 3 * + 2 -

This is in fact how the expression appears to the runtime evaluator. It loads P1, P2 and 3 on to the stack then does the multiply on the top two stack elements (P2 and 3). The result is placed on the second position on the stack, the place where P2 used to be. The top two elements then are added with the result being placed back onto the stack as its only element. Finally the constant 2 is pushed onto the stack and the subtract is performed. After an expression is evaluated the result is always the last element on the stack. This value is then returned to the data acquisition package, MINUS3.

EXPRESSION SYNTAX

The learning curve for physicists using the evaluator was minimized by making mathematical syntax basically the same as that found in FORTRAN and most other high

level languages. This permits nesting by use of parentheses, with multiplication and division being performed before addition and subtraction within one nesting level.

Input expressions may be up to 80 characters in length, and may contain the following standard operator symbols:

```

( )
** (exponentiation)
* /
+ -

```

Parameters, pseudo-parameters, constants and built-in functions may be used within each expression. A parameter is the data from a given detector. Parameters are specified as P1 thru Pn where n is the number of detectors in a given experiment. A pseudo-parameter is the output of some other hard-coded expression or dynamic expression. Pseudo-parameters are specified as Pn+1 thru P32 where n is the number of detectors in the experiment. If desired, the results of a pseudo-parameter may replace the value of a given detector input. Constants may be imbedded as part of an expression, (e.g. $P1 + 6.3$) or given identifier names of C1 through C16. Identifier names for constants permit change of values for "constants" within a run. The built-in functions SINE, COSINE, TANGENT, SQRT (square root), EXPONENT, LOG (base 10), LN (natural log), and ABSOLUTE value are supported. Expressions may be evaluated as integer or real, specified by the structure of the equation.

IMPLEMENTATION

The dynamic expression evaluator interfaces to MINUS3 in the following way:

CHAOS handles all data collection and the writing of data onto magnetic tape. ORDER handles the analysis of that data and MULTID displays the analysis results for the experimenter.

At the start of a run CHAOS is told how many parameters are in each event and other relevant information about data collection. Subsequently the experimenter tells ORDER what type of analysis is to be performed on the data. At this point the experimenter may choose any of the hard coded expressions or may use the dynamic expression evaluator. Upon encountering a dynamic expression the translator portion of the expression evaluator, CRUNCH, is called to convert the text to the intermediate code used by the runtime evaluator, MUNCH. The intermediate code is saved within ORDER for use during data analysis. During the run, experimental data is collected by CHAOS, and stored on magnetic tape. Simultaneously, ORDER feeds the data through the runtime evaluator which uses the previously interpreted expressions to produce ongoing output. This output can be formatted for viewing on a Tektronix 4014 scope, or a plotter. The output of MUNCH may be used with any of the existing display options in MINUS3 such as gating or 2-Dimensional displays.

EDITING EXPRESSIONS

An editing command was included in MINUS3 to permit alteration of expressions without retyping. This command can be invoked while an experiment is in progress or during subsequent analysis. In either case, the experimenter can retrieve the input source expressions, change text (by typing alterations beneath it), or code deletions (by typing an underscore beneath character(s) to be deleted), or additions (by typing an up-arrow at the point of insertion). The changed expression is processed back through CRUNCH, and the internal runtime table is altered.

Below is an example of defining and editing a dynamic expression (boldface denotes user response):

```
# 1 ... FORTRAN EXPRESSION EVALUATOR
  (USE FORTRAN ALGEBRAIC NOTATION WITH
  SYMBOL PN DENOTING PARAMETER N AND CN TO
  DENOTE A DYNAMIC VARIABLE N)
*** ENTER EXPRESSION (80 CHAR. MAX)
1.0+C1*P2-C2*(2.7E-4*P2**2 + P2**3)+C2*P1**4

***EDIT EXPRESSION (Y/N)?
Y
***ENTER CHANGE(S) UNDER ORIGINAL EXPRESSION
  (USE & FOR A SPACE; ^ TO INSERT; AND _ TO
  DELETE)
1.0+C1*P2-C2*(2.7E-4*P2**2 + P2**3)+C2*P1**4
          P      ^C3*  _  4  2
1.0+C1*P2-C2*P2*P2 + C3*P2**3+C4*P2**4
```

FORTRAN EXPRESSION EVALUATOR DESIGN

Unlike the XPL evaluator the FORTRAN system is not stack oriented. The FORTRAN evaluator uses a single accumulator type architecture and uses temporary storage when necessary. Each instruction given to the evaluator consists of an operator and an operand with the result being placed into the accumulator. Operator precedence, either implicit (different operators) or explicit (parentheses), requires the accumulator to be stored. In some cases this brings about unnecessary storage and retrieval of intermediate variables.

The lower throughput of the system could be attributed to the fact that all operations require an operand and the operand decode requires floating point arithmetic. Within the FORTRAN evaluator operators are designated as integers (1 through 99) referencing operations (add, subtract, store-temporary, retrieve-temporary, etc.) and external functions. The operands are either positive floating-point constants, or coded indices into arrays of defined constants or variables (parameters). The unary negative is handled by a special operator called retrieve-negate.

RESULTS

Development time for the FORTRAN evaluator was approximately two man months. Development time for the XPL expression evaluator was about half the amount of time used for the FORTRAN system. The difference in the amount of effort is directly attributable to the use of the XPL compiler writing system.

The XPL evaluator can evaluate expressions at approximately 50 microseconds per operand on a ModComp Classic 7870 with 20 microseconds overhead per expression. Below is a table of times given in microseconds for the various evaluators. Hard coded FORTRAN times are included for comparison.

EXPRESSION	Compiled FORTRAN	XPL	FORTRAN/ ASSEMBLY
X/A+Y/B+Z	24	305	500
(X/A+Y)**P-Y**P	232	605	804
A+BX+CX**2+DX**3+EX**4	79	780	1270
SQRT(X)	34	145	280

All variables and constants are 32 bit floating point numbers.

Times for the FORTRAN runtime evaluator were 50% slower than the Assembly timings.

Timings for the XPL runtime evaluator as it currently exists would be improved by rewriting the code in Assembly but this has not been done to date. The assembly language evaluator written for the FORTRAN translator cannot be used with the XPL translator since the two packages use different intermediate code.

After initial timing tests showed the XPL expression evaluator to be faster, the XPL system was integrated into the data analysis package of MINUS3. The FORTRAN expression evaluator is currently being used to provide graphics display and modeling.

The major advantage of a FORTRAN expression evaluator is portability. The XPL system is limited to machines with an XPL compiler. This is not a major disadvantage at LBL because the MINUS3 data acquisition system is only used on ModComp equipment.

FUTURE DIRECTION

Since use of the XPL dynamic expression evaluator has begun, some suggestions have been made for its improvement. The first was to expand the character size to over 80 characters. Second, inclusion of the logical operators AND, OR and NOT was suggested. Third, it was requested that the implicit specification of integer or real evaluation for each expression be made explicit. Currently, the expression is considered integer if it does not contain a floating point constant or a function call. Finally, rewriting the runtime evaluator in Assembly is still under consideration as a future upgrade.

REFERENCES

1. C. Maples, J. Sivak, MINUS3 - A General Purpose Data-Acquisition System at LBL's 88"-Cyclotron and SuperHILAC, IEEE Transactions in Nuclear Science, Vol. NS-26, No. 4, 4409 (1979)
2. W. M. McKeeman, J. J. Horning, D. B. Wortman, A Compiler Generator, (Prentice-Hall), Chap. 6, 9. (1970)