DIVISION OF COMPUTER SCIENCE

An Analysis of Automatic Differentiation on the VAX

Sandra Abbott

Technical Report No.159

July 1993

THE UNIVERSITY OF HERTFORDSHIRE

School of Information Sciences

An Analysis of Automatic Differentiation on the VAX

by

Sandra Abbott

Technical Report No. 159 **July 1993**

Abstract

The reverse accumulation algorithm is an efficient technique for implementing automatic differentiation. It is able to calculate the complete gradient vector of a scalar function to the same degree of accuracy as the target function and requires only about twice the amount of floating point arithmetic as that needed to calculate the function itself. However, when implemented on conventional architectures several overheads are incurred which degrade the performance. This report analyses the performance of an Ada implementation on a MicroVAX 3800 using VMS and discusses possible improvements.

1 Introduction

Differentiation is a fundamental mathematical tool essential in a diverse range of applications including gradient methods for optimisation and the iterative solution of non-linear equations. Automatic differentiation comprises a number of processes for obtaining accurate derivatives of functions by applying the chain rule directly to numerical values rather than to symbolic expressions. The two established methods for calculating the derivative, forward and reverse accumulation, can be compared in terms of the underlying computational graph of the function. Forward accumulation yields the function value and its derivatives simultaneously as the graph is traversed once from the independent to the dependent variables. Storage requirements are relatively low as there is no need explicitly to store the graph. However, the temporal and spatial requirements of the algorithm increase in proportion to the number of independent variables unless sparse matrix techniques are applied [Dixo90].

In reverse accumulation, the function is evaluated during the forward sweep of the graph but the derivatives are calculated on a subsequent reverse pass as the graph is traversed from the dependent to the independent variables. Memory bandwidth requirements are high because the entire computational graph must be built and stored since the reverse pass takes explicit account of the order in which the function was computed. Using reverse accumulation, the complete gradient vector of a scalar function can be calculated to the same degree of accuracy as the target function itself, and requires only about twice the amount of floating point arithmetic as that needed to calculate the function [Irim87]. All permissible parallelism in the function can also be exploited in the gradient calculation.

However, when current architectures are used to support manipulation of the computational graph during reverse accumulation, a number of overheads are incurred. As a result, implementations of reverse accumulation on architectures with good floating point support spend only about ten percent of their time performing arithmetic, with a consequent ten-fold handicap in performance. The majority of the processing time is spent manipulating the function graph [Chri92a].

This report describes a series of investigations carried out using a MicroVAX/VMS 3800, part of a VAX Cluster, at Farnborough College of Technology. The system uses a fast version of the CVAX chip implemented in 1.5 micron CMOS technology, with a 60 ns cycle time and a built-in 1KB cache. Extensive use was made of the Performance and Coverage Analyzer supplied as part of the VAXset of software engineering tools. Tests were conducted on an Ada implementation of reverse accumulation [Chri92a] with the aim of analysing the CPU-time distribution of the algorithm in order to discover how much time is spent building and traversing the graph.

2 The VAX Performance and Coverage Analyzer (PCA)

The PCA, analyses the dynamic behaviour of a program through two operational components: the Collector and the Analyser. The Collector gathers performance data of a type selected by the user and writes it to a file from where the Analyzer processes it to produce performance histograms and tables. The Collector data options include:

- Program counter (PC) sampling data: sampling the program counter every 10 milliseconds of system time or at a user-defined rate.
- CPU sampling data: sampling the PC using virtual-process time and hence ignoring time spent waiting for page faults, I/O transfers etc. This option was used during most of the data-gathering program runs.
- Counters : counting the exact number of times that specified instructions execute. This was used to check the frequency of function calls during program execution.

The Analyzer operates on the data file produced by the Collector and displays it in the format specified by the user. A range of domains and filters can be applied to the data before display in order to investigate selected areas of the program.

The most useful PCA options for this investigation were found to be CPU-sampling for data collection and histogram plotting by program address for data analysis.

3 Algorithms for implementing automatic differentiation

The theory of automatic differentiation is well documented, for example [Grie89, Irim87]. The intention here is to summarise the algorithms in [Chri92a] for ease of reference as these form the basis for the Ada package under investigation.

We start with the assumption that a function $f(x_1, \dots , x_n)$ can be constructed from a sequence of simple operations. A typical sequence would consist of unary and binary arithmetic operations but may also contain control statements so that the active part of the graph will vary depending on the values of the control variables and the nature of the inputs.

Building the graph

```
    for i from n+1 upto m do                      -- after the independent variables in the list
        $x_i = f_i(x_{t_i 1}, \dots , x_{t_i n_i})$          -- evaluate each intermediate value in the graph
    enddo                                          -- finishing with the function value at $x_m$
{$x_m = f(x_1, \dots, x_n)$}
```

where for each $i > n$, n_i is the arity of f_i and t_i is a map from $\{1, \dots , n_i\}$ into $\{1, \dots , i-1\}$.

The forward pass through the graph is used to compute the function value: the reverse pass is used to evaluate the first derivative, ∇f. We define $\bar{x}_i = \partial x_m / \partial x_i$, for $1 \leq i \leq m$. Then the vector $(x_1, \dots , x_n)$ is ∇f. The algorithm is:

Forward pass - computes the function

```
    For i from 1 upto n do                        -- for each independent variable
        $\bar{x}_i = 0$                           -- initialise the adjoints which will accumulate the
    enddo                                         -- derivative on the reverse pass
    for i from n+1 upto m do                      -- for each node following on from the independent
        $x_i = f_i(x_{t_i1}, \dots, x_{t_in_i})$  -- variables, calculate the node value
        $\bar{x}_i = 0$                           -- initialise the adjoints
    enddo
$\{x_m = f(x_1, \dots, x_n)\}$                    -- the last node holds the function value
```

Reverse pass - computes the first derivative

```
    $\bar{x}_m = 1$                               -- initialise the last adjoint ready for the reverse pass
    for i from m downto n+1 do                    -- for all intermediate variables -
        for p from 1 to $n_i$ do                  -- for each arc leading back from the node to others
            $\bar{x}_{t_ip} = \bar{x}_{t_ip} + \bar{x}_i * (D_pf_i)(x_{t_i1}, \dots, x_{t_in_i})$
        enddo                                     -- compute the contribution to the derivative
    enddo
$\{(\bar{x}_1, \dots, \bar{x}_n) = (\nabla f)(x_1, \dots, x_n)\}$   -- solution given by adjoints of independent
                                                  -- variables
```

The reverse accumulation package also contains the functionality to enable the Hessian (second derivative) matrix to be calculated. This is composed from a number of directional second derivatives, each one requiring a further forward and reverse pass of the computational graph. Hence the Hessian can be computed either in one operation by using n parallel processors, or by making a further n iterations of the graph. A full discussion and proof are contained in [Chri92a] whereas just the algorithm is reproduced here.

Forward pass - computes the function and prepares for the Hessian

```
    For i from 1 upto n do                        -- for each independent variable
        $\bar{x}_i = 0$                           -- initialise the adjoints as before
        $w_i = u_i$                               -- set up the initial vector values in the $w_i$
        $\bar{w}_i = 0$                           -- initialise the Hessian adjoints to 0
    enddo

    for i from n+1 upto m do                      -- for each node following on from the independent
        $x_i = f_i(x_{t\,1}, \dots, x_{t\,n})$    -- variables, calculate the node value
        $\bar{x}_i = 0$                           -- initialise the adjoints
        $w_i = 0$                                 -- and the $w_i$
        for p from 1 to $n_i$ do                  -- calculate 'first half" of second derivative (df/dx)
            $w_i = w_i + w_{t_ip} * (D_pf_i)(x_{t_i1}, \dots, x_{t_in_i})$
        enddo
        $\overline{w}_i = 0$                      -- initialise adjoint ready for reverse pass
    enddo
{$x_m = f(x_1, \dots, x_n)$, $w_m = (u.\nabla f)(x_1, \dots, x_n)$}
                                                 -- last node holds function value and pre-Hessian
```

Reverse pass - computes the derivative and Hessian

```
    $\overline{x}_m = 1$                          -- initialise the last adjoint ready for the reverse pass
    for i from m downto n+1 do                   -- for all intermediate variables -
        for p from 1 to $n_i$ do                 -- for each arc leading back from the node to others
          $\overline{x}_{t_i p} = \overline{x}_{t_i p} + \overline{x}_i * (D_p f_i)(x_{t_i 1}, \dots, x_{t_i n_i})$
                                                 -- compute the contribution to the derivative
          $\overline{w}_{t_i p} = \overline{w}_{t_i p} + \overline{w}_i * (D_p f_i)(x_{t_i 1}, \dots, x_{t_i n_i})$
                                                 -- compute the contribution to the Hessian
          for q from 1 to $n_i$ do
            $\overline{w}_{t_i p} = \overline{w}_{t_i p} + \overline{x}_i * w_{t_i q} * (D_p D_q f_i)(x_{t_i 1}, \dots, x_{t_i n_i})$
                                                 -- repeat for each arc connected to the node
          enddo
        enddo
    enddo
{$(\overline{x}_1, \dots, \overline{x}_n) = (\nabla f)(x_1, \dots, x_n)$, $(\overline{w}_1, \dots, \overline{w}_n) = (u.\nabla f)(x_1, \dots, x_n)$}
                                                 -- solutions given by adjoints of the independent
                                                 -- variables
```

5 Automatic differentiation by reverse accumulation - the test function

The analysis was applied to the reverse accumulation package written in Ada. This implemented the algorithm in a straightforward way using real valued adjoints at each node to contribute to the final solution. The test program, in which all the operators except assignment were overloaded, used the scalar valued Helmholtz energy function of a mixed fluid in a unit volume at absolute temperature T:

$$f(x) = RT\sum_{i=1}^{n} x_i \log (x_i / (1 - b^T x) - (x^T Ax / 8b^T x) \log ((1+(1+ 2)b^T x) / (1+(1- 2)b^T x)$$

where $0 \leq x_i, b_i, b \in R^n, A = A^T \in R^{n x n}$

With n = 100, the function generated a graph with 10206 independent variable nodes and 20818 calculated nodes. The function value, first derivative and a directional second derivative were calculated. Later versions of the package [Chri92b, Bart93] will include code to implement sparse adjoint vectors and to eliminate shared computational threads: both techniques typically apply when it is necessary to compute the derivatives of vector valued functions. Neither was included in the results that follow.

6 Automatic differentiation by reverse accumulation - the package

This section describes the Ada package for reverse differentiation used in this investigation. It consists of a number of functions which are called as the computation proceeds. Part of the package is concerned with graph building and function evaluation, part with the calculation of the first derivative and part with evaluating the Hessian.

6.1 Graph building and function evaluation

Using operator overloading, the graph is constructed as the function is evaluated. Each graph node contains the following information:

```
ID   : NODE_ID;      -- number for this node
KIND : NODE_KIND;    -- many different node types to choose
                     -- from: SCRCHK CONSTK MINUSK
                     --       INDEPK PLUSK  TIMESK
                     --       RECIPK EQUALK LESSTK
                     --       ABSK   SQRTK  EXPK
                     --       LOGK   SINK   ASINK
XF   : ITEM;         -- value of this node (item is a float)
XB   : ITEM;         -- first derivative
WF   : ITEM;         -- directional derivative
WB   : ITEM;         -- directional second derivative
D    : ITEM          -- cache for first derivative of unary op
P1   : REFNODE       -- pointer to node containing first
                     -- argument of this operation
P2   : REFNODE;      -- same for second argument
LF   : REFNODE;      -- forward link
LB   : REFNODE;      -- backward link
```

The nodes are stored as a doubly-linked list, the links being determined by the function under test and its underlying graph structure. The list, which can be considered as a serial representation of the function graph, contains the independent variables and constant values followed by the intermediate nodes required by the function and ends with the dependent variables. As the internal representation of the function is broken down into a series of unary and binary elements, each node requires a maximum of two pointers to its arguments, as shown.

The function MAKE_NODE is used to form each new node as required for constructing the graph. XF_PROPAGATE evaluates the function, starting from the beginning of the list and working forwards storing intermediate values as it proceeds.

6.2 Derivative evaluation

XB_PROPAGATE performs the reverse pass of the algorithm by moving back down the pointers P1 and P2 to place the appropriate XB values there. It starts at the dependent variable node and then moves along the backward link to the previous node, evaluating as it goes, until it reaches the independent variable nodes, whereupon the process is complete.

6.3 Evaluating the Hessian

The hessian is accumulated according to the algorithm in 4 above by using the functions WF_PROPAGATE for the forward pass and WB-PROPAGATE for the reverse pass. Full details of the algorithms and the Ada implementation are to be found in [Chri90, Chri92a].

7 Time measurements

In order to gain some measure of the time taken for graph building and traversal, the program (generated using the optimised compiler) was compared with a modified version that simply evaluated the function using real arithmetic. Both versions were batch processed several times and the minimum charged CPU-times used for comparison. The simplified program required 1.46s, compared with 41.93s for reverse accumulation during which the graph was constructed twice. Hence it would appear that graph construction and manipulation represent considerable overheads and degrade performance.

8 Results

The performance analysis required a large number of run-time investigations as listed below, the details of which follow.

1. Function call count
2. Non-optimised compilation
3. Optimised compilation
4. Order of "+" and "*" swapped in function listing
5. Conditional tests swapped in "+" and "*"
6. Order of conditional statements in "+" and "*" swapped
7. Conditional statements removed
8. Modified test function
9. Modified "*" function
10. Modified "+" function
11. Counting function calls in the incomplete function
12. Converting "+" to ADD
13. Optimised compilation with ADD function
14. Non-optimised compilation with ADD function
15. Assembler code from optimised compilation

8.1 Function call count

The PCA Collector was set for Counters and compilation was carried out with the optimise option switched off. The resulting data files were plotted by function name.

Function name	Percentage of run-time calls
MAKE_NODE	49.7
"+"	21.8
"*"	21.6
CONSTANT_VALUE	6.1

This shows that the "+" and "*" overloaded operator functions were called approximately the same number of times. Hence it would seem reasonable to assume that each function would account for roughly the same amount of CPU-time during graph construction. The difference between the two arithmetic functions is simply that "+" assigns to the node the value of the sum of its input parameters whereas "*" assigns the product.

8.2 Non-optimised Compilation

The PCA Collector was set for CPU-sampling and compilation was carried out with the optimise option switched off. The resulting data files were plotted by function name and by line number.

Function name	Percentage of run-time
WB_PROPAGATE	13.9
"*"	12.8
MAKE_NODE	12.7
WF_PROPAGATE	12.3
XB_PROPAGATE	12.2
XF_PROPAGATE	11.3
"+"	8.1

Line No	Percentage of run-time	Function
386	5.3	"*"
76	5.1	H
160	4.0	MAKE_NODE
711	3.8	WB_PROPAGATE
665	3.6	WF_PROPAGATE
...		
358	1.4	"+"
391	1.2	"*"
363	1.1	"+"

Note: The complete sequence is shown in Appendix 1.

Line No	Statement	
386	IF A.NODE.ID /= A.ID OR B.NODE.ID /= B.ID THEN ERROR	
76	A_X := A_X "+" A(I,J) "*" X(J)	
160	GRAPH.CHAIN.LAST_ALLOC := new GRAPH_NODE	
711	C.P1.WB := C.P1 WB + C.WB * C.P2.XF + C.XB * C.P2.WF	{timesk}
665	C.WF := C.P1.WF * C.P2.XF + C.P2.WF * C.P1.XF;	{timesk}
...		
358	IF A.NODE.ID/=A.ID OR B.NODE.ID/=B.ID THEN ERROR	
391	C.NODE := A.NODE.XF * B.NODE.XF	
363	C.NODE := A.NODE.XF + B.NODE.XF	

This is the first indication that function "*" occupies the processor for approximately fifty percent more time than "+" and that, in particular, the error check in line 386 is the most time consuming operation of all, taking 3.78 times longer than the (identical) line 358. See 8.7 where this discrepancy is discussed further.

8.3 Optimised Compilation

The PCA Collector was set for CPU-sampling and compilation was carried out with the optimise option switched on. The resulting data files were plotted by function name and by line number.

Function name	**Percentage of run-time**
MAKE_NODE	15.3
"*"	13.5
WF_PROPAGATE	13.3
XF_PROPAGATE	12.4
WB_PROPAGATE	12.2
XB_PROPAGATE	10.9
"+"	9.1

Line No	**Percentage of run-time**	**Function**
160	6.4	MAKE_NODE
386	6.1	"*"
76	4.7	H
665	4.2	WF_PROPAGATE
705	4.0	WB_PROPAGATE
...		
358	1.1	"+"
363	0.8	"+"
391	0.5	"*"

Line No	**Statement**	
160	GRAPH.CHAIN.LAST_ALLOC := new GRAPH_NODE	
386	IF A.NODE.ID /= A.ID OR B.NODE.ID /= B.ID THEN ERROR	
76	A_X := A_X "+" A(I,J) "*" X(J)	
665	C.WF := C.P1.WF * C.P2.XF + C.P2.WF * C.P1.XF;	{timesk}
705	C.P1.WB := C.P1.WB + C.WB	{plusk}
...		
358	IF A.NODE.ID/=A.ID OR B.NODE.ID/=B.ID THEN ERROR	
363	C.NODE := A.NODE.XF + B.NODE.XF	
391	C.NODE := A.NODE.XF * B.NODE.XF	

The optimising facility of the compiler is generally less effective when using operator overloading. The compiler is unable to discern the overloaded code and may produce an output which is less efficient than the non-optimised version. The results here, while different from those in 8.2 above, do not show dramatic changes: "*" requires 1.48 more time than "+" and line 386 requires 5.54 more time than line 358. The majority of the following results were obtained with the optimising switch disabled.

8.4 Order of "+" and "*" swapped in function listing

The PCA Collector was set for CPU-sampling and compilation was carried out with the optimise option switched off. The positions of the "*" and "+" functions in the code were swapped.

Function name	Percentage of run-time
WB_PROPAGATE	13.9
"*"	12.7
WF_PROPAGATE	12.4
MAKE_NODE	12.2
XB_PROPAGATE	12.1
XF_PROPAGATE	11.6
"+"	8.3

Line No	Percentage of run-time	Function
358	5.5	"*"
76	4.9	H
665	4.1	WF_PROPAGATE
160	4.0	MAKE_NODE
711	3.6	WB_PROPAGATE
...		
386	1.7	"+"
363	1.1	"+"
391	1.0	"*"

Line No	Statement	
358	IF A.NODE.ID /= A.ID OR B.NODE.ID /= B.ID THEN ERROR	
76	A_X := A_X "+" A(I,J) "*" X(J)	
665	C.WF := C.P1.WF * C.P2.XF + C.P2.WF * C.P1.XF;	{timesk}
160	GRAPH.CHAIN.LAST_ALLOC := new GRAPH_NODE	
711	C.P1.WB := C.P1.WB + C.WB * C.P2.XF + C.XB * C.P2.WF	{timesk}
...		
386	IF A.NODE.ID/=A.ID OR B.NODE.ID/=B.ID THEN ERROR	
363	C.NODE := A.NODE.XF + B.NODE.XF	
391	C.NODE := A.NODE.XF * B.NODE.XF	

The order in which the "*" and "+" functions appear in the reverse accumulation package was swapped round to eliminate any local effects such as page boundary regions. The same results were obtained as in the two previous tests.

8.5 Conditional tests swapped in "+" and "*"

The PCA Collector was set for CPU-sampling and compilation was carried out with the optimise option switched off. The positions of the "*" and "+" functions in the code were swapped as were the tests in the IF statement on line 358.

Function name	Percentage of run-time
WB_PROPAGATE	13.9
MAKE_NODE	12.8
WF_PROPAGATE	12.8
"*"	12.4
XB_PROPAGATE	11.8
XF_PROPAGATE	11.5
"+"	8.0

Line No	Percentage of run-time	Function
76	5.3	H
358	5.2	"*"
160	4.2	MAKE_NODE
665	4.0	WF_PROPAGATE
711	3.5	WB_PROPAGATE
...		
386	1.3	"+"
363	1.1	"+"
391	1.0	"*"

Line No	Statement	
76	A_X := A_X "+" A(I,J) "*" X(J)	
358	IF B.NODE.ID /= B.ID OR A.NODE.ID /= A.ID THEN ERROR	
160	GRAPH.CHAIN.LAST_ALLOC := new GRAPH_NODE	
665	C.WF := C.P1.WF * C.P2.XF + C.P2.WF * C.P1.XF;	{timesk}
711	C.P1.WB := C.P1.WB + C.WB * C.P2.XF + C.XB * C.P2.WF	{timesk}
...		
386	IF B.NODE.ID/=B.ID OR A.NODE.ID/=A.ID THEN ERROR	
363	C.NODE := A.NODE.XF + B.NODE.XF	
391	C.NODE := A.NODE.XF * B.NODE.XF	

With the functions still in their swapped positions, this test was designed to detect any skew in the node test of lines 358 and 386. Once again, no significant changes to the previous results were detected.

8.6 Order of conditional statements in "+" and "*" swapped

The PCA Collector was set for CPU-sampling and compilation was carried out with the optimise option switched off. The positions of the "*" and "+" functions in the code were restored and the IF statement was swapped as in (5) above.

Function name	Percentage of run-time
WB_PROPAGATE	14.1
WF_PROPAGATE	12.6
"*"	12.6
MAKE_NODE	12.6
XB_PROPAGATE	12.4
XF_PROPAGATE	11.6
"+"	8.3

Line No	Percentage of run-time	Function
386	5.2	"*"
76	4.5	H
160	3.7	MAKE_NODE
601	3.7	XB_PROPAGATE
711	3.7	WB_PROPAGATE
...		
358	1.4	"+"
391	1.3	"*"
363	1.3	"+"

Line No	Statement	
386	IF B.NODE.ID /= B.ID OR A.NODE.ID /= A.ID THEN ERROR	
76	A_X := A_X "+" A(I,J) "*" X(J)	
160	GRAPH.CHAIN.LAST_ALLOC := new GRAPH_NODE	
601	C.P1.XB := C.P1.XB + C.XB	{plusk}
711	C.P1.WB := C.P1.WB + C.WB * C.P2.XF + C.XB * C.P2.WF	{timesk}
...		
358	IF B.NODE.ID/=B.ID OR A.NODE.ID/=A.ID THEN ERROR	
391	C.NODE := A.NODE.XF * B.NODE.XF	
363	C.NODE := A.NODE.XF + B.NODE.XF	

Once again, these changes had little effect on the individual line timings.

8.7 Conditional Statements Removed

The PCA Collector was set for CPU-sampling and compilation was carried out with the optimise option switched off. The "*" and "+" functions were in their original positions in the code but both IF statements were commented out. This should help isolate the time consuming section of the code.

Function name	**Percentage of run-time**
WB_PROPAGATE	15.0
WF_PROPAGATE	13.3
XB_PROPAGATE	12.8
MAKE_NODE	12.7
XF_PROPAGATE	11.9
"*"	10.5
"+"	7.0

Line No	**Percentage of run-time**	**Function**
76	5.3	H
391	4.8	"*"
160	4.3	MAKE_NODE
711	4.2	WB_PROPAGATE
665	4.2	WF_PROPAGATE
...		
363	1.3	"+"

Line No	**Statement**	
76	A_X := A_X "+" A(I,J) "*" X(J)	
391	C.NODE.XF := A.NODE.XF * B.NODE.XF	
160	GRAPH.CHAIN.LAST_ALLOC := new GRAPH_NODE	
711	C.P1.WB := C.P1.WB + C.WB * C.P2.XF + C.XB * C.P2.WF	{timesk}
665	C.WF := C.P1.WF * C.P2.XF + C.P2.WF * C.P1.XF;	{timesk}
...		
363	C.NODE := A.NODE.XF + B.NODE.XF	

The two functions still require a markedly different amount of time but now the difference appears to be accounted for by the multiplication instruction compared with the addition instruction. This is quite likely although specific timings for individual VAX machine-code instructions were not available. Neither line 391 nor line 363 had appeared as major consumers of time in the previous experiments. It is possible that the PCA is not allocating time correctly during function calls and has been wrongly crediting the conditional test instructions with large amounts of CPU-time.

8.8 Modified test function

The PCA Collector was set for CPU-sampling and compilation was carried out with the optimise option switched on. The order of calculation of the test function was modified to consider the effect of residual partial results making some parameters readily available in the registers while others had to be fetched from memory. The modified function, again with overloaded operators, was:

(LOG((ONE + (ONE + SQRT_2) * B_DOT_X)/(ONE + (ONE - SQRT_2) * B_DOT_X))/(B_DOT_X * SQRT_8)) * X_A_X + RT * (SUM_X_LX - SUM_X * LOG(ONE - B_DOT_X))

Function name	Percentage of run-time
MAKE_NODE	15.1
"*"	13.8
WF_PROPAGATE	13.1
XF_PROPAGATE	12.3
WB_PROPAGATE	11.8
XB_PROPAGATE	10.9
"+"	9.6

Line No	Percentage of run-time	Function
386	6.3	"*"
160	4.3	MAKE_NODE
76	5.3	H
146	3.8	MAKE_NODE
665	4.2	WF_PROPAGATE
705	3.6	WB_PROPAGATE
601	3.4	XB_PROPAGATE
...		
365	1.5	"+"

Line No	Statement	
386	IF B.NODE.ID /= B.ID OR A.NODE.ID /= A.ID THEN ERROR	
160	GRAPH.CHAIN.LAST_ALLOC := new GRAPH_NODE	
76	A_X := A_X "+" A(I,J) "*" X(J)	
146	function MAKE_NODE (THIS_KIND : NODE_KIND) return REFNODE	
665	C.WF := C.P1.WF * C.P2.XF + C.P2.WF * C.P1.XF;	{timesk}
705	C.P1.WB := C.P1.WB + C.WB	{plusk}
601	C.P1.XB := C.P1.XB + C.XB	{plusk}
...		
363	C.NODE := A.NODE.XF + B.NODE.XF	

This shows that the order in which the function is evaluated has no effect on the function timings.

8.9 Modified "*" function

The PCA Collector was set for CPU-sampling and compilation was carried out with the optimise option switched off. The "*" function was modified so that internally it was identical to the "+" function. This modification, when applied to the test function, allowed the graph to be built in the normal way but produced a run-time error when the function was being evaluated. Hence the results which follow apply only to the graph building phase.

Function name	**Percentage of run-time**
MAKE_NODE	70.6
"*"	9.2
H	6.6
"+"	4.9
CONSTANT_VALUE	3.9

Line No	**Percentage of run-time**	**Function**
160	4.3	MAKE_NODE
76	5.3	H
146	3.8	MAKE_NODE
386	6.3	"*"

Line No	**Statement**
160	GRAPH.CHAIN.LAST_ALLOC := new GRAPH_NODE
76	A_X := A_X "+" A(I,J) "*" X(J)
146	function MAKE_NODE (THIS_KIND : NODE_KIND) return REF_NODE
386	IF B.NODE.ID /= B.ID OR A.NODE.ID /= A.ID THEN ERROR

While this set of results cannot be compared directly with earlier ones, it shows, at least for the graph building phase of the program, that despite the two overloaded functions being identical the "*" operation stills requires 1.87 times as much CPU-time as that for "+". The following experiment was carried out with the functions modified so that they both performed an identical "*" operation. The two functions were called approximately the same number of times, were identical in their function yet the one labelled "*" requires 1.62 times the CPU-time of "+".

8.10 Modified "+" function

The PCA Collector was set for CPU-sampling and compilation was carried out with the optimise option switched off. The "+" function was modified so that it internally it was identical to the "*" function. This modification, when applied to the test function, allowed the graph to be built in the normal way but produced a run-time error when the function was being evaluated. Hence the results which follow apply only to the graph building phase.

Function name	Percentage of run-time
MAKE_NODE	66.8
"*"	10.7
"+"	6.6

Line No	Percentage of run-time	Function
160	55.7	MAKE_NODE
146	5.3	MAKE_NODE
76	4.9	H
386	3.9	"*"

Line No	Statement
160	GRAPH.CHAIN.LAST_ALLOC := new GRAPH_NODE
76	A_X := A_X "+" A(I,J) "*" X(J)
146	function MAKE_NODE (THIS_KIND : NODE_KIND) return REF_NODE
386	IF B.NODE.ID /= B.ID OR A.NODE.ID /= A.ID THEN ERROR

8.11 Counting function calls in the incomplete function

The PCA Collector was set for Counters, the modified code as in 8.10 above was used and compilation was carried out with the optimise option switched off. The resulting data files were plotted by function name and show that the two arithmetic functions are called approximately the same number of times.

Function name	Percentage of run-time calls
MAKE_NODE	49.7
"+"	16.7
"*"	16.7

8.12 Converting "+" to ADD

The PCA Collector was set for CPU-sampling and compilation was carried out with the optimise option switched off. The "+" function was modified to become ADD so that it no longer relied on operator overloading. Internally it was identical to the "*" function. Once again, this modification, when applied to the test function, allowed the graph to be built in the normal way but produced a run-time error when the function was being evaluated. Hence the results which follow apply only to the graph building phase. The experiment was designed to test whether the PCA was able to differentiate between normal calls to + and calls to the "+" function.

Function name	Percentage of run-time
MAKE_NODE	66.8
"*"	8.9
ADD	7.3

Line No	Percentage of run-time	Function
160	58.5	MAKE_NODE
181	4.1	H
386	3.8	"*"
87	3.5	H
146	2.3	MAKE_NODE

Line No	Statement
160	GRAPH.CHAIN.LAST_ALLOC := new GRAPH_NODE
181	A(I,J) := CONSTANT_VALUE(ITEM(I+J)/100.0)
386	IF B.NODE.ID /= B.ID OR A.NODE.ID /= A.ID THEN ERROR
87	A_XX := A(I<J) "*" X(J) -- part expression added for ADD programming
146	function MAKE_NODE (THIS_KIND : NODE_KIND) return REFNODE

Running the PCA using Counters gave the following breakdown of the number of calls to the various functions:

Function name	Percentage of run-time calls
MAKE_NODE	49.9
"+"	16.7
"*"	16.6

These results look more promising as identical functions now occupy similar amounts of time.

8.13 Optimised compilation with ADD function

The PCA Collector was set for CPU-sampling and compilation was carried out with the optimise option switched on. The ADD function was used in place of the overloaded "+" and the internal construction was returned to addition so that the test function would not give a run_time error. The resulting data files were plotted by function name and by line number.

Function name	**Percentage of run-time**
MAKE_NODE	16.7
"*"	13.7
WF_PROPAGATE	13.0
XF_PROPAGATE	12.2
WB_PROPAGATE	11.8
XB_PROPAGATE	10.7
ADD	9.4

Line No	**Percentage of run-time**	**Function**
160	6.8	MAKE_NODE
386	5.7	"*"
146	3.8	MAKE_NODE
76	4.7	H
705	3.5	WB_PROPAGATE
665	3.5	WF_PROPAGATE

Line No	**Statement**	
160	GRAPH.CHAIN.LAST_ALLOC := new GRAPH_NODE	
386	IF A.NODE.ID /= A.ID OR B.NODE.ID /= B.ID THEN ERROR	
146	function MAKE_NODE (THIS_KIND : NODE_KIND) return REFNODE	
76	A_X := A_X "+" A(I,J) "*" X(J)	
705	C.P1.WB := C.P1.WB + C.WB	{plusk}
665	C.WF := C.P1.WF * C.P2.XF + C.P2.WF * C.P1.XF	{timesk}

With the ADD function replacing the overloaded "+" operation the results showed a ratio of 1.45 between "*" and ADD compared with 1.48 in experiment (3). Similarly in the non-optimised version in the next test compared with experiment (2), the ratios were 1.48 and 1.58 respectively.

8.14 Non-optimised compilation with ADD function

For completeness, the previous experiment was repeated with the compiler set for no optimisation.

Function name	Percentage of run-time
WB_PROPAGATE	14.0
WF_PROPAGATE	12.7
"*"	12.6
MAKE_NODE	12.0
XB_PROPAGATE	11.9
XF_PROPAGATE	11.6
ADD	8.5

Line No	Percentage of run-time	Function
386	5.6	"*"
665	3.9	WF_PROPAGATE
160	3.7	MAKE_NODE
711	3.7	WB_PROPAGATE
705	3.7	WB_PROPAGATE

Line No	Statement	
386	IF A.NODE.ID /= A.ID OR B.NODE.ID /= B.ID THEN ERROR	
665	C.WF := C.P1.WF * C.P2.XF + C.P2.WF * C.P1.XF;	{timesk}
160	GRAPH.CHAIN.LAST_ALLOC := new GRAPH_NODE	
711	C.P1.WB := C.P1 WB + C.WB * C.P2.XF + C.XB * C.P2.WF	{timesk}
705	C.P1.WB := C.P1.WB + C.WB	{plusk}

The results show that the changes have had little effect on run times as the results are almost the same as those in 8.2.

8.15 Assembler code from optimised compilation

As an example of the low level code produced by an instruction manipulating a linked-list, the following line from WF_PROPAGATE has been expanded using the optimise option:

```
line 665     C.WF := C.P1.WF * C.P2.XF + C.P2.WF * C.P1.XF;     {timesk}

             tstl    r7                              -- initial check that offset in R7
             bneq    vcg.10                          -- is non-zero
             jsb     ADA$RAISE_CONSTRAINT
     vcg.10:
             movl    r7,r3                           -- then used to access a second
             tstl    25(r3)                          -- value which is similarly tested
             bneq    vcg.11                          -- before becoming
             jsb     ADA$RAISE_CONSTRAINT
     vcg.11:
             movl    25(r3),r4                       -- the first operand pointer in R4
             tstl    29(r3)                          -- process repeated for second ...
             bneq    vcg.12
             jsb     ADA$RAISE_CONSTRAINT
     vcg.12:
             movl    29(r3),r2                       -- operand pointer in R2
             mulf3   13(r4),5(r2),r5                 -- multiply first indirect values
             mulf3   13(r2),5(r4),r2                 -- multiply second indirect values
             addf3   r5,r2,13(r3)                    -- add then store by indirection
             brb     $CASE_JOIN349
             tstl    r0
```

Actual timings for individual instructions were not available.

The code for executing this Ada statement involves a large number of data manipulation and testing instructions which access the operand pointers and check their validity before the necessary arithmetic is carried out. The requirement for tracking through long chains of pointers to access the data is inefficient and time consuming. This is especially true because the same pointers were used when the graph was constructed so we should know where the operands are and when they will be required again during the reverse pass. The need for some sort of pointer cache or pipeline is indicated here.

9 Discussion

The PCA allows an in-depth analysis to be made of the performance of the reverse accumulation algorithm for automatic differentiation. In particular it records how the execution time is shared between the lines of code. Hence, instructions which take a long time to execute can be identified. As expected, assignment statements which need to access data via long chains of pointers occupy much CPU-time.

However, the results are not conclusive and several anomalies need to be resolved. In particular there appears to be a discrepancy when dealing with the "+" and "*" overloaded operator functions and some time was spent in analysing this situation. The ADA code for "*" is as follows:

```
function "*" (A, B : VARY) return VARY is
    C : VARY;
begin
      if A.NODE.ID /= A.ID OR B.NODE.ID /= B.ID then
         ERROR ("deallocated variable as parameter to VARY operation");
      end if;
    C.NODE :=      MAKE_NODE(TIMESK);              -- (PLUSK) for "+"
    C.ID      :=   C.NODE.ID;
    C.NODE.XF      := A.NODE.XF * B.NODE.XF;       -- + here for "+"
    C.NODE.P1      := A.NODE;
    C.NODE.P2      := B.NODE;
    return C;
end "*";
```

The two functions, "*" and "+", are identical except for the arithmetic assignment to the function value. Both functions contain code which checks that valid parameters have been passed and issues an error message when necessary. It was this part of each function that consumed most time and the amount required did not coincide with the frequency with which each function was called. In all measurements made on the code, it was found that both functions were called with approximately the same frequency and yet the error check for "*" consistently required of the order of fifty percent more time than the error check for "+". This remained the case even when the positions of the functions in the program code were swapped (in case a page boundary always penalised one and not the other) and when the function bodies were changed so that they were identical.

No firm conclusion has been reached as to why these effects occurred and it has not been possible to rule out the option that the effect is only illusory, being a product of the PCA itself. It is clear that a lower level analysis tool is required, which will monitor the movement of the code through the processor and record, amongst other things, the number of cache hits and misses that occur. The PCA is not sufficiently fine-grained to give detailed information regarding the graph construction. However, it has been able to provide some useful information in generally determining where the program is spending time.

It is clear that the algorithm is spending a large proportion of the execution time manipulating the graph. As the majority of the graph handling is effected through linked-lists, it would seem appropriate to consider various ways in which the architecture could provide more support for this type of data structure. In the past, a great deal of effort was expended to provide efficient support for floating point operations. This satisfied an obvious need and is now included in most architectures. The need for linked-list support has, perhaps, not been so obvious but it now appears that it deserves at least as much attention. It is likely that automatic differentiation will be widely adopted by numerical mathematicians once modern

programming languages (e.g., Fortran90, C++) with facilities for operator overloading supersede the established numerical languages (e.g., Fortran77). In general, older languages require a preprocessor to implement automatic differentiation. However, once the new languages are established, a wide range of applications and algorithms, many previously considered infeasible because of the higher derivatives required, are likely to be introduced. Developments in image processing and program code analysis both require large dynamic data structures: image processing for feature extraction, and code analysers to run a program backwards and forwards in order to locate errors. Thus, automatic differentiation is just one of a number of techniques requiring fast and efficient access to dynamic data structures. It should also be pointed out that recent developments in both forward and reverse accumulation require the use of sparse data structures and these, too, are implemented using linked-lists.

In discussion [Chri91], a number of ways of matching the architecture to the algorithm have been suggested. Implicit within the reverse accumulation technique is the need to be able to traverse a computational graph in exactly the opposite order to that in which it was built. Hence we must be able to store each new version of each variable and archive the old version for later. We need some way of storing the sequence of operations executed so that this information can be reversed to accumulate the derivatives.

This work has concentrated on analysing one implementation of reverse accumulation on a conventional architecture and has highlighted the need for architectural support for dynamic data structures. More work is needed to investigate similar algorithms using other languages and on other architectures.

References

[Bart93] M.C. Bartholomew-Biggs, L. Bartholomew-Biggs, D. B. Christianson, Optimisation and Automatic Differentiation in Ada: Some Practical Experience, *to be published*, 1993.

[Chri90] D. B. Christianson, Automatic Hessians by Reverse Accumulation in Ada, unpublished, 1990.

[Chri91] D. B. Christianson, private communication, 1991.

[Chri92a] D. B. Christianson, Automatic Hessians by Reverse Accumulation, IMA Journal of Numerical Analysis, 12, pp 135-150,1992.

[Chri92b] D.B. Christianson, L.C.W. Dixon, Reverse Accumulation of Jacobians in Optimal Control, Technical Report, The University of Hertfordshire, 1992.

[Dixo87] L. C. W. Dixon, Automatic Differentiation and Parallel Processing in Optimisation, Numerical Optimisation Centre Technical Report No. 180, The Hatfield Polytechnic, 1987.

[Dixo90] L. C. W. Dixon, Z. Maany, and M. Mohsenina, Automatic Differentiation of Large Sparse Systems, Journal of Economic Dynamics and Control, 14, North-Holland, 1990.

[Grie89] A Griewank, On Automatic Differentiation, in Mathematical Programming 88, Klewer Academic Publishers, Japan, 1989

[Irim87] M. Iri and K. Kubota, Methods of Fast Automatic Differentiation and Applications, Research Memorandum RMI 87-02, Department of Mathematical Engineering and Information Physics, Faculty of Engineering, University of Tokyo, 1987.

Appendix 1 Non-optimised Compilation Timings

The following list gives full details of the twenty five most time-consuming statements observed during investigation 8.2 in which the PCA Collector was set for CPU-sampling and compilation was carried out with the optimise option switched off. The resulting data files were plotted by function name and by line number.

Line No	**Percentage of run-time**	**Function**
386	5.3	"*"
76	5.1	H
160	4.0	MAKE_NODE
711	3.8	WB_PROPAGATE
665	3.6	WF_PROPAGATE
601	3.5	XB_PROPAGATE
712	3.5	WB_PROPAGATE
705	3.2	WB_PROPAGATE
146	3.1	MAKE_NODE
607	3.0	XB_PROPAGATE
661	2.8	WF_PROPAGATE
554	2.8	XF_PROPAGATE
550	2.5	XF_PROPAGATE
264	2.4	SCRATCH_GRAPH
544	2.4	XF_PROPAGATE
655	2.4	WF_PROPAGATE
608	2.2	XB_PROPAGATE
172	2.0	MAKE_NODE
389	1.7	"+"
355	1.7	"*"
383	1.6	"*"
583	1.6	"*"
706	1.5	WB_PROPAGATE
358	1.4	"+"
391	1.2	"*"
363	1.1	"+"