COMPUTERSYSTEMSLABORATORY

DEPARTMENTS OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
STANFORD UNIVERSITY · STANFORD, CA 94305

Data Buffers for Execution Architectures

Donald Alpert

Technical Report No. 83-250

November 1983

The work described herein was supported in part by the Army Research Office-Durham under contract #DAAG-29-82-K-0109 using facilities supported by NASA under contract NAGW419.

Data Buffers for Execution Architectures

by

Donald Alpert

Technical Report No. 83-250

November 1983

Computer Systems Laboratory

Departments of Electrical Engineering and Computer Science

Stanford University

Stanford, California 94305

Abstract

Directly Executed Language (DEL) architectures are derived from idealized representations of high-level languages. DEL architectures show dramatic reduction in the number of instructions and memory references executed when compared to traditional architectures, offering the potential for designing highly cost-effective DEL processors. This report explains the design considerations for the data buffer in a DEL microprocessor. Simulation techniques were used to evaluate the performance of different sized buffers for a set of Pascal test programs. The results show that **a** buffer with 256 words typically faults on less than 5% of storage allocations.

Key Words and Phrases: buffer memory, contour memory, directly executed language, DEL, execution architecture, microprocessor design

Table of Contents

List of Figures

List of Tables

1. Introduction

During the past decade many studies have identified the inefficiencies of traditional computer architectures for supporting applications programmed in high-level languages. The objects and operators specified by high-level language programs have no direct representation in the machine resources of registers and ALU that define traditional architectures. Execution of even a simple high-level statement requires a proliferation of instructions to position the data in registers and store results back to memory. For IBM/370 and DEC PDP-11 architectures about one in four instructions perform a function required by the high-level source program; the remaining instructions are overhead introduced by the architecture [3].

Directly Executed Languages (DELs) are a class of architectures originally developed by Michael Flynn and Lee Hoevel of Stanford University [4]. DEL architectures directly represent the objects and operators specified in a high-level language program in a form that is theoretically minimal, but still permits ready execution. The contour memory is an innovative technique introduced in DEL architectures to represent high-level source programs efficiently. The contour is a table containing the value or address of each object referenced in a source procedure.

To understand better the cost/performance tradeoffs of implementing a contour buffer for a DEL microprocessor, we have studied the contour sizes for a set of Pascal test programs and evaluated the performance of different sized contour buffers by tracing execution of the programs. This report contains the statistical results of that study, along with discussion of other buffer design considerations. The contour buffer is also compared with related data buffering techniques described in the literature.

2. DEL Principles

In a DEL architecture the operations exactly correspond to the operators specified in the programming language, for example integer and floating point arithmetic, array address calculation, complex arithmetic for FORTRAN, and set operations for Pascal. The operands in a DEL architecture are objects (i.e., variables, constants, and labels) named in the source program. For each subroutine or procedure in the program a table, known as the ***contour, is*** used to access the DEL operands. Every object referenced in the source procedure has an entry in the contour to hold the value or address of the corresponding DEL operand The contour entry holds the value for simple local variables, constants, and labels; the address is used for non-local variables, arrays, and records, all of which are indirectly accessed The name contour is derived from Johnston's [5] use of the term to model the effects of lexical scoping in block-structured languages.

2.1 lnst ruction Representation

The close correspondence between the source program and its DEL representation makes compilation straightforward. Every operator in the source program generates a single DEL instruction, which consists of several bit-packed fields specifying the operands, operation, and format. Figure 2-l shows the representation of a DEL instruction with the fields packed from right to left.

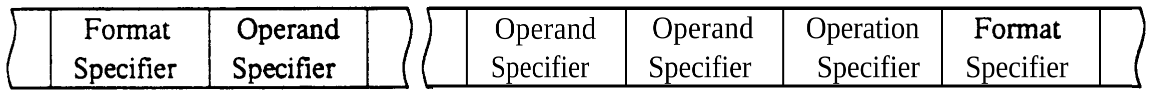

Figure 2-1: DEL Instruction Representation

The format specifies the number, order, and use of operands in the instruction. The DEL instruction formats have two properties, called ***transformational completeness,*** that minimize the number of operand fields in an instruction. First, an operand is specified only once in an instruction even if it serves more than once as source or destination. Second, an expression evaluation stack is implicitly used to avoid introducing temporary operands that are not mentioned in the source program. For example, Table 2-1 lists the transformationally complete set of formats for dyadic operators. The most general case

$$a+b \rightarrow c$$

is represented in stack architectures with four instructions.

```
push a
push b
add
pop c
```

A simple, register-oriented architecture replaces the pushes with register loads and the pop with **a** register store. Three-address architectures capture the case above concisely; but, the evaluation of complex expressions or statements with repeated identifiers, such as a+ a $\rightarrow$ b, introduces significant redundancy. The DEL formats have the advantages of stack architectures for evaluating expressions and the advantages of one, two, and three-address architectures for reducing instruction count.

Format	Transformation	No. Explicit Operands
F1	A op B $\rightarrow$ C	3
F2	A op B $\rightarrow$ B	2
F3	A op B $\rightarrow$ A	2
F4	A op A $\rightarrow$ B	2
F5	A op A $\rightarrow$ A	1
F6	A op B $\rightarrow$ T	2
F7	A op T $\rightarrow$ B	2
F8	T op A $\rightarrow$ B	2
F9	T op A $\rightarrow$ A	1
F10	A op T $\rightarrow$ A	1
F11	A op A $\rightarrow$ T	1
F12	A op T $\rightarrow$ T	1
F13	T op A $\rightarrow$ T	1
F14	T op T $\rightarrow$ A	1
F15	T op U $\rightarrow$ A	1
F16	T op U $\rightarrow$ U	0

A, B, and C are explicit operands.
T is the top of stack; U is below T in the stack

Table 2-1: Transformationally Complete Dyadic Formats

The number of bits used to encode a field in a DEL instruction is $\lceil \log_2(N) \rceil$, where N is the number of distinct field values, This is the minimum number of bits required without resorting to frequency encoding. Thus, the DEL representation for the source program is minimal because only explicit operators and objects from the source program appear in DEL instructions, and the instruction fields are fully packed.

2.2 Example

For an example of DEL representation, consider the following procedure in a Pascal program

```
procedure p( a: integer;  var b: integer);
        begin
        b:=(b+a)*3;
        end;
```

The DEL contour for the procedure (see Figure 2-2) has three entries for the parameters ***a*** and ***b*** and the constant 3. The single statement in the procedure body generates two DEL instructions.

```
F6 a b +            (a+b → T)
F8 3 b *            (T*3 → b)
```

Compare this to IBM/370, which uses six instructions.

```
L   8,b             address of b → register 8
L   7,0(8)          value of b → register 7
A   7,a             a+b → register 7
LA  5,3(0)          3 → register 6
MR  6,5             (a+b)*3 → register 7
ST  7,0(8)          register 7 → b
```

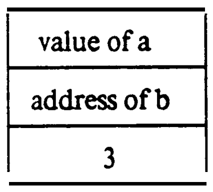

Figure 2-2: DEL Contour

2.3 Evaluation

DEL architectures have been developed for FORTRAN (DELtran [4]) and Pascal (Adept [10]). The Adept architecture was compared against IBM/370, HP1000, and P-Code for a set of five Pascal test programs. The programs were FFT, Kalman filter, puzzle (an unofficial benchmark developed by Forest Baskett), quicksort, and a program that compiles and interprets a subset of Pascal. Table 2-2 shows average results for several measures of the programs.

By eliminating the overhead instructions of the traditional architectures, the DEL architecture has reduced the static program size by a factor of2 and the number of instructions executed by a factor

Measure	Adept	IBM/370	HP1000	P-Code
code size	1	2.40	2.22	3.60
instructions executed	1	3.42	3.55	3.57
instruction bytes fetched	1	3.29	2.57	4.12
data bytes fetched	1	5.82	4.80	5.81
data bvtes stored	1	20.15	12.56	9.19

Table 2-2: Architectural Measures Relative to Adept

of 3, while lowering the bandwidth for fetching instructions by a factor of 3. In addition, the DEL contour has reduced main memory bandwidth for data traffic by a factor of 5.

3. Contour Memory

The favorable results of emulating DEL architectures encourage the belief that DEL processors can be highly cost-effective. Consequently, an effort has recently begun to design a 32-bit DEL microprocessor using a 4μm nMOS technology.

The contour memory provides one of the major performance advantages for DEL architectures compared to traditional architectures by reducing the data memory traffic. In a DEL microprocessor the contour memory serves to locate most of the operand references on-chip. Thus, slower references to main memory are avoided, and the memory bus is free for fetching instructions. Recognizing the significance of the contour memory, we decided to study the contour implementation for a DEL microprocessor. The purpose of the study is twofold: define the operations involved in allocating and releasing contour storage and determine the amount of storage required to implement the contour efficiently.

3.1 Contour and Structure Stacks

The contour contains a single word to store the value or address of each object named in a procedure of the high-level source program, several words of linkage information, and a stack of temporary values for expression evaluation. This contour organization applies to DELs for block-structured languages, such as Pascal, C, and Ada

When a procedure is called during execution of a program, storage is allocated for the procedure's contour on the ***contour stack.*** If any structured variables, such as arrays, records, and sets, are declared in the procedure, then it is also necessary to allocate storage for these variables on a second stack, the ***structure stack.*** For each structured variable, the contour contains the address where its value is stored in the structure stack. Roth of these stacks are located in the memory address space of the DEL processor. (Although the contour is most often referenced using an operand specifier from an instruction, it is important that the contour stack be addressable in memory because the address of a contour entry is required for reference parameters and non-local references.)

The DEL processor contains three pointers for maintaining the contour and structure stacks (Figure 3-l). The Environment Pointer (EP) contains the address of the first contour entry, which is also the address immediately below the linkage area A contour entry is located by subtracting the operand specifier field in an instruction from EP. Top (T) contains the lowest address allocated for the contour, which may be part of evaluation stack. T is decremented before every push and incremented after every pop. The Structure Stack Pointer (SSP) contains the lowest address allocated in the structure stack

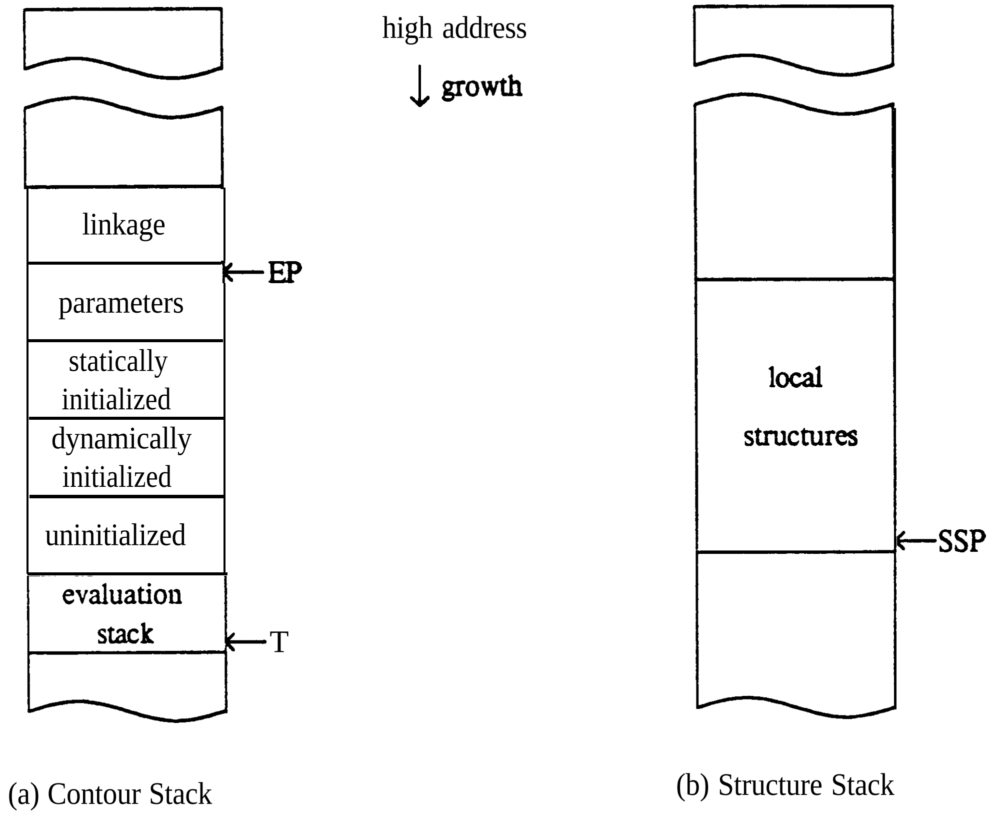

Figure 3-1: Contour and Structure Stacks

3.2 Contour Allocation

When a CALL instruction is executed, several operations are necessary before control is transferred to the called procedure: The linkage area is allocated and initialized; the actual parameters are passed; the statically initialized contour entries are copied from the header of the called procedure; the dynamically initialized contour entries are evaluated using descriptors in the header of the called procedure; storage for the uninitialized contour entries is allocated; and, storage for the called procedure's structured variables is allocated.

The linkage area includes the following information, which is required to return to the caller.

- Instruction Pointer (IP), which points to the instruction following the CALL
- value of EP for the caller (the ***dynamic link)***
- address of the most recently allocated contour for the immediately nesting procedure ***(the static link)***
- value of SSP
- static nesting level
- field width for operand specifiers

A DEL architecture for the language C does not require the static link or static level because nesting procedures is not permitted.

The actual parameters are passed by storing the value or address of entries in the caller's contour into the callee's contour. The parameters are specified as operands for the CALL instruction. The header of the called procedure specifies the number of parameters and contains a bit for each parameter specifying whether call-by-value or call-by-reference is used

The procedure header also contains copies of the statically initialized contour entries, the values of which are known when the program is compiled or linked The contour entries for constants, global variables, and labels (other than procedure parameters) are statically initialized by copying their values from the procedure header.

Some contour entries must be initialized with values calculated when the program is executed Entries containing the address of a local structured variable or non-local variables are calculated using descriptors in the procedure header. (Non-local variables are those declared outside the current procedure, but not at the outermost, global level.) The descriptor specifies whether the value of the variable is in the contour or structure stack, the difference in nesting level between the called procedure and the procedure in which the variable is declared, and the displacement of the variable's value from EP or SSP.

Contour entries for simple local variables are uninitialized. The procedure header specifies the number of uninitialized contour entries and the quantity of storage to allocate for local structures, as well as the procedure's static level and the field width for operand specifiers.

When a RETURN instruction is executed, IP, EP, SSP, and other linkage information are restored. Functions return a value from the callee's contour, which is pushed onto the caller's evaluation stack

4. Experiment and Test Programs

Six Pascal test programs were selected to gather data about contour sizes and to simulate the performance of a contour buffer (see Table 4-l). Bench is a computation-intensive, composite benchmark. Bench consists of several smaller programs collected by John Hennessy of Stanford University, including Towers of Hanoi, eight queens, matrix multiplication, and others. Ccal is an interactive desk calculator program written by Warren Cory of Stanford University. Compare is a program, written by James F. Miner of the University of Minnesota that compares two files and reports the differences. Macro is the macro expansion preprocessor for the SCALD computer-aided design system developed by T. M. McWilliams and L. C. Widdoes of Stanford University [6]. Pasm is the P-Code assembler for the Stanford Emulation Laboratory, written by Donald Alpert [1]. Pcomp is a compiler from Pascal to P-Code, written by Sasan Hazeghi of the Stanford Linear Accelerator Center. In total, the test programs represent 16,696 lines of code and 403 contours,

Program	Number of Contours	Number of Lines	Use
bench	48	9 3 3	composite benchmark
ccal	27	799	interactive calculator
compare	28	968	file comparison
macro	179	7484	macro processor
pasm	17	1112	P-code assembler
pcomp	104	5400	Pascal to P-code compiler

Table 4-1: Pascal Test **Programs**

The first step in analyzing the test programs was modification of a compiler from Pascal to U-Code [7] so that it generated pseudo-comments reporting the sizes for several storage regions allocated by the compiler. When the test programs are compiled to U-Code with the modified compiler, the contents of each contour can be readily determined A program was developed to gather statistics about contour sizes, and the results are reported in the Section 5.

(There is one minor source of innacuracy in determining the contour sizes by examining the U-Code version of a program rather than the Pascal source. The Pascal compiler for a DEL architecture must allocate storage for simple temporary variables in three cases: ***with*** statements, for statements where the final value is not constant, and value parameters that are calculated on the expression stack. The compiler can allocate one location for several temporary uses if the different values are not simultaneously live. When analyzing the U-Code, the storage for the first two types of temporary values is allocated independently from storage for the third type. Thus, there may be more temporary storage locations counted than necessary, although the effect on the reported results is negligible.)

Using the contour statistics gathered in one pass through the U-Code version of a test program, a modified U-Code version was produced in a second pass. The modified U-Code version was identical to the original except that each call and return of the source program was supplemented with instructions to produce a run-tune trace of contour allocation. The modified U-Code version was then translated to an object module and executed. After executing the programs with contour tracing, the trace records were analyzed by a program that simulated contour buffers of varying size. These dynamic contour statistics are reported in Section 6.3.

The five test programs other than bench required input during execution. . Ccal was used to calculate the mean and standard deviation of the integers between 1 and 100. Compare was used to compare two versions of the pasm program that differed in 12 lines. Macro was used to expand 46 lines of macro definitions for one portion of a processor. Pcomp was used to compile the pasm program, then the P-Code output was assembled by pasm itself.

5. Contour Sizes

The six test programs were analyzed to determine contour sizes. Table 5-1 shows separately for each program the mean contour size and the mean number of entries in the different contour regions. The linkage region is not included in the results because its size is constant for all contours and because its size depends only on the implementation, not the source programs. The size reported for the evaluation stack is the maximum possible stack depth. Two averages are reported for the entire test set. The program average is calculated by giving equal weight to each of the program means. The contour average is calculated by giving equal weight to each of the 403 contours in the set of test programs.

Program	Mean Number of Contour Entries					
	Total	Params	Static Init	Dynamic Init	Uninit	Eval Stack
bench	20.3	1.3	15.2	0.2	2.0	1.6
ccal	33.6	1.1	28.2	0.1	2.9	1.3
compare	24.2	1.2	18.8	0.6	2.2	1.4
macro	29.2	1.5	22.6	0.4	3.4	1.3
pasm	60.5	0.5	50.5	3.8	4.4	1.3
pcomp	51.5	0.8	42.8	1.5	5.2	1.2
program average	36.5	1.1	29.7	1.1	3.3	1.3
contour average	35.1	1.2	28.2	0.8	3.6	1.3

Table 5-l: Contour Sizes

As shown in Table 5-1, the mean contour size is 36.5 entries when the program means are equally weighted or 35.1 entries when the contours are weighted equally. (In the first case the standard deviation of contour size is 16.0, in the second case 36.3.) Preliminary evaluation of the DEL microprocessor design indicates that the contour memory buffer occupies more area than any other functional unit. Hence, several ways of reducing contour size are considered.

Examining the use of contour entries in more detail shows that 30.4% of entries are for labels. Most of the labels are used for short branches to implement simple control constructs like ***if-then, while***, or ***repeat***. The contour entries for branch labels can be eliminated if the DEL architecture is extended to include branch instructions, that contain a field specifying the branch displacement relative to the Instruction Pointer. Procedure labels for CALL instructions would still be represented using an operand specifier for a contour entry. Table 5-2 shows the effect of eliminating branch labels on contour size.

Program	Mean Number of Contour Entries		
	Branch Labels	Branch Displacements	Percent Reduction
bench	20.3	15.5	23.7
ccal	33.6	24.3	27.8
compare	24.2	19.5	19.4
macro	29.2	23.5	19.3
pasm	60.5	42.2	30.2
pcomp	51.5	39.0	24.4
program average	36.5	27.3	25.2
contour average	35.1	27.1	22.9

Table 5-2: Effect of Branch Displacements on Contour Size

With the elimination of branch labels the mean contour size is reduced by 25.2% to 27.3 entries when the program means are weghted equally. When the contours are weighted equally, the mean contour size is reduced by 22.9% to 27.1 entries.

Placing branch displacements in the instruction complicates the implementation because a mechanism must be included for extracting the displacement field and adding it to IP. The time to calculate the branch destination can also add to the delay when a branch is taken. With branch labels, all operands (including branch destinations) are specified directly in the contour or indirectly through the contour. Despite the additional complication and possible performance degradation, the dramatic reduction in contour size argues convincingly for the use of branch displacements. Adept [10], a DEL architecture for Pascal, uses branch displacements; but DELtran [4], a DEL architecture for FORTRAN, uses branch labels.

Using branch displacements rather than branch labels has other implications for the DEL architecture and computer system. Code can be position independent, making relocation and sharing easier. And, the linker needs to initialize many fewer values in the procedure headers. The rest of this report assumes branch displacements are used.

In addition to the mean contour size, another concern is maximum contour size. If a contour is larger than the storage on the DEL microprocessor chip, then either the program cannot execute or performance is severely degraded. For three of the programs (bench, ccal, and macro) the largest contour was for the outermost (global) contour. We believe that the largest contour for a Pascal program is often at the outermost level because Pascal requires that global variables be declared within the scope of the program body; external variables are not allowed Simple variables are often declared globally in Pascal programs because their values are shared by several procedures, although the variables are not referenced in the main program body. Storage for variables that are declared within a procedure but are not referenced locally can be moved from the contour stack to the structure stack, thus reducing the size of the contour. Where those variables are referenced in nested procedures, the contour contains the variable's address. Table 5-3 shows the effects of eliminating unreferenced variables from the contour.

Program	Global Contour Size		Maximum Contour Size	
	With Unreferenced Variables	Only Referenced Variables	With Unreferenced Variables	Only Referenced Variables
bench	89	73	89	73
ccal	117	110	117	110
compare	40	36	58	58
macro	166	74	166	125
pasm	28	15	145	145
pcomp	126	68	129	129

Table 5-3: Maximum Contour Size

Although the size of the global contour has been reduced for some programs, the maximum contour size is not significantly changed. The test programs, and likely Pascal programs in general, contain initialization procedures that reference almost all the global variables. Again, this is because of Pascal's restrictions; variables cannot be initialized at compile-time. For example, the largest contour in the test program pasm is for a procedure that initializes all of the P-Code instruction mnemonics. Note also that a one-pass compiler has difficulty in relocating storage for unreferenced simple variables from the contour stack to the structure stack; references to variables are found in nested procedures before it is known whether the variables are locally referenced. In conclusion, we recommend that simple variables be stored in the contour of the procedure where they are declared, whether or not they are referenced locally.

The contour size can be effectively reduced by recognizing that only a portion of the evaluation stack is in use at most times. The contour buffering algorithm described in Section 6.2 allocates only the required portion of the evaluation stack

6. Contour Buffer

When a DEL processor executes a CALL instruction, storage is allocated on the contour stack for the called procedure's contour the storage is released when the corresponding RETURN instruction is executed. The allocation and release of storage on the contour stack was simulated for execution of the six test programs, and Table 6-1 summarizes the results. The simulation assumed that the contour linkage area contains 4 words. (The format for the linkage area is explained in Section 6.2.) Also, the values reported for contour size do not include the evaluation stack. Instead, at each call the number of entries actively in use for the caller's evaluation stack was allocated before the callee's contour on the contour stack

Program	Number of Contours	Contour. Size		Contour Stack Depth	
		Mean	Maximum	Mean	Maximum
bench	316655	12.8	92	253.1	540
ccal	17003	18.5	120	185.9	483
compare	10506	22.1	60	106.7	172
macro	9666	24.3	169	362.5	534
pasm	15172	49.5	149	194.8	325
pcomp	55983	33.1	133	1198.7	2756

Table 6-1: Contour Allocation During Program Execution

For the test programs, the contour stack grew to a maximum depth of 2756 entries. Of course, for other programs the stack can grow even deeper, and the DEL architecture should not place unreasonable constraints on the size of the contour stack. Consequently, it is impractical to store the entire contour stack in a single-chip microprocessor. It is possible to implement the contour memory and CPU in separate chips, though doing so increases system ‘cost and reduces performance because of longer access time to the contour. Rather, a DEL microprocessor should integrate a contour buffer memory. The buffer, though smaller than the maximum contour stack, contains the contour entries most frequently referenced by the processor.

Several approaches are possible for implementing the contour buffer. An associative cache can be used for the buffer, but with two major disadvantages: storage for the address tags is expensive, and the tag comparison introduces considerable delay to the critical access time for each operand. Nevertheless, the performance of a contour cache in conjunction with a cache for instructions is being studied. The approach taken here is to implement the contour buffer as a random-access memory that stores the top portion of the contour stack

When a contour is allocated, if the buffer is too full to store the new contour (buffer overflow), then the oldest contour entries in the buffer are copied to the contour stack in memory until sufficient room is available in the buffer for the new contour. After a contour is released, if the buffer is missing some entries from the top contour on the stack (buffer underflow), the missing entries are copied from memory to the buffer. If the contour on top of the stack is too large to fit in the buffer, the CPU flushes the buffer and executes using the contour stack in memory. Other techniques for supporting contours larger than the buffer are considered in Section 6.4.

6.1 Call Runs

The contour buffer is effective when few memory references are typically required for each contour allocation and release. However, if the program executes a long sequence of CALLS without an intervening RETURN, the buffer becomes quickly filled and then overflows repeatedly. Similarly, if the program executes a long sequence of RETURNS with no intervening CALL, the buffer becomes soon emptied and then underflows repeatedly. This aspect of program calling behavior was examined before developing a detailed simulation of contour buffer performance.

We define a *call run* to be the execution of a RETURN instruction followed by the execution of one or more CALL instructions up to the next RETURN. The length of a call run is the number of CALLS in the run. Table 6-2 shows the distribution of call run lengths for the test programs. The runs are typically short. None of the programs had a mean run length larger than 2. For each of the test programs, at least 75% of the runs contained only 1 or 2 CALLS. Hence, long sequences of contour allocations without intervening releases are not expected to degrade buffer performance.

6.2 Buffering Algorithm

The simulated contour buffer implements an array of N words at locations 0,1,...,N-1. The contour stack entry at memory address i is directly mapped to buffer location i mod N. The CPU includes three registers that store pointers used for contour allocation and buffer maintanence: the Environment Pointer (EP), Top (T), and Buffer Pointer (BP). EP and T were previously described in Section 3.1. BP contains the maximum address of any contour entry contained in the buffer. Figure

Cumulative Frequency for Call Run Lengths					
Program	Run Length				
	1	2	3	4	5
bench	.655	.878	.928	.951	,961
ccal	.477	.774	.930	.975	.987
compare	.223	.993	.998	1.000	1.000
macro	.776	.949	.988	.997	1.000
pasm	.563	.999	.999	1.000	1.000
pcomp	.585	.885	.955	.965	.990

Table 6-2: Distribution of Call Run Lengths

6-1 shows the contour buffer and two possible configurations for EP, T, and BP. Notice that the buffer wraps around, with location 0 following location N-l, so that it is possible for contour entries at lower memory addresses in the contour stack to map into higher locations in the buffer (see Figure 6-lb).

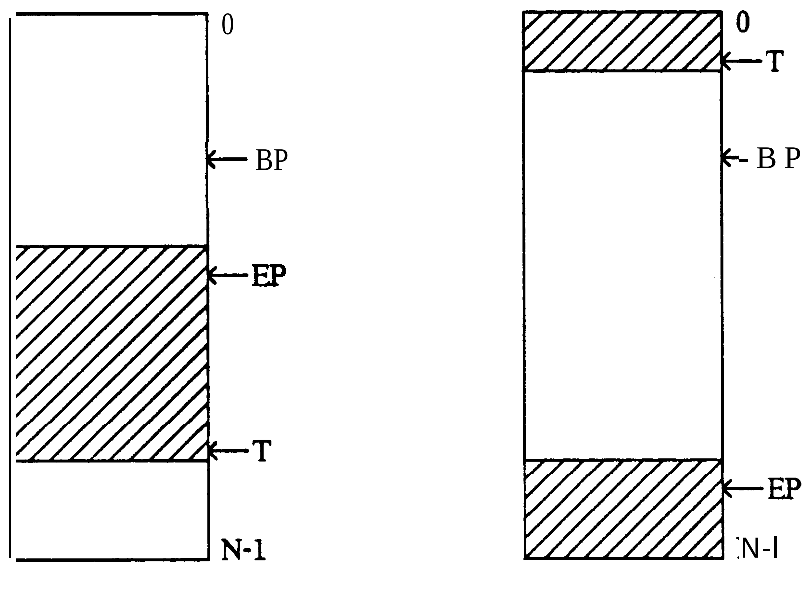

Figure 6-1: Contour Buffer

The linkage area of the contour (previously described in Section 3.2) contains four words with the format shown in Figure 6-2. The static and dynamic links are represented in only half a word because the high-order address bits of the contour stack are fixed. Section 6.4 explains that locating the contour in a reserved memory segment simplifies support for memory-mapped references to the contour buffer. The linkage area also contains a mode bit that specifies whether the CPU is operating with the contour in the buffer or in memory.

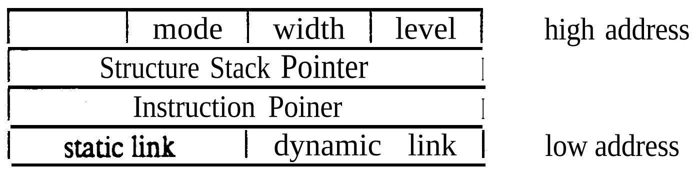

Figure 6-2: Linkage Format

The operations performed in executing a CALL instruction were explained in Section 3.2. Only the CPU's actions for maintaining the contour buffer are considered here. First, the CPU tests whether the callee's contour, including its maximum evaluation stack specified in the procedure header, can fit into the buffer. By reserving sufficient room for the maximum evaluation stack at this point, the buffer can only overflow when a CALL is executed. If the contour can fit, then the CPU checks whether the buffer overflows and, if necessary, copies some old buffer entries to memory and adjusts BP. If the contour cannot fit, the CPU flushes all the valid buffer entries, adjusts BP, and operates from the contour stack in memory.

When a RETURN instruction is executed, the CPU tests whether the caller's operating mode locates the contour in the buffer or memory. If the contour is located in the buffer and some of its entries have overflowed to memory, then the CPU copies the missing entries from memory and adjusts BP. Figure 6-3 shows the detailed buffering algorithm expressed in a Pascal-like syntax.

6.3 Buffer Performance

The contour buffering algorithm was simulated for buffers with 64, 128, 256, 512, and 1024 words. Three performance measures were defined and calculated for the buffers: the buffer fault ratio, the buffer traffic per contour, and the buffer traffic *ratio*. ***The buffer fault mtio*** is defined to be the fraction of allocated contours that cause the buffer to overflow. In calculating the buffer fault ratio, contours larger than the buffer count as overflows. The simulation results, shown in Table 6-3, indicate that a buffer with 256 words overflows on less than 3% of contour allocations for all programs except pcomp. The program average shown in the table is the mean buffer fault ratio

```
const
        LINKAGE = size-of-linkage-area;
        N = size-of-buffer;
        MAXMEMORY = size_of_memory;
        DYNAMICLINK= displacement-of-dynamiclink-from-ep;
        STATICLINK= displacement-of-staticlink-from-ep:
        MODE = displacement-of-mode-from-ep:
type
        address = 0..MAXMEMORY-1;
        bufferlocation = 0..N-1;
        contourmodes = (inbuffer,inmemory);
var
        buffer: array[bufferlocation] of word:
        memory: array[address] of word;
        ep,t,bp: address;
        contourmode: contourmodes;

procedure alloc(contoursize,maxstackdepth: integer):
var   i,temp: address;
begin
temp := ep;
ep := t - LINKAGE - 1;
t := ep - contoursize + 1:
if contourmode = inmemory then bp := t - 1;
if (LINKAGE + contoursize + maxstackdepth) <= N then
        begin (* new contour can fit in buffer *)
        if (bp - N) >= (t - maxstackdepth) then
                begin (* buffer overflow, copy to memory *)
                for i := bp down to (t - maxstackdepth + N) do
                        memory[i] := buffer[i mod N];
                bp := t - maxstackdepth + N - 1;
                end:
        buffer[(ep + DYNAMICLINK) mod N] := temp;
        buffer[(ep + MODE) mod N] := contourmode;
        contourmode := inbuffer;
        end
else begin (* new contour cannot fit in buffer *)
        (* flush valid entries from buffer *)
        for i := bp downto ep + LINKAGE + 1 do
                memory[i] := buffer[i mod N];
        memory[ep + DYNAMICLINK]:= temp;
        memory[ep + MODE] := contourmode;
        contourmode := inmemory;
        bp := t - 1:
        end;
end:
```

Figure 6-3: Contour Buffering Algorithm

```
procedure release:
var   i: address;
begin
t := ep + LINKAGE + 1;
if contourmode = inbuffer then
      begin
      contourmode := buffer[(ep + MODE) mod N];
      ep := buffer[(ep + DYNAMICLINK) mod N];
      end
else begin (• contourmode = inmemory *)
      contourmode := memory[ep + MODE];
      ep := memory[ep + DYNAMICLINK]
      bp := t - 1;
      end:
if (contourmode = inbuffer) and (bp < ep + LINKAGE) then (*underflow*)
      begin (* copy top of contour stack from memory to buffer *)
      for i := bp + 1 to ep + LINKAGE do
            buffer[i mod N] := memory[i];
      bp := ep + LINKAGE;
      end;
end;
```

Figure 6-3, continued

with the programs equally weighted (The contour average is not shown because bench allocated so many more contours than the other programs.) Note that the number of overflows and underflows can differ; however, the numbers never differed by more than a factor of 2 for any of the programs and buffer sizes. The smallest buffers show some anomalous results: they can have better buffer fault ratios than larger buffers. These anomalies can be explained by recognizing that when a contour larger than the buffer is allocated, the buffer is flushed, so subsequent allocations are less likely to overflow. Although these anomalies are interesting to note, they occur only for buffer sizes with unacceptable performance.

The second measure of buffer performance is the buffer traffic per contour, which is defined to be the number of words transferred between the buffer and memory (on overflows and undefflows) divided by the number of contours allocated. The buffer traffic per contour and buffer fault ratio measures can be used with a model of a DEL processor's performance to estimate the time penalty and memory bandwidth for maintaining the contour buffer. Table 6-4 shows the buffer traffic for each of the test programs and simulated buffers. On average, a buffer with 256 words requires an extra 1.8 words transferred per CALL instruction for maintaining the contour buffer.

The buffer traffic ratio is the most revealing measure for the contour buffer in absence of a model for DEL processor performance. Even with an infinite buffer, some memory references are

Buffer Fault Ratio					
Program	Buffer Size				
	64	128	256	512	1024
bench	.308	.147	.025	.000	.000
ccal	.417	.621	.015	.000	.000
compare	.220	.001	.000	.000	.000
macro	.345	.148	,010	.000	.000
pasm	.304	.651	.000	.000	.000
pcomp	.412	.438	.154	.041	,006
program average	.334	.334	.034	.007	.001

Table 6-3: Buffer Fault Ratio

Buffer Traffic per Contour					
Program	Buffer Size				
	64	128	256	512	1024
bench	7.09	3.75	0.54	0.00	0.00
ccal	9.93	18.8	0.48	0.00	0.00
compare	7.18	0.01	0.00	0.00	0.00
macro	11.98	3.43	0.26	0.01	0.00
pasm	15.79	26.50	0.01	0.00	0.00
pcomp	23.71	25.93	9.49	2.54	0.34
program average	12.61	13.06	1.80	0.43	0.06

Table 6-4: Buffer Traffic per Contour

required to initialize the contour when a CALL is executed. The buffer traffic ratio is defined to be the number of words transferred between the buffer and memory (on overflows and underflows) divided by the number of memory words fetched for the statically and dynamically initialized contour regions. Table 6-5 shows that on average a buffer with 256 words requires less than 10% more memory references for contour allocation than an infinite buffer would.

On the basis of the simulation results, it appears that a buffer with 256 words provides good performance. A buffer with 128 words may be acceptable for a low-performance processor. Several other implementation issues remain to consider.

Buffer Traffic Ratio					
Program	Buffer Size				
	64	128	256	512	1024
bench	1.37	0.72	0.11	0.00	0.00
ccal	0.86	1.63	0.04	0.00	0.00
compare	0.54	0.00	0.00	0.00	0.00
macro	0.71	0.20	0.02	0.00	0.00
pasm	0.41	0.68	0.00	0.00	0.00
pcomp	0.93	1.02	0.37	0.10	0.01
program average	0.67	0.71	0.09	0.02	0.00

Table 6-5: Buffer Traffic Ratio

6.4 Implementation Issues

One trouble for the buffer design is supporting contours larger than the buffer. We propose that the CPU have an operating mode in which the contour is located in memory. Introducing this operating mode is a straightforward solution with other advantages for improved testing and availability, as the CPU can run at degraded performance with the on-chip buffer disabled, Other techniques are possible, such as treating the allocation of large contours as a run-time error or mapping only a portion of large contours into the buffer. Treating the allocation of large contours as an error is unacceptable for a commercial implementation, although it is appropriate for a first, experimental design. Mapping a portion of large contours into the buffer presents a number of problems. If the lower contour entries are mapped to the buffer, then the entire buffer must be swapped to memory whenever a contour is allocated or released on top of a large contour, possibly causing excessive traffic. If the upper contour entries are mapped to the buffer, then it is difficult to distinguish contour references in the buffer from those in memory. In any case, if large contours occur frequently, performance can only be poor. None of the test programs allocated more than 2 contours larger than 128 words.

The DEL microprocessor should be able to handle memory-mapped contour references located in the on-chip buffer. There are three ways that a contour entry can be referenced by its memory address - when a simple local variable is a reference parameter for a procedure call, when a simple variable is referenced in a nested procedure, or when following the chain of static links in executing a CALL instruction. It is possible for a compiler to identify the first two cases and to relocate the storage for the simple variable's value from the contour stack to the structure stack; however, this is difficult for a one-pass compiler. It is preferable that the implementation detect memory references to the portion of the contour stack stored in the buffer. For read references, the CPU can initiate the memory transaction concurrently with examining the buffer. When necessary, the buffer's data bypasses the memory's data For write references, the CPU can also initiate the memory transaction concurrently with examining the buffer. The data is written to memory and, when necessary, to the buffer as well.

For the purpose of determining whether memory references map to the contour buffer, it is helpful to reserve a segment of the processor's logical address space for the contour stack For example, if memory addresses are 32 bits, contour addresses can have 1s in the 16 most significant bits. Although this limits the size of the contour stack, the limitation is not of practical concern. Whenever memory is referenced, the CPU checks whether the upper address bits are all ls. (This can be done using a single gate.) If so, the CPU checks whether the address is less than the contents of BP, in which case the address maps into the buffer. The function is summarized below.

```
function  address-maps-to-buffer(memory_address:  address:
              var bufferloc:  bufferlocation):  boolean;
begin
if (memory-address > 0xFFFF0000) and (memory-address <= BP) then
      begin
      address-maps-to-buffer := true;
      bufferloc := (memory-address mod N);
      end
else address-maps-to-buffer := false;
```

Thus far, the discussion of buffer implementation has concentrated on execution of a single high-level program and has ignored considerations of the system environment, such as multiprocessing. Because the design of a DEL computer system involves many issues beside the contour buffer, only a brief and general treatment is presented here. Primarily, it is necessary to copy the valid buffer contents to memory when the running process is forced to wait. This can be accomplished either as part of an instruction that saves the entire process context or by software, if access to BP, EP, and T registers is made available. The software routine would read the buffer contents using memory-mapped addresses and then write back to the same addresses with the data stored through to memory. Additionally, it is necessary to copy the top contour from the stack in memory to the buffer when a process is selected to run. This can be implemented as part of an INTERRUPT RETURN instruction, sharing the microcode for RETURN.

To conclude this discussion of contour buffer implementation, note that many of the buffer operations, such as locating a contour entry using an operand specifier, handling buffer overflow and underflow, and mapping memory addresses to the contour, involve modulus calculations. Often these calculations do not require division. For example, to locate a contour entry using an operand specifier, the buffer location of the first entry (EP mod N) can be calculated whenever CALL is executed, and saved in a register. Still, if the buffer size is a power of 2, all of the modulus operations are easily performed by truncation. We recommend that the buffer size be a power of 2.

7. Related Research

The study of DEL contour memory presented in this report is related to the work of several other researchers. Scott Wakefield, in earlier developing Adept, treated many of the issues concerning contour memory discussed here. Other researchers have studied techniques for effectively buffering data in a VLSI microprocessor. Although they were also concerned with supporting high-level programming languages, their approaches differed from DELs. All of these approaches, however, have one goal in common: The buffer size should be transparent to the architecture, so that improvements in technology can be exploited without modifying software.

Wakefield developed an experimental DEL system, including definition of the architecture, implementation of a Pascal compiler, and microcoding an interpreter in the Stanford Emulation Laboratory [10]. Wakefield's work extends initial research by Michael Flynn and Lee Hoevel [4], expanding their concepts of a DEL for FORTRAN to one for a block-structured language. Although Wakefield measured some characteristics of contour size and referencing behavior in his broad treatment of the Adept architecture, our study provides a more comprehensive test workload and more detailed investigation of contour implementation.

Richard Sites recognized the potential of VLSI for integrating large quantities of short-term data memory with a CPU [9]. He compared the advantage of general-purpose registers, which have very fast access but require the compiler to control their use, with an associative cache memory, which is slower than registers but largely invisible to compilation strategy. He recommended implementing a circular buffer for multiple register sets. In this respect his approach is similar to the contour buffer: a register set is allocated at each call and released at each return. But, the architecture supports a fixed number of registers, unlike the DELs' variable number of contour entries that exactly matches the requirements of the source program

David Patterson and Carlo Sequin have directed the RISC (Reduced Instruction Set Computer) project at U. C. Berkeley [8]. Their approach to VLSI processor design is the antithesis of the DEL approach. They advocate reducing the functionality of the architecture to vertical microcode; the intention is to make the processor simple and fast. Their small processor is coupled with a register stack buffer, using a buffering scheme similar to Sites's. However, their design offers two improvements over Sites's proposal: some registers are globally accessible to all procedures and other registers, used to pass parameters and return values, overlap between the caller and callee. Coincidentally, the RISC I microprocessor integrates 138 registers, roughly equal to the number of contour entries recommended for a DEL microprocessor.

David Ditzel and H. R McLellan describe a proposed stack cache for the Bell Laboratories' C Machine [2]. The stack cache stores the top elements of the stack of activation records in a circular buffer. Although the stack-cache buffering algorithm is similar to the DEL contour buffer, the stack cache stores structures as well as simple variables Storing structures in the stack buffer makes the buffer larger, increases the size of operand specifiers in the instructions, and forces the buffer to be byte-addressable. Also, the C machine architecture includes instructions for explicitly maintaining the buffer. For example, the CATCH instruction is always the first instruction executed after returning from the callee, informing hardware to fill the buffer on underflow. Their results show that a buffer with 1024 words provides acceptable performance.

8. Conclusions

The research described in this report shows that an economical data buffer of 256 words can efficiently support the contour memory for a DEL microprocessor. A buffer with 256 words typically handles more than 95% of contour allocations without overflowing to memory. For low-performance processors, a buffer with 128 entries may be suitable. The contour buffer is significantly more effective if the DEL architecture represents short branches with displacement fields in instructions rather than with label operands in the contour.

We are expanding the simulation of contour allocation to develop more complete traces of DEL referencing activity and to further evaluate DEL memory hierarchies. Our work on contour buffer design is coordinated with implementation studies of other DEL microprocessor functional units; the goal is to integrate a single-chip CPU.

References

[1] Donald Alpert.
A Pascal P-Code Interpreter for the Stanford Emmy.
Technical Note 164, Computer Systems Laboratory, Stanford University, September, 1979.

[2] David R. Ditzel and H. R McLellan.
Register Allocation for Free: The C Machine Stack Cache.
In *Proceedings, Symposium on Architectural Support for Programming Languages and Operating Systems,* pages 48-56. March, 1982.

[3] Michael J. Flynn.
Directions and Issues in Architecture and Language.
Computer 13(10):5-22, October, 1980.

[4] Michael J. Flynn and Lee W. Hoevel.
Execution Architecture: The DELtran Experiment.
IEEE Transactions on Computers C-32(2):156-175, February, 1983.

[5] John B. Johnston.
The Contour Model of Block Structured Processes.
Sigplan Notices 6:55-82, February, 1971.

[6] T. M. McWilliams and L. C. Widdoes, Jr.
SCALD: Structured Computer-Aided Logic Design.
Technical Report 152, Digital Systems Laboratory, Stanford University, March, 1978.

[7] Peter Nye.
U-Code An Intermediate Language for Pascal and Fortran.*
S-l Document PAIL-8, Stanford University, May, 1982.

[8] David A. Patterson and Carlo H. Sequin.
RISC I: A Reduced Instruction Set VLSI Computer.
In *Conference Proceedings, The 8th Annual Symposium on Computer Architecture,* pages 443-458. May, 1981.

[9] Richard L. Sites.
How To Use 1000 Registers.
In Caltech Conference On VLSI, pages 527-532. January, 1979.

[10] Scott Wakefield.
Studies in Execution Architectures.
Technical Report No. 237, Computer Systems Laboratory, Stanford University, January, 1983.