Technical Report
CMU/SEI-87-TR-021
ESD-TR-87-172

Timing Variation in Dual Loop Benchmarks

Neal Altman
Nelson Weiderman

October 1987

Technical Report
CMU/SEI-87-TR-21
ESD-TR-87-172
October 1987

Timing Variation in Dual Loop Benchmarks

Neal Altman
Member of the Technical Staff
Ada Embedded Systems Testbed Project

Nelson Weiderman
Project Leader
Ada Embedded Systems Testbed Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Timing Variation in Dual Loop Benchmarks

Abstract

Benchmarks that measure time values using a standard system clock often employ a dual loop design. One of the important assumptions of this design is that textually identical loop statements will take the same amount of time to execute. This assumption was tested on two bare computers with Ada® test programs and has been demonstrated to be inaccurate in these specific test cases.

1. Dual Loop Benchmarks: Purpose and Assumptions

Benchmarks are tests designed to measure the capabilities of a computer system. They are used to compare different computer systems and determine the suitability of a computer system for particular tasks. Benchmarks show a wide variation in what they are intended to measure, how they are designed, and how they are implemented. Many benchmarks produce outputs that are measurements of the time required to perform some task. A common technique is to write a program that performs some interesting bit of work (e.g., a rendezvous) sandwiched between calls to a system timer.

Benchmarks that use time as a unit of measure vary widely in the time they require to run. Some tasks are brief and can take fractional parts of a second to execute. Others measure durations of minutes or hours. The ability of computer clocks to measure this range of times also varies widely. A system clock available to a benchmark designer may be accurate only to a tenth of a second, far too slow to measure an event in the millisecond or microsecond range. More accurate timing devices are often available, but as an option rather than as a standard component. For benchmarks intended for general use, a **dual loop benchmark** design is often used to permit the benchmark to execute on an unmodified system.

Dual loop benchmarks handle the problem of imprecise clocks by extending the duration of the test to a length that the clock can readily measure. The time required for a test is extended by repeating the test numerous times between calls to the system timer. Repetition is usually programmed by inserting the test in a loop, where the number of repetitions may be conveniently changed. The increased time duration of the test series can be measured easily, and the time for the individual test can be determined by computing the average value for the test series. Introducing a loop construct into the test adds time, which must be factored out. This is done with a second loop, a control loop, which contains only the loop construct and not the actual test. The time required for the benchmark is assumed to be the value obtained by subtracting the control loop time from the test loop time. An Ada skeleton for a dual loop benchmark appears as follows:[1]

[1]This Ada program fragment requires that the constant SOME_VALUE and a procedure TEST be added before the program can actually be run. In actual practice, precautions must be taken to ensure optimization by the compiler does not alter the essential program structure. For example, because the empty loop contains no executable statements, it might be removed by a compiler.

```
with CALENDAR; use CALENDAR;

procedure DUAL_LOOP_EXAMPLE is

NUMBER_OF_TESTS         : constant INTEGER := SOME_VALUE;
START_TEST              : CALENDAR.TIME;
STOP_TEST               : CALENDAR.TIME;
START_CONTROL           : CALENDAR.TIME;
STOP_CONTROL            : CALENDAR.TIME;
AVERAGE_TIME            : DURATION;

begin

-- Test loop
START_TEST := CALENDAR.CLOCK;
for INDEX1 in 1..NUMBER_OF_TESTS loop
          TEST;                  -- Test.
end loop;
STOP_TEST := CALENDAR.CLOCK;

-- Control loop
START_CONTROL := CALENDAR.CLOCK;
for INDEX1 in 1..NUMBER_OF_TESTS loop
    null;                         -- No Test.
end loop;
STOP_CONTROL := CALENDAR.CLOCK;

AVERAGE_TIME := ((STOP_TEST - START_TEST) -
                        (STOP_CONTROL - START_CONTROL))
                        / NUMBER_OF_TESTS;
end DUAL_LOOP_EXAMPLE;
```

A critical assumption made by the dual loop benchmarking scheme is that textually equivalent code constructs require the same amount of time to execute. In other words, time required by the loop constructs and control loops are substantially identical.

2. Testing the Validity of the Dual Loop Design for Timing Benchmarks

The assumption that textually equivalent loops require similar amounts of time was subjected to test with Ada compilers for two bare machines: a DEC MicroVAX II^{TM} computer using VAXELN^{TM} Ada Ver. 1.1, and a Motorola 68020 single board computer using Systems Designers' SD Ada-Plus^{TM} Ver. 2B.01. Times were obtained using the routines in package CALENDAR. For DEC Ada, SYSTEM.TICK was 0.01 seconds. For SD Ada-Plus, SYSTEM.TICK was approximately 0.0078 seconds (actually @Math{2@up(-7)} seconds).

The test program, CAL2, used the format of the Ada dual loop skeleton, but it increased the number of loops to five. Each loop was inserted into a procedure. The source code for each loop was made as similar as possible. Only the names of the procedures containing the test loops were allowed to differ between loops. By calling the procedures in different sequences, the order of loop execution was varied (e.g., first order, 1-2-3-4-5; next order, 5-4-3-2-1; then, 2-5-1-3-4). This variation tested the hypothesis that the execution time for a loop may be affected by the run sequence. Rather than using completely empty loops, a call to a single subroutine was placed in each loop, and appropriate checks were made to ensure that the subroutine call was not optimized out of the loop by the Ada compiler during program translation. The light loading factor was imposed arbitrarily, but it matched the test loop of a benchmark measuring subroutine call overhead. Output of test results was initiated only after testing was complete. There were two versions of CAL2, one for VAXELN and one for the 68020, reflecting the differences in the I/O packages available under the two compilers. The VAXELN version is included as Appendix C.a (page 19), and the 68020 version as Appendix C.b (page 22). Note that the 68020 version uses the package TARGET_IO rather than TEXT_IO.

The assembly language translations produced by each compiler were examined. The code for the loops proved to be identical except for names of variables, procedures, and labels.

The programs were run three times on each target and showed a consistent pattern. Times for individual loops were consistent, while times between loops showed noticeable variation (Tables 2-1 and 2-2). The timings were sensitive to the number of loop iterations, the exact format of the loop, the location of program code in memory, and other factors.[2] The examples included here show a maximum difference between loops of about 12%. The raw output is included in Appendix B.

CAL2 showed a consistent pattern of variation on each of the tested systems. The MicroVAX/VAXELN Ada combination showed one "slow loop" and four "fast loops" with similar times. The 68020/SD-Ada combination showed two "slow loops" and three "fast loops." Again, the two categories of loops shared similar times. The order of execution of the individual loops had no effect on the times.

[2] Complete information is provided in [1].

		Loop 1	Loop 2	Loop 3	Loop 4	Loop 5	Variation
Trial A	Mode	4.38	4.38	4.91	4.38	4.38	.53 (12.1%)
	Mean	4.38	4.38	4.91	4.38	4.38	.53 (12.1%)
	Range	.01	.02	.01	.01	.01	
	(20 samples, 100,000 iterations/loop)						
Trial B		Loop 1	Loop 2	Loop 3	Loop 4	Loop 5	Variation
	Mode	4.37	4.37	4.91	4.37	4.37	.54 (12.4%)
	Mean	4.37	4.37	4.91	4.37	4.37	.54 (12.4%)
	Range	.02	.01	.01	.01	.01	
	(20 samples, 100,000 iterations/loop)						
Trial C		Loop 1	Loop 2	Loop 3	Loop 4	Loop 5	Variation
	Mode	4.37	4.37	4.91	4.37	4.37	.54 (12.4%)
	Mean	4.37	4.37	4.91	4.37	4.37	.54 (12.4%)
	Range	.02	.01	.00	.01	.01	
	(20 samples, 100,000 iterations/loop)						

Table 2-1: CAL2 Test Results from VAXELN Ada (time in seconds)

The cause of the variation in times was analyzed. For the MicroVAX, testing established that the loop position in memory was the critical factor. The virtual memory space of the MicroVAX is divided into 512-byte pages, which correspond to identically sized physical pages. The slow loop happened to span a page boundary and consequently ran more slowly due to the overhead inherent in shifting between pages; the loop changed as the program size changed. A suggestion that the variation was caused by the byte alignment of individual loops with respect to the four-byte MicroVAX word was considered, but the byte alignment of the loops was identical (compared to the start of word boundaries).

The 68020 processor accessed memory by word (four bytes), while the SD-Ada compiler placed the loop statement without regard to word boundaries. As a consequence, certain loops were aligned more advantageously and required fewer memory accesses to execute.

		Loop 1	Loop 2	Loop 3	Loop 4	Loop 5	Variation
Trial A	Mode	2.055	2.258	2.055	2.312	2.055	.256 (12.5%)
	Mean	2.054	2.259	2.055	2.310	2.054	.256 (12.5%)
	Range	.008	.008	.008	.007	.008	
	(20 samples, 100,000 iterations/loop)						

		Loop 1	Loop 2	Loop 3	Loop 4	Loop 5	Variation
Trial B	Mode	2.055	2.250	2.055	2.312	2.055	.257 (12.5%)
	Mean	2.055	2.250	2.055	2.310	2.055	.255 (12.4%)
	Range	.008	.008	.008	.007	.000	
	(20 samples, 100,000 iterations/loop)						

		Loop 1	Loop 2	Loop 3	Loop 4	Loop 5	Variation
Trial C[3]	Mode	2.086	2.062	2.133	2.055	2.180	.125 (6.1%)
	Mean	2.084	2.066	2.133	2.055	2.181	.126 (6.1%)
	Range	.008	.031	.000	.000	.008	
	(20 samples, 100,000 iterations/loop)						

Table 2-2: CAL2 Test Results from SD ADA-Plus on the 68020 (time in seconds)

[3]Minor changes to the source code forced the recompilation of CAL2 for Trial C. Note the difference in times when Trial C is compared to Trials A and B. The source code for Trial C is included in Appendix C.

3. Conclusion

It is not clear that the variation observed in these examples will be seen on all systems or that some variation in loop timings is sufficient to completely invalidate the technique. However, practitioners who simply prepare and run dual loop benchmarks without validation may garner results that are not accurate. This source of variation appears to be dependent on the specific hardware/software combination under test; thus, the amount of variation will vary depending upon the hardware, the system software, the format of the benchmark, and the specific load points selected by the interaction of these components. As a consequence, the accuracy of a dual loop benchmark depends upon a highly specific set of circumstances and cannot be controlled by a general technique when the benchmark is written.

Dual loop benchmarking is based on the assumption that the time taken to execute two textually identical loops will be substantially identical. Simple tests have demonstrated that textually identical loops exhibit substantial variation in execution time on specific test systems. The consequence of this variation is that benchmark programs using the dual loop paradigm to measure the execution time of a particular Ada feature (such as a subroutine call) can and do produce negative values. The positive values produced by such test suites can be erroneously accepted as accurate despite unbounded relative errors.

References

[1] Altman, Neal.
Factors Causing Unexpected Variation in Ada Bechmarks.
Technical Report CMU/SEI-87-TR-22, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213, October, 1987.

Appendix A: Specific Configurations Tested

A.a. MicroVAX/VAXELN

System Type:	MicroVAX II (two identical configurations, SEIYB and SEIYC)
Manufacturer:	Digital Equipment Corporation
Processor:	KA-630
Peripherals:	Console terminal, KWV11 real-time clock; DRV11J parallel interface
Ada Compiler:	DEC VAX™ Ada Ver. 1.3-23 (under MicroVMS™ Ver. 4.5); VAXELN Ada Ver. 1.1 (under MicroVMS Ver. 4.5)
Run Time:	VAXELN, Ver. 2.3; VAXELN Ada, Ver. 1.1
Vendor:	Digital Equipment Corporation

A.b. MC68020/SD-Ada

System Type:	MVME™133 single board processor in Motorola VME bus enclosure
Manufacturer:	Motorola Microsystems
Processor:	MC68020, 12.5 Mhz.
Peripherals:	Console terminal, two RS232 host connections
Ada Compiler:	SD Ada-Plus VMS™ x 68020, Release 2B.01 (under MicroVMS Ver. 4.5)
Run Time:	SD-Ada VMX® x 68020, Release 2B.01
Vendor:	Systems Designers plc.

Appendix B: Raw Data

B.a. CAL2 for the MicroVAX/VAXELN

```
CAL2--Multiple executions of identical loops--time in seconds:

Run on SEIYB on 4/6/87.  Build parameters were:
characteristic /nofile /noserver /debug=none
program CAL2 /kernel_stack=40 /user_stack=40 /job_priority=0 -
        /process_priority=0 /argument=("CONSOLE:",  -
        "25""NA XXXXXXXX""::PS:[NA.REASON_T.CAL2]CAL2_VAXELN.LOG", "CONSOLE:")
device XQA /register=%O774440 /vector=%O120 /priority=4


Test #  LOOP_1  LOOP_2  LOOP_3  LOOP_4  LOOP_5  Calling Order
     1    4.38    4.39    4.91    4.38    4.38  1-2-3-4-5
     2    4.38    4.37    4.91    4.38    4.38  5-4-3-2-1
     3    4.37    4.38    4.91    4.38    4.38  2-5-1-3-4
     4    4.37    4.38    4.91    4.38    4.38  4-1-5-2-3
     5    4.37    4.38    4.91    4.38    4.37  1-2-3-4-5
     6    4.37    4.38    4.91    4.38    4.38  5-4-3-2-1
     7    4.38    4.38    4.91    4.37    4.38  2-5-1-3-4
     8    4.38    4.38    4.91    4.38    4.38  4-1-5-2-3
     9    4.37    4.38    4.91    4.38    4.38  1-2-3-4-5
    10    4.38    4.38    4.91    4.37    4.38  5-4-3-2-1
    11    4.38    4.38    4.91    4.38    4.37  2-5-1-3-4
    12    4.37    4.38    4.91    4.38    4.38  4-1-5-2-3
    13    4.38    4.37    4.91    4.38    4.38  1-2-3-4-5
    14    4.38    4.38    4.91    4.37    4.38  5-4-3-2-1
    15    4.38    4.38    4.91    4.38    4.37  2-5-1-3-4
    16    4.38    4.38    4.91    4.37    4.38  4-1-5-2-3
    17    4.37    4.38    4.91    4.38    4.37  1-2-3-4-5
    18    4.38    4.38    4.91    4.38    4.38  5-4-3-2-1
    19    4.38    4.37    4.91    4.38    4.38  2-5-1-3-4
    20    4.38    4.37    4.90    4.38    4.38  4-1-5-2-3
```

```
CAL2--Multiple executions of identical loops--time in seconds:

Run on SEIYC on 5/29/87.  Build parameters were:
characteristic /nofile /noserver /debug=none
program CAL2 /kernel_stack=40 /user_stack=40 /job_priority=0 -
        /process_priority=0 /argument=("CONSOLE:",  -
        "25""NA XXXXXXXX""::PS:[NA.REASON_T.CAL2]CAL2_VAXELN.LOG", "CONSOLE:")
device XQA /register=%O774440 /vector=%O120 /priority=4


Test #  LOOP_1  LOOP_2  LOOP_3  LOOP_4  LOOP_5  Calling Order
     1    4.39    4.38    4.92    4.38    4.38  1-2-3-4-5
     2    4.37    4.37    4.91    4.37    4.37  5-4-3-2-1
     3    4.37    4.37    4.91    4.37    4.37  2-5-1-3-4
     4    4.37    4.37    4.91    4.37    4.37  4-1-5-2-3
     5    4.37    4.37    4.91    4.37    4.37  1-2-3-4-5
     6    4.37    4.37    4.91    4.37    4.37  5-4-3-2-1
     7    4.37    4.37    4.91    4.37    4.37  2-5-1-3-4
     8    4.37    4.37    4.91    4.37    4.37  4-1-5-2-3
     9    4.37    4.37    4.91    4.37    4.37  1-2-3-4-5
    10    4.37    4.37    4.91    4.37    4.37  5-4-3-2-1
    11    4.37    4.37    4.91    4.37    4.37  2-5-1-3-4
    12    4.37    4.37    4.91    4.37    4.37  4-1-5-2-3
    13    4.37    4.37    4.91    4.37    4.37  1-2-3-4-5
    14    4.37    4.37    4.91    4.37    4.37  5-4-3-2-1
    15    4.37    4.37    4.91    4.37    4.37  2-5-1-3-4
    16    4.37    4.37    4.91    4.37    4.37  4-1-5-2-3
    17    4.37    4.37    4.91    4.37    4.37  1-2-3-4-5
    18    4.37    4.37    4.91    4.37    4.37  5-4-3-2-1
    19    4.37    4.37    4.91    4.37    4.37  2-5-1-3-4
    20    4.37    4.37    4.91    4.37    4.37  4-1-5-2-3
```

```
CAL2--Multiple executions of identical loops--time in seconds:

Run on SEIYC on 5/29/87.  Build parameters were:
characteristic /nofile /noserver /debug=none
program CAL2 /kernel_stack=40 /user_stack=40 /job_priority=0 -
        /process_priority=0 /argument=("CONSOLE:",  -
        "25""NA XXXXXXXX""::PS:[NA.REASON_T.CAL2]CAL2_VAXELN.LOG", "CONSOLE:")
device XQA /register=%O774440 /vector=%O120 /priority=4


Test #  LOOP_1  LOOP_2  LOOP_3  LOOP_4  LOOP_5  Calling Order
     1    4.39    4.38    4.91    4.38    4.38  1-2-3-4-5
     2    4.37    4.37    4.91    4.37    4.37  5-4-3-2-1
     3    4.37    4.37    4.91    4.37    4.37  2-5-1-3-4
     4    4.37    4.37    4.91    4.37    4.37  4-1-5-2-3
     5    4.37    4.37    4.91    4.37    4.37  1-2-3-4-5
     6    4.37    4.37    4.91    4.37    4.37  5-4-3-2-1
     7    4.37    4.37    4.91    4.37    4.37  2-5-1-3-4
     8    4.37    4.37    4.91    4.37    4.37  4-1-5-2-3
     9    4.37    4.37    4.91    4.37    4.37  1-2-3-4-5
    10    4.37    4.37    4.91    4.37    4.37  5-4-3-2-1
    11    4.37    4.37    4.91    4.37    4.37  2-5-1-3-4
    12    4.37    4.37    4.91    4.37    4.37  4-1-5-2-3
    13    4.37    4.37    4.91    4.37    4.37  1-2-3-4-5
    14    4.37    4.37    4.91    4.37    4.37  5-4-3-2-1
    15    4.37    4.37    4.91    4.37    4.37  2-5-1-3-4
    16    4.37    4.37    4.91    4.37    4.37  4-1-5-2-3
    17    4.37    4.37    4.91    4.37    4.37  1-2-3-4-5
    18    4.37    4.37    4.91    4.37    4.37  5-4-3-2-1
    19    4.37    4.37    4.91    4.37    4.37  2-5-1-3-4
    20    4.37    4.37    4.91    4.37    4.37  4-1-5-2-3
```

B.b. CAL2 for the MC68020/SD-Ada

```
Date: Friday, 24 April 1987 10:43:30 EST
From: John.Slusarz@sei.cmu.edu
To: na@sei.cmu.edu

*** Note:  Leading zeros added to fractional portions of times which
           required them.  This is a fix of the output problem with the
           original version of CAL2_SD.  NWA  5/28/87  ***

CAL2_SD--Multiple executions of identical loops--time in seconds:
Test #  LOOP_1  LOOP_2  LOOP_3  LOOP_4  LOOP_5  Calling Order
     1   2.055   2.258   2.055   2.312   2.055  1-2-3-4-5
     2   2.055   2.258   2.055   2.312   2.055  5-4-3-2-1
     3   2.055   2.258   2.055   2.312   2.055  2-5-1-3-4
     4   2.055   2.258   2.055   2.312   2.055  4-1-5-2-3
     5   2.055   2.258   2.055   2.312   2.055  1-2-3-4-5
     6   2.055   2.258   2.055   2.312   2.055  5-4-3-2-1
     7   2.055   2.258   2.055   2.312   2.055  2-5-1-3-4
     8   2.055   2.258   2.055   2.312   2.055  4-1-5-2-3
     9   2.055   2.258   2.055   2.312   2.055  1-2-3-4-5
    10   2.055   2.258   2.055   2.312   2.055  5-4-3-2-1
    11   2.055   2.258   2.055   2.312   2.055  2-5-1-3-4
    12   2.047   2.258   2.055   2.305   2.047  4-1-5-2-3
    13   2.055   2.258   2.055   2.312   2.055  1-2-3-4-5
    14   2.055   2.266   2.055   2.305   2.055  5-4-3-2-1
    15   2.055   2.258   2.055   2.312   2.055  2-5-1-3-4
    16   2.055   2.258   2.047   2.305   2.055  4-1-5-2-3
    17   2.047   2.258   2.055   2.305   2.047  1-2-3-4-5
    18   2.055   2.258   2.055   2.305   2.047  5-4-3-2-1
    19   2.055   2.266   2.055   2.305   2.055  2-5-1-3-4
    20   2.055   2.258   2.055   2.312   2.055  4-1-5-2-3

Another run :
**********
```

```
CAL2_SD--Multiple executions of identical loops--time in seconds:
Test #  LOOP_1  LOOP_2  LOOP_3  LOOP_4  LOOP_5  Calling Order
     1   2.055   2.250   2.055   2.312   2.055  1-2-3-4-5
     2   2.055   2.250   2.055   2.312   2.055  5-4-3-2-1
     3   2.055   2.250   2.055   2.312   2.055  2-5-1-3-4
     4   2.055   2.250   2.055   2.312   2.055  4-1-5-2-3
     5   2.055   2.250   2.055   2.312   2.055  1-2-3-4-5
     6   2.047   2.250   2.055   2.305   2.055  5-4-3-2-1
     7   2.055   2.250   2.055   2.305   2.055  2-5-1-3-4
     8   2.055   2.250   2.055   2.312   2.055  4-1-5-2-3
     9   2.055   2.250   2.055   2.312   2.055  1-2-3-4-5
    10   2.055   2.250   2.055   2.312   2.055  5-4-3-2-1
    11   2.055   2.250   2.055   2.312   2.055  2-5-1-3-4
    12   2.055   2.250   2.055   2.312   2.055  4-1-5-2-3
    13   2.055   2.250   2.055   2.312   2.055  1-2-3-4-5
    14   2.055   2.250   2.047   2.305   2.055  5-4-3-2-1
    15   2.055   2.250   2.055   2.312   2.055  2-5-1-3-4
    16   2.055   2.250   2.055   2.305   2.055  4-1-5-2-3
    17   2.055   2.258   2.055   2.305   2.055  1-2-3-4-5
    18   2.055   2.250   2.055   2.312   2.055  5-4-3-2-1
    19   2.055   2.250   2.055   2.312   2.055  2-5-1-3-4
    20   2.055   2.250   2.055   2.312   2.055  4-1-5-2-3
```

```
Date: Thursday, 28 May 1987 13:28:54 EDT
From: John.Slusarz@sei.cmu.edu
To: na@sei.cmu.edu

**********
CAL2_SD--Multiple executions of identical loops--time in seconds:
Test #  LOOP_1  LOOP_2  LOOP_3  LOOP_4  LOOP_5  Calling Order
     1   2.086   2.055   2.133   2.055   2.180  1-2-3-4-5
     2   2.078   2.062   2.133   2.055   2.188  5-4-3-2-1
     3   2.078   2.086   2.133   2.055   2.188  2-5-1-3-4
     4   2.086   2.062   2.133   2.055   2.180  4-1-5-2-3
     5   2.086   2.062   2.133   2.055   2.180  1-2-3-4-5
     6   2.086   2.062   2.133   2.055   2.180  5-4-3-2-1
     7   2.086   2.086   2.133   2.055   2.180  2-5-1-3-4
     8   2.086   2.078   2.133   2.055   2.180  4-1-5-2-3
     9   2.078   2.070   2.133   2.055   2.180  1-2-3-4-5
    10   2.086   2.086   2.133   2.055   2.180  5-4-3-2-1
    11   2.086   2.055   2.133   2.055   2.180  2-5-1-3-4
    12   2.086   2.062   2.133   2.055   2.180  4-1-5-2-3
    13   2.086   2.055   2.133   2.055   2.180  1-2-3-4-5
    14   2.078   2.062   2.133   2.055   2.180  5-4-3-2-1
    15   2.078   2.055   2.133   2.055   2.188  2-5-1-3-4
    16   2.086   2.062   2.133   2.055   2.180  4-1-5-2-3
    17   2.078   2.086   2.133   2.055   2.180  1-2-3-4-5
    18   2.086   2.055   2.133   2.055   2.180  5-4-3-2-1
    19   2.086   2.062   2.133   2.055   2.180  2-5-1-3-4
    20   2.086   2.055   2.133   2.055   2.180  4-1-5-2-3

loop alignment data :

loop 1  line 77 :       E82
loop 2  line 94 :       1004
loop 3  line 111 :      1186
loop 4  line 128 :      1308
loop 5  line 145 :      148A


Loops 2 and 4 have starting alignment on 32 bit boundary
Loops 1,3,5 have starting alignment not on 32 bit
```

Appendix C: Test Programs

C.a. CAL2 Source Code for the MicroVAX/VAXELN

```
--
-- =========================================================================
-- CAL2 is a benchmark calibration routine intended to test the assumption
-- that textually identical loops will take (approximately) the same amount
-- of time to execute.
--
-- The routine was devised to verify that benchmarks which depend on a
-- dual test and control loop structure will execute correctly on the
-- target system.
--
-- Test format is to call five functions (LOOP_1..LOOP_5) executed in
-- succession.  Each function returns a DURATION value, obtained using
-- the Ada CALENDER.CLOCK routine.  The time is obtained by subtracting
-- the time as the routine is entered from the time just
-- prior to the return to caller.  Between the two calls, a tight loop is
-- executed LOOP_REPETITIONS times.  The loop contains a single call to
-- procedure PROC.  PROC simply serves to place a light load in each of
-- the timing loops.
--
-- The test calls are made in a number of arbitrary orders to allow
-- detection of any effects relating to the total number of machine
-- cycles, as opposed to the ordering of the LOOP routines.
--
-- The test sequence is executed TEST_REPETITIONS times to allow for system
-- intialization effects (and possibly interruptions during execution).
--
-- Results are output at the conclusion of all tests.
--
-- Programming notes:
--    o The package T_ROUTINE contains a small routine PROC, which simply
--      assigns a fixed value to the single integer argument, ARG.  It is
--      isolated in a package to prevent its being optimized to an inline
--      assignment.
--
-- Known bugs:
--      <none>
--
-- Who  Date            Remarks
-- ---  ----            -------
-- NWA  16 June 87      Corrected comments.
-- NWA  4 April 87      Adapted from bencharking test routine TEST_9.
-- =========================================================================

package T_ROUTINE is
    procedure PROC(ARG: in out INTEGER);
end T_ROUTINE;

with T_ROUTINE; use T_ROUTINE;
with CALENDAR; use CALENDAR;
with TEXT_IO; use TEXT_IO;

procedure CAL2 is

    package TIME_IO is new FIXED_IO(DURATION); use TIME_IO;
    package INT_IO is new INTEGER_IO(INTEGER); use INT_IO;

    LOOP_REPETITIONS:   constant INTEGER := 100000;
    TEST_REPETITIONS:   constant INTEGER := 5;
    SEQUENCE_COUNT:     constant INTEGER := 4;
    LOOP_COUNT:         constant INTEGER := 5;

    LOOP_TIMES:         array (1..TEST_REPETITIONS, 1..SEQUENCE_COUNT,
                                1..LOOP_COUNT) of DURATION;

    SEQUENCE_LENGTH:    constant INTEGER := (LOOP_COUNT * 2) - 1;
    CALLING_SEQUENCE:   array (1..TEST_REPETITIONS, 1..SEQUENCE_COUNT) of
                            STRING(1..SEQUENCE_LENGTH);


-- ----------------------------------------------------------------------

    function LOOP_1 return DURATION is

        START_TIME:     TIME;
        END_TIME:       TIME;
        A_VALUE:        INTEGER := 12;

    begin
        START_TIME := CLOCK;
        for INDEX in 1..LOOP_REPETITIONS loop
            PROC(A_VALUE);
        end loop;
        END_TIME := CLOCK;
        return END_TIME - START_TIME;
    end LOOP_1;

-- ----------------------------------------------------------------------

    function LOOP_2 return DURATION is

        START_TIME:     TIME;
        END_TIME:       TIME;
        A_VALUE:        INTEGER := 12;

    begin
        START_TIME := CLOCK;
        for INDEX in 1..LOOP_REPETITIONS loop
            PROC(A_VALUE);
        end loop;
        END_TIME := CLOCK;
        return END_TIME - START_TIME;
    end LOOP_2;

-- ----------------------------------------------------------------------

    function LOOP_3 return DURATION is

        START_TIME:     TIME;
        END_TIME:       TIME;
        A_VALUE:        INTEGER := 12;

    begin
        START_TIME := CLOCK;
        for INDEX in 1..LOOP_REPETITIONS loop
            PROC(A_VALUE);
        end loop;
        END_TIME := CLOCK;
        return END_TIME - START_TIME;
    end LOOP_3;

-- ----------------------------------------------------------------------

    function LOOP_4 return DURATION is

        START_TIME:     TIME;
        END_TIME:       TIME;
        A_VALUE:        INTEGER := 12;

    begin
        START_TIME := CLOCK;
        for INDEX in 1..LOOP_REPETITIONS loop
            PROC(A_VALUE);
        end loop;
        END_TIME := CLOCK;
        return END_TIME - START_TIME;
    end LOOP_4;

-- ----------------------------------------------------------------------
```

```
    function LOOP_5 return DURATION is

        START_TIME:     TIME;
        END_TIME:       TIME;
        A_VALUE:        INTEGER := 12;

    begin
        START_TIME := CLOCK;
        for INDEX in 1..LOOP_REPETITIONS loop
            PROC(A_VALUE);
        end loop;
        END_TIME := CLOCK;
        return END_TIME - START_TIME;
    end LOOP_5;

-- ---------------------------------------------------------------------------

begin
    for CURRENT_TEST in 1..TEST_REPETITIONS loop

-- Calling sequence one:
        CALLING_SEQUENCE(CURRENT_TEST, 1) := "1-2-3-4-5";
        LOOP_TIMES(CURRENT_TEST, 1, 1) := LOOP_1;
        LOOP_TIMES(CURRENT_TEST, 1, 2) := LOOP_2;
        LOOP_TIMES(CURRENT_TEST, 1, 3) := LOOP_3;
        LOOP_TIMES(CURRENT_TEST, 1, 4) := LOOP_4;
        LOOP_TIMES(CURRENT_TEST, 1, 5) := LOOP_5;

-- Calling sequence two:
        CALLING_SEQUENCE(CURRENT_TEST, 2) := "5-4-3-2-1";
        LOOP_TIMES(CURRENT_TEST, 2, 5) := LOOP_5;
        LOOP_TIMES(CURRENT_TEST, 2, 4) := LOOP_4;
        LOOP_TIMES(CURRENT_TEST, 2, 3) := LOOP_3;
        LOOP_TIMES(CURRENT_TEST, 2, 2) := LOOP_2;
        LOOP_TIMES(CURRENT_TEST, 2, 1) := LOOP_1;

-- Calling sequence three:
        CALLING_SEQUENCE(CURRENT_TEST, 3) := "2-5-1-3-4";
        LOOP_TIMES(CURRENT_TEST, 3, 2) := LOOP_2;
        LOOP_TIMES(CURRENT_TEST, 3, 5) := LOOP_5;
        LOOP_TIMES(CURRENT_TEST, 3, 1) := LOOP_1;
        LOOP_TIMES(CURRENT_TEST, 3, 3) := LOOP_3;
        LOOP_TIMES(CURRENT_TEST, 3, 4) := LOOP_4;

-- Calling sequence four:
        CALLING_SEQUENCE(CURRENT_TEST, 4) := "4-1-5-2-3";
        LOOP_TIMES(CURRENT_TEST, 4, 4) := LOOP_4;
        LOOP_TIMES(CURRENT_TEST, 4, 1) := LOOP_1;
        LOOP_TIMES(CURRENT_TEST, 4, 5) := LOOP_5;
        LOOP_TIMES(CURRENT_TEST, 4, 2) := LOOP_2;
        LOOP_TIMES(CURRENT_TEST, 4, 3) := LOOP_3;
    end loop;

    PUT_LINE("CAL2--Multiple executions of identical loops--time in seconds:");
    NEW_LINE;
    PUT_LINE("Test #  LOOP_1  LOOP_2  LOOP_3  LOOP_4  LOOP_5  Calling Order");
    for INDEX_1 in 1..TEST_REPETITIONS loop
        for INDEX_2 in 1..SEQUENCE_COUNT loop
            PUT((((INDEX_1 - 1) * SEQUENCE_COUNT) + INDEX_2), 6);
            for INDEX_3 in 1..LOOP_COUNT loop
                PUT(LOOP_TIMES(INDEX_1, INDEX_2, INDEX_3), 5, 2);
            end loop;
            PUT("  ");
            PUT(CALLING_SEQUENCE(INDEX_1, INDEX_2));
            NEW_LINE;
        end loop;
    end loop;
end CAL2;

package body T_ROUTINE is

procedure PROC(ARG: in out INTEGER) is
```

```
begin
    ARG := 42;
end PROC;

end T_ROUTINE;
```

C.b. CAL2 Source Code for the MC68020/SD-Ada

```
--
-- =====================================================================
-- CAL2 is a benchmark calibration routine intended to test the assumption
-- that textually identical loops will take (approximately) the same amount
-- of time to execute.
--
-- CAL2_SD is a modified version which uses the restricted I/O facilites
-- provided by the SD compiler (Ver. 2B01).
--
-- The routine was devised to verify that benchmarks which depend on a
-- dual test and control loop structure will execute correctly on the
-- target system.
--
-- Test format is to call five functions (LOOP_1..LOOP_5) executed in
-- succession.  Each function returns a DURATION value, obtained using
-- the Ada CALENDER.CLOCK routine.  The time is obtained by subtracting
-- the time as the routine is entered from the time just
-- prior to the return to caller.  Between the two calls, a tight loop is
-- executed LOOP_REPETITIONS times.  The loop contains a single call to
-- procedure PROC.  PROC simply serves to place a light load in each of
-- the timing loops.
--
-- The test calls are made in a number of arbitrary orders to allow
-- detection of any effects relating to the total number of machine
-- cycles, as opposed to the ordering of the LOOP routines.
--
-- The test sequence is executed TEST_REPETITIONS times to allow for system
-- intialization effects (and possibly interruptions during execution).
--
-- Results are output at the conclusion of all tests.
--
-- Programming notes:
--    o The package T_ROUTINE contains a small routine PROC, which simply
--      assigns a fixed value to the single integer argument, ARG.  It is
--      isolated in a package to prevent its being optimized to an inline
--      assignment.
--
-- Known bugs:
--      <none>
--
-- Who  Date            Remarks
-- ---  ----            -------
-- NWA  16 June 87      Corrected comments.
-- JAS  18 May 87       Fixed so that SD I/O will function correctly.
-- NWA  6 April 87      Modified to work with SD compiler.
-- NWA  4 April 87      Adapted from bencharking test routine TEST_9.
-- =====================================================================

package T_ROUTINE is
    procedure PROC(ARG: in out INTEGER);
end T_ROUTINE;

with T_ROUTINE; use T_ROUTINE;
with CALENDAR; use CALENDAR;
with TARGET_IO; use TARGET_IO;

procedure CAL2_SD is

    LOOP_REPETITIONS:   constant INTEGER := 100000;
    TEST_REPETITIONS:   constant INTEGER := 5;
    SEQUENCE_COUNT:     constant INTEGER := 4;
    LOOP_COUNT:         constant INTEGER := 5;
    LOOP_TIMES:         array (1..TEST_REPETITIONS, 1..SEQUENCE_COUNT,
                                1..LOOP_COUNT) of DURATION;

    SEQUENCE_LENGTH:    constant INTEGER := (LOOP_COUNT * 2) - 1;
    CALLING_SEQUENCE:   array (1..TEST_REPETITIONS, 1..SEQUENCE_COUNT) of
                            STRING(1..SEQUENCE_LENGTH);

    TEMP_FLOAT:         FLOAT;
    TEST_NUMBER:        INTEGER;
    T_VALUE_INT_PART:   INTEGER;
    T_VALUE_FRAC_PART:  INTEGER;


-- ----------------------------------------------------------------------

    function LOOP_1 return DURATION is

        START_TIME:     TIME;
        END_TIME:       TIME;
        A_VALUE:        INTEGER := 12;

    begin
        START_TIME := CLOCK;
        for INDEX in 1..LOOP_REPETITIONS loop
            PROC(A_VALUE);
        end loop;
        END_TIME := CLOCK;
        return END_TIME - START_TIME;
    end LOOP_1;

-- ----------------------------------------------------------------------

    function LOOP_2 return DURATION is

        START_TIME:     TIME;
        END_TIME:       TIME;
        A_VALUE:        INTEGER := 12;

    begin
        START_TIME := CLOCK;
        for INDEX in 1..LOOP_REPETITIONS loop
            PROC(A_VALUE);
        end loop;
        END_TIME := CLOCK;
        return END_TIME - START_TIME;
    end LOOP_2;

-- ----------------------------------------------------------------------

    function LOOP_3 return DURATION is

        START_TIME:     TIME;
        END_TIME:       TIME;
        A_VALUE:        INTEGER := 12;

    begin
        START_TIME := CLOCK;
        for INDEX in 1..LOOP_REPETITIONS loop
            PROC(A_VALUE);
        end loop;
        END_TIME := CLOCK;
        return END_TIME - START_TIME;
    end LOOP_3;

-- ----------------------------------------------------------------------

    function LOOP_4 return DURATION is

        START_TIME:     TIME;
        END_TIME:       TIME;
        A_VALUE:        INTEGER := 12;

    begin
```

```
        START_TIME := CLOCK;
        for INDEX in 1..LOOP_REPETITIONS loop
            PROC(A_VALUE);
        end loop;
        END_TIME := CLOCK;
        return END_TIME - START_TIME;
    end LOOP_4;

-- ---------------------------------------------------------------------------

    function LOOP_5 return DURATION is

        START_TIME:     TIME;
        END_TIME:       TIME;
        A_VALUE:        INTEGER := 12;

    begin
        START_TIME := CLOCK;
        for INDEX in 1..LOOP_REPETITIONS loop
            PROC(A_VALUE);
        end loop;
        END_TIME := CLOCK;
        return END_TIME - START_TIME;
    end LOOP_5;

-- ----------------------------------------------------------------------------

begin
    for CURRENT_TEST in 1..TEST_REPETITIONS loop

-- Calling sequence one:
        CALLING_SEQUENCE(CURRENT_TEST, 1) := "1-2-3-4-5";
        LOOP_TIMES(CURRENT_TEST, 1, 1) := LOOP_1;
        LOOP_TIMES(CURRENT_TEST, 1, 2) := LOOP_2;
        LOOP_TIMES(CURRENT_TEST, 1, 3) := LOOP_3;
        LOOP_TIMES(CURRENT_TEST, 1, 4) := LOOP_4;
        LOOP_TIMES(CURRENT_TEST, 1, 5) := LOOP_5;

-- Calling sequence two:
        CALLING_SEQUENCE(CURRENT_TEST, 2) := "5-4-3-2-1";
        LOOP_TIMES(CURRENT_TEST, 2, 5) := LOOP_5;
        LOOP_TIMES(CURRENT_TEST, 2, 4) := LOOP_4;
        LOOP_TIMES(CURRENT_TEST, 2, 3) := LOOP_3;
        LOOP_TIMES(CURRENT_TEST, 2, 2) := LOOP_2;
        LOOP_TIMES(CURRENT_TEST, 2, 1) := LOOP_1;

-- Calling sequence three:
        CALLING_SEQUENCE(CURRENT_TEST, 3) := "2-5-1-3-4";
        LOOP_TIMES(CURRENT_TEST, 3, 2) := LOOP_2;
        LOOP_TIMES(CURRENT_TEST, 3, 5) := LOOP_5;
        LOOP_TIMES(CURRENT_TEST, 3, 1) := LOOP_1;
        LOOP_TIMES(CURRENT_TEST, 3, 3) := LOOP_3;
        LOOP_TIMES(CURRENT_TEST, 3, 4) := LOOP_4;

-- Calling sequence four:
        CALLING_SEQUENCE(CURRENT_TEST, 4) := "4-1-5-2-3";
        LOOP_TIMES(CURRENT_TEST, 4, 4) := LOOP_4;
        LOOP_TIMES(CURRENT_TEST, 4, 1) := LOOP_1;
        LOOP_TIMES(CURRENT_TEST, 4, 5) := LOOP_5;
        LOOP_TIMES(CURRENT_TEST, 4, 2) := LOOP_2;
        LOOP_TIMES(CURRENT_TEST, 4, 3) := LOOP_3;
    end loop;

    OUT_STRING(VDU_PORT,
        "CAL2_SD--Multiple executions of identical loops--time in seconds:");
    NEW_LINE(VDU_PORT);
    OUT_STRING(VDU_PORT,
        "Test #  LOOP_1  LOOP_2  LOOP_3  LOOP_4  LOOP_5  Calling Order");
    NEW_LINE(VDU_PORT);
    for INDEX_1 in 1..TEST_REPETITIONS loop
        for INDEX_2 in 1..SEQUENCE_COUNT loop
            TEST_NUMBER := ((INDEX_1 - 1) * SEQUENCE_COUNT) + INDEX_2;
            OUT_DECIMAL_INTEGER(VDU_PORT, TEST_NUMBER, 6);
            for INDEX_3 in 1..LOOP_COUNT loop
                TEMP_FLOAT := FLOAT(LOOP_TIMES(INDEX_1, INDEX_2, INDEX_3));
                T_VALUE_INT_PART := INTEGER(TEMP_FLOAT);
                TEMP_FLOAT := FLOAT(LOOP_TIMES(INDEX_1, INDEX_2, INDEX_3))
                    * 100.0;
                T_VALUE_FRAC_PART := INTEGER(TEMP_FLOAT) rem 100;
                OUT_DECIMAL_INTEGER(VDU_PORT, T_VALUE_INT_PART, 2);
                OUT_STRING(VDU_PORT, ".");
                OUT_DECIMAL_INTEGER(VDU_PORT, T_VALUE_FRAC_PART, 2);
            end loop;
            OUT_STRING(VDU_PORT, "  ");
            OUT_STRING(VDU_PORT, CALLING_SEQUENCE(INDEX_1, INDEX_2));
            NEW_LINE(VDU_PORT);
        end loop;
    end loop;
end CAL2_SD;

package body T_ROUTINE is

procedure PROC(ARG: in out INTEGER) is

begin
    ARG := 42;
end PROC;

end T_ROUTINE;
```

Table of Contents

1. Dual Loop Benchmarks: Purpose and Assumptions **3**

2. Testing the Validity of the Dual Loop Design for Timing Benchmarks **5**

3. Conclusion **9**

References **11**

Appendix A. Specific Configurations Tested **13**

A.a. MicroVAX/VAXELN 13

A.b. MC68020/SD-Ada 13

Appendix B. Raw Data **15**

B.a. CAL2 for the MicroVAX/VAXELN 15

B.b. CAL2 for the MC68020/SD-Ada 16

Appendix C. Test Programs **19**

C.a. CAL2 Source Code for the MicroVAX/VAXELN 19

C.b. CAL2 Source Code for the MC68020/SD-Ada 22

List of Tables

Table 2-1: CAL2 Test Results from VAXELN Ada (time in seconds) 6
Table 2-2: CAL2 Test Results from SD ADA-Plus on the 68020 (time in seconds) 7