HPC Fortran Compilers

Lee Higbie

Arctic Region Supercomputing Center

ABSTRACT:

Fortran, a largely forgotten language outside of HPC, continues to be important for computationally intensive problem solving on supercomputers. This paper provides a detailed investigation of several of the Fortran compilers that are most heavily used in supercomputing. The investigation looks at the performance of the compilers on small code snippets. During the investigation, some problems with using PAPI on small code blocks were uncovered; these are also discussed. The paper makes recommendations targeted at compiler developers and program optimizers.

KEYWORDS: Compiler Comparison, PAPI, Timing, Optimization, Performance

Introduction

Fortran is the dominant language for numeric supercomputer applications such as weather, climate and airfoil/aircraft modeling. These applications often use millions of compute hours per year. For example, the Arctic Region Supercomputing Center allocates approximately 16 core-years[1] annually to weather forecasting for the state of Alaska. Most Fortran compilers offer extensive "code optimization" options to reduce modeling cost.

Many HPC users prefer to spend their time working in their field of specialization instead of learning about "another compiler" or a new language construct. The result is that many programs are compiled with default optimization or -Ox, for the biggest x mentioned in the first screen or two of the compiler man page. If there is an option with "fast," that may be added.

With this background, it seemed reasonable to compare the compilers available on our machines. Most Fortran compiler performance studies have evaluated compiler performance based on the execution time of a few, mostly large programs.[1-4] This study has approached the analysis of Fortran compilers from a different angle. By studying at the performance on a large number of very small code blocks. Each code snippet used in this study is one or more loops, only a few of which have more than five lines of code.[2] We think that the performance of one or more major applications is probably a better indication of useful compiler quality than performance of small blocks of code, but such performance measures provide little or no help to the compiler-optimization writers or to the analysts programming or tuning an application to execute more efficiently.

The complete program is available at [6]. Several of the loops are shown later in this paper and a sampling of others is in the appendix.

We think our approach will be of greater interest to compiler and optimizer creators and code developers because we highlight specific well or poorly compiled source code structures. The loops are small enough that optimizer writers can evaluate the compiler's in-

1 Because the CPUs in supercomputers have multiple computational "cores," this term is usually used to describe the basic computational resource on supercomputers. Nodes, which typically have between 4 and 32 cores on 1 to 8 chips, are the basic unit of allocation on many large systems.

2 The entire program source code and performance result spreadsheets are at www.arsc.edu/~higbie/CompilerTests

ternal optimization operations and people doing code development or optimization can see the types of structure that prevent compilers from producing efficient code.

This paper used all the Fortran 90/95 compilers on the XT5 supercomputers at the Arctic Region Supercomputing Center at the University of Alaska Fairbanks: those from Cray, the Gnu Project, PathScale and The Portland Group.[3] The study consisted of comparing the execution speed of several hundred code snippets on each compiler.[4] Some statistics on the relative performance of the compiled code are included in the figures.

The results were surprising for four reasons.

1. The execution time, as reported by the PAPI[5] papif_get_real_cyc, varied widely from one run to the next.

2. Changing the optimization from normal to high often did little to improve the performance on these small code blocks.

3. For each compiler tested, there were some snippets that ran substantially slower with "high" optimization.

Caveats

Any study like this is a snapshot of specific compiler versions on specific code.[3] Making general inferences about compiler performance is unlikely to be useful. At a different time, i.e., with other compiler versions, the relative results are likely to change.

We did not attempt to comprehensively test compiler options. Only a few switches or options were tried and they were only tried on our small code snippets. How well a compiler does on a large program, specifically on your own program, is probably a better compiler metric.

Another performance-enhancing option for some users is to take advantage of thread-level parallelism and use OpenMP. For these small loops, performance was often slowed by auto-parallelization so we do not report on it further.

There are many sources for information on how to efficiently utilize compilers, but we doubt that most users spend time studying these options. This documentation seems targeted at analysts.

Background

Each compiler tested has many, often dozens of "optimization" options.[6] Facing the complexity of selecting the way to compile, we suspect most production program users try high optimization, and if that doesn't work (the program doesn't run or runs inaccurately), back off to the default optimization.

ARSC's XT5s have a typical supercomputer architecture with hundreds of nodes, each with two or more x86 family processor chips. Thus, we do not think

3 Terminology for compilers mentioned in this paper:
"Cray" = "Cray Fortran," version 7.0.3
"Gnu" = "The gcc Fortran 90 compiler," gfortran, version 4.1.2(prerelease)
"PathScale" = "Qlogic PathScale Compiler Suite," versions 3.2
"PGI" = "Portland Group's pgf90 compiler," versions 7.2-3

4 We made no attempt to analyze the quality of diagnostics or the acceptability of Fortran dialects with these compilers. We did run into some minor issues with Cray Fortran and Gnu Fortran. These compilers produced fatal errors for a few statements that the PGI and PathScale compilers accepted:
a. Gnu Fortran would not accept concatenated strings on stop statement such as: stop 'Overflow at ' // here
b. Neither compiler would not allow unused parameters from an include file that were out of range. PAPI has two flags that are set to the value of 2^{31}, which caused warnings in the other compilers but a fatal error for both Cray and Gnu compilers.
c. One part of the inter-loop data re-initialization used .xor. For gfortran, this had to be changed to .neqv.
d. The test program was written in Fortran 95 and nearly all the code files had name suffix .F95. For the Cray Fortran compiler, the files *had to be renamed* with .F90 as the suffix.

5 Acronym Decoding

ARSC	Arctic Region Supercomputing Center
CNL	Compute node Linux
HPC	High performance computing
MFLOPS	Mega-floating point operations/second
MOPS	Mega-operations/second
PAPI	Performance application programming interface

6 It's unlikely that any significant program has been optimized. Requesting "optimization" from a compiler means requesting that it generate code that runs faster (or sometimes takes less space). "Hey, make it run a little faster" just doesn't have the nice ring of "optimize."

they pose any special compilation difficulties the way a more unusual architecture, such as vector or cell processors, might. ARSC has two XT5s, a small one named Ognip and a large one named Pingo.[7] These tests were run standalone on nodes with 8 CPU cores and 32 GB of memory.

Test Procedure

The program was compiled using each of the compilers and a few of the most common options:

-O2, or the default optimization level for each compiler

-fast (or its equivalent), an option that is is supposed to produce faster-running code. This option often invokes several more specific types of "optimization." Cray recommends the -tp barcelona switch for the Portland Group compiler and PathScale has a switch, -ffast-math, that looked useful. We used both options in addition to "-fast" for the fast code versions.

Because code quality is difficult to assess directly and the size of source code structure space is so large and highly dimensioned, we planned to use execution time as a surrogate for compiler quality. We feel that this is a proper measure, in the sense that code execution time *is* what compiler optimization is all about, at least for HPC.[8]

Further, we doubt that the management of the memory hierarchy can be determined except by its execution time behavior. I.e., to measure how well the code generated by a compiler utilizes the memory system, we believe one has to use code execution time.

Because of variability of PAPI clock cycle time, many loop-time measurements were made for each loop. The sidebar describes some of these issues.

Compiler Differences

Table1 shows the execution speed ratios on Ognip comparing statistics for the loop times compiled with

7 A Pingo is a large frost heave, typically a kilometer across and dozens of meters high. An Ognip is a Pingo that has collapsed (melted interior). Both words are derived from Inuit. In Alaska, some people (incorrectly?) call smaller frost heaves "pingos." Pingos are common in northern Canada, but rare in Alaska. Pingos form in areas of permafrost.

8 DoD's large Challenge Projects are often required to show the efficient operation on the machines they use. In this context, efficiency is usually measured only by the scaling of the program to large numbers of MPI tasks.

While collecting data, we realized there was large variability in the PAPI-reported clock ticks. The calls to `papif_get_real_cyc()` include some operating system overhead, which can vary widely and systematically.

It seems to us there should be an easy, portable, low-overhead way to access the system clock, but we could find none, nor could we find any standard-Fortran high resolution timer. The PAPI function was the only one that appeared adequate for timing small loops and was available on all machines and compilers.

Because of the seemingly random operating system overhead added to PAPI calls, we experimented to find a way to determine the actual loop execution time. The procedure we settled on was to time each test loop three times in succession to guarantee that the time from program-start to loop timing varied:

```
DO I=1, noTimings     ! = 1 to 3
   timeStmp(tstNo, 1, I) = compTim()
   <loop being timed>
   timeStmp(tstNo, 2, I) = compTim()
   call checkResult
   call reinitialize
enddo
```

This loop structure is repeated for each of the 708 loops, see [6]. The calls to checkResult and reinitialize have the side effect of flushing data from the cache before the next loop timing. For *some* of the test loops, the code block above was embedded in an outer loop that doubled the iteration count of the test loop 20 times, from about 35 to 35M iterations. In the sample graphs at the end of the paper or those at [6], you can see timing ratios for blocks of loops with increasing iteration counts.

Using the smallest timing from three re-executions of a loop appears to produce repeatable and reasonable results. The entire program was run 15+ times for each compiler and the minimum of the 15 minimal-times is the value used for this paper.

The test code program does not perform any I/O until the last few code snippets.

Time Ratio -fast over -O2				
Statistic / Compiler	**Cray**	**Gnu**	**PathScale**	**PGI**
Maximum Time Ratio	1.35	1.50	2.28	1.17
99th Percentile	1.20	1.25	1.30	1.13
95th Percentile	1.05	1.14	1.15	1.05
50th Percentile	1.00	1.00	0.94	1.00
5th Percentile	0.96	0.92	0.39	0.94
1st Percentile	0.86	0.80	0.32	0.86
Minimum Time Ratio	0.72	0.56	0.04	0.61

Table 1: Statistics of the execution-time ratios of the loops compiled with -O2 and -fast.

"-fast" to those compiled with -O2. If the ratio is greater than 1.0, then -O2 code performed faster than -fast. The column heading show the compiler and the "optimization" selections compared.

We checked the assembly code produced by the code snippets producing the extremal values in the table above. In some cases we could see why the code was substantially faster or slower.

The Gnu compiler with the -fast option had the best speedup, nearly twice as fast, on the "loop"

```
DO I=1, Nparhd        ! = 1 to 128
  DO J=1, NSomeDat    ! = 1 to 32
    DO K=1, nFewDat   ! = 1 to 15
      XP1(I,J) = XP1(I,J) +        &
         XP2(I,K)* XP3(K,J)
    enddo
  enddo
enddo
```

For this loop the -fast option caused preloading the XP2 values and fully unrolling the inner loop (the loop iteration counts are parameters).

The PathScale compiler's -fast option slowed the execution of

```
j = 0
DO I=1, nFewDat
  K = nFewDat - I + 1
  J= J+1
  T1(I,J,I) = T2(I,I,K)  *         &
        F1(I,J,M,I)      *         &
        FV1(I,NP2,J,K,I) *         &
        FV2(I,I,I,J,K)
enddo
```

by more than a factor of 2. In this case, PathScale unrolled the loop with -fast. Our guess is that cache misses slowed execution of the unrolled code. At the other extreme, -fast increased execution speed of

```
v1 = 2
DO I = 1, NPARHD
    XS1(I) = XS2(I)**V1
enddo
```

by a factor of 25 for PathScale. In this case the compiler called a different function, vrs4_powf(), to evaluate the expression, instead powf(). No other compiler used the vrs4_powf() function, one that computes four exponentiations at a time. In effect, it unrolled the loop.

The Portland Group compiler was slowed with -fast by almost 20% on the set of loops

```
DO I=1,13
   XS1(I) = 1.0
enddo
DO I=14,330      ! note overlap
   XS1(I) = -1.0
enddo
DO I=34,nData    ! nData = 600
   XS1(I) = 10.0
enddo
```

apparently because of loop unrolling. At the other extreme, it gave a 40% speed up on

```
k = 1
DO I=1, nParHD
    IF(ls1(I)) THEN
        XS1(I) = XS2(k)
        k=k+1
    ENDIF
enddo
```

Inter-compiler Time Ratio Statistics						
Statistic / Compiler	**O2: CRI/PGI**	**O2: Gnu/PGI**	**O2: Path/PGI**	**fast: CRI/PGI**	**fast: Gnu/PGI**	**fast: Path/PGI**
Maximum Time Ratio	1.23	3.86	29.42	1.51	3.99	3.07
99th Percentile	1.09	3.19	2.84	1.18	3.32	1.81
95th Percentile	1.04	1.94	1.74	1.06	2.00	1.52
50th Percentile	1.00	1.05	1.02	1.00	1.06	0.97
5th Percentile	0.95	0.73	0.58	0.95	0.72	0.28
1st Percentile	0.86	0.43	0.30	0.88	0.30	0.09
Minimum Time Ratio	0.79	0.00	0.00	0.71	0.00	0.00

Text 1: Statistics on the loop timings of our XT5 compilers to the PGI compiler, the Ognip and Pingo default.

Using our loop-by-loop technique for measuring compiler-to-compiler differences does not seem appropriate as we have noted. In fact, despite our efforts to use performance as an accurate surrogate for compiled code quality, we may have bad time values instead of compiled-code quality differences. Compiler writers may be interested in specific areas where their compiler's relative performance is poor, so they can improve it. Table 2 should not be viewed as comparing compiled code quality.

For example, the zero entries in the Table 2 result from the random number generator, which took substantially longer in the PGI-compiled code than with the others. PGI's code may be doing substantially better or more anticipatory work than the others.

As with the intra-compiler comparison above, we made an effort to see how the compilers "optimized" or failed to optimize code, by looking at the assembly language output for the loops producing the maxima or minima in Table 2. Here we summarize those cases where this yielded useful insight.

The PGI compiler outperformed the Gnu compiler at O2 and fast optimization levels by the widest margin on a character string copies. The Gnu compiler compiled the code while PGI made a single call to __c_mcopy1. With -fast, Gnu unrolled the loop, but __c_mcopy1 was still nearly four times faster.

The PGI compiler had the best performance relative to the Pathscale compiler for fast optimization on

```
DO I=1, nData
  ls1(I) = CH1(I:I) .EQ. CH2(I:I)
ENDDO
```

For this loop, it appears that PGI -fast is preloading the data, probably reducing cache miss time to achieve more than three times the performance of PathScale -fast.

The loop where the PGI -O2 compilation code ran orders-of-magnitude slower than either Gnu or PathScale (but quite close to Cray's compiler) is

```
DO I = 1, NPARHD
   XS1(I) = XS2(I)*XS3(i)
   CALL random_seed()
enddo
```

Pathscale called ranf_4 and Gnu called _gfortran_random_ seed while PGI called pghpf_rseed, which apparently slowed the execution tremendously for both levels of optimization.

Summary:

If performance on a code is not as expected, the easiest optimization is often to vary the compiler or compiler options. Changing from -fast to -O2 or conversely, may yield good results, especially for programs with loop counts near 64. If your program will compile with another compiler, this analysis suggests you should try it or try it on some of the hot-spot routines.

The big and long term program improvement is from enhancing the algorithms in heavily used code blocks and cleaning up their code. We feel certain that clean code will always be easier to analyze and optimize, both to programmers working on it and to compilers—it is not difficult to confuse a compiler. Clean, understandable code is the best defense against poor compiler performance.

Observations:
1. PAPI's `papif_get_real_cyc()` appears to have a variable, sometime thousands of clock cycle, overhead. Perhaps the incorrect clock cycle counts would improve if there were separate `get_start_cycle` and `get_end_ cycle` functions.

Our idea is `get_start_cycle` would collect the system clock value, hopefully putting any variable operating system operations before the clock value is captured; we would use it as the starting time. At the end of code snippet timing, we would use `get_end_cycle`, which would return the clock value just as quickly as possible from the function, before nearly all operating system overhead.

An alternative that we prefer is for the Fortran standard to specify a function to directly access the system clock and a complementary function to report its period. The Fortran standard should specify functions with minimal overhead and maximal reproducibility.

2. Our loop results, as shown in Figures 1 and 2, suggest that for most Fortran programs any of these compilers should produce similar performance. For any code, one compiler may be better or worse, possibly spectacularly so and small performance improvements on production codes can be worthwhile. Changing from -O2 or default optimization to -fast, or conversely may be worthwhile. Experiment with optimization *and* compiler choices, even on a routine-by-routine basis.

References and Bibliography

1. Appleyard, John, "Comparing Fortran compilers," ACM SIGPLAN Fortran Forum, v.20 n.1, p.6-10, April 2001
2. Higbie, Lee, "Speeding up FORTRAN (CFT) programs on the CRAY-1," Cray Research Inc. Tech Note 2240207. 1978
3. Kozin, Igor N., "Performance comparison of compilers for AMD Opteron," Dec, 2005, www.cse.scitech.ac.uk/ disco/Benchmarks/Opteron_compilers.pdf
4. Polyhedron Software, "32 bit Fortran execution time benchmarks," http://www.polyhedron.com/benchamdwin and http://www.polyhedron.com/pb05-win32-f90bench_p40html
5. Higbie, Lee, Tom Baring, Ed Kornkven, "Simple Loop Performance on the Cray X1," CUG 2005, Tuesday. Also available at www.arsc.edu/~higbie/LoopTests

About the Author

Lee Higbie is an HPC Specialist at the Arctic Region Supercomputing Center, P.O. Box 756020, Fairbanks, AK 99775-6020, +1-907-450-8688, higbie@arsc.edu. He has been working in supercomputing and parallelization for almost five decades.

Appendix: A few of the 708 loops

In these snippets, names that begin with x are real*4, d are real*8, c are complex*8, l are logical.

I. Loops with doubling trip counts. Each of these was timed for k = 1, 2, 4, 8, 16, ...2 ** 20.

1. Done for several data types (real*4, int*1, int*4, complex, etc.)

```
DO I=1, nSomeDat * k
   XL1(I) = XL2(I) + XL3(I)
enddo
```

2.

```
DO I=1, (nSomeDat + 3) * k
   XL1(I) = XL2(I) * v2 +       &
        xl2(i+1) * v1 +         &
        xl2(i+2) * v3 +         &
        XL2(I + 3)
enddo
```

3. Done for several common functions

```
DO I = 1, nSomeDat * k
    XL1(I) = sin(XL2(I))
enddo
```

4. Note that here the "vector length" is decreasing and the stride is doubling.

```
DO I=1, nLongDat, k
   XL1(I) = XL2(I) + xl3(i)
enddo
```

II. A few loops were checked with the loop trip count varying but formed in a way that the compiler could know it at compile time.

1. This loop was repeated with he constant 2 replaced by integers from 1 to 11. nFewDat is a parameter = 15. The loop was also repeated with nFewDat replaced by a parameter whose value was 32.

```
DO I = 1, 2 * nFewDat
   XS1(I) = XS2(I) + XS3(I)
enddo
```

2. This loop, like the one above, was rewritten for many, fixed and known iteration counts.

```
DO I = 1, 2 * nFewDat
   XS1(I) = sin(XS2(I))
enddo
```

III. Many loop nests, loops with conditionals and series of loops were checked. Often the best strategy is to re-order the loops, combine several loops or break a loop into multiple loops.

1.

```
DO I=1, NPARHD
   DO J=1, NSomeDat
     DO K=1, nFewDat
        XP1(I,J) = XP1(I,J) +    &
              XP2(I,K) * XP3(K,J)
     enddo
    enddo
enddo
```

2.

```
DO I=1,NPARHD
   XS1(I) = 0.0
enddo
do i = 1, NPARHD
    XS2(I) = 1.0
enddo
DO I = 1,NPARHD
   XS3(I) = 2.0
enddo
DO I=1,NPARHD
    XL1(I) = -1.0
enddo
DO I=1,NPARHD
    XL2(I) = -12.0
enddo
```

3.

```
DO I=1,nData
   IF(I.LT.11) XS1(I) = 0.0
   IF((I.GE.11) .AND.           &
         (I.LT.33))  XS1(I) = 1.0
   IF(I.GE.33) XS1(I) = 3.0
enddo
```

IV. The compilation was done as a single unit, so the compiler can expand in-line, pass the sign of parameters, etc to the sub-programs.

1.

```
DO I = 1, NPARHD
   XS1(I) = VCTFN(XS2(I),XS3(I))
enddo
```

2.

```
CALL VCTSB2(XS2, XS3, XL2,       &
               2, NPARHD)
```

3. In this case, the programmer viewed the call to random_seed as a no-op that could be lifted out of the loop.

```
DO I=1, NPARHD
    xs1(I) = xs2(I)*xs3(I)
    CALL random_seed()
    xl1(I) = xs2(I) - xs3(I)
enddo
```

V. The following sample Fortran 90 loop was tried with several structure layouts.

```
DO I=1,Ndata
  bo1(i)%XS1(2) = bo1(i)%XS2(1) &
             + bo1(i)%XS3(3)
enddo
```

Figure 1: Graph of log of execution time ratios for all loops. Values above zero imply -O2 was faster than -fast.

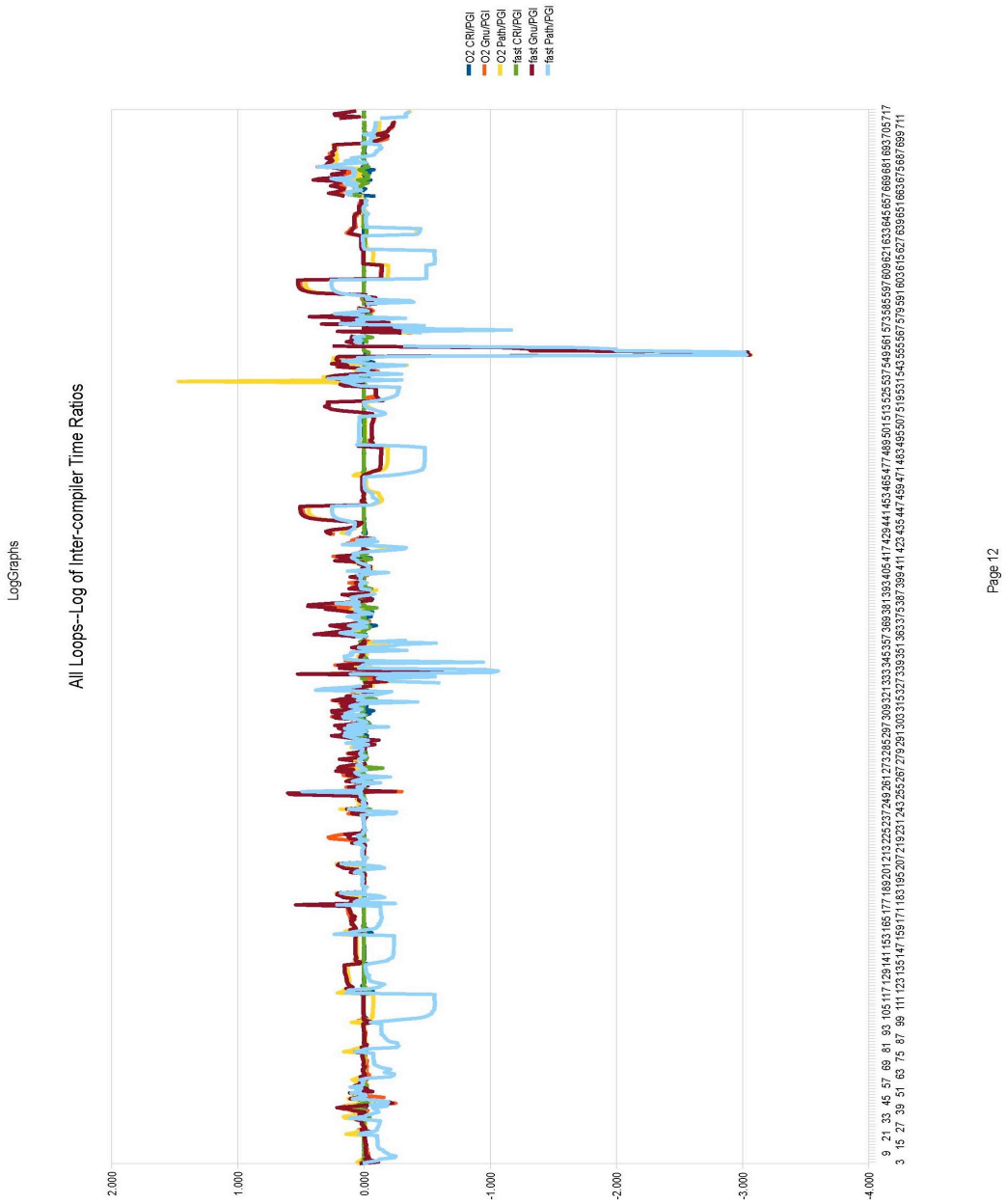

Figure 2: Log Time Ratios of Compilers to PGI Compiler