

Performance Metrics for Ocean and Air Quality Models on Commodity Linux Platforms

George Delic

george@hiperism.com
HiPERiSM Consulting, LLC
Durham, North Carolina

Abstract. This report examines performance of four models with two compilers on the Intel Pentium 4 Xeon processor using the PAPI performance event library to collect performance counter values. The intent is to identify performance metrics that indicate where performance inhibiting factors occur when the codes execute. Results for operations, instructions, cycles, cache, table look-aside buffer misses, branching instructions, and I/O operations are discussed in detail. Results support the thesis that codes optimized for vector register computer architectures also benefit from commodity cache-based architectures and the compilers that optimize for them. Consequences for Air Quality Models are analyzed by comparing results against two ocean models with a view to potential performance benefits on commodity architectures.

Introduction

This is a report on a project to evaluate industry standard fortran 90/95 compilers for IA-32 Linux™ commodity platforms when applied to Air Quality Models (AQM). The goal is to determine the optimal performance and workload though-put achievable with commodity hardware. Only a few AQM's have been successfully converted to OpenMP (CAMx, [1]) or MPI (CMAQ, [2]) and considerable work remains to be done on others. In exploring the potential for parallelism it has been interesting to discover the problems with serial performance on several AQM codes. For this reason we have searched for more precise metrics of performance as an aid to measuring progress in performance enhancement. The historical analogy is the programming environment on Cray architectures which enabled the development of performance attributes for either individual codes or workloads using hardware performance counters [3,4]. Since commodity processors also have performance counters, software interfaces, such as PAPI [5], may be used to read them.

This study applied the PAPI library in understanding what delivered performance is for two AQM's, ISCST3 [6] and AERMOD [7], and what the optimal achievable performance can be. For the latter, as a base-line, two Ocean models with good vector character have been included. These are used to measure the optimal performance to be expected on commodity hardware with available compiler technology. The intent in this report is to identify performance metrics that show *where* performance inhibition occurs on the hardware architecture, and not to so suggest *what* needs to be done

to remedy the situation. Remedial actions to enhance performance require a discussion of source code analysis and this is beyond the scope of this study. Rather the focus here is on the quantitative measurement of factors known to influence performance of executing applications: processor clock rate, pipeline design, branch prediction ability, cache events, and memory and I/O architecture. Not all of these factors can be quantified for an executing application, but sufficient detail is presented on several of these factors to demonstrate the value of specific performance metrics as aids in performance monitoring of an application. Without such performance metrics the task of measuring progress in performance improvement for an application is not quantifiable.

In addition to performance metrics (as derived from hardware performance counter values) some comments on I/O storage performance are also included because of the special character of I/O requirements in AQM's.

Both ISCST3 and AERMOD are (pollutant) dispersion models used in health risk assessment since they provide pollutant concentrations and deposition rates. Both these AQM's have a wide-spread use and account for much computer time although neither has yet been successfully converted to parallel form, despite good parallelism potential. However, they are used in "task farming" mode on cluster computers and through-put performance is an important issue.

Results are reported here for the two ocean models and the two AQM's on Intel Pentium 4 Xeon processors. This report is divided into sections describing the choice of hardware, operating system, and compilers, choice of benchmarks, overview of benchmark results, performance inhibitors, and I/O performance in AQM's, Conclusions and consequences for AQM performance are also summarized.

Choice of Hardware, Operating System, and Compilers

The hardware used for the results reported here is the Intel Pentium 4 Xeon (P4) processor (3GHz, 1MB L3 cache) in a dual configuration on a Super Micro SuperServer 6023P-i platform with the Intel E7501 chipset. This configuration has 4GB of memory and a 533MHz System Bus shared by both processors. The operating system (OS) is HiPERiSM Consulting, LLC's modification of the Red Hat (RH) Linux™ 9.0 SMP kernel. Four compilers are now available for this combination of hardware and OS from Absoft, Intel, Lahey, and The Portland Group, STMicroelectronics. All four compilers have previously been compared in benchmarks [8]. In this report the focus is on the Intel and Portland compilers for the four models described in the previous section. For performance data collection the PAPI library was installed with the appropriate kernel patch to enable performance counter data collection. The PAPI library offers a total of 103 events for which counter information may be collected. However, the number of events that are available is dependent on the architecture and for the Intel Pentium 4 Xeon (IA32) only the 26 listed in the Appendix are available. In addition, some higher level (derived) events may be computed (e.g. Mflips, Mflops, etc.) from these. The execution times reported here are obtained from the "process time" reported by PAPI and event rates were obtained by dividing the absolute counter value by the PAPI process time. However, since only four performance

counters are available on the Pentium 4 Xeon, some 17 separate executions were required to collect all 26 counter values for a given application. All executions were run in dedicated mode, but a minor variability in process time was observed to be of the order of a few percent.

The Xeon architecture offers Streaming Single-Instruction-Multiple-Data Extensions, SSE2, (SSE) to enable vectorization of loops operating on multiple elements in a data set with a single operation. Both compilers discussed here specifically enable SSE through a compiler switch and this has been used in tests.

It is expected that different compilers will deliver different performance (as will different choices of compiler switches). This analysis used the selection shown in Table 1. This describes the four sets of compiler switches used: noopt (all optimization disabled), opt (the default switches with some optimization), vect (vector transformations enabled), and SSE (SSE2 instructions enabled). Note that with the Intel compiler, vector code transformation is only enabled when the SSE compiler switch is enabled. There is no direct comparison between both compilers for the vect choice of switches for vectorization without hardware SSE instruction use. This is a compiler option with the Portland compiler that enables vector-friendly code transformations such as loop tiling or idiom recognition. However, this possibility is not available with the Intel compiler and for this reason no vect compiler option is shown in this case.

Table 1. Compiler command and switches.

Compiler and version	Compiler command and selected switches	Key
Intel 8.1.023	ifort -tpp7 -O0 -Ob0 -unroll0 -FI	noopt
	ifort -tpp7 -O3 -Ob2 -prefetch- -FI	opt
	ifort -tpp7 -xW -O3 -Ob0 -prefetch- -FI	sse
Portland 5.2-2	pgf90 -O0 -tp p7	noopt
	pgf90 -O2 -tp p7	opt
	pgf90 -fast -Mvect -tp p7	vect
	pgf90 -fast -Mvect=sse -tp p7	sse

Choice of Benchmarks

The choice of benchmarks includes four codes. Two of these are ocean models: the Stommel Ocean Model (SOM) and the Princeton Ocean Model (POM). These are useful as tests of the vector capabilities of compilers and architectures. Both have regular data structures with loop nests of strong vector character. Consequences for performance as problem size scales is investigated for the POM. The AQM's chosen were ISCST3 [6] and AERMOD [7] which describe pollutant dispersion and deposition. Two other important AQMs used in atmospheric chemistry and pollutant transport [1,2] are the subject of future study.

Stommel Ocean Model (SOM)

The compute kernel of the SOM is a double-nested loop that performs a Jacobi iteration sweep over a two-dimensional finite difference, $N \times N$, grid and the number of iterations is set to 100 [8]. For this study the problem grid size is set at $N=8000$ since performance was remarkably consistent in the range $N=2000$ to 8000. The SOM is an unusual algorithm in that virtually all the arithmetic work is concentrated in this loop nest. However, it is a useful test of compiler vectorization ability and the optimal delivered floating point performance for a compiler and architecture combination.

Princeton Ocean Model (POM)

The Princeton Ocean Model (POM) is a legacy Fortran 77 code with compute kernels consisting of over five hundred vectorizable loops [8]. Typically these are triple-nested loops (i,j,k) that perform operations over a three-dimensional finite difference grid. The vertical zones over the k range form the outermost loop in the nest. The number of iterations varies with the choice of data set as shown in Table 1. The inner loop structure presents compilers with good prospects for vectorization.

Table 2. Problem sizes and scaling for the POM.

GRID	i_{\max}	j_{\max}	k_{\max}	Scaling
1	100	40	15	1
2	128	128	16	4.4
3	256	256	16	17.5

ISCST3

The Industrial Source Complex Short Term (ISCST3) Model is the U.S. EPA's current regulatory model for many new source reviews and other permitting applications [6]. It is a legacy Fortran 77 code developed between 1989 and 1992, and has since been converted (in part) to the Fortran 90 standard. As such, and typical of that generation of environmental models, it was developed on a PC platform, with a small memory requirement, poor vector character, and I/O bound performance characteristics. Nevertheless, ISCST3 enjoys widespread use.

AERMOD

The U.S. EPA has proposed AERMOD as a replacement for ISCST3. The code has good potential for parallelism, but the conversion task is complicated due to an elaborate call tree that also inhibits vectorization due to multiple levels of procedure calls within loop structures. AERMOD and other AQM's are available at the U.S. EPA's Support Center for Regulatory Air Models [7].

Overview of Benchmark Results

Previous reports in this series [8] have reported only the time for an application. However, with the availability of the Performance Application Programming Interface (PAPI, [5]), it is now possible to read hardware performance counter values and derive performance metrics from them. The Intel Pentium 4 Xeon processor has 4 performance counters and this section presents results for process time, operations, instructions and clock cycles. These are summarized in derived performance attributes, such as million floating point (fp) operations (Mflops), and cycles per instruction (mean time between instruction issue). It was found that on this architecture, for fp applications, the number of Mflips (million fp instructions) were close to the Mflops rate, with each fp instruction resulting in one floating point operation. The exception is when SSE is enabled, and for this case the Mflops rate was estimated from the operation count measured (without SSE) and the SSE process time. Values for vector/SIMD instruction rates are shown in the Performance Inhibition section.

Overview of SOM Results

Particularly striking for SOM is the boost in performance for both compilers as higher levels of optimization are enabled. This is demonstrated in Figure 1 showing the process time and in Figure 2 showing the Mflops rate. In the latter case the Intel compiler with SSE optimization enabled shows just over 0.9 Gflops performance. The increasing rate in fp performance is expected with the decrease in process time, since the total number of operations remains fixed. However, it is interesting to observe that as optimization level rises, Figure 3 shows a sharp and steady decrease in the total instruction count. But, even so, the corresponding mean number of cycles between instruction issue increases in the range 1 to 6, as shown in Figure 4. Longer intervals between instruction issue that have positive correlation with correspondingly increasing operation rates are the hallmarks of efficient vector performance [3,4]. Thus, while SOM may represent an unusually felicitous performance case, it does show what typical values of performance metrics are likely to be on commodity processors for good vector code.

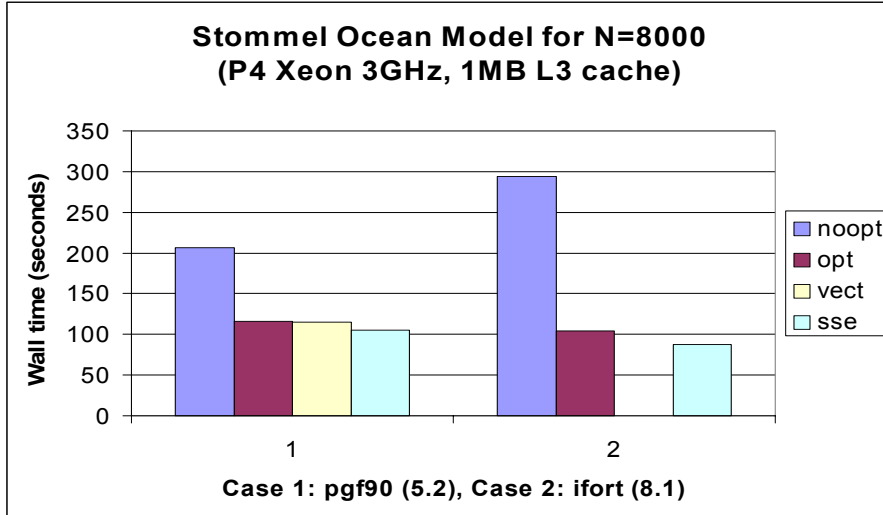


Fig. 1. Execution times (seconds) for the SOM algorithm with pgf90 and ifort compilers on the Pentium 4 Xeon processor for the choices of switches shown in Table 1.

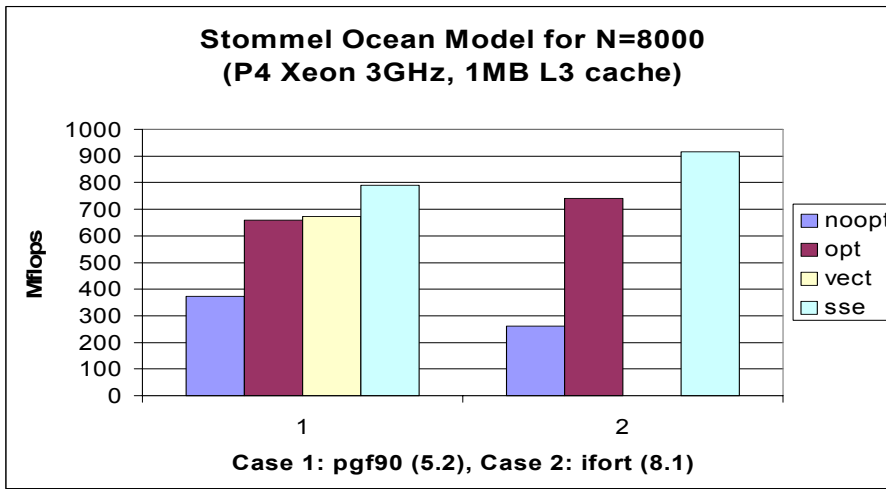


Fig. 2. Mflops for the SOM algorithm with pgf90 and ifort compilers on the Pentium 4 Xeon processor for the choices of switches shown in Table 1.

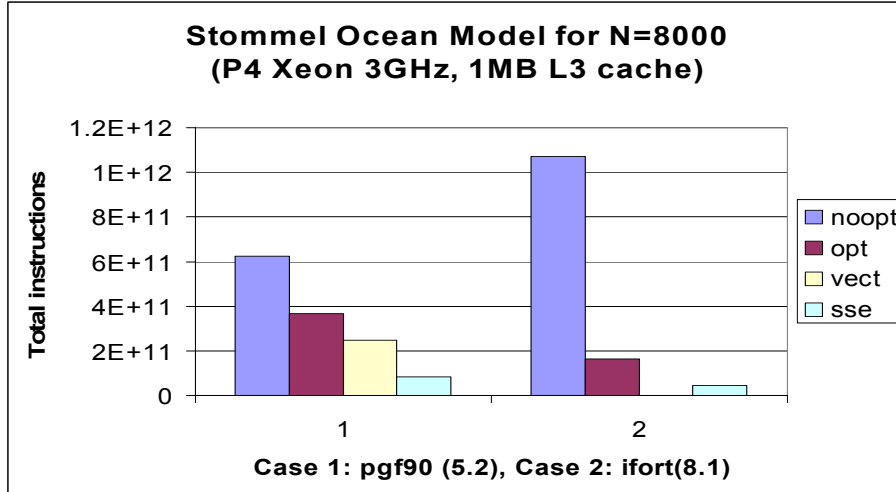


Fig. 3. Total instructions for the SOM algorithm with pgf90 and ifort compilers on the Pentium 4 Xeon processor for the choices of switches shown in Table 1.

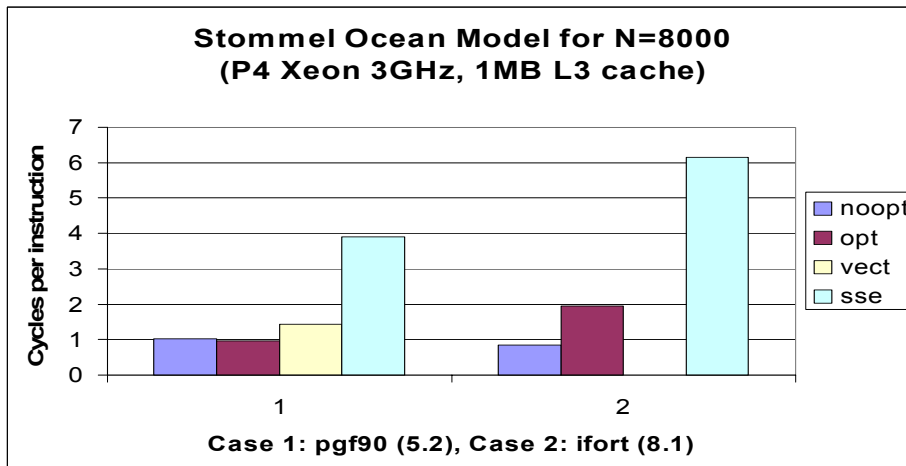


Fig. 4. Cycles per instruction for the SOM algorithm with pgf90 and ifort compilers on the Pentium 4 Xeon processor for the choices of switches shown in Table 1.

Overview of POM Results

The Princeton Ocean Model is a good example of a “real world” model that enjoyed wide-spread use and specialized enhancements for vector performance. In this

respect it is a good representative benchmark of a legacy vector fortran application. Figure 5 shows the process times reported by PAPI, and in view of the range of performance for the three problem sizes shown in Table 2, the numerical values are listed in Table 1. In the case of POM the increase in Mflops rates shown in Figure 6 is regular only for GRID 2 and GRID 3. For all three problem sizes an increasing optimization level shows that the total instruction count decreases (Figure 7) with a corresponding increase in the mean number of cycles between instruction issue (Figure 8). It is notable that for the POM application there is a steady decrease in the Mflops rate as the problem size increases (Figure 6) while the mean number of cycles between instruction issues rises steadily in the range 2 to 9 (Figure 8). This is in contrast to the SOM where there was a positive correlation between both performance attributes.

Table 3. Execution times (seconds) for the POM algorithm with the pgf90 compiler on the Pentium 4 Xeon processor for the choices of switches shown in Table 1.

GRID	noopt	opt	vect	sse
1	190.1	149.6	168.6	145.1
2	2724.0	1566.0	1527.0	1191.8
3	13199.9	7567.9	7603.5	5985.6

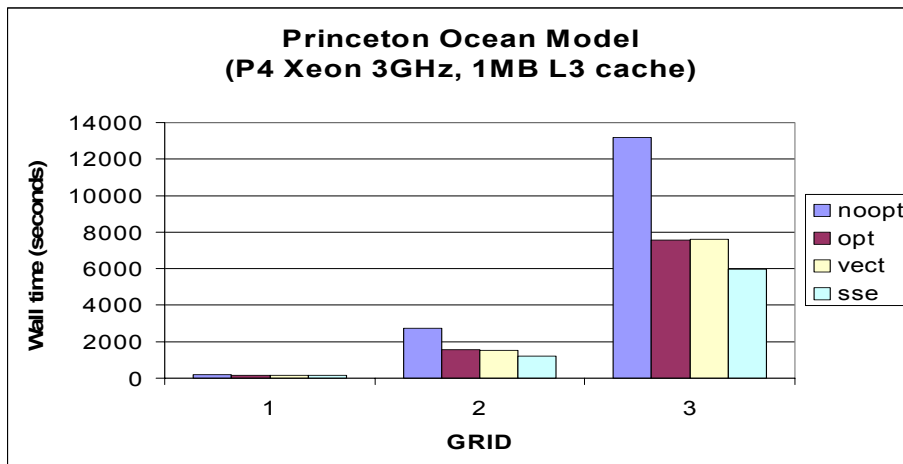


Fig. 5. Execution times (seconds) for the POM algorithm with the pgf90 compiler on the Pentium 4 Xeon processor for the choices of switches shown in Table 1.

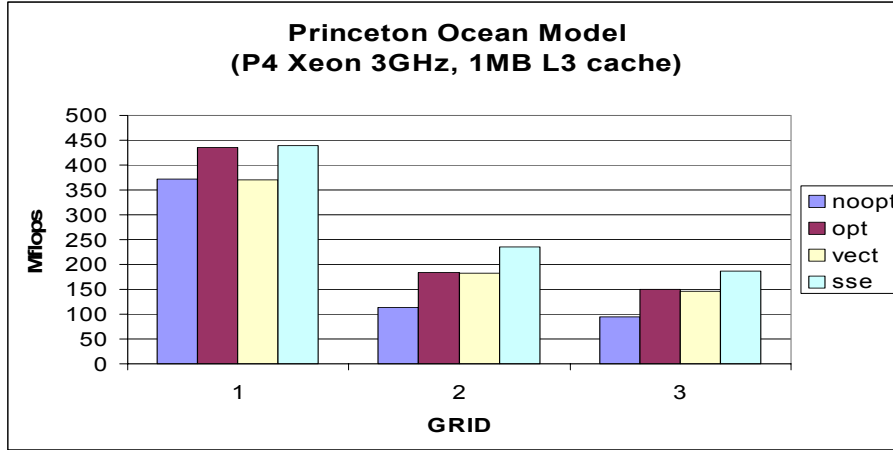


Fig. 6. Mflops for the POM algorithm with the pgf90 compiler on the Pentium 4 Xeon processor for the choices of switches shown in Table 1.

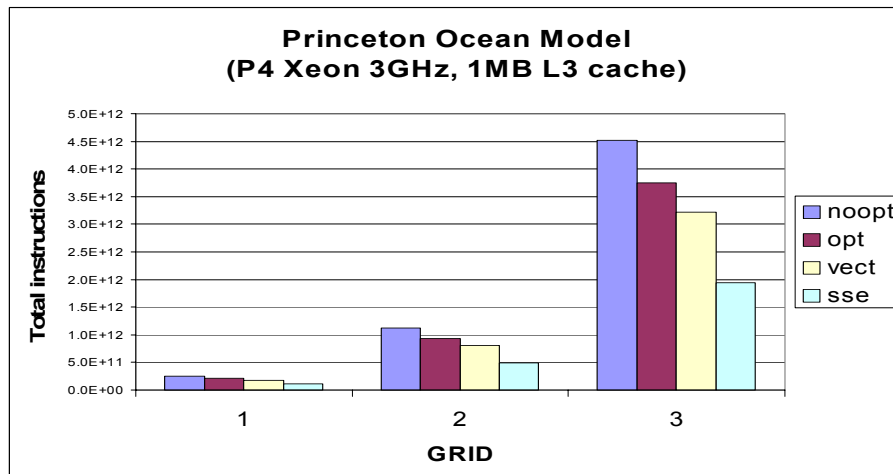


Fig. 7. Total instructions for the POM algorithm with the pgf90 compiler on the Pentium 4 Xeon processor for the choices of switches shown in Table 1.

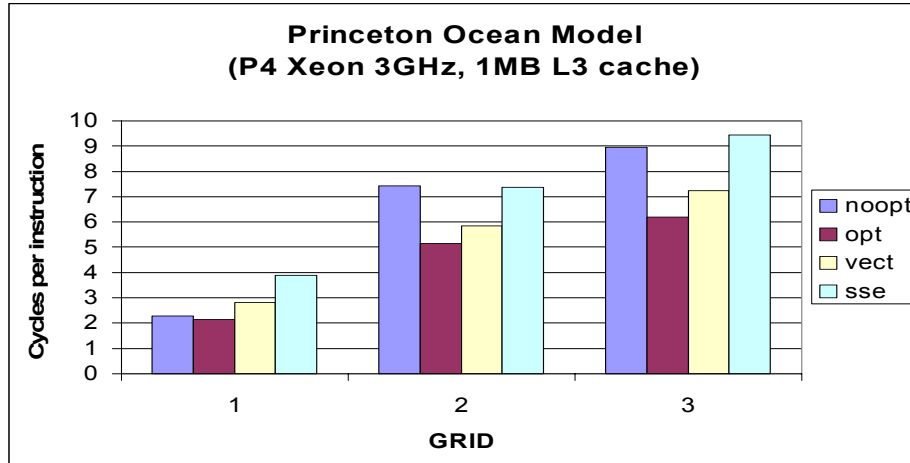


Fig. 8. Cycles per instruction for the POM algorithm with the pgf90 compiler on the Pentium 4 Xeon processor for the choices of switches shown in Table 1.

The interesting behavior for POM Mflops rates as problem size increases suggests more analysis is required to study memory and cache usage and this is discussed in the Performance Inhibition section.

Overview of ISCST3 Results

In contrast to the previous two examples the ISCST3 code shows small changes in process time (Figure 9), or Mflops rates (Figure 10), as optimization levels are increased. Here also, there is a reversal of the differences in the performance results of the two compilers compared to the SOM results. In particular the Intel compiler lags very significantly behind the other compiler. The total instruction counts (Figure 11) are not widely different for the two compilers. However, the observed number of cycles per instruction (Figure 12) is significantly different, with larger values for the Intel compiler case. For either compiler the mean number of cycles between instruction issue is 2 or less, which is much lower than the good vector performance of SOM or POM. The resolution to this anomalous behavior is revealed in the sections on Performance Inhibition and I/O Performance.

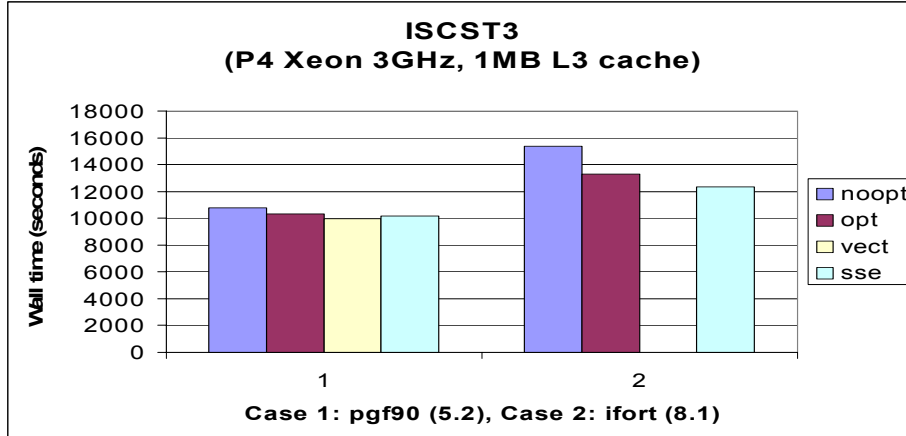


Fig. 9. Execution times (seconds) for the ISCST3 code with pgf90 and ifort compilers on the Pentium 4 Xeon processor for the choices of switches shown in Table 1.

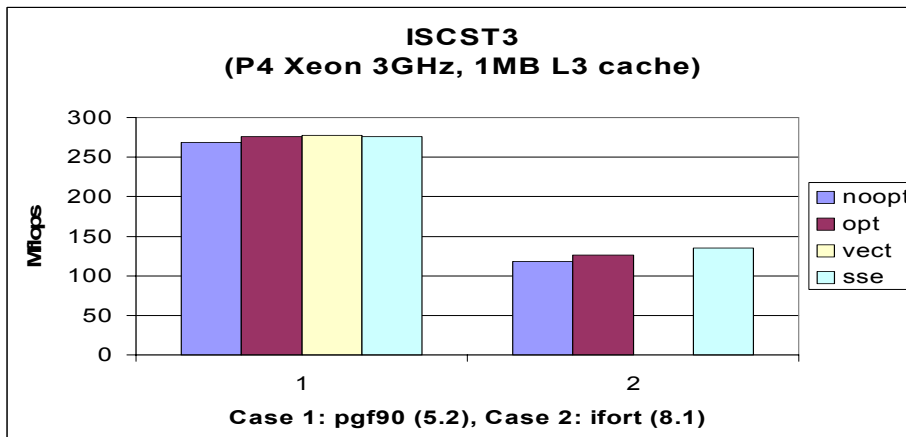


Fig. 10. Mflops for the ISCST3 code with pgf90 and ifort compilers on the Pentium 4 Xeon processor for the choices of switches shown in Table 1.

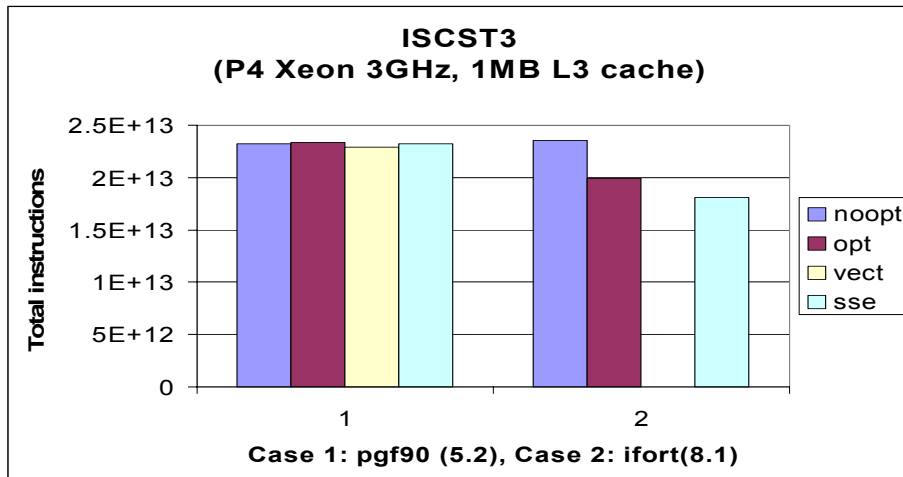


Fig. 11. Total instructions for the ISCST3 code with pgf90 and ifort compilers on the Pentium 4 Xeon processor for the choices of switches shown in Table 1.

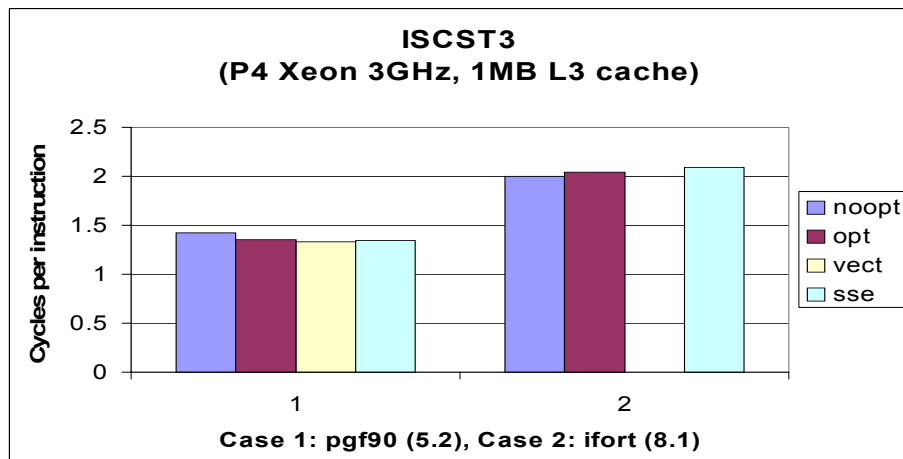


Fig. 12. Cycles per instruction for the ISCST3 code with pgf90 and ifort compilers on the Pentium 4 Xeon processor for the choices of switches shown in Table 1.

Overview of AERMOD Results

The results for the AERMOD code are remarkably similar to those for ISCST3. There is little change in process time with increasing optimization (Figure 13) or Mflops rates (Figure 14). Again the Intel compiler lags the Portland pgf90 compiler. There is some variability in instruction counts (Figure 15) between the two compilers, but not a great deal. Also, the mean number of cycles between instruction issue is

typically 2, or less (Figure 16), suggesting poor vector code character. The reasons for these results are described in the section on Performance Inhibitors.

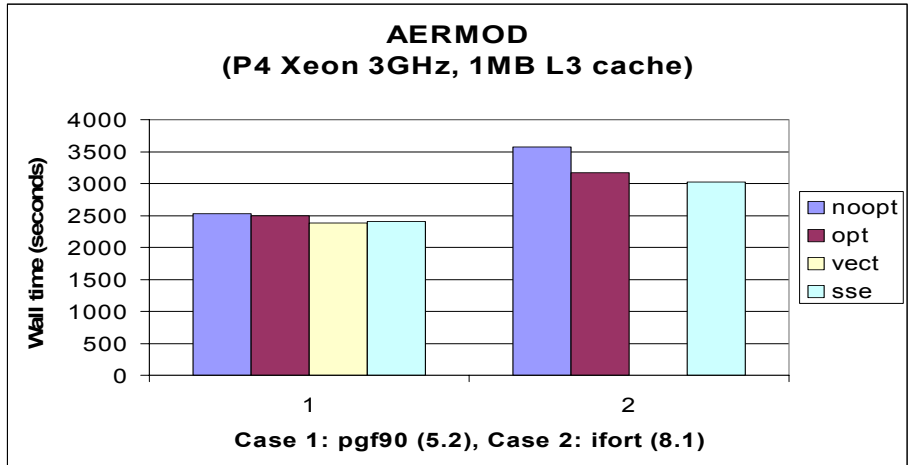


Fig. 13. Execution times (seconds) for the AERMOD code with pgf90 and ifort compilers on the Pentium 4 Xeon processor for the choices of switches shown in Table 1.

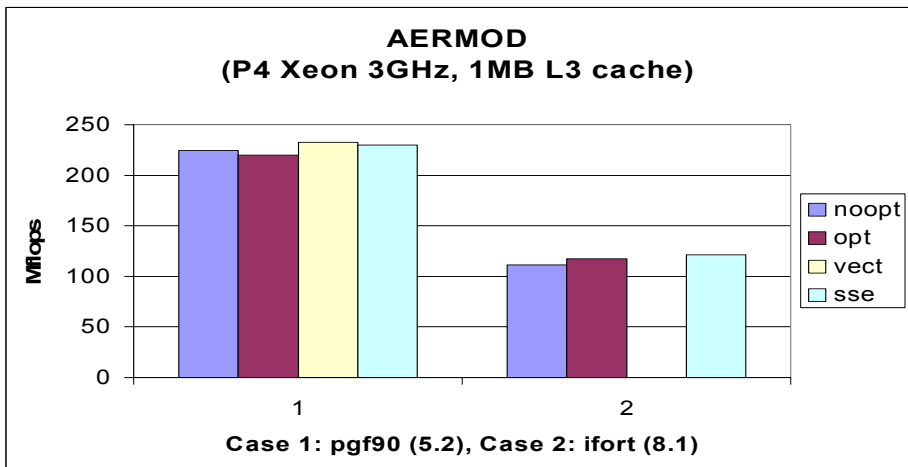


Fig. 14. Mflops for the AERMOD code with pgf90 and ifort compilers on the Pentium 4 Xeon processor for the choices of switches shown in Table 1.

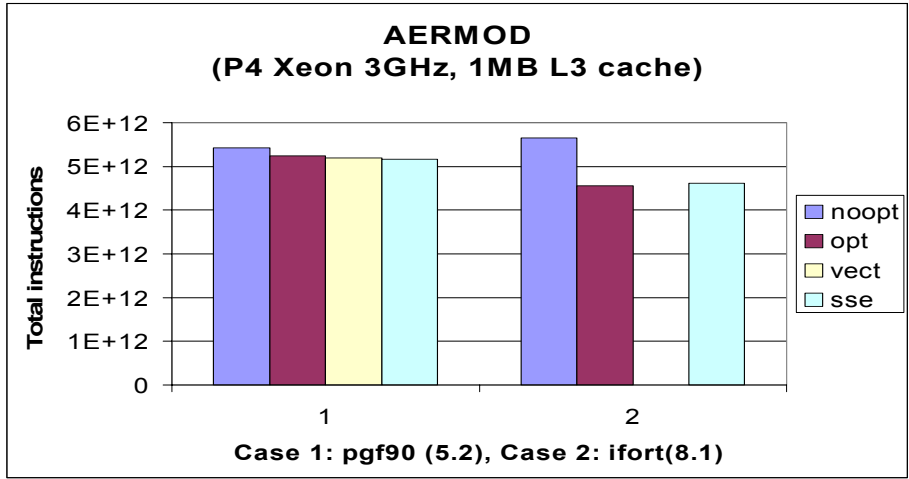


Fig. 15. Total instructions for the AERMOD code with pgf90 and ifort compilers on the Pentium 4 Xeon processor for the choices of switches shown in Table 1.

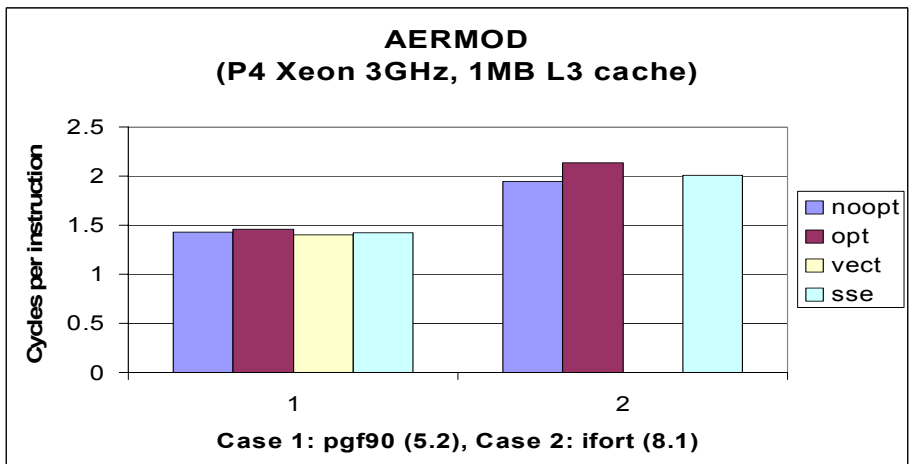


Fig. 16. Cycles per instruction for the AERMOD code with pgf90 and ifort compilers on the Pentium 4 Xeon processor for the choices of switches shown in Table 1.

Performance Inhibition

This section presents a deeper analysis of the performance inhibiting factors observed in the above discussion. This discussion is restricted to three codes: SOM, POM and AERMOD with the Portland compiler because of the burden of PAPI data counter collection requiring 17 separate executions. The following summary is separated into groups of counters that give salient clues to performance inhibition (with explicit ref-

erence to the counter names in the Appendix). The division is into events for stalled cycles, cache, table lookaside buffer, and branching instructions. The focus will be to explain (a) the steady decline in the Mflops rate for POM with increasing problem size, and (b) the total lack of performance gain for AERMOD with increasing levels of compiler optimization. The issue with POM was clear in Figure 6, but is also reflected in the vector/SIMD instruction rates (PAPI_VEC_INS) in Figure 17.

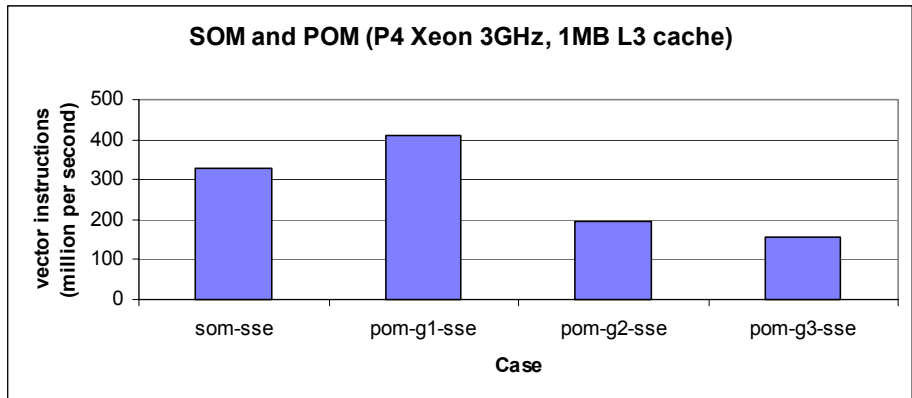


Fig. 17. Vector/SIMD instructions issued (million per second) for SOM and POM (grids 1 to 3) when SSE is enabled.

Memory access and stalled cycles

All three applications, SOM, POM and AERMOD, display vigorous memory activity as shown for load and store instructions (PAPI_LST_INS) in Figure 18 each with the four groups of compiler switches in Table 1. This shows that both high and low Mflops rates can be associated with large memory instruction rates. However, a correlation exists between Mflops rates and cycles stalled on any resource (PAPI_RES_STL) as shown in Figure 19. This shows a cluster of points due to SOM at the left most margin (and extreme right), with two distinct trend lines due to POM and AERMOD below 500 Mflops. Stalled cycles refer to processor stalls when one instruction depends on another. These result in lost performance so it is important to determine which resource is responsible. A stalled processor pipeline can be due to either data not being available in a register (or cache), in which case it needs to be fetched from memory, or, because of mispredicted logical branches, which results in a pipeline flush with a restart using the correct branch. In either case performance is degraded.

The next subsection discusses cache events and the one following shows branch instructions.

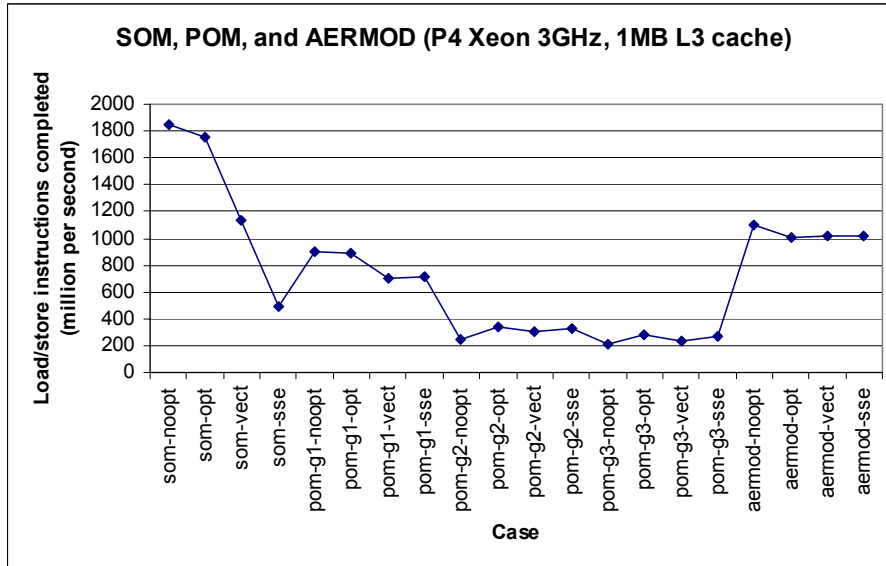


Fig. 18. Load/store instruction rates (million per second) for SOM, POM and AERMOD with the four sets of compiler switches listed in Table 1 in each case. Note that SOM and AERMOD show the largest values.

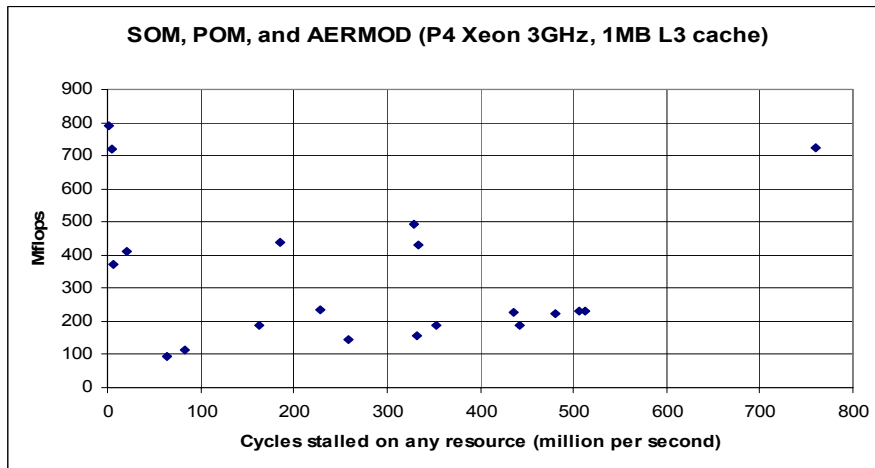


Fig. 19. Mflops versus the rate of stalled cycles for SOM, POM and AERMOD.

Cache events

The Pentium 4 Xeon has three levels of cache: L1, L2 and L3, and the PAPI library allows for the collection of counts for eight cache events (see the Cache Access cate-

gory in the Appendix). For a trend of MFlops versus L1 data cache misses per second (PAPI_L1_DCM) there are groupings similar to those of Figure 19 with a SOM cluster above 700 Mflops and an AERMOD cluster close to 200 Mflops. If attention is restricted to POM for GRID 1, 2 and 3 a clear trend becomes obvious. Figure 20 shows two frames: (a) clusters for all three grids, and (b) results for GRID 2 and 3 with a trend line. It is clear that while the smallest problem size (GRID 1) shows the highest Mflops rates and presumably, L1 cache-friendly behavior, the two larger problem sizes (GRID 2 and 3) fall on a clear trend line in a region of smaller Mflops ranges. This trend is also visible for L2 (Figure 21) and L3 (Figure 22) cache events but there is a convergence of the GRID 1 cluster towards the same trend line. This analysis does not show *why* there is performance inhibition for GRID 2 and 3, but it does show *where* performance enhancement activity could focus and what expected improvement is possible. The implication is that application data locality and a good match to cache properties is vital to good performance.

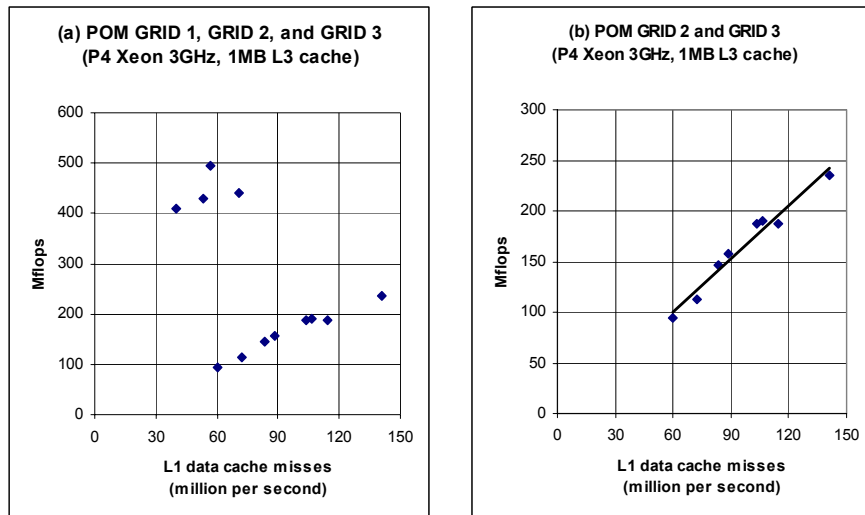


Fig. 20. Mflops versus L1 data cache misses (million per second) for the POM code with (a) all three problem sizes, and (b) only the two larger problem sizes. Note the difference in the Mflops scale.

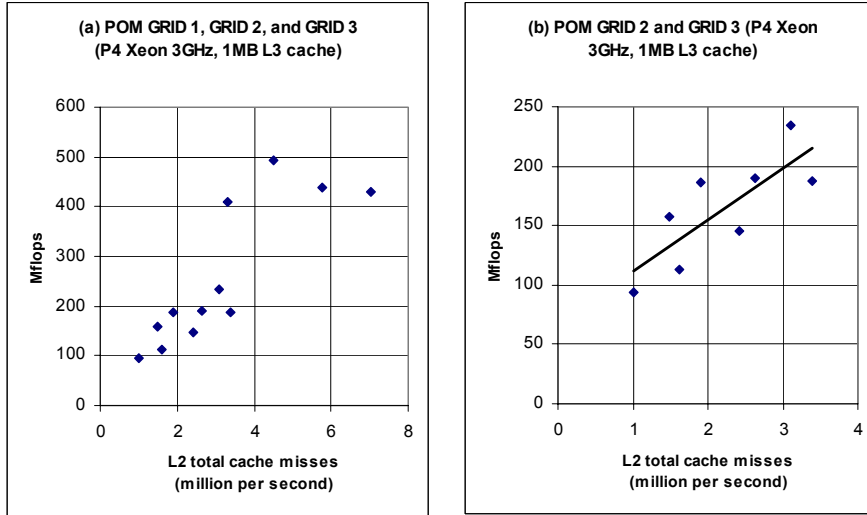


Fig. 21. Mflops versus L2 total cache misses (million per second) for the POM code with (a) all three problem sizes, and (b) only the two larger problem sizes. Note the difference in scales.

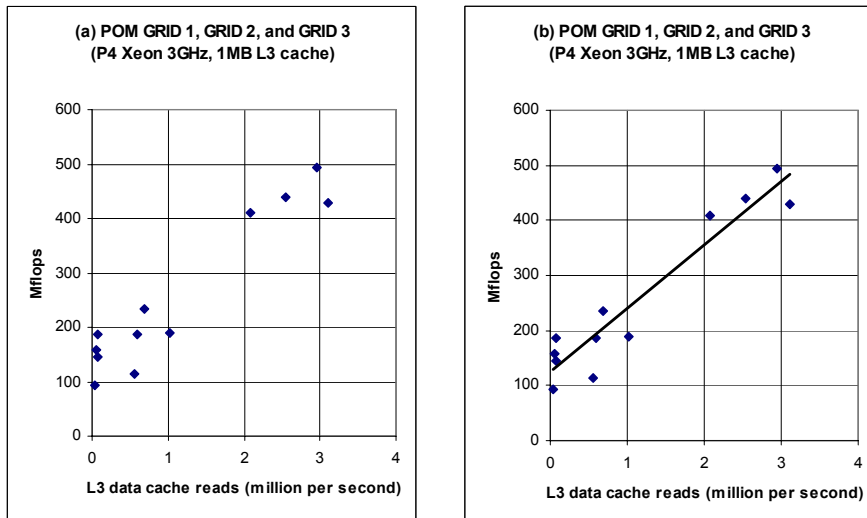


Fig. 22. Mflops versus L3 data cache reads (million per second) for the POM code for all three problem sizes, (a) without and (b) with a trend line.

Table Lookaside Buffer and Branching Events

PAPI offers counters for translation lookaside buffer (TLB) miss events for both instruction and data. The TLB is a small buffer (or cache) to which the processor pre-

sents a virtual memory address and looks up a table for a translation to a physical memory address. If the address is found in the TLB table then there is a hit (no translation is required) and the processor continues. The TLB buffer is usually small, and efficiency depends on hit rates as high as 98%. If the translation is not found (a TLB miss) then several cycles are lost while the physical address is translated. The methodology of handling a TLB miss can vary and is not discussed further here. Suffice it to say that any TLB misses are another form of degraded performance.

The results of instruction TBL misses for the AERMOD code are particularly revealing. Figure 23 shows rates for instruction TLB misses for SOM, POM (all three grids), and AERMOD, each with the four groups of compiler switches in Table 1. The instruction TLB misses (PAPI_TLB_IM) measured for AERMOD are some 85 to 100 times larger than those for SOM or POM so it is clear that this behavior is the performance inhibiting factor for AERMOD.

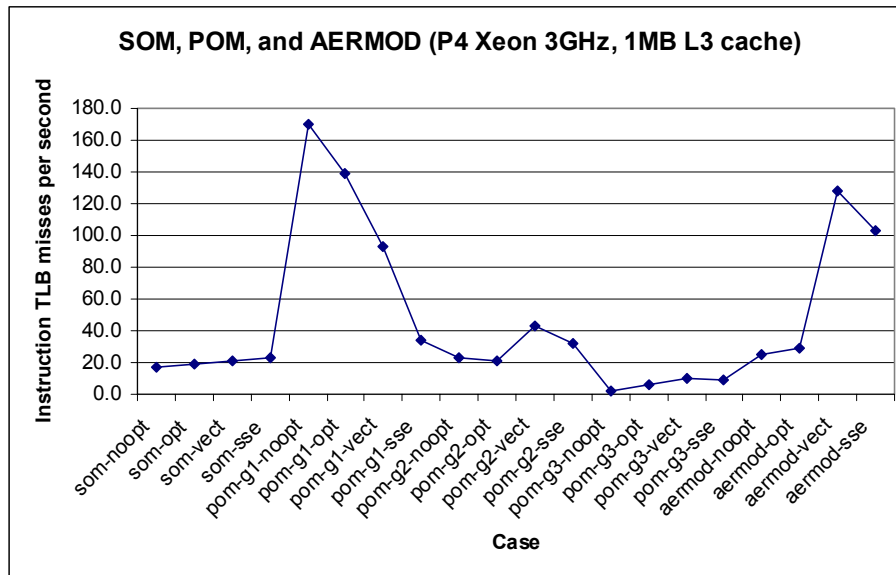


Fig. 23. Instruction TLB miss rates for SOM, POM (all three problem sizes), and AERMOD, with the four sets of compiler switches from Table 1 in each case. Note that the values for AERMOD (last four points on the right) have been scaled down in magnitude by 1/100 to fit the scale shown.

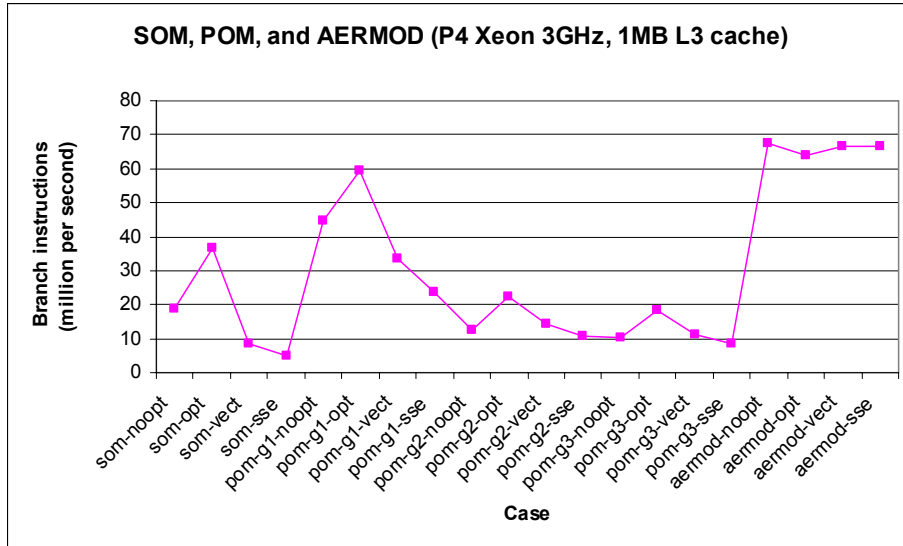


Fig. 24. Branch instruction rates (million per second) for SOM, POM (all three problem sizes), and AERMOD, with the four sets of compiler switches in each case. Note that the values for SOM (first four points on the left) and AERMOD (last four points on the right) have been scaled down in magnitude by 1/5 to fit the scale shown.

For branch instructions PAPI offers five counters and Figure 24 shows the rate of branch instruction issue for SOM, POM (all three grids), and AERMOD, each with the four groups of compiler switches in Table 1. The rate of branch instruction issue rate for AERMOD is at least three times larger than that for SOM, but this may not be as critical, or as important an issue as the TLB miss rate discussed above.

I/O Performance in Air Quality Models

Performance results for I/O was captured in time series snapshots with CSPM [9]. Unfortunately, CSPM includes all system I/O in the time interval. Nevertheless, in dedicated mode, patterns due to the application are clearly visible when I/O activity is coherent. Figure 25 shows such a snapshot where the window is approximately 11.5 minutes wide with subdivisions at 40 second intervals. The right and left margins are, respectively, at 67 and (approximately) 55.5 minutes after execution start. A distinct group of storage reads is visible at the right. This group now appears at the left in the snapshot shown in Figure 26 taken at 75 minutes after execution start and several more groups of storage reads have appeared. Note the cyclical pattern of reads: 100 to 120 seconds of no I/O activity followed by alternating intervals of similar duration with intense read activity. This demonstrates that ISCST3 is I/O bound and this storage access pattern leads to poor performance even on advanced architectures such as the IBM Power 3.

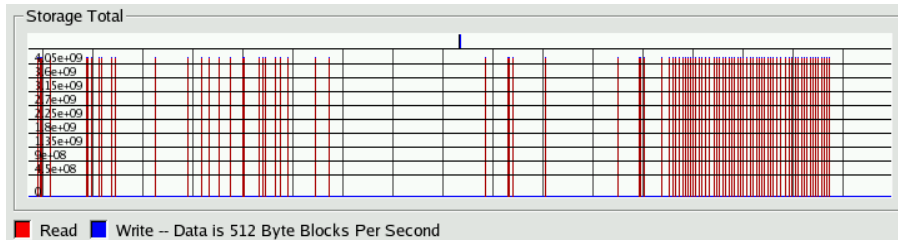


Fig. 25. Time series of ISCST3 storage reads and writes produced by CSPM at 67 minutes after execution start.

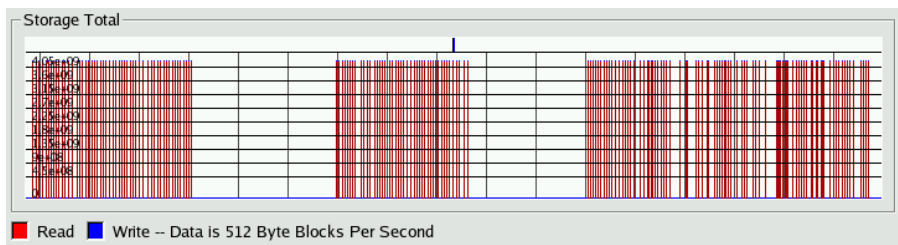


Fig. 26. Time series of ISCST3 storage reads and writes produced by CSPM at 75 minutes after execution start.

Figure 27 shows a CSPM I/O snapshot for AERMOD where the window is approximately 11.5 minutes wide with subdivisions at 40 second intervals. The right and left margins are, respectively, at 37 and (approximately) 25.5 minutes after execution start. A distinct group of storage reads is visible at the right, but is of lower density compared to the ISCST3 case. Note the cyclical pattern of reads: sparse I/O activity over intervals of several minutes followed by alternating intervals of duration less than a minute showing more intense read activity. These tools have also been applied in the analysis of I/O performance in other major Air Quality models. It is characteristic of AQM codes that they produce voluminous output files and read large input data sets, and therefore optimization of I/O is an important consideration in improving overall performance.

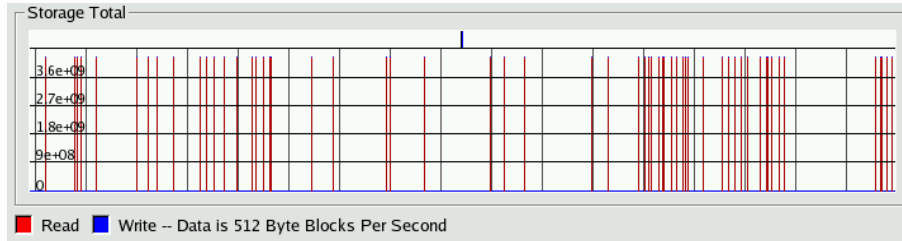


Fig. 27. Time series of AERMOD storage reads and writes produced by CSPM at 37 minutes after execution start.

Conclusions

The PAPI and CSPM interfaces were applied to measure performance attributes of two fortran compilers on the Intel Pentium 4 Xeon platform for four different cases. Two of the cases, SOM and POM, displayed good vector code characteristics that encourages strong compiler optimizations such as vectorization, idiom recognition, etc. The best performance results were for the SOM case where most of the arithmetic work appears in a double loop nest. As a result when the SOM code is presented to either compiler, it benefits from the full power of the optimization transformations they can apply. The POM, while still a good vector code, benefits less, from compiler optimizations because work is distributed over hundreds of loops in some ten subroutines that account for 90% of the total time. Both these examples showed that good vector performance corresponds to a mean number of cycles between instruction issue in the range 4 to 9 depending on optimization levels. Mflop rates measured in these two cases were in the range 200 to 900 Mflops. It was observed for the POM case that performance declined as problem size scaled upward. The differentiator for this behavior was memory access where the larger problem sizes showed a simple correlation between Mflops and data cache misses in a lower performance band compared to the smallest problem size.

By contrast the two AQM's, studied here, ISCST3 and AERMOD, delivered inferior performance. At best, Mflop rates were of the order of 250 and the mean number of clock cycles between instruction issue were of the order of 2 or less. Also a difference in the behavior of the two compilers with these AQM models was observed. Investigation of the I/O behavior showed that in the case of ISCST3, and to a lesser degree in AERMOD, this was a performance inhibiting factor. A deeper analysis of AERMOD with PAPI events showed that a high level of instruction TLB misses is the prime cause of degraded performance.

A fundamental result is that models developed to effectively utilize vector register architectures, also receive very significant performance boosts from optimization and SSE2 instructions on cache-based commodity hardware. This was observed to be the case for both compilers with switches to enable SSE2 instructions. The negligible performance improvements from increasing levels of compiler optimization for the two AQM's suggest that significant code restructuring should be attempted if the full po-

tential benefit of current (and future) commodity processors is to be reached. In this respect the performance metrics accessible with the PAPI interface should provide quantitative information along the path to improved performance during source code analysis.

References

- [1] CAMx was developed by ENVIRON Corp. and is available at <http://www.camx.com>.
- [2] CMAQ was developed in the Atmospheric Modeling Division (AMD) of the NOAA Air Resources Laboratory (ARL) in collaboration with the U.S. EPA's National Exposure Research Laboratory (NERL) and is distributed by CMAS at <http://www.cmascenter.org>.
- [3] George Delic, Performance Attributes for Code and Workload Analysis on Cray X-MP and Y-MP Systems, *International Journal of Supercomputer Applications*, 7 (1993), pp 304-336.
- [4] George Delic and R. I. Haller, Factor Analysis of Applications Performance Data for the Cray Y-MP, *International Journal of Supercomputer Applications and High Performance Computing*, 10 (1996), pp. 91-113.
- [5] Performance Application Programming Interface, <http://icl.cs.utk.edu/papi>. Note that the use of PAPI requires a Linux kernel patch (as described in the distribution).
- [6] ISCST3, <http://home.pes.com/iscst3.htm>.
- [7] U.S. EPA, Technology Transfer Network, Support Center for Regulatory Air Models <http://www.epa.gov/scram001/>.
- [8] See <http://www.hiperism.com> for Technical Reports HCTR-2004-2 (SOM) and HCTR-2004-3 (POM).
- [9] Complete System Performance Monitor <http://cspm.sourceforge.net/index.htm>.

Appendix

Table 4. PAPI preset Event Definitions by Category for the Intel Pentium 4 Xeon.

CATEGORY	NAME	DESCRIPTION
Conditional Branching	PAPI_BR_INS	Branch instructions
	PAPI_BR_MSP	Conditional branch instructions mispredicted
	PAPI_BR_NTK	Conditional branch instructions not taken
	PAPI_BR_PRC	Conditional branch instructions correctly predicted
	PAPI_BR_TKN	Conditional branch instructions taken
Floating Point Operations	PAPI_FP_INS	Floating point instructions
	PAPI_FP_OPS	Floating point operations
Instruction Counting	PAPI_TOT_CYC	Total cycles
	PAPI_TOT_IIS	Instructions issued
	PAPI_TOT_INS	Instructions completed
	PAPI_VEC_INS	Vector/SIMD instructions
Cache Access	PAPI_L1_DCM	L1 data cache misses
	PAPI_L1_LDM	L1 load misses
	PAPI_L2_DCR	L2 data cache reads
	PAPI_L2_LDM	L2 load misses
	PAPI_L2_STM	L2 store misses
	PAPI_L2_TCM	L2 total cache misses
	PAPI_L3_DCR	L3 data cache reads
	PAPI_L3_LDM	L3 load misses
Data Access	PAPI_LD_INS	Load instructions
	PAPI_LST_INS	Load/store instructions completed
	PAPI_RES_STL	Cycles stalled on any resource
	PAPI_SR_INS	Store instructions
TLB Operations	PAPI_TLB_DM	Data translation lookaside buffer misses
	PAPI_TLB_IM	Instruction translation lookaside buffer misses
	PAPI_TLB_TL	Total translation lookaside buffer misses