

Performance Analysis of AERMOD on Commodity Platforms

George Delic

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

Abstract. This report examines performance of the AERMOD Air Quality Model in two versions and two data sets with three compilers on Intel and AMD processors using the PAPI performance event library to collect hardware performance counter values where possible. The intent is to identify performance metrics that indicate where performance inhibiting factors occur when the codes execute on commodity hardware. Results for operations, instructions, cycles, cache, table look-aside buffer misses, and branching instructions, are discussed in detail. An execution profile and source code analysis uncovers the causes of performance inhibiting behavior and how they lead to bottle-necks on commodity hardware. Based on this analysis, source code modifications are proposed with a view to potential performance enhancement.

Introduction

This is a progress 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 throughput achievable with commodity hardware for such models because they are in widespread use on these platforms. New results are presented for AERMOD in two versions and with two data sets. These results give insight into the algorithm's performance on commodity architectures. Important performance bottle-necks are identified with the aid of proprietary software to collect and compute performance metrics using a publicly available hardware performance interface. These studies are intended to quantitatively measure performance differences as hardware and programming environments change and to relate these differences to the underlying causes.

Choice of Hardware, Operating System, and Compilers

The hardware used for the results reported here is the Intel Pentium 4 Xeon (P4) and Pentium Xeon 64EMT (P4e) processors. These have processor clock rates of 3GHz and 3.4GHz, respectively. Each is in a dual configuration with a corresponding front side bus (FSB) of 533MHz and 800MHz shared by each pair of processors. The operating system (OS) is HiPERiSM Consulting, LLC's modification of the Linux™

2.6.9 kernel to include a patch that enables access to hardware performance counters. This modification allows the use of the Performance Application Programming Interface (PAPI) performance event library [1] to collect hardware performance counter values as the code executes. Additional results are included for processors from Advanced Micro Devices, Inc. (AMD), however, for Linux systems without the PAPI kernel patch.

For platforms with the PAPI kernel patch, the performance metrics are defined with a view to giving insight into how the application is mapped to the architectural resources by a compiler. The compilers used in these cases were the Portland pgf90/95 (release 6.0), Intel ifort (release 9.0), and Absoft f90/f95 (release 9.0) and the choices of optimization switches are shown in Table 1. For simplicity mnemonics pgf-60, ifc-60, and abf-90 are introduced for the respective compilers. This table also shows that there is wide variability in the compilation times. In a later section additional compilers and optimization switches are also added for these and other non-PAPI platforms (Table 4). It is expected that different compilers will deliver different performance (as will different choices of compiler switches).

All these architectures offer Streaming Single-Instruction-Multiple-Data Extensions (SSE) to enable vectorization of loops operating on multiple elements in a data set with a single operation. This is enabled through a compiler switch (see Table 1) and has been used in these tests.

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.

Table 1. Compiler command and switches.

Compiler and platform*	Compiler optimization switches	Switch group mnemonic**
pgf95 (P4:115s)	-fast -Mvect=sse -tp p7	pgf-60-sse-p4
pgf95 (P4e:383s)	-fast -Mvect=sse -tp p7-64	pgf-60-sse-p4e
ifort (P4:27s)	-tpp7 -xW -O3 -Ob0 -prefetch- -FI	ifc-90-sse-p4
ifort (P4e:31s)	-tpp7 -xW -O3 -Ob0 -prefetch- -FI	ifc-90-sse-p4e
f90 (P4:52s)	-cpu:p7 -O3 -s -ffixed	abf-90-sse-p4
f90 (P4e:68s)	-cpu:p7 -O3 -s -ffixed	abf-90-sse-p4e
* Note that the P4 platform has a 32-bit Linux OS kernel and compilers, while the P4e has a 64-bit Linux OS kernel and 64-bit versions of each compiler. Compile times (in seconds) are shown in the parentheses.		
** This mnemonic is used in the following discussion.		

Choice of Benchmarks

The choice of benchmarks includes two versions of the AERMOD code [2] with “small” scenarios that complete in less than an hour. The second code version (AERMOD 04300) is the production version currently in use at the U.S. EPA and is publicly available. Most of the detailed performance metric results reported here are

for the earlier version (in Figs. 1 to 9 and Tables 1 to 3). A separate section includes results for AERMOD 04300 with a control input data set provided by the U.S. EPA. Also, for AERMOD 04300 (in Figs. 10, 11, and Table 4), some results for AMD platforms are included. AERMOD performance results and analysis for earlier versions of the Portland (5.2) and Intel (8.1) compilers have been presented elsewhere [3].

AERMOD is predominantly a Fortran 77 code developed over ten years ago but has since used (in small part) Fortran 90 features. As such, and typical of that generation of environmental models, AERMOD was developed on a PC platform, with a small memory requirement, poor vector character, and I/O bound performance characteristics. 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 (some details of code structure are described in the section on the AERMOD 04300 execution profile). Despite these difficulties parallelization efforts are underway. The AERMOD code describes pollutant dispersion and deposition and is now an official regulatory model for new source reviews and other permitting applications. AERMOD and other AQM's are available at the U.S. EPA's Support Center for Regulatory Air Models (EPA-SCRAM) [2]. AERMOD, enjoys constant use with scenarios that may require weeks of wall clock time and for this reason there is considerable interest in finding ways to improve performance.

Hardware Performance Events

The PAPI [1] interface defines over a hundred hardware performance events, but not all of these events are available on all platforms. For the Intel hardware under discussion the number of hardware events that can be collected are, respectively, 28 (P4) and 25 (P4e) and Table 2 shows only events that are common to them. Not all events can be collected in a single execution due to the fact that the number of hardware counters is small (typically four). Thus, multiple executions are needed to collect all available events on any given platform. Performance metrics are defined using the PAPI events and measured in the expectation that they will give insight into how resource utilization differs between compilers. The process time (PTIME) reported here is obtained from the hardware performance counter interface. The AMD platforms discussed in a later section did not have the PAPI interface because the Linux kernel did not have the required kernel patch. In all cases timing information was also obtained by calls to the Fortran `system_clock` procedure.

Performance Metrics

Rate performance metrics

Rate metrics have the suffix “_rate” (except for MFLOPS) and some examples include `TOT_CYC_rate`, `TOT_INS_rate`, `BR_INS_rate`, `L1_ICA_rate`, and

TLB_ICM_rate. This naming convention uses the corresponding PAPI event name in Table 2 divided by the process time (usually in units of million per second). The following discussion will use those rate metrics of relevance in identifying bottle-necks in AERMOD.

Table 2. PAPI events common to the Intel P4 and P4e.

Category	PAPI Name	Description
Conditional Branching	PAPI_BR_INS	Total 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_L1_ICA	L1 instruction cache accesses
	PAPI_L1_ICM	L1 instruction cache misses
	PAPI_L2_LDM	L2 load misses
	PAPI_L2_STM	L2 store misses
	PAPI_L2_TCM	L2 total cache 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
Translation lookaside buffer (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

Profiling and code performance

While not a metric, execution profiling is useful in determining where “hot spots” occur in the source code by measuring (cumulative) time consumed during the code execution. A profile of AERMOD is discussed to identify the code characteristics and correlate them with the values of performance metrics.

AERMOD Performance Results

Operations, instructions, and cycles

Figure 1 shows the process time for AERMOD on P4 and P4e platforms. The left and right hand half of Figure 1 shows, respectively, the P4 and P4e results. Each group of executions corresponds to the choice of compiler switches listed in Table 1. Comparing the results on the two platforms shows the shortest times are for the abf-90 cases with 1,597 and 1,226 seconds, respectively. The inter-procedural analysis (IPA) optimizations have not been enabled because ifc-90 will not produce an executable when fortran and C language modules are used together and `-ipo` is enabled. The C language modules are required to collect performance events. However, with these removed, and the `-ipo` switch enabled for ifort, the execution time on the P4e for ifc-90 is only reduced from 1,509 to 1,464 seconds (a 3% change). Compared to this the Absoft f90 v9.0 compiler is still 16% faster. Figure 2 shows the instruction rates for instructions issued and completed by AERMOD on P4 and P4e platforms with the three compilers. The obvious feature is that the abf-90 compiler has the lowest values on both platforms.

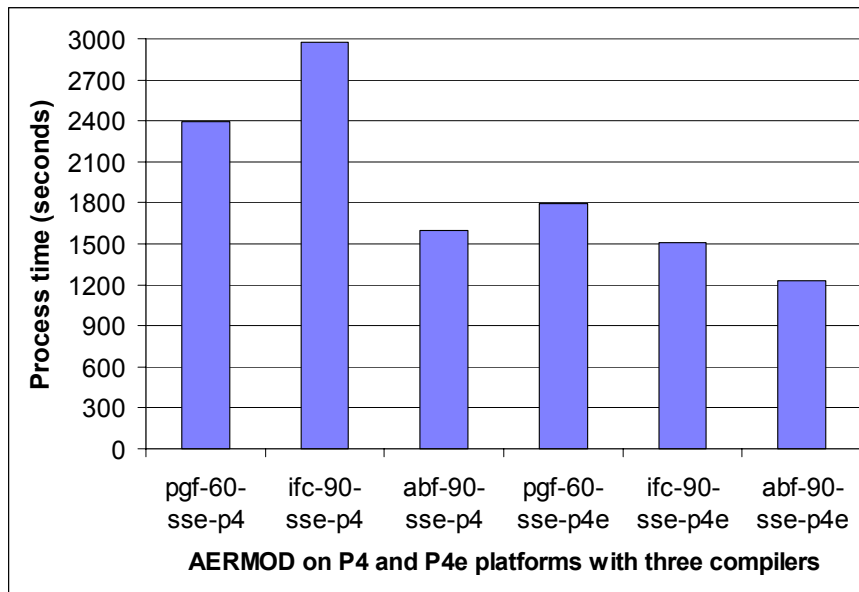


Fig. 1. Process time (seconds) for AERMOD with pgf, ifc, and abf compilers on P4 and P4e processors (left and right half, respectively). Each compiler has the group of compiler switches defined in Table 1.

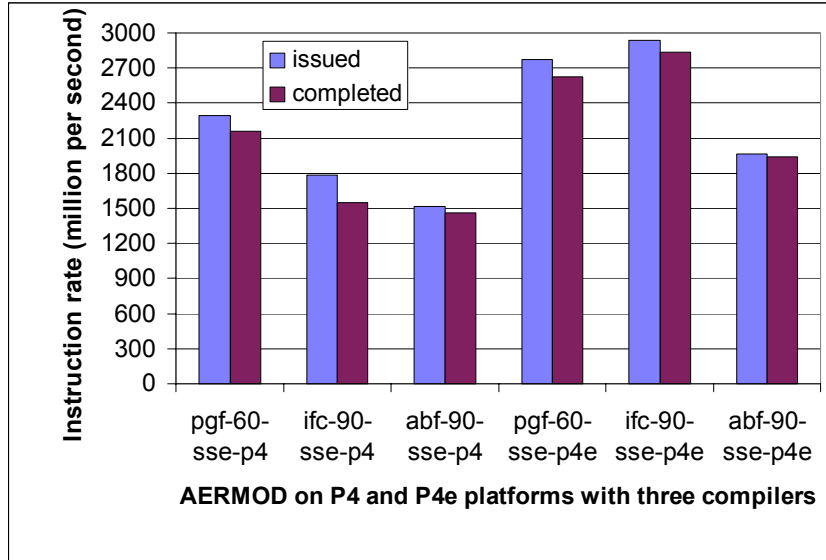


Fig. 2. Rates of instructions issued and completed for AERMOD with pgf, ifc, and abf compilers on P4 and P4e processors (left and right half, respectively). Each compiler has the group of compiler switches defined in Table 1.

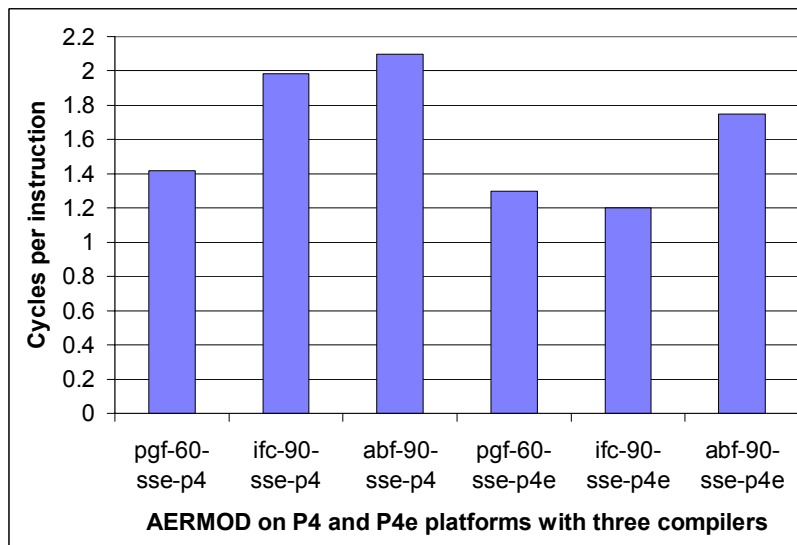


Fig. 3. Cycles per instruction for AERMOD with pgf, ifc, and abf compilers on P4 and P4e processors (left and right half, respectively). Each compiler has the group of compiler switches defined in Table 1.

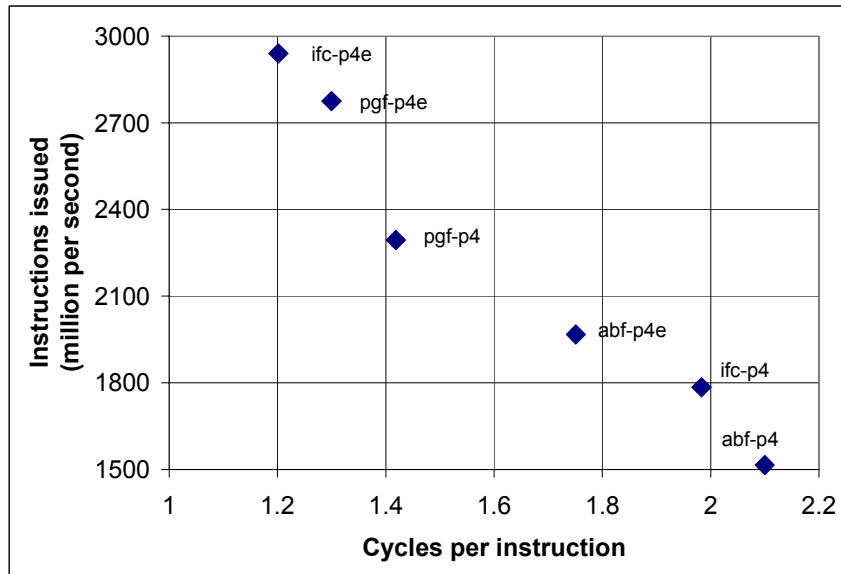


Fig. 4. Instruction issue rate versus cycles per instruction for AERMOD with pgf, ifc, and abf compilers on P4 and P4e processors. Each compiler has the group of compiler switches defined in Table 1.

Another interesting metric related to instructions is the number of cycles per instruction (CPI). This is the mean number of cycles between instruction issue. Figure 3 shows the CPI of all three compilers on both platforms and the largest CPI value is for the compiler with the lowest execution time. For another view, Figure 4 shows instructions issued versus CPI. The CPI metric is not an unambiguous indicator since, for example, it does not sufficiently differentiate ifc-p4 and abf-p4. However, the problem size is fixed in this benchmark and therefore the same amount of total work is performed by all three compilers. They differ only in how they allocate resources to perform this work, i.e., how each compiler maps the application to the architecture. Thus, qualitatively, a larger CPI value means that more operations are performed per instruction. It is instructive to further investigate the possible sources of this execution time difference between the three compilers at the hardware event level.

The sse optimizations allow the compiler on 64 bit hardware (with a 64 bit kernel) to use the enhanced sse instruction set. This approach takes advantage of the availability of considerably more hardware resources on the P4e compared to the P4. For this reason all three compilers have sse optimizations enabled. However, this gives little performance gain for AERMOD because of the lack of vector loop structure and the predominance of control transfer instructions (as discussed below). One side effect when sse instructions predominate is that the values reported by the PAPI event counter PAPI_MFLOPS under-estimates Mflops. This is because this counter uses floating point operation counts and not sse events. Nevertheless a simple esti-

mate of Mflops is shown in Table 3 where it is evident that the range is approximately a factor of two from pgf-60-sse-p4 to abf-90-sse-p4e.

Table 3. Mflops estimates for P4 and P4e.

Switch group mnemonic	Execution time (seconds)	Mflops
pgf-60-sse-p4	2391	230
ifc-90-sse-p4	2976	185
abf-90-sse-p4	1597	344
pgf-60-sse-p4e	1793	307
ifc-90-sse-p4e	1509	365
abf-90-sse-p4e	1226	449

One type of control transfer instructions, namely, branch instruction rates, is shown in Figure 5 for AERMOD on P4 and P4e platforms. The left and right hand half of Figure 5 shows, respectively, the P4 and P4e results. The lowest values observed are for the Absoft f90 compiler. It was noted previously [3] that AERMOD reports branch instruction rates that are more than an order of magnitude larger than those shown by good vector code on the same platforms. Therefore, the fact that the Absoft compiler optimizations reduce the branch instruction rates correlates positively with higher Mflops rates. Presumably a reduction of this type of control transfer instruction is due to a more efficient use of hardware resources when compared to the other two compilers.

Memory footprint

In comparing performance of compilers and processors the memory behavior is of special interest. Figure 6 shows instruction rates for load (LD_INS_rate), store (SR_INS_rate), and the sum of the two (MEM_TOT_rate). In general, for AERMOD, the rate of total memory instructions issued is voluminous. A high rate of memory instruction issue need not be an indicator of a performance bottleneck. Benchmarks with good vector character that deliver of the order of 1Gflop on a P4 can also show high memory access rates [3]. However, what is interesting (and important for the case of AERMOD) is that the compiler with the lowest execution time is also the one with the lowest memory instruction rate. The fact that these two metrics correlate should not be surprising because the commodity architectures compromise on memory bandwidth and latency. Thus, a memory intensive application, without a dominant vector code character (as is AERMOD), is performance constricted on commodity architectures where memory bandwidth is limited by the FSB and cache design. The consequence of AERMOD's memory footprint is that the path to memory can become a limiting critical resource and this is explored in the next two sections.

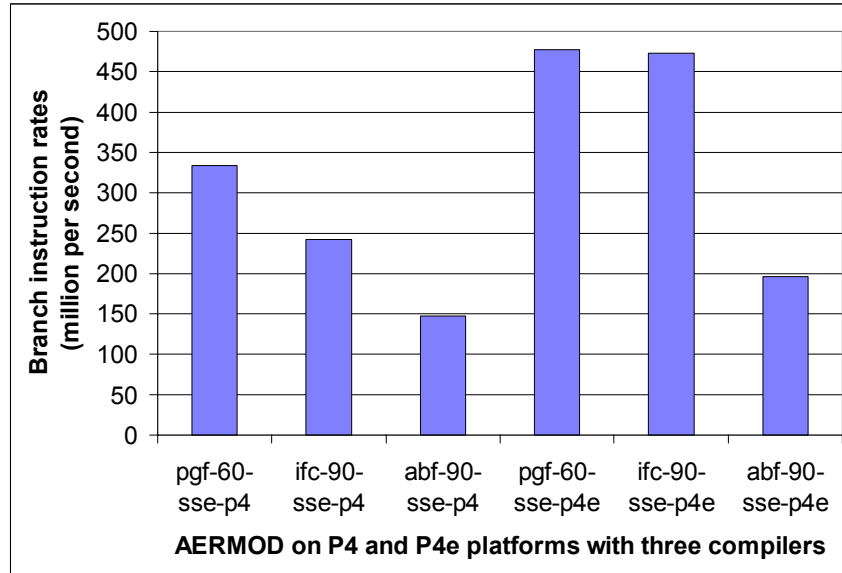


Fig. 5. Branch instruction rates for AERMOD with pgf, ifc, and abf compilers on P4 and P4e processors (left and right half, respectively). Each compiler has the group of compiler switches defined in Table 1.

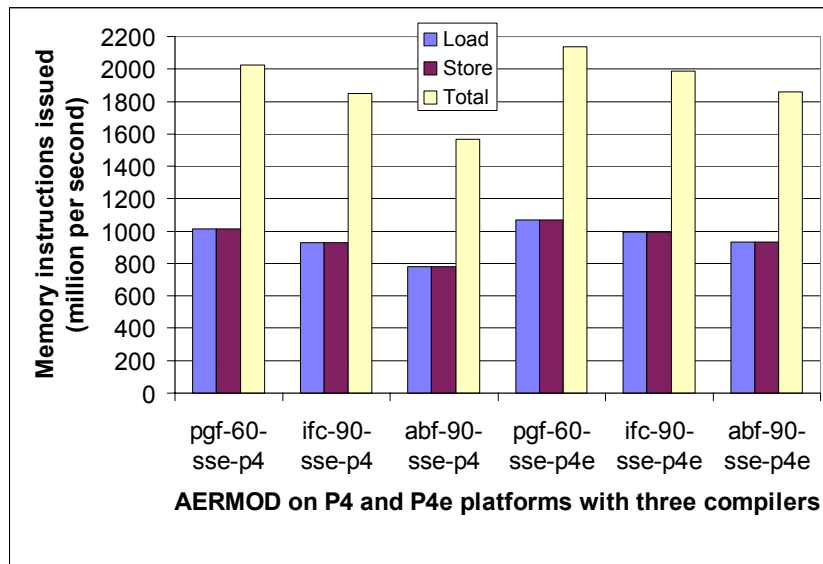


Fig. 6. Memory instruction rates for AERMOD with pgf, ifc, and abf compilers on P4 and P4e processors (left and right half, respectively). Each compiler has the group of compiler switches defined in Table 1.

TLB cache usage

Between the processor and the first level of cache (L1) there is the TLB cache. The translation lookaside buffer (TLB) is a small buffer (or cache) to which the processor presents 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 computed) 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. Therefore TLB misses degrade performance. PAPI offers counters for TLB miss events for both instruction and data (see Table 2). In the case of AERMOD it is the instruction TLB misses that are critical because of the voluminous incidence of control transfer instructions. Figure 7 shows the instruction TLB miss rates observed for AERMOD with the three compilers on the P4 and P4e platforms. From this graph it is clear that the execution with the shortest time has the lowest number of instruction TLB cache misses in each group of three compilers. There is no such simple trend with the data TLB miss rates. To see this correlation more clearly, Figure 8 shows the execution time versus the instruction TLB miss rate for the three compilers on the P4e platform. Higher instruction TLB miss rates suggest that the processor pipeline stalls more frequently because of a higher rate of control transfer instructions. It appears that the Absoft compiler is more efficient in reducing instruction TLB miss rates through optimization and resource allocation compared to the other two compilers. However, a complete explanation of AERMOD behavior is more subtle, and depends also on cache performance.

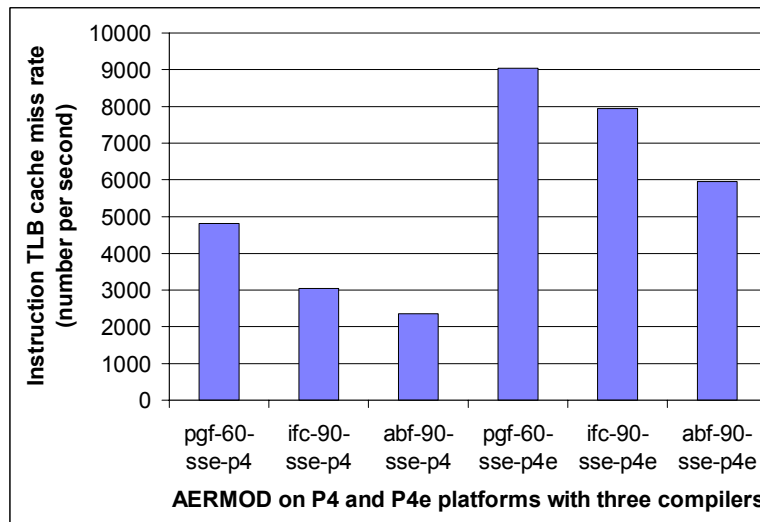


Fig. 7. Instruction TLB cache miss rates for AERMOD with pgf, ifc, and abf compilers on P4 and P4e processors (left and right half, respectively). Each compiler has the group of compiler switches defined in Table 1.

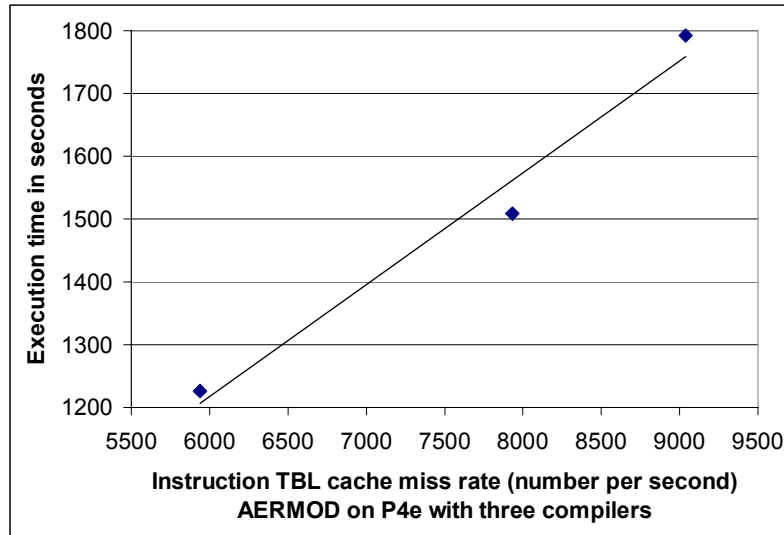


Fig. 8. Execution time versus the instruction TLB cache miss rate for AERMOD with pgf, ifc, and abf compilers on the P4e processor. Each compiler has the group of compiler switches defined in Table 1. A linear regression line is added to show that process time increases with increasing instruction TLB miss rate.

Cache usage

Both the P4 and P4e platforms discussed here have L1 and L2 caches. A cache miss on either of these occurs when data or instructions are not found in the cache and an excursion to higher level cache, or memory, is necessitated. Cache misses result in lost performance because of increasing latency in the memory hierarchy. Memory latency is smallest at the register level and increases by an order of magnitude for a L1 cache reference, and another order of magnitude to access L2 cache. In the case of AERMOD this analysis will focus on the L1 cache behavior.

There is another view of the penalties associated with excursions to cache by the processor. Figure 9 shows the relationship between instruction TLB miss rates and L1 instruction cache access rates. Clearly, increased instruction TLB rates also lead to increased L1 instruction cache access rates.

This suggests that the extremely high data TLB misses for AERMOD are a critical source of performance limitations. They lead to very high memory instruction rates due in part to high TBL instruction miss rates and also in part due to correlated L1 instruction cache access rates. This behavior results directly from the profile of the AERMOD execution and is ameliorated by the efficiency of the Absoft compiler in minimizing the consequences of this behavior.

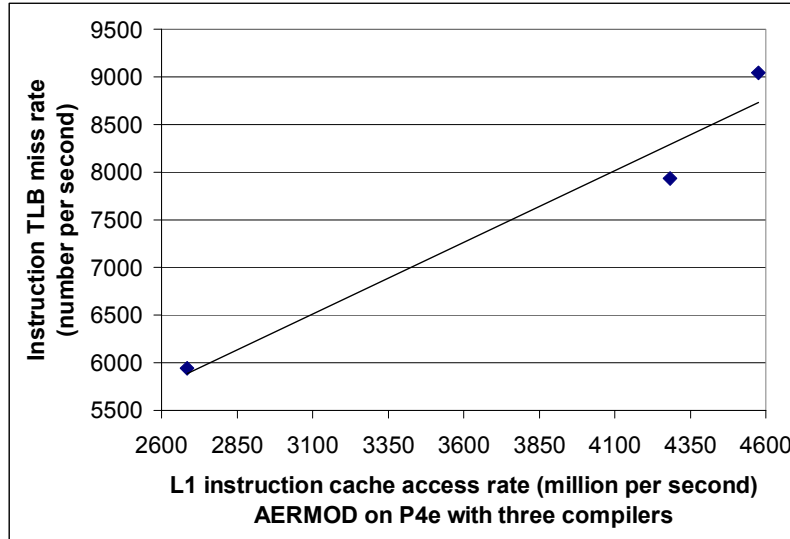


Fig. 9. Instruction TLB cache miss rate versus L1 cache access rate for AERMOD with pgf, ifc, and abf compilers on the P4e processors. Each compiler has the group of compiler switches defined in Table 1. A linear regression line is added to show that the cache access rate increases with increasing instruction TLB miss rate.

Additional results for AERMOD 04300

In this section additional benchmark results are presented that include AMD platforms and more choices of compiler options as listed in Table 4. This analysis was undertaken to explore which compiler and switch choices result in the shortest process time. From the numerous alternatives for compiler switches tested those shown in Table 4 represent the best choice for each compiler. Figure 10 shows the times for the AERMOD 04300 production version case on both 64 bit and 32 bit platforms in right and left parts, respectively. The “baseline benchmark” case is the result for the compiler version in current use at the U.S. EPA. The best results (on all platforms) were with the Absoft f90 v9.0 compiler and these are annotated in Figure 10. In view of the diversity of compilers and platforms these timing results are also shown as a frequency graph in Figure 11. Three out of four observations in the lowest time bin are for the Absoft f90 compiler and the spread in performance from top to bottom ranges by a factor of 2.5.

Comparing results on a platform similar to that used for production at the U.S. EPA (x86_32 AMD) the Portland compiler has a process time some some 67% longer than the Absoft result. This is an impressive throughput performance gain by simply changing the compiler. It should be observed that the current pgf90 release is 6.x and this could reduce the difference. However, when this release was tested on an Intel platform (x86_32 Intel) the corresponding process time was still 34% longer than that for Absoft f90. For the 64-bit platforms the Absoft compiler performed best on the In-

tel 64EMT (x86_64 Intel) when compared with the AMD Opteron (x86_64 AMD) where the Portland compiler was not installed. In the 64 bit Intel case the Intel and Portland compilers have a 14% and 39% longer process time when compared to the Absoft f90 compiler. Thus, the overall best result of 799 seconds is for the Intel Pentium 4 Xeon 64EMT with the Absoft compiler.

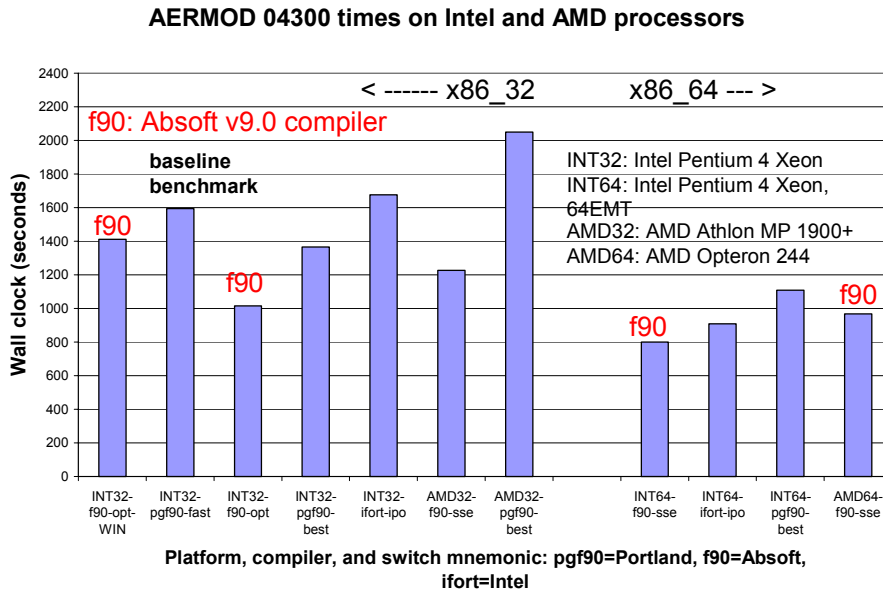


Fig. 10. Run time from Table 4 for AERMOD 04300 with three compilers on five platforms with 32 bit platforms shown on the left and 64 bit platforms on the right. The “baseline benchmark” represents the choice of compiler and switches in use at the U.S. EPA at this time. The switch mnemonics are defined in Table 4 where the corresponding compiler switches are listed.

AERMOD 04300 Execution Profile and Source Code Analysis

AERMOD consists of just under 50,000 lines of fortran code of which 49% are comments and 51% is executable code. There are some 400 subprograms and a calling tree that is some six levels deep. For the profile study the production version, AERMOD 04300, is used with a control data set provided by the U.S. EPA. The structure of the code in both versions is fundamentally the same, but the control data set used here provides a somewhat different source coverage. Nevertheless the iterative structure of the calculation is common to all AERMOD input data: emissions and deposition data is processed in I/O and calculation for each hour of each day in each year included in the scenario. As is discussed below, the iteration over the hour loop repeats the same calculation.

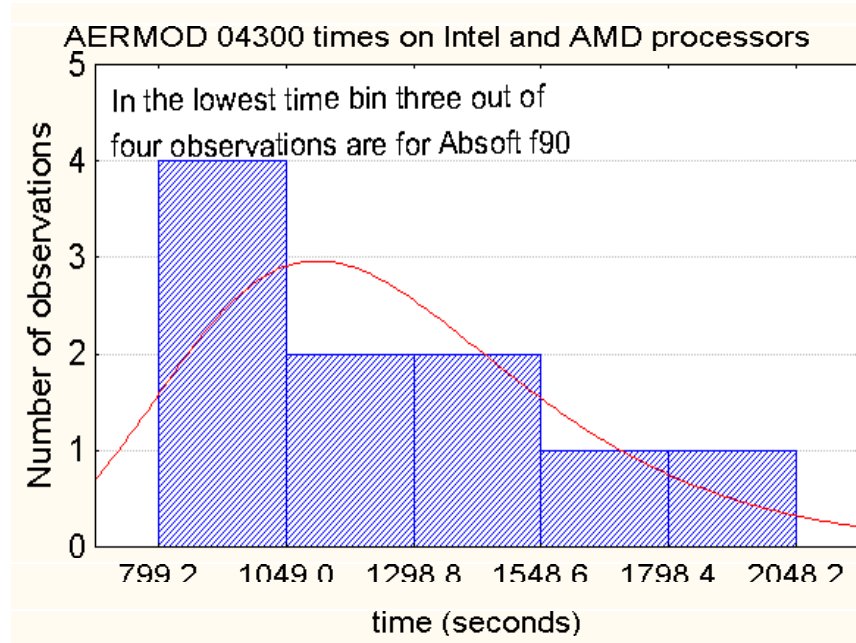


Fig. 11. Run times from Table 4 for AERMOD with three compilers on five platforms as a frequency graph. Of the four observations in the lowest time bin, 3 are for the Absoft f90 compiler and 1 is for the Intel compiler.

The execution profile of AERMOD 04300 is easily performed with the `-Mprof=lines` compiler switch in the `pgf90/95` compiler. Results are shown in Table 5 with those functions that account for 83% of the cumulative process time. Once the important functions are identified code inspection shows some reasons why vector instructions are scarce in AERMOD and why control transfer instructions are so numerous.

In Table 5 the top four routines account for 55% of the total execution time and there is a long list of called procedures. In these routines (and most others shown in the table) execution time is spent in procedures that have less than 100 lines of executable code (typically 58) but do little arithmetic calculation. Furthermore, these procedures occur at the leaves of a deep calling tree and they invariably have no loop structure but consist of simple arithmetic statements and conditional code blocks. These are the reasons for lack of vectorizable loops and the high rates of branching instructions. The other important feature in Table 5 is the voluminous number of calls between these procedures that do little computation per call. Hence one of the causes of high rates of instruction TLB misses.

Of the list of functions shown in Table 5 those that perform little computation per call have a very high calling overhead and should be in-lined to reduce the high cost of control transfer instructions. In-line method are available at the compiler command line level. However, experimentation with more than one level of in-lining produced

little added improvement, suggesting compiler technology is limited when it comes to deep calling trees such as those in AERMOD. Also, as noted in the previous section, interprocedural optimizations (when tried) gave little improvement in performance. This experience with AERMOD suggests that the enhancement of vectorization opportunities combined with manual inlining is best done by source code modifications

Table 4. Compilers, switches, and run times for AERMOD 04300 with three compilers on five platforms as follows: Intel Pentium 4 Xeon (x86_32 Intel) and 64EMT (x86_64 Intel), AMD Athlon MP 1900+ (x86_32 AMD) and Opteron 244 (x86_64 AMD).

Platform	Com- piler	Ver- sion	Time (sec)	Switch mne- monic	Compiler switches
x86_32 Intel (Win)	Absoft	9.0	1411.2	opt	-cpu:p7 -O3 -YEXT_NAMES=LCS - YEXT_SFX=_ -s -YCFRL=1 -ffixed
x86_32 Intel	Port- land	6.0	1595.2	fast	-fast -tp p7 -Minfo -Mlfs -Bstatic
x86_32 Intel	Absoft	9.0	1016.0	opt	-cpu:p7 -O2 -YEXT_NAMES=LCS - YEXT_SFX=_ -s -YCFRL=1 -ffixed
x86_32 Intel	Port- land	6.0	1366.0	best	-fastsse -Mscalarsse -Mcache_align - Minline=size:100 -tp p7 -Minfo -Mlfs - Bstatic
x86_32 Intel	Intel	9.0	1675.2	ipo	-tpp7 -xW -O3 -Ob2 -ipo -static -FI
x86_32 AMD	Absoft	9.0	1226.7	sse	-O3 -YEXT_NAMES=LCS - YEXT_SFX=_ -s -YCFRL=1 -ffixed
x86_32 AMD	Port- land	5.1	2048.2	best	-Mscalarsse -Mcache_align - Minline=size:100 -tp athlonxp -Minfo - Mlfs -Bstatic
x86_64 Intel	Absoft	9.0	799.2	sse	-O3 -YEXT_NAMES=LCS - YEXT_SFX=_ -s -YCFRL=1 -ffixed
x86_64 Intel	Intel	9.0	908.1	ipo	-tpp7 -xW -O3 -Ob2 -ipo -static -FI
x86_64 Intel	Port- land	6.0	1108.5	best	-fastsse -Mscalarsse -Mcache_align - Minline=size:100 -tp p7-64 -Minfo - Bstatic
x86_64 AMD	Absoft	9.0	968.2	sse	-O3 -YEXT_NAMES=LCS - YEXT_SFX=_ -s -YCFRL=1 -ffixed

To better understand the behavior summarized in the execution profile of Table 5 consider the segment of the call tree for the iterative compute kernel in AERMOD as shown in Figure 12. This segment was extracted from the extensive static call tree produced by Fortran Lint source code analysis software [4]. The procedure HRLOOP is called for each day in each year included in the model scenario. It has a DO WHILE loop over each hour of each day and this loop reads hourly pollutant emissions in a DO loop over a list of sources (that can be a long list). Towards the end of the DO WHILE loop averages are computed and written to multiple disk files. At the heart of the loop is the call to CALC to perform arithmetic operations. The type of operation depends on the pollutant source type: point source (PCALC), volume

source (VCALC), area source (ACALC), etc., with optional calls to other processes (PVMRM_CALC). Each of these procedures has an elaborate call tree of its own, however, they all have some common characteristics as summarized in the caller/callee map displayed in Table 6. All the procedures called by CALC in Figure 12 call IBLVAL, GINTRP, LOCATE, and SIGZ. Furthermore, IBLVAL and SIGZ also have multiple calls to GINTRP and LOCATE. Thus the high volume of calls to the top four procedures shown in Table 5 has a simple explanation. Inspection of the heavily called procedures GINTRP and LOCATE show some remarkable inefficiencies. The source code for GINTRP has a single executable arithmetic statement, while LOCATE has a DO WHILE loop containing a conditional block to search an array for the index that brackets a preassigned numerical value. Thus, at least two major examples of high instruction TBL misses are identified: voluminous calls to GINTRP to execute one statement, and numerous calls to LOCATE for a loop over a conditional block. While a good compiler will apply code transformations to ameliorate this poor code structure, the best approach is to eliminate it entirely from the compute kernel of AERMOD.

Table 5. AERMOD 04300 P4 Profile for the Portland compiler.

Function	Number of calls	Time (%)
iblval	398,477,537	37%
gintrp	840,173,126	7%
locate	419,637,478	6%
sigz	606,942,693	5%
plumef	178,865,297	4%
rmssig	606,942,693	3%
szsfcl	606,828,941	3%
vertcbl	213,855,840	3%
centroid	398,477,537	3%
heff	506,954,878	2%
aer_achi	178,865,297	2%
pdf	318,971,409	2%
pwidth	190,012,381	2%
adisz	547,742,975	2%
sigy	259,569,689	2%
		83%

Another important source of control transfer instructions is I/O but this has not been studied here and is still an open issue pending a suitable tool for application I/O metrics. Needless to say, hoisting such I/O out of an iteration in the compute kernel is highly advisable, even if it is at the cost of an increase in memory usage (AERMOD's memory usage is modest and well below 1GB).

The summary in Figure 12 and Table 6 does not represent all the procedures called in the compute kernel but could be considered typical. Further analysis of the other procedures listed in the execution profile of Table 5 yields similar results.

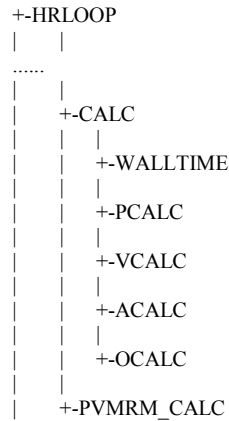


Fig. 12. Segment of the call tree showing the first and second levels of the call tree contained by the DO WHILE loop over each hour of the day.

Table 6. AERMOD 04300 Caller and Callee example for first four procedures in Table 5.

Callee →	IBLVAL	GINTRP	LOCATE	SIGZ
Caller ↓				
PCALC	X	X	X	X
VCALC	X			X
ACALC	X			
OCALC	X	X	X	
PVMRM_CALC		X	X	
IBLVAL		X	X	
SIGZ		X	X	

Conclusions

This study showed the utility of metrics based on PAPI hardware events. Such metrics point to detailed behavioral characteristics of executing code on commodity platforms. They also help to differentiate compilers and provide clues to possible source code structures that could change performance if modified.

Specifically, this performance analysis of AERMOD, shows that it is a memory intensive application with large rates of control transfer instructions such as branching logic and high procedure calling overhead. These features result in large observed rates for branching instructions and instruction TLB misses. In combination these two characteristics of the AERMOD code place a limit on the optimal performance possible from AERMOD on commodity platforms. This is because, by design, commodity hardware solutions offer a cost effective compromise between processor clock rates, cache size, and bandwidth (or latency) to memory. Nevertheless, the important result

of this analysis is that by simply changing hardware platforms and compilers it is possible to see performance enhancement by as much as a factor of two-and-a-half. At this time the best results were delivered by the Absoft f90 v9.0 compiler on the P4e platform.

In its present form, AERMOD gains mostly from improvements in the scalar performance of the hardware. But, despite these observations, a profile of AERMOD performance, followed by code inspection, does suggest that there is scope for performance improvement beyond the range it currently delivers on the P4 and P4e platforms.

An execution profile and source code analysis of the type briefly described in the previous section suggests that the next steps in this project are several in number. The first step is source code modification to simplify the call tree by inline of calls to trivial procedures and removal of some conditional code blocks (to reduce branching instructions and TLB misses). The second step is to expose looping structure to enhance the opportunity for vector code generation by compilers (to increase use of SSE instructions). Examples of changes for this step would include replace DO WHILE by an indexed loop and hoisting of I/O out of the loop. These steps, in combination, are expected to improve serial performance of AERMOD and benefit the large user community that uses the model on stand-alone workstations. Once these serial optimizations are successfully completed, the third step is parallelization of the iterative parts of AERMOD to provide reduced wall clock times for large model studies performed at the U.S. EPA's Linux clusters. These three steps are not trivial, but the rewards are expected to be very significant for model through-put and enhance progress in improving air quality.

Acknowledgments

Part of this work was performed for Lockheed Martin Information Technology, Inc., under the U.S. EPA ITS ESE contract, in support of the U.S. EPA OAQPS/EMAD group and the enthusiastic help from the group members is gratefully acknowledged. Many thanks are due to the Absoft Corporation for making available their fortran compilers and the excellent technical support throughout this arduous study.

References

- [1] 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).
- [2] U.S. EPA, Technology Transfer Network, Support Center for Regulatory Air Models <http://www.epa.gov/scram001/>.
- [3] Delic, 2005: George Delic, Performance Metrics for Ocean and Air Quality Models on Commodity Linux Platforms, presented at the 6th International Conference on Linux Clusters: The HPC Revolution 2005, Chapel Hill, NC, April 26-28, 2005, <http://www.linuxclustersinstitute.org/Linux-HPC-Revolution/Archive/2005techpapers.html>.
- [4] Fortran Lint is described in the Product pages at <http://www.hiperism.com>.