© Copyright, HiPERiSM Consulting, LLC Monday, September 12, 2022

PERFORMANCE METRICS FOR AIR QUALITY MODELS ON COMMODITY PLATFORMS

George Delic*
HiPERiSM Consulting, LLC, Durham, NC

1. 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 though-put achievable with commodity hardware for such models because they are in wide-spread use on these platforms. New results are presented for CMAQ 4.4 that 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. In this report only serial (single processor) results are presented and MPI performance is the subject of a future study.

2. CHOICE OF HARDWARE, OPERATING SYSTEM, AND COMPILERS

The hardware used for the results reported here is the Intel Pentium 3 (P3), Pentium 4 Xeon (P4), and (in the sequel) Pentium Xeon 64EMT (P4emt) processors. These have processor clock rates of 933MHz, 3GHz and 3.4 GHz, respectively. Each is in a dual configuration with a corresponding system bus (FSB) of 133MHz, 533MHz and 800HMz shared by each pair of processors. The operating system (OS) is HiPERiSM Consulting, LLC's modification of the Linux™ 2.4.20 (P3) and 2.6.9 (P4, P4emt) kernels to include a patch that enables access to hardware performance counters. This modification allows the use of the PAPI performance event library (PAPI, 2005) to collect hardware performance counter values as the code executes. The compiler used was the Portland pgf90/95 with release 4.0 (P3) and 6.0 (P4 and P4emt) for the four groups of optimization switches shown in Table 1.

Corresponding author: George Delic, HiPERiSM Consulting, LLC, P.O. Box 569, Chapel Hill, NC 27514-0569; e-mail: george@hiperism.com

All three 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. The Portland compiler specifically enables SSE through a compiler switch (sse in Table 1) and this has been used in these tests.

TABLE 1. Compiler command and switches

Compiler and version*	Compiler optimization switches	Switch group mnemonic
pgf90/95 4.0 (6.0)	–O0 –tp p6(p7)	noopt
	–O2 –tp p6(p7)	opt
	–fast –Mvect –tp p6(p7)	vect
	–fast –Mvect=sse –tp p6(p7)	sse

* Version 4.0 has a P3 target architecture and 6.0 has P4 with the target switch for the latter shown as (p7).

3. CHOICE OF BENCHMARKS

The choice of benchmarks includes two of the tutorial examples that come with the CMAQ distribution. These four examples are: days 1 and 2 at 32 km resolution (Cases 1 and 2), and days 1 and 2 at 8km resolution (Cases 3 and 4). The performance results for Cases 1 (32km) and 3 (8km) are presented here.

4. HARDWARE PERFORMANCE EVENTS

The PAPI (PAPI, 2005) 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, 46 (P3), 28 (P4), and 25 (P4emt). However, not all events can be collected in a single execution due to the fact that the number of hardware counters is small (two for P3 and four for P4). Thus, multiple executions are needed to collect all available events on any given platform. Table 2 lists only events that are common to these three platforms grouped by category. The process time (PTIME) reported here is obtained from the hardware performance counter interface.

TABLE 2. PAPI events common to the Intel P3, P4 and P4emt.

Category	Description	Name
Floating Point Operations	Floating point instructions	PAPI_FP_INS
	Floating point operations	PAPI_FP_OPS
Instruction Counting	Total cycles	PAPI_TOT_CYC
	Instructions issued	PAPI_TOT_IIS
	Instructions completed	PAPI_TOT_INS
Data Access	Cycles stalled on any resource	PAPI_RES_STL
Cache Access	L1 data cache misses	PAPI_L1_DCM
	L1 load misses	PAPI_L1_LDM
	L1 instruction cache accesses	PAPI_L1_ICA
	L1 instruction cache misses	PAPI_L1_ICM
	L2 load misses	PAPI_L2_LDM
	L2 store misses	PAPI_L2_STM
	L2 total cache misses	PAPI_L2_TCM
TLB Operations	Instruction translation lookaside buffer misses	PAPI_TLB_IM

5. PERFORMANCE METRICS

5.1 Rate performance metrics

Rate metrics have the suffix "_rate" (except for MFLOPS) and some examples include TOT_CYC_rate, TOT_INS_rate, and RES_STL_rate. This naming convention uses the corresponding PAPI event name in Table 2 divided by the process time with units of million per second. The following discussion will use those rate metrics of relevance in identifying bottle-necks in CMAQ.

5.2 Ratio performance metrics

In addition to rate metrics, ratios of PAPI events define a set of ratio metrics. Table 3 lists a few examples of ratio metrics used in the following discussion to identify performance bottle-necks in CMAQ. Other rate metrics are introduced as needed.

TABLE 3. Examples of ratio metrics common to the Intel P3 and P4 Xeon.

Description	Name
Memory instructions versus total instructions	MEM_INS_TOT
Memory instructions per floating point instruction	MEM_INS_FPINS
Respectively, L1 instruction, data, and total cache misses per floating point operation	L1_ICM_FPOP L1_DCM_FPOP L1_TCM_FPOP
L2 total cache misses per floating point operation	L2_TCM_FPOP

5.3 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 CMAQ is discussed to identify the compute intensive routines and their code characteristics.

6. CMAQ PERFORMANCE RESULTS

6.1 Operations, instructions, and cycles

Figures 1 and 2, respectively, show the process time and Mflops (million floating point operations per second) for CMAQ on P3 and P4 platforms. This (and all subsequent figures unless otherwise noted) show four executions for each of the Case 1 and Case 3 benchmarks defined in Section 3. Each group of four executions corresponds to the same choice of compiler switches listed in Table 1. Comparing Case 1 (C1) and Case 3 (C3) shows that the higher resolution grid (C3) typically has a 7% lower performance. The left and right hand parts show, respectively, the P3 and P4 results. As expected, the P4 process time is less than the corresponding P3 time. However, on both processors the higher level optimizations (vect and sse) give negligible performance gain suggesting little (if any) vector character in CMAQ code. When sse switches were enabled the measured vector instruction counts were less than four percent of all floating point instructions issued. Thus the gain in performance for CAMQ between P3 and P4 processors can be attributed largely to the improvement in scalar hardware performance. As examples of the latter, Fig. 3 shows the change in the instruction completion rate and a similar increase is observed in the rate at which cycles are completed.

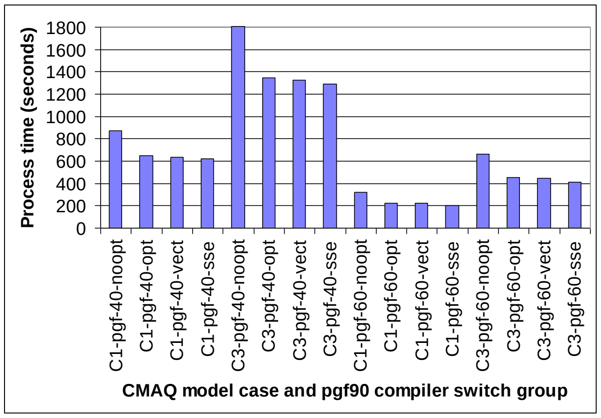

Fig. 1. Process time for CMAQ model tutorial cases 1 (C1) and 3 (C3) on P3 and P4 processors. Each case has the four groups of compiler switches defined in Table 1.

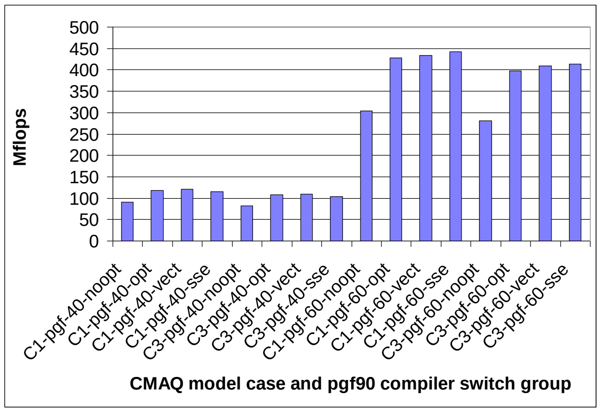

Fig. 2. Mflops for CMAQ model tutorial cases 1 (C1) and 3 (C3) on P3 and P4 processors. Each case has the four groups of compiler switches defined in Table 1.

Of special interest is the memory behavior of CMAQ and Fig. 4 shows the number of cycles stalled on any (memory) resource per unit time (the RES_STL_rate metric). Not surprisingly, there is a distinct improvement for the P4 over the P3 due to the FSB bandwidth and cache size increases. However, there is variability in this metric for the P4 and a deeper investigation of the CMAQ memory footprint is appropriate.

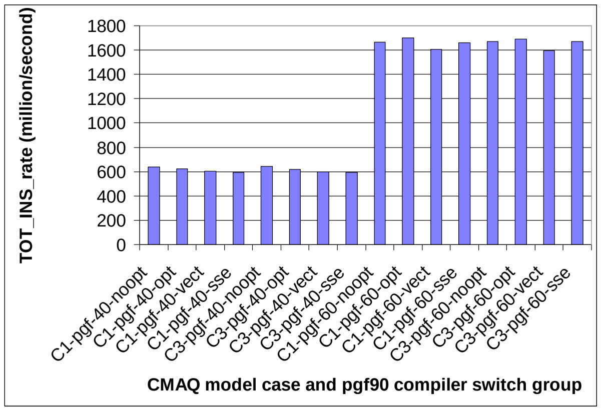

Fig. 3. Instructions completed in million per second for CMAQ model tutorial cases 1 (C1) and 3 (C3) on P3 and P4 processors. Each case has the four groups of compiler switches defined in Table 1.

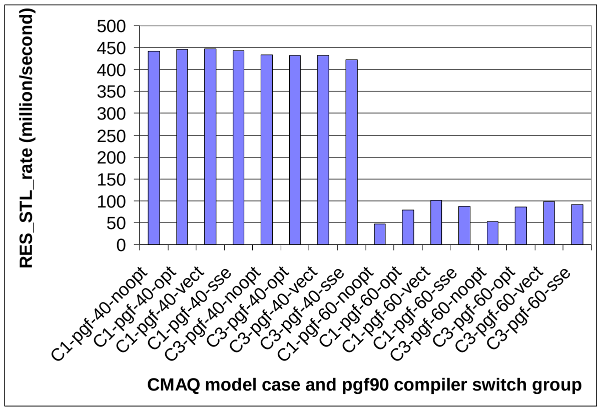

Fig. 4. Cycles stalled on any resource in million per second for CMAQ model tutorial cases 1 (C1) and 3 (C3) on P3 and P4 processors. Each case has the four groups of compiler switches defined in Table 1.

6.2 Memory footprint

The CMAQ memory footprint has some revealing characteristics. First of all, as shown in Fig. 5, the rate of total memory instructions issued (loads plus stores) is voluminous. Only results for the P4 processor are shown in Figs. 5 to 7 because not all of these events are available on the P3. High rates of memory instruction issue, in itself, 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.

However, an interesting differentiator is the ratio metric MEM_INS_FPINS which is the number of memory instructions issued per floating point instruction. Figure 6 shows that with noopt compiler switches this number is greater than 7. Even on allowing optimization this metric is still of the order of 4. Note that Case 3 (8km grid) has a slightly higher value than than Case 1 (32km grid). Notably, the compiler offers no reduction with the higher optimization levels (vect, sse).

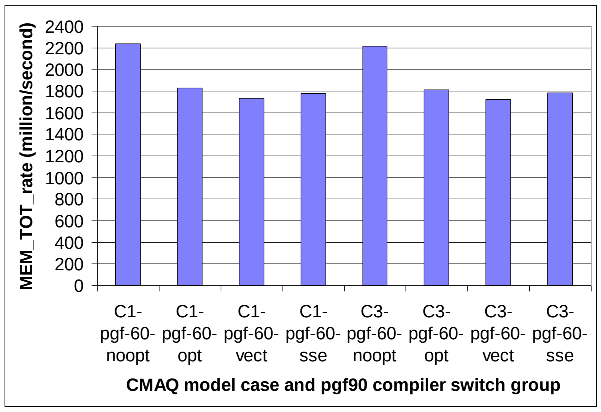

Fig. 5. Total memory instructions in million per second for CMAQ model tutorial cases 1 (C1) and 3 (C3) on the P4 processor. Each case has the four groups of compiler switches defined in Table 1.

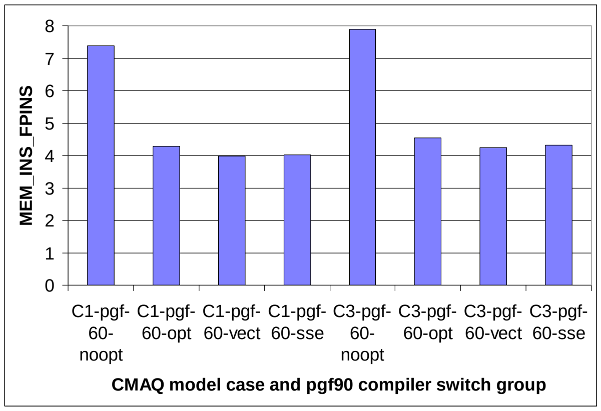

Fig. 6. Number of memory instructions per floating point instruction for CMAQ model tutorial cases 1 (C1) and 3 (C3) on the P4 processor. Each case has the four groups of compiler switches defined in Table 1.

The results of the MEM_INS_FPINS metric suggests that CMAQ is a memory-intensive algorithm. This is not necessarily an issue on proprietary architectures where high memory bandwidth is available and multiple loads or stores per cycle are possible. However, a memory intensive application, without a dominant vector character, is performance constricted on commodity architectures where memory bandwidth is limited by the FSB and cache design.

The consequence of CMAQ's memory footprint is that cache can become a limiting critical resource. 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 are another form of degraded performance. PAPI offers counters for TLB miss events for both instruction and data. In the case of CMAQ it is the data TLB misses that are critical. For example, comparison of the sse results for C1 and C3 in Fig. 7 shows a 20% increase in TLB misses and this correlates with a Mflops reduction of 6.6% (see Fig. 2).

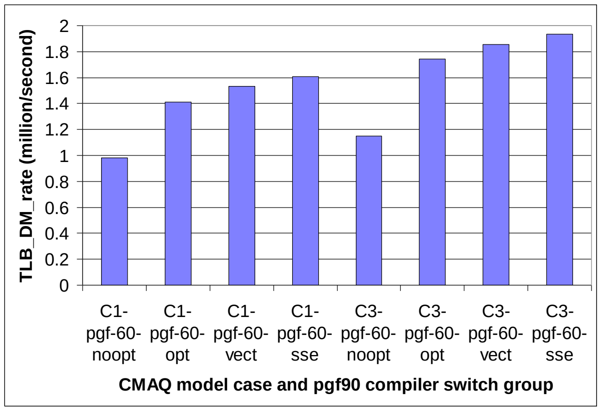

Fig. 7. TLB data cache misses in million per second for CMAQ model tutorial cases 1 (C1) and 3 (C3) on the P4 processor. Each case has the four groups of compiler switches defined in Table 1.

6.3 Cache usage

A code that is memory intensive will be sensitive to cache misses on commodity architectures. Both the P3 and P4 platforms discussed here have L1 and L2 caches, but these are larger on the P4 platform. Nevertheless, a cache miss 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 CMAQ the rate of L1 cache misses are shown in Fig. 8 for P3 and P4 platforms. Despite the larger cache on the P4 platform the incidence is much larger, due in part, to the higher rate of instruction issue. Even though the L1 cache on the P4 is larger than that of the P3, L1 data cache misses are a source of degraded performance for CMAQ.

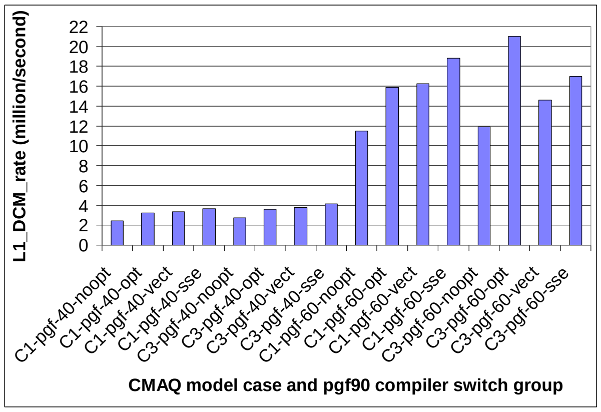

Fig. 8. L1 Data cache miss rate in million per second for CMAQ model tutorial cases 1 (C1) and 3 (C3) on P3 and P4 processors. Each case has the four groups of compiler switches defined in Table 1.

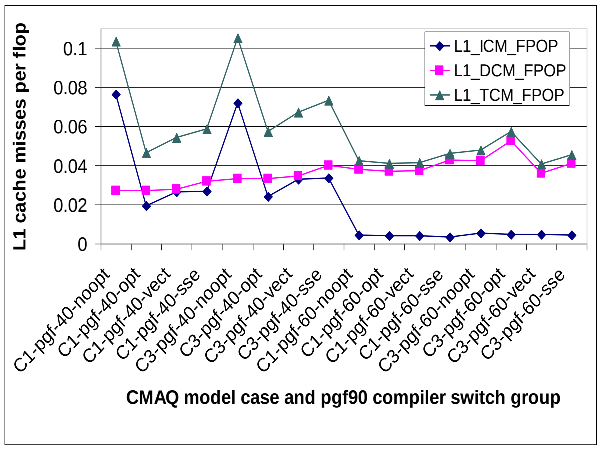

Fig. 9. L1 instruction, data, and total cache misses per floating point operation for CMAQ model tutorial cases 1 (C1) and 3 (C3) on P3 and P4 processors. Each case has the four groups of compiler switches defined in Table 1.

Since CMAQ has four memory operations per flop it is more instructive to inspect the number of L1 cache misses per flop. This is shown in Fig. 9 for both L1 instruction and data cache misses for P3 and P4 platforms. Interestingly, on the P3, for the opt, vect and sse switch groups, the result of this ratio metric is similar for both instruction and data cache misses. However, for the P4 the number of instruction cache misses per flop is much smaller than the value for L1 data cache misses. The total number of L1 cache misses per flop are generally larger for the P3 when compared to the P4. It is noteworthy that the value of this metric for L1 data cache misses is of the order of 0.04 on the P4. One L1 data cache miss per 20 flops seems to be a small value. But for a memory intensive code with negligible vector instructions, the latency cost of memory access is the limiting factor on performance improvement above the opt level of compiler optimizations.

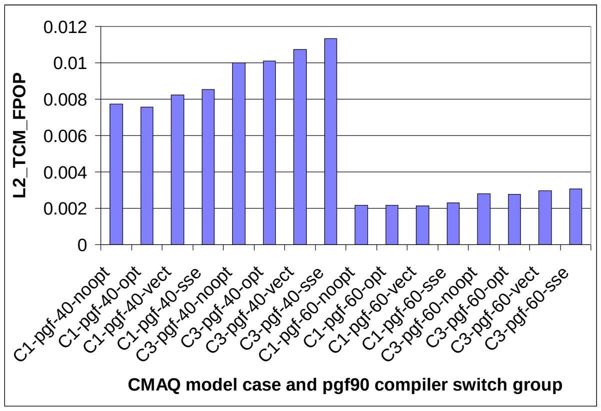

Fig. 10. L2 total cache misses per floating point operation for CMAQ model tutorial cases 1 (C1) and 3 (C3) on P3 and P4 processors. Each case has the four groups of compiler switches defined in Table 1.

7. CMAQ EXECUTION PROFILE

An execution profile of CMAQ is easily performed with the –Mprof=lines compiler switch in the pgf90/95 compiler. Results for Case 3 (8km grid) with the sse optimization switches is shown in Table 4. This shows those functions accounting for 68% of the cumulative process time with some that have a high calling overhead. Once the important functions are identified code inspection shows some reasons why vector instructions are scarce in CMAQ. The top two routines account for 31% of the process time but inhibit loop vectorization because of either exits out of the loop range (hrsolver), or I/O operations inside otherwise vectorizable loops (cksummer). Subroutine kmtab (and others not shown) account for a negligible amount of the process time but have a very high calling overhead. These routines (and also vppm) should be inlined to reduce the cost of control transfer instructions. Inline methods are well documented in the PGI User's Guide. Likewise interprocedural optimizations should be applied with the –Mipa compiler switch. Both of these simple steps should give significant reductions of process time and will be tested at a later date.

However, the removal of vectorization inhibitors in the top two routines would require source code modifications.

TABLE 4. CMAQ P4 profile for Case 3 (sse)

Function	Number of calls	Time (%)
hrsolver	2387952	16
cksummer	2473	15
hppm	117832	14
vppm	27461448	9
ludcmp	2016360	6
hrcalcks	2387952	4
calcact	9362289	2
kmtab	59699539	2
		…….
		68

8. CONCLUSIONS

This performance analysis of CMAQ, for two of the 4.4 release tutorial problems, at two grid resolutions, shows that this is a memory intensive application with some 4 memory operations to each floating point operation. Also, a negligible number of vector instructions are measured at the highest optimization level. In combination these two characteristics of the CMAQ code place a limit on the optimal performance possible from CMAQ 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.

In its present form, CMAQ gains mostly from improvements in the scalar performance of the hardware. But, despite these observations, a profile of CMAQ performance, followed by code inspection, does suggest that there is scope for performance improvement beyond the 400 to 450 Mflop range it currently delivers on the P4 for the tutorial problems that come with the 4.4 release.

REFERENCES

CMAQ, 2005: 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.

PAPI, 2005: *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).