ANALYSIS OF CMAQ 4.6.1 PERFORMANCE AND EFFICIENCY

Jeffrey O. Young*
U.S. EPA, Office of Research and Development, National Exposure Research Laboratory, Atmospheric Model Development Branch, Research Triangle Park, North Carolina 27711, USA

George Delic**
HiPERiSM Consulting, LLC, P.O. Box 569, Chapel Hill, NC 27514, USA

1. INTRODUCTION

CMAQ performance and workload throughput have always been issues of concern for Air Quality modelers. These concerns have only grown with the migration to commodity processors in the past decade because of compromises in architecture design that have impaired performance and workload throughput. With the new generation of multi-core processors a decline in efficiency is to be expected without some remedial action to mollify the performance bottle-necks that CMAQ encounters at runtime. The present study was undertaken to identify where, and to what extent, performance is inhibited. Since CMAQ is a data-intensive application the search for memory path "hot-spots" could be expected to provide clues because of the expense of memory access on commodity platforms. The following sections present the details of this study which focused only on CMAQ with the EBI chemistry solver. Work is in progress on improving the Rosenbrock and Gear solvers (Delic, 2008). The ultimate goal of this work is to significantly reduce the wall-clock time of large-scale CMAQ model simulations in future releases.

2. CHOICE OF HARDWARE AND OPERATING SYSTEM

The hardware systems chosen were HiPERiSM Consulting, LLC's 8 CPU SGI Altix® with the Itanium2® (ia64) and Intel® Pentium 4 Xeon, 64EMT (x86_64) processors. Clock rate and cache capacity differed with ia64 having 1.5GHz, and data cache sizes of 256KB (L2), 4MB (L3), and x86_64 having 3.4GHz, and data cache size 1MB (L2). CMAQ was executed in serial mode on both platforms and with MPI mode on the ia64 cluster. Both platforms used the SUSE 10 Linux release with proprietary modifications and interfaces to the Performance Application Programming Interface performance event library (PAPI, 2005) to collect hardware performance counter values as the code executes (Delic, 2003-2006). Results for selected performance metrics are presented to demonstrate how CMAQ code is mapped to architectural resources by compilers.

Two compilers were used: Intel (ia64 and x86_64 platforms) and Portland Group (x86_64 platform). In both cases high level (vector and inter-procedural) optimizations were applied and are designated by mnemonics ipo (Intel) or ipa (Portland) for switch groups shown in Table 2.1.

Table 2.1. Compiler switches.

Platform	Compiler (version)	Switch group	Switches
x86_64	Portland(1) (7.0)	ipa	-fastsse –Mscalarsse -Mipa=fast -tp p7-64
x86_64	Intel(2) (6.0)	ipo	-tpp7 -xW -O3 -Ob2 -ipo
ia64	Intel(2) (10.1)	ipo	-O3 -Ob2 -ipo

1) Portland additional compiler switches include: -Mfixed –Mextend –mcmodel=medium and link flag – mcmodel=medium.
2) Intel additional compiler switches include: -fixed –extend_source 132 –fno-alias and link flags –static (without ipo), or –ipo –static (with ipo).

3. EPISODES STUDIED

The model episodes selected for this analysis were for January 10 and August 14, 2006 (hereafter Winter and Summer, respectively). Both used the CB05 mechanism with Chlorine extensions and the Aero 4 version for PM modeling. The EBI solver was used for the gas chemistry. Both episodes were run for a full 24 hour scenario on a 279 X 240 Eastern US domain at 12 Km grid spacing and 34 vertical layers. It should be noted that these episodes required a minimum memory capacity of 8GB and therefore could only be executed on 64-bit operating systems that are capable of addressing more than 2GB (this excluded 32-bit operating systems from consideration).

**Corresponding author:* Jeffrey O. Young, U.S. EPA; e-mail: Young.Jeffrey@epa.gov.
**Work performed under contract to ORD, U.S. EPA.

4. RUNTIME PROFILE

Each 24-hour episode produced in excess of 22GB of output to disk. Typical wall-clock times depended on the platform and compiler chosen. They are in the range 32 to 48 hours for serial execution and less for MPI enabled execution (depending on the number of processes chosen).

The details of total runtime for the two scenarios are shown in Table 4.1 for ia64 and x86_64 platforms. The number in parentheses represents the number of MPI processes for the ia64 case and these results are also shown in Fig. 4.1. Note that the runtime of the Winter episode takes 13% longer.

Table 4.1. Process time in seconds for the Intel compiler with the ipo compiler switch group.

Winter		Summer	
x86_64	ia64	x86_64	ia64
131,606	162,325(1)	116,999	143,578(1)
	90,068(2)		82,406(2)
	53,975(4)		46,100(4)
	28,162(8)		25,048(8)

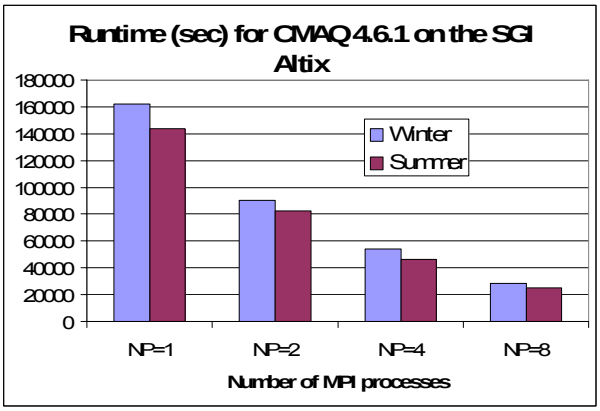

Fig 4.1: Runtime in seconds for CMAQ 4.6.1 for the Winter and Summer episodes with the Intel ipo compiler switch group on the Itanium2 platform in serial and parallel modes.

To determine where time is spent during CMAQ runtime, profiles runs where conducted with the two compilers using gprof (Intel) and pgprof (Portland Group), respectively. In the case of gprof, profiles were performed on both platforms. Since the results were consistent across platforms and compilers, typical results of pgprof are shown here for the Summer episode. Table 4.2 shows a (truncated) sort ranked on time per procedure with a decaying dispersion from 10% to 2% of the total runtime. Only 24% of the total runtime is spent in gas chemistry operations (procedures with hr prefix).

Table 4.2. Profile sorted on time per procedure for the 24 hour Summer episode on the Pentium 4 Xeon 64EMT with the Portland Group compiler using the ipa compiler switch group.

Function	seconds	%	Σ%
hrsolver	21,975	10	10
matrix	19,716	9	19
calcact	17,156	8	27
funcg5a	13,067	6	33
hppm	10,214	5	38
kmtab	9,060	4	42
hrcalcks	7,498	4	46
hrrates	7,301	3	49
hrprodloss	6,573	3	52
y_yamo	6,179	3	55
calcmr	5,929	3	58
cksummer	4,927	2	60
km298	4,881	2	62
hrg2	4,784	2	64
vdiff	4,466	2	66
isoinit3	4,367	2	68
ibacpos	4,173	2	70
hrg1	4,018	2	72
calcph	3,584	2	74

Table 4.3. Runtime profile sorted on number of calls per procedure for the 24 hour Summer episode on the Pentium 4 Xeon 64EMT with the Portland compiler using the ipa compiler switch group.

Function	calls	%	Σ%
kmtab	19,614,348,165	4	4
lbacpos	19,614,348,165	2	6
calcmr	16,000,736,938	3	9
calcph	14,547,357,820	2	11
km298	10,663,991,856	2	13
calcact	9,149,927,307	8	21
funcg5a	7,842,130,668	6	27
hrrates	6,882,708,845	3	30
hrprodloss	6,882,708,845	3	33
hrg2	6,882,708,845	2	35
hrg1	6,882,708,845	2	37
hrg4	6,882,708,845	1	38
hrg3	6,882,708,845	1	39
km273	4,330,128,825	1	40
km248	2,362,506,156	1	41

Table 4.3 shows the profile ranked by number of procedure calls. While this display is also truncated it does show that 41% of the total runtime is spent in 15 procedures where the total number of calls per procedure ranges from 2.4×10^3 to 19.6×10^3 million calls. What is important is that the top six and last two procedures listed in Table 4.3 are from module isocom.f and not the gas chemistry.

In the episodes considered here the top 7 to 9 routines account for half of the total execution time.

5. HARWARE PERFORMANCE METRICS

5.1 Operations

In this section selected results of hardware performance metrics are summarized for execution of the serial version of CMAQ on the ia64 platform with the Intel compiler. Some metrics are rates (in million events per second) whereas others are ratios of two events, and for details see presentations by Delic (Delic 2003-2006, 2005, 2006). Table 5.1 shows the Mflops (FP_OPS_rate) achieved for both episodes. Note the lower Mflops rate for the Winter episode. Also shown is FP_STAL_rate which is a measure of the number of cycles per unit time that the floating point (FP) units are stalled. The ratio FP_STAL_rate to the TOT_CYC_rate shows that for some 20%-22% of cycles the FP units are stalled (i.e. not performing arithmetic work). The TOT_CYC_rate tracks the clock speed of the two processors: 1.5.Mhz (ia64) and 3.4Mhz (64EMT).

Table 5.1. Rate metrics for floating point units

Metric (million/sec)	Summer	Winter
FP_OPS_rate	597	507
FP_STAL_rate	292	315
TOT_CYC_rate	1415	1409

5.2 Memory access and TLB cache

Memory access is shown in Table 5.2 where the total memory instruction rate (MEM_TOT_rate) is large. However, this alone is not an indicator of poor performance unless other issues coincide. One such is the rate of TLB cache misses and CMAQ shows an extraordinarily large number of data TLB cache misses (instruction TLB cache misses are negligible). This indicator suggests that memory latency is expanded as the CPU waits for a data address translation and fetch from higher up the memory hierarchy. Note that for the Winter episode the TLB data miss rate is significantly higher even though the memory rate is lower.

Table 5.2. Operations for memory access

Metric (million/sec)	Summer	Winter
MEM_TOT_rate	495	417
TLB_DM_rate	12.7	13.7

5.3 L1 cache

The ia64 memory hierarchy includes L1, L2, and L3 cache and Table 5.3 shows the total L1 cache access rates (L1_TCA_rate) and also the total L1 cache miss rates which are seen to be negligible even though the access rates are large. Virtually all L1 cache accesses are reads (L1_TCR_rate).

Table 5.3. Rate metrics for L1 cache access

Metric (million/sec)	Summer	Winter
L1_TCA_rate	673	565
L1_TCM_rate	11.5	13.6
L1_TCR_rate	673	565

Cache is divided into instruction and data parts and Table 5.4 shows the ratio of instruction access to data cache accesses (ICA/DCA). For L1 cache it is instruction cache access that dominates. This is due to the high calling overhead noted in Section 4: a control transfer such as a procedure call will induce new instruction fetches. An issue of concern is that there is typically one L1 cache access per FP operation. This value could be traced to a memory bottle-neck or lack of vector code. Successful vector code constructs tend to give fewer cache accesses per flop because they enhance data locality.

Table 5.4. Ratio metrics for L1 cache access

Metric (ratios)	Summer	Winter
ICA / DCA	14.7	10.6
TCA / FPOP	1.2	1.1

5.4 L2 cache

As is seen in Table 5.5, L2 cache access rates are also large, and are dominated by reads, with negligible cache misses.

Table 5.5. Rate metrics for L2 cache access

Metric (million/sec)	Summer	Winter
L2_TCA_rate	648	484
L2_TCM_rate	9.3	13.4
L2_TCR_rate	532	370

Table 5.6 shows that, conversely to L1 cache, here it is the data cache accesses that dominate over instruction accesses by more than an order of magnitude. This behavior could be traced to memory references caused by use of numerous scalar variables in several of the frequently called procedures in CMAQ code. Again, as for L1 cache, there is one L2 cache access per FP operation (flop), thereby enhancing sensitivity to memory latency effects.

Table 5.6. Ratio metrics for L2 cache access

Metric (ratios)	Summer	Winter
DCA / ICA	63.4	39.5
TCA / FPOP	1.1	1.0

5.5 Memory Bottlenecks and MPI performance

This section extends the analysis to MPI parallel execution on the ia64 platform. While it may appear difficult to generalize, the above analysis, when extended to the parallel CMAQ execution case, suggests that memory bottlenecks occur with increasing severity as the MPI parallel process count increases. One example of this is a measured increase from 1.1 to 1.7 in mean L1 cache reads per flop for both episodes. When this behavior is correlated with the dominance of L1 instruction cache access rates, and that over 32% to 56% (depending on the episode, or number of MPI processes) of L2 instruction cache accesses result in a miss, then there may be an apparent explanation for where the memory bottle-neck occurs in the hardware. It is important to note that the memory access rates (particularly the mean load rates) change little as the MPI parallel process count increases whereas there is a corresponding sharp decline (and large spread) in the Mflop rate (for both episodes). As the MPI parallel process count increases, the increase in L1 cache reads per flop correlates with memory loads remaining at a steady rate (with small spread) while floating point rates decline. Fig. 5.1 show the correlation between Mflops and L1 cache reads per flop. There is an obvious steady decline in the Mflop rate as the number of MPI processes increases. A similar result holds for L1 cache reads per memory instruction. This demonstrates that the load balance between arithmetic and memory operations tilts sharply to the latter as the number of MPI processes increases.

6. EFFICIENCY

For MPI execution of both episodes Fig. 6.1 shows the parallel speedup observed and it is notable that this diverges substantially from the linear result. This trend may be traced to a decrease in efficiency of CMAQ due to the increasing effect of memory latency.

Results for two efficiency metrics are presented in Table 6.1 and two conclusions are that (a) parallel execution shows a 11% to 28% decrease in efficiency relative to the serial execution for flops per cycle, and (b) Parallel execution shows a 17% to 19% decrease in efficiency for flops per cycle as the number of MPI processes increase from 2 to 8. This second observation is one explanation for the scaling result of Fig. 6.1. A second explanation is the memory bottle-neck identified in Section 5.5. It is interesting to observe that the Winter episode has the lower efficiency values, and this accounts for why it takes longer to complete a 24 hour scenario.

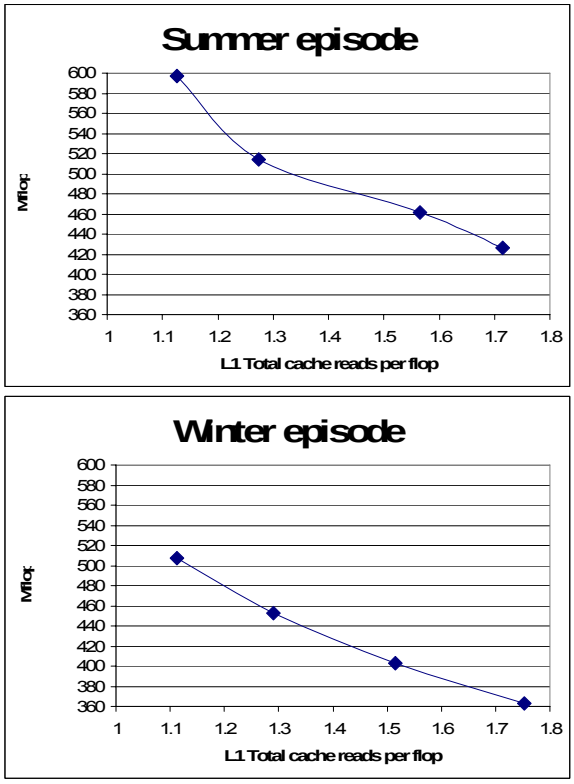

Fig 5.1. Parallel results for CMAQ 4.6.1 for the Summer (upper) and Winter (lower) episodes with the Intel ipo compiler switch group on the Itanium2 platform. These show the correlation of Mflops and L1 total cache reads per flop as the number of MPI parallel processes increments from 1, 2, 4 and 8 from left to right. The serial result corresponds to the left most point in each curve.

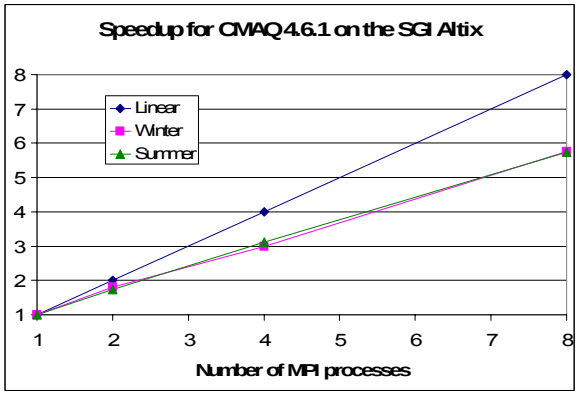

Fig 6.1. Parallel speedup for CMAQ 4.6.1 for the Summer and Winter episodes with the Intel ipo compiler switch group on the Itanium2 platform.

Table 6.1: Efficiency metrics for CMAQ 4.6.1 for the Summer and Winter episodes with the ipo compiler switch groups using the Intel compiler on the x86_64 and Itanium2 platforms.

Efficiency metric (episode)	x86_64	ia64	
	Serial	Serial	MPI parallel N = 2 N = 4 N = 8
Flops per cycle (Winter)	NA	0.360	0.32 0.28 0.26
Memory instructions per cycle (Winter)	0.172	0.298	0.29 0.28 0.28
Flops per cycle (Summer)	NA	0.422	0.36 0.32 0.30
Memory instructions per cycle (Summer)	0.198	0.350	0.34 0.33 0.32 0.32

To compare efficiency of the two platforms used here, Table 6.1 also shows results for memory efficiency with the same compiler. The Mflops were not measured for the x86_64 platform. Note the much higher memory efficiency metric for the ia64 platform.

7. CMAQ CODE

7.1 Problem areas

The three fundamental problem areas are:

1. Insufficient use of vector instructions
2. Excessive control transfer instructions
3. Inefficient memory access

The workload spread over the top procedures listed in Table 4.2 ranges from 3% to 10% of the total runtime. Source code inspection of the most frequently called procedures identified in Table 4.3 reveals that three have 15 lines (or less) of executable code, and others are dominated by scalar arithmetic and logical operations. Thus, not surprisingly, these procedures invariably have negligible vector loop structure and consist predominantly of simple arithmetic statements and conditional code blocks. Such circumstances can be expected to lead to CPU pipeline stalls as cache lines are flushed and new instructions are fetched from memory. Also, the poor vector structure in the CMAQ EBI solver code severely limits the potential compiler optimization opportunities on commodity architectures. Table 7.1 correlates the problem area noted above for each procedure.

Table 7.1: Top procedures in CMAQ 4.6.1 that account for 40% of the total runtime in Table 4.3. The "LOC" column gives an estimate of the lines of code in each procedure.

name	Problem area			Estimated LOC
	1	2	3	
(ibacpos)		Y		6
(kmtab)		Y		29
(calcmr)	Y		Y	150
(km273)		Y	Y	14
(calcph)		Y		16
(km298)		Y	Y	14
(calcact)	Y		Y	127
(hrg3)	Y		Y	60
(hrg2)	Y		Y	319
(hrg4)	Y		Y	60
(hrrates)	Y		Y	224
(hrprodloss)	Y		Y	500
(hrg1)	Y		Y	314
(funcg5a)	Y		Y	50

7.2 Remedies

A detailed analysis of compiler optimization abilities was undertaken and the result was that both compilers gave only limited success in optimizing the CMAQ code in the EBI solver version. As a consequence the onus for improving runtime performance falls on the code designers. In this study no code intrusive optimizations were attempted. However, the ability of compilers to perform inlining procedures was investigated to see the effects of reducing control transfer instructions originating from calling overhead. Automatic inline options did not inline the most frequently called procedures identified in Tables 4.3 and 7.1. Nevertheless, both compilers used here allow (in principle) the forced inlining of named procedures. These features were explored but failed to function entirely successfully. However, the Portland Group compiler was able to force the inlining of kmtab, ibacpos, and funcg5a, and showed a noticeable reduction in runtime. Obvious procedure groups that are targets for optimizations through code transformations and inline actions include call trees leading to:

- Subroutine hrsolver and procedures called from there: hrdata, hrrates, hrprodloss, hrg1, hrg2, hrg3, and hrg4.
- Subroutine calcact which is called from numerous places in CMAQ
- Similarly for calcmr, calcph, kmtab, ibacpos, etc.

The issue of adding vector code to the CMAQ EBI solver will require major code restructuring. By contrast the Rosenbrock and Gear solvers have inherent vector structure and optimization opportunities that are promising (Delic, 2008).

8. CONCLUSIONS FOR CMAQ

8.1 Serial execution

- Delivers more efficient performance when compared to parallel mode
- Exhibits performance-inhibiting source code constructs
- Does not match commodity hardware effectively because of:
  - Insufficient use of vector instructions
  - Excessive control transfer instructions
  - Inefficient memory access

8.2 Parallel execution

- With increasing number of parallel processes:
  - The load balance of memory to floating point operations increases
  - Parallel speedup increasingly departs from linearity
  - The average CPU idle time increases for all CPU's

8.3 Efficiency and architectures

For CMAQ, the ia64 (Itanium2) architecture is more efficient than is the x86_64 (64EMT) platform despite the difference in clock rate.

8.4 Next steps

Opportunities abound for significant performance enhancement of CMAQ through optimizations and code restructuring. Specifically, next steps should:

- Inline the most frequently called procedures that do little work
- Streamline the gas chemistry solver group of procedures
- Seek opportunities higher up the call tree for code restructuring to enhance vector instructions and data locality

8.5 Can compilers help?

The compilers used here generate efficient code for floating point and memory operations whenever they find efficient vector code constructs, but they do have very limited procedure inlining optimization support.

8.6 Will future hardware help?

Future hardware offers more cores per CPU socket, and unless data parallel structures are uncovered by developers, CMAQ with the EBI solver will deliver even lower efficiency than reported here. The situation is aptly summarized: "The road to high performance [will be] via multiple processors per chip … this signals a historic switch from relying solely on instruction level parallelism (ILP) … to thread-level parallelism (TLP) and data-level parallelism (DLP). Whereas the compiler and hardware conspire to exploit ILP implicitly without the programmer's attention, TLP and DLP are explicitly parallel, requiring the programmer to write parallel code to gain performance" (Hennessy, 2007)

REFERENCES

Delic, 2003-2006: see presentations at the Annual CMAS meetings.

Delic, 2005: 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.

Delic, 2006: 7th International Conference on Linux Clusters: The HPC Revolution 2006, Norman, OK, May 2-4, 2006.http://www.linuxclustersinstitute.org/Linux-HPC-Revolution/Archive/2006techpapers.html, and in Commodity Cluster Symposium, Baltimore, MD, July 26-27, http://www.arl.hpc.mil/events/Clusters2006

Delic, 2008: Parallel algorithms for CMAQ's gas chemistry solver procedures, (work in progress).

Hennessy, 2007, J.L. Hennessy and D.A. Patterson, *Computer Architecture, 4th Ed.*, 2007.

PAPI, 2005: *Performance Application Programming Interface*, http://icl.cs.utk.edu/papi. The use of PAPI requires a Linux kernel patch (as described in the distribution). Note however, that this patch has been included in the SGI Altix Suse Linux kernel.