Performance Characteristics of the SPEC OMP2001 Benchmarks*

Vishal Aslot and Rudolf Eigenmann
Purdue University
Department of Electrical and Computer Engineering

Abstract

Parallel computing is becoming mainstream with the advent of general purpose cost effective Shared-memory Multiprocessor (SMP) systems. At the same time, new developments in the parallel programming environment allow rapid and efficient programming of these systems. To this end, OpenMP has emerged as a flexible and a fairly comprehensive set of compiler directives, library routines, and environment variables to facilitate parallel programming of the SMP systems in Fortran and C/C++. The Standard Performance Evaluation Corporation (SPEC) has created a benchmark suite of eleven applications, named SPEC OMP2001, to be used for the performance evaluation and comparison of moderate size SMP systems. Each of the benchmarks in SPEC OMP2001 is either automatically or manually parallelized using the OpenMP directives. In this paper, we present basic static and runtime characteristics of these benchmarks. We present data gathered using high resolution timers and the hardware counters available on our SMP system. We explain some of the performance characteristics of these benchmarks quantitatively and qualitatively within the framework of our measured data and a quantitative model.

1 Introduction

With the breakthroughs in the standard off-the-shelf microprocessor and memory technologies, and their use in building cost effective Shared-memory Multiprocessor (SMP) systems, SMP systems have gained prominence in the market place. As their popularity grows, a need for more sophisticated, yet flexible development and runtime environments are called for to facilitate rapid and efficient development of parallel applications. Over the years, a variety of parallel programming paradigms such as custom compiler directives to mark parallel regions, MPI, POSIX thread programming, and data-parallel paradigm, just to name a few, have emerged. While each one has its benefits, for small to medium range SMPs, either directive based programming or POSIX thread programming have gained prominence. Since most compilers implement parallelization directives as threads, these two ways of programming parallel machines are related.

While a large number of vendor specific parallelization directives have served the SMP user community, there was a dire need for standardization. The OpenMP API [8] (Application Programmer's Interface) has fulfilled the need by providing a flexible, scalable, and fairly comprehensive set of compiler directives, library routines, and environment variables to incrementally write parallel programs. OpenMP is still evolving to better accommodate needs of the parallel programmers.

As SMPs become more commonplace, it is important to be able to evaluate the performance of these systems using a standard set of benchmarks. Several parallel benchmark suites over the past 20 years have attempted to fill the void including SPLASH 2 [11], Parkbench [5], and the Perfect Benchmarks [3]. In contrast to these efforts, the Standard Performance Evaluation Corporation (SPEC) has released a new set of benchmark targeted towards modern SMP systems. The suite has been named SPEC OMP2001. It contains eleven programs written in Fortran and C, which have been made parallel using the OpenMP API. The goal of SPEC OMP2001 is to provide a benchmark suite that

- is portable across mid-range parallel computer platforms,
- can be run with relative ease and moderate resources,
- represents modern parallel computer applications, and
- addresses scientific, industrial, and customer benchmarking needs.

*This material is based upon work supported in part by the National Science Foundation under Grant No. 9703180, 9975275, and 9974976. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

SPEC OMP2001 has recently been released. Our work is the first attempt at studying the performance characteristics of these applications in reasonable detail. In this paper, we present the basic static and runtime characteristics of these benchmarks. We present data gathered using high resolution timers and the hardware counters available on our SMP system. We explain some of the performance characteristics of these benchmarks quantitatively and qualitatively within the framework of our measured data and a basic quantitative model.

The remainder of the paper is organized as follows. Section 2 gives an overview of the benchmark applications. Section 3 briefly presents the runtime environment in which we carried out our experiments. Section 4 presents basic timings and speedups of the benchmarks and the most time-consuming parallel regions within each benchmark. Also in this section, we identify the parallel regions that perform poorly. Section 5 discusses more detailed measurements and some of the derived runtime characteristics of the benchmarks. The data presented in this section creates a framework within which we discuss the performance of the benchmarks. Section 6 presents a basic model that attempts to explain observed performance quantitatively and qualitatively. Finally, section 7 concludes the paper.

2 Overview of the SPEC OMP2001 Benchmarks

SPEC OMP2001 is a collection of 11 applications. All applications except *gafort* are floating-point applications taken directly from the SPEC CPU2000 benchmark suite. Each application is either automatically or manually parallelized by inserting OpenMP directives to mark parallel regions of the code. The reference data set for each application has been scaled using time and memory-constrained scaling. Each benchmark consumes up to 2 GB of memory at runtime and runs for up to 10 hours on a modern single processor system. An overview of these applications is presented in Table 1. The reference data set is targeted towards moderate size SMP systems.

A more comprehensive discussion of the applications and their development effort is presented in [2].

Table 1: Overview of the SPEC OMP2001 Benchmarks

Code	Applications	Lang.	# of lines
ammp	Chemistry/biology	C	13500
applu	Fluid dynamics/physics	Fortran	4000
apsi	Air pollution	Fortran	7500
art	Image Recognition neural networks	C	1300
fma3d	Crash simulation	Fortran	60000
gafort	Genetic algorithm	Fortran	1500
galgel	Fluid dynamics	Fortran	15300
equake	Earthquake modeling	C	1500
mgrid	Multigrid solver	Fortran	500
swim	Shallow water modeling	Fortran	400
wupwise	Quantum chromodynamics	Fortran	2200

3 Methodology

3.1 Experimental Setup

We ran the benchmarks on a quad processor Sun Enterprise 450 SMP system. The basic configuration is shown in Table 2.

Table 2: Hardware and Software Setup

CPU	480 MHz UltraSPARC II
No. of CPUs	4
Memory	4 GB
Instruction Cache	16 KB, 32 byte line
Data Cache	16 KB, 32 byte line
External Cache	Unified, 8 MB, 64 byte line
Interconnect	1.72 GB/Sec Peak Throughput
Address Bus	1 Bus for a pair of CPUs
Data Bus	1 Bus for a pair of CPUs
Operating System	Solaris 5.8
Page Size	8KB
Compiler	Sun Forte 6.1 Kuck & Associate's GuideC

All measurements were taken in single user mode. We executed each benchmark with the reference data set from the SPEC OMP2001 Toolkit environment. All of the executions validated within the tolerances defined by the SPEC OMP2001 Toolkit.

3.2 Instrumentation

We have developed custom multipurpose instrumentation libraries that allow us to measure execution time, fork-join overhead time, and the hardware counters with relatively low overhead. Our libraries utilize a high resolution timer available on Solaris OS. We can measure the execution time and the fork-join overhead time in nanoseconds. By measuring the most time-consuming parallel and serial sections of the program, we account for over 99% of the execution time for each program. For instrumentation overhead and instrumented coverage, refer to Table 3. In most cases, the overhead introduced by the instrumentation is less than 1%. *Equake* and *fma3d* show overhead of over 2%, because each invokes the important parallel regions many times, which results in a large number of calls to instrumentation library routines. However, for the analysis purpose, the overheads are within a tolerable range.

Table 3: Instrumented Coverage and Instrumentation Overhead of SPEC OMP2001 Benchmarks

Code	Instrumented Coverage (%)	Instrumentation Overhead (%)
ammp	99.1	< 0.1
applu	99.9	< 0.1
apsi	99.8	0.7
art	99.9	< 0.1
equake	99.9	2.7
fma3d	99.4	2.2
gafort	99.9	< 0.1
galgel	95.5	0.6
mgrid	99.9	0.7
swim	99.4	0.3
wupwise	99.8	< 0.1

As mentioned earlier, we measure the execution times using high resolution timers. In addition, in order to account for lost cycles and explain the speedup in more detail, we used the performance counters available on the UltraSPARC II processors [10]. Using hardware counters allows us to collect vital runtime statistics on a real system. Since each benchmark runs for several hours on a real system with the reference data set, it would take a prohibitively long time to run them through software simulators such as RSIM [9] and Wisconsin Wind Tunnel II [7].

The hardware counters on UltraSPARC II can measure up to 20 different events related to pipeline stall cycles, stall cycles due to memory system latencies, memory system performance metrics, and the coherency protocol performance metrics. We can measure up to two events per run of a benchmark, because there are two physical counters per processor. In order to measure several events per program run, we multiplexed 2 counters over all measurable events. Since each of the benchmarks runs for several hours, and we sample once every 500ms, the measured numbers are statistically stable and represent the true characteristics of the program very closely. In this paper, we present the results of the overall program executions. A more detailed program section by program section analysis is presented in [1].

In order to measure the execution time, we instrumented each OMP PARALLEL and OMP END PARALLEL section but not the worksharing constructs inside each section. A number of OMP PARALLEL DO constructs have been converted to OMP PARALLEL/OMP DO pair to allow instrumentation inside the parallel regions. Also, we measured the fork-join overhead time by instrumenting around OMP PARALLEL/OMP END PARALLEL sections. The fork-time is defined as the time spent in OMP PARALLEL, and the join-time is the time spent in OMP END PARALLEL. Finally, in order to identify load imbalance among processors, we instrumented inside OMP PARALLEL and converted OMP END DO constructs to OMP END DO NOWAIT where ever possible, which removes the implicit barrier in the former.

Table 4: Basic Runtime Characteristics of SPEC OMP2001 Applications

Code	Parallel Coverage (%)	Total Runtime (sec)		# of Parallel Sections
		Seq.	4 CPU	
ammp	99.11	16841	5898	7
applu	99.99	11712	3677	22
apsi	99.84	8969	3311	24
art	99.82	28008	7698	3
equake	99.15	6953	2806	11
fma3d	99.45	14852	6050	92/30[1]
gafort	99.94	19651	7613	6
galgel	95.57	4720	3992	31/32
mgrid	99.98	22725	8050	12
swim	99.44	12920	7613	8
wupwise	99.83	19250	5788	10

[1] static sections / sections called at runtime

4 Basic SPEC OMP2001 Performance Characteristics

We ran each benchmark on 1, 2, and 4 processors and measured the execution time of the overall application as well as instrumented program sections. The execution time allowed us to compute parallel coverage, overall speedup of the application, and program section by program section speedup. Since all applications are mostly loop based, we can use the term "loop" interchangeably with "parallel region". Parallel coverage is defined as the percentage of serial program execution time enclosed by a parallel construct. Speedup, in this paper, is defined as the ratio of *1-processor* execution time to *n-processor* execution time. Parallel coverage allows us to calculate maximum theoretical speedup, or Amdahl's speedup, for a given parallel program or even a single parallel region for a fixed size data set. For a program with parallel coverage of p and n processors, Amdahl's speedup is given as,

$$speedup = \frac{1}{(p/n) + (1-p)}, 0 \leq p \leq 1, n \geq 2$$

The overall speedup of the benchmarks is shown in Figure 1, whereas the parallel coverage, serial and 4 processor execution times, and number of parallel regions are shown in Table 4. It is clear from the table that all benchmarks except *galgel* have parallel cover-

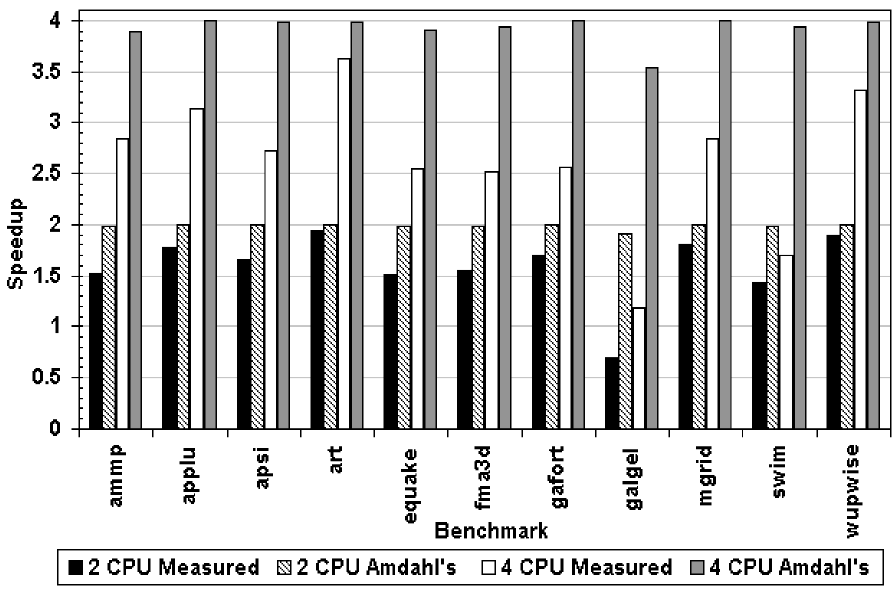

Figure 1: Overall Measured and Amdahl's Speedup on 2 and 4 Processors

age over 99%. Also, *galgel* has the shortest execution time.

We have instrumented a total of 172 program sections in all benchmarks. The execution profile for each section indicates the time spent in a section as a percentage of the overall execution time. Table 5 shows basic characteristics of all the loops that contribute more than 2% of the total execution time. Speedup of a single processor execution indicates *parallelization cost.* For most loops, the single processor speedup is close to 1.0. However, one or more loops in all benchmarks show speedups far below 1.0. In particular, *galgel*, *ammp*, *fma3d*, and *equake* demonstrate significant parallelization cost, with *galgel* suffering the most. The 2 and 4 processor speedups indicate how well each individual loop scales. Speedup of a high execution profile loop impacts the overall speedup more than a loop with low execution profile. In *swim*, *galgel*, *equake*, *fma3d*, and *ammp*, the major loops show speedups below 3.0 on 4 processors.

Finally, the average execution time is a measure of how much speedup one can expect from the parallel execution of a loop. Since the fork-join time for a loop is typically in microseconds under stable runtime conditions, a loop that runs for more than a few milliseconds should speedup almost linearly if other effects (load imbalance, coherence traffic, cache performance, bus contention) are ignored. Several important loops in *wupwise* and *applu* show significant speedup on 4 processors. However, for some of the benchmarks, despite of long average loop execution times, the speedups are poor. This holds for *galgel*, *swim*, and *fma3d*.

5 Detailed Performance Characteristics

In this section, we present more detailed performance characteristics derived from the hardware counter measurements. Table 6 shows the percentage of overhead cycles due to stalls in the processor pipeline and the lost speedup because of it ("Loss", see section 6.1). "Seq." numbers show contribution of the stalls in a serial execution, and "4 CPU" numbers show per processor contribution on 4 processors (rather than the cumulative contribution of all the processors). These stalls are a result of mispredicted branches and floating-point (FP) dependence between instructions. Table 7 delineates the lost cycles as a percentage of the overall execution cycles due to memory system stalls. In particular, the table outlines stalls due to instruction cache (IC) miss ("IC Miss"), dependence between a load and an earlier incomplete store ("Load RAW"), wait time on an earlier load whose result is not yet available to the instruction in the execute stage ("Load Use"), and stalls due to a full store buffer ("Store Buff."). "Loss" column, as in the case of pipeline stalls, shows the lost speedup due to the memory system stalls.

In addition to the stall cycles, we have derived a number of important performance metrics from the raw hardware counter measurements. Specifically, Table 8 shows instructions per cycle (IPC), com-

Table 5: Basic Runtime Characteristics of Major Parallel Sections

Code	Parallel Section Name[1]	Number of Invocations	Execution Profile (%)	Average Time (sec) Seq.	Average Time (sec) 1 CPU	Speedup 1 CPU	Speedup 2 CPU	Speedup 4 CPU
ammp	mmfvupdate-#5	202	96.37	80.15	110.94	0.72	1.54	2.92
applu	ssor-do#3	50	76.62	177.24	176.87	1.00	1.82	3.50
	rhs-do#3	51	7.17	16.26	18.71	0.87	1.67	2.47
	rhs-do#4	51	6.19	14.04	13.75	1.02	1.89	3.02
	rhs-do#2	51	3.36	7.62	10.60	0.72	1.31	1.91
	rhs-do#1	51	2.01	4.56	5.03	0.91	1.42	1.71
apsi	run-do#40	50	8.37	15.12	15.42	0.98	1.76	2.99
	run-do#60	50	8.31	15.02	15.29	0.98	1.76	3.02
	run-do#30	50	8.30	15.00	15.27	0.98	1.76	3.02
	run-do#20	50	8.30	14.99	15.27	0.98	1.76	2.94
	dvdtz-do#40	50	7.66	13.84	14.72	0.94	1.87	3.46
	dkzmh-do#30	51	7.15	12.67	11.00	1.15	2.24	4.30
	dudtz-do#40	50	7.06	12.75	13.19	0.97	1.16	2.24
	dcdtz-do#40	50	6.48	11.70	11.02	1.06	2.14	2.74
	dtdtz-do#40	50	6.45	11.65	12.75	0.91	1.86	2.36
	wcont-do#30	50	6.05	10.94	12.58	0.87	1.00	1.87
	run-do#100	50	6.03	10.89	11.38	0.96	1.69	2.80
	run-do#70	50	3.80	6.86	7.02	0.98	1.71	2.84
	run-do#50	50	2.93	5.29	5.26	1.00	1.75	2.72
	dkzmh-do#40	51	2.86	5.07	5.83	0.87	1.73	3.28
	smooth-do#10	152	2.14	1.27	1.27	1.00	1.88	3.13
	leapfr-do#30	185	2.04	1.00	1.00	1.00	1.59	1.91
art	scanreco-#0	1	99.83	27836.26	27897.07	1.00	1.94	3.64
equake	smvp-#0	3334	65.96	1.41	1.70	0.83	1.47	2.46
	main-#3	3334	31.37	0.67	0.79	0.85	1.63	2.99
fma3d	platq-do#2	522	75.46	21.95	26.54	0.83	1.57	2.83
	solve-do#6	2092	11.88	0.86	0.87	0.99	1.73	2.27
	solve-do#4	522	5.16	1.50	1.62	0.93	1.27	1.45
	solve-do#2	523	3.26	0.95	0.92	1.04	1.32	1.86
gafort	shuffle-do#10	1000	35.01	6.84	7.09	0.96	1.65	2.11
	gafort-do#45	1000	26.17	5.11	5.14	0.99	1.88	3.13
	mutate-jump	250	18.57	14.51	19.82	0.73	1.42	2.41
	evalout-do#30	250	12.76	9.97	9.93	1.00	1.96	3.64
	newgen-do#94	250	7.35	5.74	6.03	0.95	1.81	2.71
galgel	syshtN-do#1234	117	25.49	10.35	32.04	0.32	0.64	1.22
	sysnsn-do#123	117	22.97	9.33	55.73	0.17	0.33	0.63
	lapak-do#7	89815	12.26	0.01	0.01	1.00	2.53	3.57
	lapak-do#1	1056	8.93	0.40	0.408	0.99	2.00	3.60
	lapak-do#5	2945833	7.76	1.0E-4	0.01	0.96	1.56	2.19
	lapak-do#3	39996	6.39	0.01	0.01	1.00	1.40	1.34
	lapak-do#4	10962	3.36	0.01	0.02	1.00	4.19	9.66
	lapak-do#10	144	2.92	0.96	0.963	1.00	1.96	3.77
mgrid	resid-do#600	18250	50.47	0.63	0.64	0.99	1.88	2.86
	psinv-do#600	18000	23.33	0.30	0.31	0.95	1.85	3.28
	rprj3-do#100	15750	10.08	0.15	0.14	1.02	1.91	3.20
	interp-do#400	15750	5.27	0.08	0.09	0.83	1.62	2.47
	interp-do#800	15750	5.14	0.07	0.12	0.61	1.19	1.93
swim	calc3-do#300	1198	34.71	3.76	4.03	0.93	1.52	1.70
	calc2-do#200	1200	30.72	3.32	3.31	1.00	1.50	1.77
	calc1-do#100	1200	28.27	3.05	3.22	0.95	1.49	1.77
	swim-do#400	1200	5.62	0.61	1.28	0.47	0.81	1.29
wupwise	muldoe-do#1	402	43.92	20.97	20.61	1.02	2.03	3.86
	muldeo-do#1	402	41.54	19.84	20.31	0.98	1.95	3.70
	zaxpy-do#1	1604	5.55	0.66	0.67	0.99	1.50	1.73
	zdotc-do#1	801	4.25	1.02	1.47	0.69	1.34	2.15
	zcopy-do#1	806	2.73	0.65	0.66	0.99	1.52	1.73

[1] Naming Scheme: Either Subroutine/FileName-do#*LoopLabel* or Subroutine/FileName-do#*LoopNumber* from the top of the subroutine

Table 6: Basic Pipeline Overheads in SPEC OMP2001 Benchmarks

Code	Branch Mispred. %			FP Dependence %		
	Seq.	4 CPU	Loss	Seq.	4 CPU	Loss
ammp	0.69	0.90	0.01	10.39	7.35	-0.03
applu	0.04	0.03	0.00	2.00	1.56	0.00
apsi	0.82	0.65	0.00	6.53	4.85	0.00
art	3.30	4.25	0.05	12.07	10.34	0.00
equake	0.56	0.77	0.01	5.16	6.09	0.08
fma3d	0.21	0.14	0.00	17.88	10.49	-0.03
gafort	1.69	1.31	0.00	3.64	2.53	0.00
galgel	0.55	0.29	0.00	1.39	0.59	0.00
mgrid	0.02	0.02	0.00	0.04	0.03	0.00
swim	0.00	0.00	0.00	0.33	0.41	0.00
wupwise	2.62	2.02	0.00	5.02	3.67	0.00

Table 7: Basic Memory System Overheads in SPEC OMP2001 Benchmarks

Code	IC Miss %			Store Buff. %		
	Seq.	4 CPU	Loss	Seq.	4 CPU	Loss
ammp	0.03	0.05	0.00	0.49	0.87	0.02
applu	0.12	0.11	0.00	25.54	20.42	0.09
apsi	1.88	1.61	0.01	2.85	6.15	0.16
art	0.00	0.05	0.00	0.31	0.67	0.02
equake	0.25	0.14	0.00	1.99	2.25	0.03
fma3d	6.29	9.47	0.21	7.62	3.87	-0.04
gafort	0.02	0.02	0.00	1.23	1.03	0.01
galgel	0.16	0.08	0.00	13.10	5.89	0.08
mgrid	0.10	0.10	0.00	2.54	4.64	0.13
swim	0.09	0.09	0.00	52.20	54.74	1.12
wupwise	0.24	1.22	0.04	9.03	9.64	0.11
Code	**Load Use %**			**Load RAW %**		
	Seq.	4 CPU	Loss	Seq.	4 CPU	Loss
ammp	29.25	38.40	0.57	0.02	0.19	0.01
applu	49.81	49.62	0.56	1.14	0.92	0.00
apsi	22.48	30.15	0.53	0.27	0.31	0.00
art	38.73	36.65	0.11	4.63	3.57	-0.02
equake	70.68	66.98	0.51	1.16	1.01	0.01
fma3d	31.57	40.62	0.80	0.90	0.89	0.01
gafort	51.65	55.65	0.74	5.55	3.76	-0.01
galgel	21.37	20.45	0.51	0.24	0.17	0.00
mgrid	69.21	65.80	0.99	3.94	3.25	0.04
swim	28.86	28.57	0.56	3.75	4.82	0.11
wupwise	24.83	32.08	0.53	0.22	1.76	0.06

munication to computation ratio [4] ("Comm/Comp Ratio"), and memory access to computation ratio ("MemAccess/Comp Ratio"). The instruction cache (IC) hit rates, the first-level data cache (DC) hit rates, and the secondary or external cache (EC) hit rates are shown in Table 9.

The communication to computation ratio as presented here approximates the communication in an application – that is, the amount of data that the processors must share with each other in order to perform computations. The ratio in this paper is based on the number of copy-backs between caches, the number of write-backs from the external cache (EC) to the memory, and the line size of the external cache. Also, the ratio as presented here is total communication as opposed to per processor communication. This ratio may be lower than the actual communication, because we did not account for the data transfer from the memory to the external cache (EC). Such a transfer is triggered by either an EC miss or by read-for-ownership transaction where the cache line is not present in any other cache. Finally, since we could not measure the number of computation-related operations per second, the ratio is computed based on the number of instructions from the *sequential* execution of the program.

The memory access to computation ratio is a measure of relative proportion and distribution of memory access and computation-related instructions. We show the ratio per processor in a 4 processor execution. As in "Comm/Comp Ratio," we used the number of instructions from the sequential execution. Also, we assume that each data cache access is double precision (8 bytes long on our system), and each reference to the instruction cache brings 4 instructions, each one of 4 bytes in length. Hence, the ratio may be larger than the true ratio, because not every data access is double precision and not every instruction is 4 bytes long.

6 Performance Evaluation and Discussion

Based on the data presented in sections 4 and 5, we will explain partial speedup loss using a quantitative model similar to the Speedup Component Model [6]. The model will explain speedup loss for the overall program.

6.1 A Quantitative Model

Performance of a parallel application depends on more factors than its sequential counterpart. We can divide the total number of execution cycles of a parallel program into several components: cycles spent in performing useful work, stall cycles waiting for local and remote data accesses, stall cycles due to pipeline bubbles, extra cycles spent in parallelization overhead (fork-join overhead), cycles due to load-imbalance, cycles spent while executing additional code in the parallel program, which is not present in the sequential version, as well as cycles consumed by extra instructions due to conservative code generation by the compiler.

Within our experimental framework, we can measure the most important pipeline and memory system stall cycles, fork-join overhead cycles, and load-imbalance cycles. The memory stalls account for the local cache misses, some of which trigger coherence traffic. Since our system is an SMP, there are no true remote data accesses. Some programs have synchronization points such as OMP CRITICAL and calls to OMP_SET_LOCK/OMP_UNSET_LOCK, which are in addition to the implicit barriers in OMP DO and OMP END PARALLEL. We did not measure exact numbers of cycles spent waiting at these points. However, if there is contention to acquire a lock, or if the wait time is significant while acquiring and releasing a lock, it will show up as load imbalance and higher average execution time for the loop. We discovered that the Sun compiler makes shared variables "volatile" inside a parallel region, which results in more conservative register allocation. We have found the fork-time in all applications to be in the order of microseconds, as expected. In our model, we will account for the total fork-time.

Thus, our quantitative model for the SPEC OMP2001 benchmarks consists of the following components: $Speedup_{lost} = SpeedupLoss_{memory} + SpeedupLoss_{pipeline} + SpeedupLoss_{fork-time} + SpeedupLoss_{other}$. Figure 2 shows the speedup components for each of the benchmarks. The basic observation is that a given overhead *reduces* the speedup if the associated stall cycles *increase* in a parallel execution with respect to the stall cycles in a serial program execution. We compute each component as $\frac{p \cdot StallCycles_p - StallCycles_1}{TotalCycles_p}$. Here, p is the number of processors, which is also the maximum speedup for a program with 100% parallel coverage. $StallCycles_p$ and $StallCycles_1$ are the stall cycles due to either "memory," "pipeline," "fork," or "other" component, which is measured on one of the p processors for the parallel execution and on a single processor for the sequential execution, respectively. Finally, $TotalCycles_p$ is the total number of execution cycles for the entire program on p processors (dividing by $TotalCycles_p$ assures that we compute the *speedup component* rather than a raw difference in stall cycles). Inherent in our model are the assumptions that there is no load imbalance among processors, and that every stall cycle is counted only once under one of the stall categories. Both of these assumptions are true in our study. While Figure 2 outlines the *overall* speedup loss due to each of the three components, a more detailed breakdown is shown as "Loss" in Tables 6 and 7. Also, since all programs have a parallel coverage below 100%, the maximum speedup is not p but is Amdahl's speedup. Thus, the difference between the top edge of a bar in Figure 2 and the speedup of 4.0 represents speedup loss due to the serial regions of the program. For more details on the quantitative model, refer to [1].

6.2 General Comments on Performance

Figure 2 shows that all benchmarks are memory bound. The pipeline stalls are insignificant except in *equake* and *art*. Similarly, the fork overhead is also an insignificant component of the lost speedup except for *galgel*. "Other" component summarizes impact of the effects that we did not measure in our experiments. Such effects include pipeline and memory system stalls not measured by the hardware counters (e.g. stalls due to MEMBAR instruction), the join-times at the barrier in OMP END PARALLEL clause, and the extra time spent in computation due to less aggressive compiler optimizations. In Figure 2, speedup loss due to "Other" component is less than 0.5 except in the case of *galgel*.

Table 8 provides several useful insights into the performance of the benchmarks. We did not find any trend in the IPC data. It is, however, important to notice that IPC is fairly low across the board even on a modern superscalar processor. The communication to computation ratio as well as the memory access to computation ratio both show noticeable increase on 4 processors. The communication to computation ratio is under 1.0 for most benchmarks except *applu*, *fma3d*, and *swim* on 4 processors. The memory access to computation ratio is almost 8 bytes per instruction for all benchmarks in the sequential execution, which is equivalent to one double precision access per instruction. The ratio increases even more in the parallel execution. This is an indication that the benchmarks are memory bound as mentioned earlier. Increase in the memory access to computation ratio on 4 processors is a result of a higher number of memory references in the parallel execution. We attribute this effect to "volatile" references in the parallel regions to access the shared data as well as extraneous instructions such as spinning on a barrier. This

Table 8: IPC and Basic Ratios of SPEC OMP2001 Benchmarks on Sun E450

Code	IPC		Comm/Comp Ratio Bytes/Instruction			MemAccess/Comp Ratio Bytes/Instruction		
	Seq.	4 CPU	Seq.	4 CPU	% Change	Seq.	4 CPU	% Change
ammp	1.13	0.95	0.01	0.17	957.3	6.26	8.55	36.5
applu	0.34	0.54	0.60	1.92	218.6	7.67	18.93	146.8
apsi	1.06	0.95	0.05	0.31	530.7	8.15	10.25	25.8
art	0.80	0.69	0.00	0.09	>1000	8.29	8.73	5.3
equake	0.37	0.38	0.16	0.69	335.0	7.53	12.41	64.8
fma3d	0.48	0.59	0.22	1.20	416.8	6.79	15.03	121.2
gafort	0.68	0.69	0.21	0.91	316.4	8.64	11.16	29.2
galgel	1.30	0.87	0.06	0.43	572.0	7.51	21.98	192.4
mgrid	0.73	0.80	0.20	0.82	300.2	6.53	17.67	170.7
swim	0.39	0.35	0.78	3.15	302.9	6.66	14.25	113.7
wupwise	0.88	0.85	0.05	0.20	276.6	7.10	10.22	44.1

Table 9: Cache Hit Rates of SPEC OMP2001 Benchmarks on Sun E450

Code	IC Hit Rate %			DC Hit Rate %			EC Hit Rate %		
	Seq.	4 CPU	Change	Seq.	4 CPU	Change	Seq.	4 CPU	Change
ammp	100.02	100.97	0.95	81.16	86.68	6.79	98.66	96.64	-2.04
applu	101.13	97.70	-3.39	61.01	81.97	34.37	89.05	93.14	4.59
apsi	99.22	103.34	4.15	83.42	85.53	2.52	98.52	94.36	-4.22
art	104.00	107.49	3.36	66.74	72.98	9.35	97.40	89.40	-8.21
equake	99.52	102.36	2.85	75.03	83.99	11.94	86.82	85.70	-1.30
fma3d	92.15	91.04	-1.21	84.96	89.81	5.71	96.31	97.22	0.95
gafort	99.72	99.11	-0.62	89.87	92.30	2.70	95.27	95.65	0.40
galgel	100.19	101.83	1.63	67.82	88.02	29.78	98.60	99.16	0.57
mgrid	100.15	98.50	-1.65	69.13	84.24	21.86	93.87	94.48	0.65
swim	97.61	100.65	3.11	49.49	74.72	50.97	87.46	87.70	0.27
wupwise	99.74	99.37	-0.37	90.16	91.77	1.79	95.11	95.57	0.49

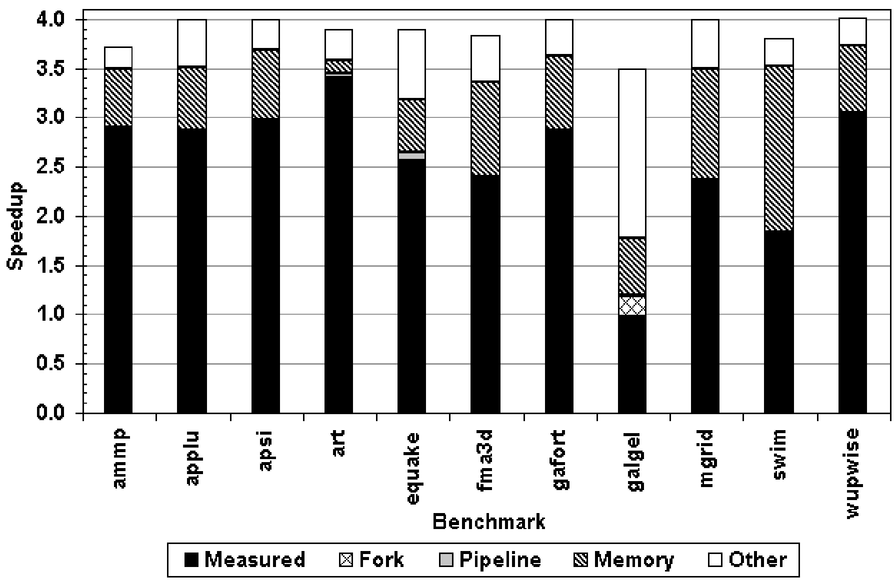

Figure 2: Speedup Component Model for SPEC OMP2001 Benchmarks on 1 and 4 Processors

increase in memory references amplifies the memory bound limitation of the benchmarks.

The UltraSPARC II processors have a first-level write-through 16KB direct-mapped on-chip data cache and a 8MB direct-mapped external cache. The instruction cache is 2-way set associative 16KB on-chip cache. Since each benchmark in the SPEC OMP2001 suite has a large working set size, the first-level cache hit rate is very poor as portrayed by Table 9. For most benchmarks, the hit rate for DC is lower than 90%. For several benchmarks, even the EC hit rate is below 95% on 4 processors. Poor cache performance is the most important reason for the long wait times on load instructions, which translates to lower IPC.

In the following subsections, we will discuss performance of several benchmarks individually.

6.3 ammp, applu, apsi, art, mgrid, and wupwise

ammp, *apsi*, and *mgrid* speedup between 2.5 and 3.0 on 4 processors, whereas *applu*, *art*, and *wupwise* speedup by a factor over 3.0 on 4 processors. Their loss of speedup can be explained mainly by the memory system stalls. Impact of the pipeline stalls and the fork-time overhead is negligible. We did not find load imbalance in any of the parallel regions in these benchmarks.

6.4 equake

In *equake*, increased floating-point dependence is a key reason for pipeline stalls, which leads to the speedup loss of about 0.1. We did not find load imbalance in *equake*. Also, Table 7 shows that almost 70% of the total execution time is spent waiting for the loads to finish. This is true for sequential and parallel executions of the program. One of the key reasons for long load latencies is the poor performance of the secondary cache. *equake* shows the lowest "EC Hit Rate". Long load latencies are responsible for almost 14% of the lost speedup.

6.5 gafort

From Table 5, we can see that "shuffle-do#10" is the most time consuming loop in *gafort* whose "1 CPU" speedup is lowered by calls to OMP_SET_LOCK/OMP_UNSET_LOCK as well as possible false-sharing in the lock array. The most noticeable characteristic of *gafort* is that it spends over 50% of the execution time waiting for the loads to complete as shown in Table 7. In the parallel execution of the program, "Load Use" increased to 55%. Thus, the memory system stalls explain a significant portion of the lost speedup in *gafort*. *gafort* did not exhibit load imbalance in any of the loops.

6.6 galgel

galgel shows the lowest speedup among all of the benchmarks. Its speedup loss comes partly from memory stalls and fork-time overhead. However, the "Other" component in *galgel* is the key reason for the lost speedup. According to Table 5, for the top two parallel regions, average execution time of the parallel execution on 1 processor increased by 220% from the average time for the sequential execution. However, these regions speed up almost perfectly on 2 and 4 processor executions with respect to the 1-processor parallel execution. Thus, *galgel* loses most of its performance as a result of conversion from sequential to parallel. A closer examination of the top two parallel regions reveal that even though the loop bodies are quite small, they contain MATMUL, TRANSPOSE, and DOT_PRODUCT library calls. We did not further analyze the performance of these calls.

6.7 swim

Based on Figure 2, we found that *swim*, although it performs well on 2 processors, does not perform comparable on 4 processors mainly because of memory system stalls. From Table 7, we can conclude that *swim* is a very memory-intensive program. The store buffer seems to be a major bottleneck on our platform. During the sequential execution of the program, nearly 80% of the execution time is spent in two memory system stalls: stalls due to full store buffer and long latencies on loads. Table 9 shows 49% first-level data cache hit rate and a secondary cache hit rate below 90%. Even though the first-level data cache hit rate improves by 50% in the parallel execution, the secondary cache (EC) hit rate does not improve. The fork overhead and the pipeline stalls do not impact the speedup in *swim*.

swim has fourteen 110MB large arrays, which are used and defined during every time-step. These arrays are responsible for excessive capacity and conflict misses. Even though within subroutines CALC1 and CALC2, these arrays show good temporal as well as spatial locality, from CALC1 to CALC2 within the same time-step, temporal locality is not maintained. Our preliminary study shows that by rearranging the calculations across CALC1 and CALC2, such that definitions and uses of CU, CV, H, and Z arrays are close, we may be able to improve the temporal locality in *swim*.

Finally, the fork-time does not hamper the performance significantly. We also found that there is no load imbalance among processors in the most important parallel regions.

6.8 fma3d

fma3d loses nearly 20% of its speedup due to long load latencies. However, it also shows the lowest IC hit rate among all benchmarks (Table 9). Lower IC hit rate is reflected as almost 10% "IC Miss" in Table 7.

7 Conclusions

We have presented basic performance characteristics of the SPEC OMP2001 benchmark suite. We used a high resolution timer on Solaris 5.8 as well as the hardware counters on the UltraSPARC II processors.

We identified major parallel regions in each benchmark and delineated their speedups on 1, 2, and 4 processors. Finally, using a simple quantitative model, we explained partial speedup loss. Across the board, we found the memory system stalls as the most important reason for the speedup loss. While for *galgel* fork-time is important in determining the speedup, *equake* and *art* suffered speedup loss due to pipeline stalls. Finally, *fma3d* is the only benchmark with substantial instruction cache misses as a reason for speedup loss. We did not find load imbalance in any parallel region in any of the benchmarks.

Our detailed measurements suggest the following reasons for the increase in memory latency of the parallel applications. Although there is an expected amount of coherence-related cache misses, the hit ratio of the level one cache increases for all parallel applications. On the other hand, there is a small decrease in level 2 cache hits for several applications. Most importantly, the number of memory operations increases substantially for the parallel applications, compared to their serial variants.

References

[1] Vishal Aslot. Performance Characterization of the SPEC OMP2001 Benchmarks. Master's thesis, Purdue University, 2001.

[2] Vishal Aslot, Max Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B. Jones, and Bodo Parady. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In *Workshop on OpenMP Applications and Tools, WOMPAT 2001, Lecture Notes in Computer Science, 2104*, pages 1–10, 2001.

[3] M. Berry, et. al. The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. *Int'l. Journal of Supercomputer Applications*, 3(3):5–40, Fall 1989.

[4] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware and Software Approach.* Morgan Kaufmann Publishers, Inc., 1999.

[5] R. W. Hockney and M. Berry (Editors). PARKBENCH Report: Public International Benchmarking for Parallel Computers. *Scientific Programming*, 3(2):101–146, 1994.

[6] Seon Wook Kim and Rudolf Eigenmann. Where Does the Speedup Go: Quantitative Modeling of Performance Losses in Shared-Memory Programs. *Parallel Processing Letters*, 10(2 and 3):227–238, 2000.

[7] S. Mukherjee, S. Reinhardt, B. Falsafi, M. Litzkow, S Huss-Lederman, M. Hill, J. Larus, and D. Wood. Wisconsin Wind Tunnel II: A Fast and Portable Parallel Architecture Simulator. *Workshop on Performance Analysis and Its Impact on Design (PAID)*, July 1997.

[8] OpenMP Forum, http://www.openmp.org. *OpenMP: A Proposed Industry Standard API for Shared Memory Programming*, October 1997.

[9] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. RSIM Reference Manual. Version 1.0. Technical Report 9705, Electrical and Computer Engineering Department, Rice University, July 1997.

[10] Sun Microsystems Inc. *UltraSPARC User's Manual*, 1997.

[11] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, 1995.