View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Crossref

Scientific Programming 11 (2003) 105–124
IOS Press

Quantitative performance analysis of the SPEC OMPM2001 benchmarks

Vishal Aslot[a] and Rudolf Eigenmann[b]
[a]*Purdue University, Department of Electrical and Computer Engineering, 2600 Gracy Farms Lane, Austin, Texas 78758, USA*
Tel.: +1 512 833 5545; Fax: +1 512 838 3484; E-mail: vaslot@ecn.purdue.edu
[b]*Purdue University, Department of Electrical and Computer Engineering, 1285 Electrical Engineering Building, West Lafayette, Indiana 47907, USA*
Tel.: +1 765 494 1741; Fax: +1 765 494 6440; E-mail: eigenman@ecn.purdue.edu

Accepted 17 July 2002

Abstract. The state of modern computer systems has evolved to allow easy access to multiprocessor systems by supporting multiple processors on a single physical package. As the multiprocessor hardware evolves, new ways of programming it are also developed. Some inventions may merely be adopting and standardizing the older paradigms. One such evolving standard for programming shared-memory parallel computers is the OpenMP API. The Standard Performance Evaluation Corporation (SPEC) has created a suite of parallel programs called SPEC OMP to compare and evaluate modern shared-memory multiprocessor systems using the OpenMP standard. We have studied these benchmarks in detail to understand their performance on a modern architecture. In this paper, we present detailed measurements of the benchmarks. We organize, summarize, and display our measurements using a Quantitative Model. We present a detailed discussion and derivation of the model. Also, we discuss the important loops in the SPEC OMPM2001 benchmarks and the reasons for less than ideal speedup on our platform.

1. Introduction

With the breakthroughs in 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, 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 paradigms have emerged. While each one has its benefits, for small to medium range SMPs, directive-based programming and 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 [6] (Application Programming 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 the needs of parallel programmers.

As SMPs become more commonplace, it is important to be able to evaluate their performance with a standard set of benchmarks. Several parallel benchmark suites over the past 20 years have attempted to fill the void, including SPLASH 2 [10], Parkbench [9], and the Perfect Benchmarks [5]. More recently, the Standard Performance Evaluation Corporation (SPEC) has released a new set of benchmarks targeted towards modern SMP systems, called SPEC OMP. The suite contains SPEC OMPM2001 (a medium, 2 GB data set) and SPEC OMPL2001 (a large, 7 GB dataset). The data

ISSN 1058-9244/03/$8.00 © 2003 – IOS Press. All rights reserved

set sizes define the maximum memory requirements for a single-processor run. In this paper we analyze SPEC OMPM2001. It contains eleven programs written in Fortran or C, which have been made parallel using the OpenMP API. More information about the benchmarks and the parallelization effort can be found in [15].

The SPEC OMPM2001 suite has been released in June 2001. This paper aims at characterizing the benchmarks on a modern, commercial multiprocessor system. To this end, we present detailed measurements using timers and hardware counters on our platform. We summarize the data using a Quantitative Model and derive this model in detail. We discuss the individual loops in the benchmarks. Finally, we analyze the reasons for the difference between measured and ideal speedups.

The remainder of the paper is organized as follows. We give a brief overview of the important OpenMP constructs in Section 2. Section 3 briefly presents the runtime environment in which we carried out our experiments. We describe the key concepts behind our Quantitative Model and derive it in detail in Section 4. Section 5 presents overall measurements for the benchmarks. In Section 6, we discuss the important loops of several benchmarks and reasons for their speedup loss. Section 7 summarizes the discussions in Sections 5 and 6. Finally, Section 8 concludes the paper.

2. Overview of OpenMP

The OpenMP standard is a set of directives, library functions, and environment variables to write shared-address-space (SAS) parallel programs in Fortran and C languages. The OpenMP API resulted from standardizing vendor-specific directives for writing parallel programs. OpenMP encompasses some of the key concepts behind writing shared-address-space programs with a few simple directives and library functions. We briefly introduce the OpenMP constructs referred to in this paper. See [6,7] for details on the OpenMP standard.

In OpenMP, a parallel region is declared by placing an `OMP PARALLEL/OMP END PARALLEL` directive around it. Such a region will be executed by every participating processor. A group of participating processors is called a team of threads. Usually, OpenMP programs create one thread per processor. Therefore, the following description refers to *processor* where the OpenMP standard would use the more abstract term *thread*. Variables within the parallel region are declared private per processor or shared among the processors with a `PRIVATE` or a `SHARED` clause after `OMP PARALLEL`, respectively. The private variables declared in this fashion are undefined at the beginning of the parallel region and are undefined at the end. Thus, they should be used only within the parallel region. On the other hand, `THREADPRIVATE` variables are private to each processor, but their values persist from one parallel region to the next. `PRIVATE`, `SHARED`, and `THREADPRIVATE` are known as the data environment clauses.

If a `for`-loop in C or a `DO`-loop in Fortran have independent iterations, which can be executed by different processors without generating incorrect results, the iterations can be easily partitioned among the available processors using `omp for` or `OMP DO/OMP END DO` construct. The construct is placed immediately before the loop and is called a worksharing construct. If the parallel region contains only one worksharing construct and does not contain any serial code either, it may be possible to combine `OMP PARALLEL` and `OMP DO` and use `OMP PARALLEL DO` instead.

In a worksharing construct, it is possible for the programmer to instruct the compiler to divide the iterations among the processors in a specific way. One possibility is to divide the iterations equally so that each processor executes the same number of iterations. This is called block or static scheduling. It is a default in the OpenMP standard with the above worksharing constructs. Block scheduling assumes that each iteration does roughly the same amount of work, and hence, all processors will perform the same amount of computation. When such is not the case, the programmer can specify either dynamic or guided scheduling. While these two kinds of scheduling are different in implementation, the basic concept is the same: each processor fetches more iterations once it is finished with its current share of iterations. Thus, a slower processor may do less work and a faster one may perform more. However, each processor works for about the equal amount of time. This helps avoid load-imbalance. The type of scheduling can be specified using a `SCHEDULE` clause next to an `OMP DO` construct.

Since the OpenMP constructs are inherently multithreaded, it is necessary to provide some form of mutual exclusion and global synchronization. In the OpenMP standard, the mutual exclusion is provided by enclosing a critical section of code within either the `OMP CRITICAL/OMP END CRITICAL` directives or by using the OpenMP library functions `omp_set_lock/omp_unset_lock`. The code

within these constructs is executed by each processor. If part of the code within a parallel region needs to be serialized among the processors, but only one processor must execute it and the others must skip it, an `OMP SINGLE/OMP END SINGLE` construct is appropriate. Lastly, when *all* processors must arrive at a certain point in the program before continuing on, an `OMP BARRIER` directive can be used. No processor can continue past the barrier until all processors have finished executing the code before it. Thus, it is a global synchronization construct. A barrier is implicitly defined in `OMP END PARALLEL`, `OMP END DO`, and `OMP END SINGLE` among many other constructs. Because a barrier introduces overhead, it is desirable to remove it where possible from `OMP END DO`. This can be done by placing a `NOWAIT` clause next to it. This is typically done for the last worksharing construct in a parallel region. In many instances it is necessary, not only to synchronize all processors, but also to guarantee that all memory operations before the synchronization point are complete, and all processors have a consistent view of memory [16]. An `OMP FLUSH` directive updates the global view of memory for all processors by completing all memory operations.

Finally, reduction operations are common in parallel programming, where several processors update a single scalar or array variable. The reduction operations can be performed by inserting a `REDUCTION` clause following a worksharing construct in OpenMP. The OpenMP standard supports many most frequently used reduction operators such as $+$, $-$, MIN, and MAX. The OpenMP standard, revision 2.0 and above also includes array reduction operations, which are not yet supported by all compilers.

3. Experimental setup

We ran the benchmarks on a quad processor Sun Enterprise 450 SMP system from Sun Microsystems Inc.. The basic configuration of the system is shown in Table 1.

All measurements were taken in single-user mode. We executed each benchmark with the full data set.[1] from the released version of the SPEC OMPM2001 Toolkit environment. All of the executions validated within the tolerances specified by the SPEC OMPM2001 Toolkit.

[1] The full data set is called Reference set in the SPEC toolkit

Table 1
Basic hardware and software setup

Machine Model	Sun Enterprise 450
CPU	480 MHz UltraSPARC II
No. of CPUs	4
Used CPUs	4
Memory per Node	4 GB
Instruction Cache	16 KB
Data Cache	16 KB, 32 byte line, direct mapped, write through, write no-allocate
External Cache	8 MB, 64 byte line, unified, write allocate, inclusion with data L1
Peak Mem. Bandwidth	1.78 GB/sec
Operating System	Solaris 5.8
Page Size	8KB
Fortran Compiler	Sun Forte 6, update 1
C Compiler	Kuck & Associate's GuideC 4.0 with Sun Backend

In order to account for over 99% of the total execution time, we instrumented all time-consuming parallel and serial sections of the programs with a high-resolution timer. The overhead introduced by instrumentation is 2% or less for all benchmarks, which is within a tolerable range. In order to gain additional insight into the performance of the programs, we enhanced our instrumentation libraries to measure the hardware counters on the UltraSPARC II processors. Each UltraSPARC II processor has two 32-bit hardware counters. There are up to 22 distinct hardware events that can be measured with these counters. Our library handles the overflow of the counters correctly. We measured all hardware events for the sequential and the 4-processor parallel executions.

Just as in the case of timers, we have measured the hardware events per program section as well as per processor. In order to measure the events per processor and inside a parallel region, we applied several modifications to the region. In order to measure the parallel execution time, we instrumented each `OMP PARALLEL` and `OMP END PARALLEL` section but not the worksharing constructs inside each section. Because we wanted to instrument at the parallel region level rather than worksharing construct level, a number of `OMP PARALLEL DO` worksharing constructs had to be converted to `OMP PARALLEL/OMP DO` pair, which then allowed instrumentation at the parallel region level. Also, we measured the fork-join and the load-imbalance times by instrumenting around `OMP PARALLEL/OMP END PARALLEL` directives. We define the fork-time as the time spent while entering a parallel region. The time spent in `OMP END PARALLEL` construct is a sum of the join-time

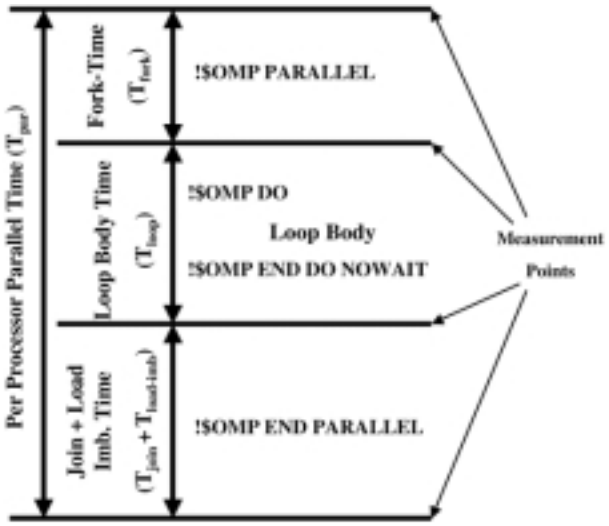


Fig. 1. Measurement of different loop times.

and the load-imbalance time. Since the join-time is typically very small (several microseconds), we expect that the time spent in `OMP END PARALLEL` matches the load-imbalance time closely. Also, in order to measure the load-imbalance, we appended a `NOWAIT` clause to `OMP END DO` wherever possible. The `NOWAIT` clause removes the implicit barrier in `OMP END DO`, permitting each thread to reach the barrier in `OMP END PARALLEL` as soon as it finishes useful work. Figure 1 shows how we attribute times.

We compiled each benchmark using the SPEC Toolkit. We used the `-fast` flag on all C and Fortran benchmarks. Also, with the Fortran benchmarks, we used `-O5` and `-xprofile`. We found more specific optimization flags to achieve peak performance for *gafort*. The C programs used GuideC as the OpenMP compiler combined with Sun's C compiler. The only optimizations for the C programs were `-xfast` and `-xalias_level=strong`. We used the same set of optimization flags to generate the sequential and the parallel versions of the programs. We did not make any algorithmic changes to the parallel version that would enhance or degrade the speedup. We compiled and ran the exact same code in the sequential and the parallel versions. We simply used or not used a `-openmp` flag with the Fortran programs to generate either a parallel or a sequential version, respectively. Compiling with GuideC automatically generates parallel code. To compile a sequential version of a C program, we used Sun's C compiler directly.

4. A quantitative model

We introduce a model in order to quantify our performance observations. More specifically, we want to exhibit quantitatively the reasons that limit scalability of parallel programs. The basic idea is to analyze the difference between measured and ideal speedup of the parallel program. The model will subdivide this difference into *speedup components*, which represent the overhead factors responsible for suboptimal performance. The issues in doing so are to (1) define a complete and orthogonal set of overhead factors, (2) measure the factors or derive them from measured data, and (3) compute the model values thereof. The following subsections present our solutions to these issues. The presented model refines the Speedup Component Model introduced in [13]. Another model that attempts to quantify the performance of loop-dominated scientific applications is presented in [3]. Similar to the model in [3] that focuses on explaining performance difference between the upper bound on the best achievable performance and the realized performance, our model also attempts to explain the difference between the ideal performance and the measured performance.

4.1. Overhead Factors

The total execution time of a parallel program or program section can be divided into the following factors:

1. Time spent performing useful work,
2. stalls due to waiting for data accesses,

3. stalls due to pipeline bubbles (e.g., branch misprediction stalls),
4. idle time due to serial program sections and load-imbalance,
5. parallelization overhead, such as fork and join operations,
6. time spent in extra computation, not present in the serial code (such as initialization and final sum of parallel reductions or OpenMP intrinsic function calls),
7. time attributable to less optimal parallel code generation (e.g., due to more conservative compilation)
8. time spent at low-level synchronization points, such as fence instructions (e.g., `MEMBAR` and `STBAR` in the UltraSPARC II processor)

We divide the overhead factors into two categories. The first category includes overheads also present in the serial program (factors 2 and 3). The second category includes overheads only present in the parallel program execution (factors 4 – 8).

While we believe these overhead factors to be reasonably complete, there are second-order effects. For example, even in single-user mode, a program may be interrupted by low-level system processes. Our measurements will need to ensure that such effects are negligible. To account for possible inaccuracies, we will introduce an additional factor that represents not-modeled effects.

The orthogonality of the overhead factors needs careful consideration. For example, idle time due to load-imbalance and due to serial program execution must be distinguished clearly. This will become important for the measurement of the factors, discussed in the next subsection. Orthogonality is a further issue in that overhead factors may hide each other. For example, a code section may exhibit 40% memory stall cycles and 10% pipeline stalls. The programmer may be able to reorder the computation so that the memory stalls decrease by 20%. However, as a result the pipeline stalls may increase by 10%. While it is important for our instruments to attribute each processor cycle to exactly one time factor, this requirement tends to hide other factors. Note, it would be incorrect to conclude that removing a certain speedup component via some improvement would necessarily lead to a speedup increase proportional to the affected speedup component. Instead, the speedup components quantify the relative importance of overhead factors in the measured program execution. When interpreting our results we will revisit this fact.

Table 2
Measured parameters

No.	Parameter	Notation	Device
1	Sequential or Serial Time	T_{ser}	Timer
2	Parallel Time	T_{par}	Timer
3	Loop Body Time	T_{body}	Timer
4	Fork Time	T_{fork}	Timer
5	IC Miss Stalls	T_{IC}	HWC
6	Store Buffer Stalls	T_{SB}	HWC
7	Load Use Stalls	T_{LD}	HWC
8	Load Stalls on RAW	T_{RAW}	HWC
9	Branch Mispredicitons Stalls	T_{BM}	HWC
10	Floating-Point Dependence Stalls	T_{FP}	HWC
11	Amdahl's Time	T_{amdahl}	Timer

4.2. Measuring and Deriving Overheads

We have instrumented our programs with calls to the Sun UltraSPARC II hardware counter libraries at the points indicated in Fig. 1. From these measurements we obtained the following performance factors. All factors apply to individual processors.

Category 1 (overheads present in both serial and parallel code):

1. memory system stalls (T_{memory}), further subdivided into stalls due to instruction cache misses (T_{IC}), store buffer full (T_{SB}), dependence on earlier incomplete load (T_{LD}), and load dependent on an earlier store (T_{RAW}).
2. pipeline stalls (T_{pipeline}), further subdivided into stalls due to branch misprediction (T_{BM}), and floating-point dependence (T_{FP}).

Category 2 (overheads present in parallel code only):

1. load-imbalance ($T_{\mathrm{load-imb}}$),
2. serial program sections (T_{amdahl}),
3. fork and join overhead ($T_{\mathrm{fork}}, T_{\mathrm{join}}, T_{\mathrm{fj}}$)

Table 2 lists the overheads and times that we obtained through direct measurement. T_{ser} and T_{par} represent the serial and parallel execution time, respectively. T_{amdahl} represents time spent by all but one processors during serial program sections. The used hardware counters ensure that each machine cycle is attributed to exactly one overhead factor, satisfying the orthogonality criterion discussed in Section 4.1. Within our overhead factors, only the memory system and pipeline stalls can hide each other to a certain extent. However, load-imbalance and fork-join overheads are orthogonal to all other categories.

Table 3 lists overheads and parameters that we derived from measured values. $T_{\mathrm{join}} = T_{\mathrm{par}} - T_{\mathrm{fork}} -$

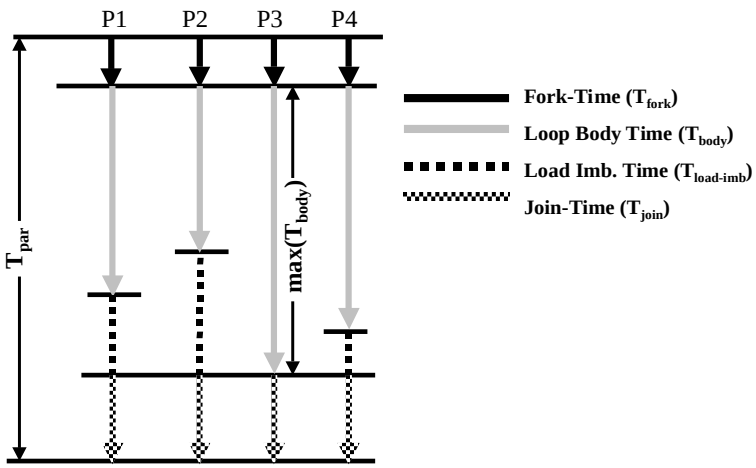


Fig. 2. Illustration of load-imbalance.

Table 3
Derived parameters

No.	Parameter	Notation	Unit
1	Speedup	$Speedup$	Unitless
2	Useful Computation	T_{comp}	Cycles
3	Memory Stalls	T_{memory}	Cycles
4	Pipeline Stalls	T_{pipeline}	Cycles
5	Join Time	T_{join}	Cycles
6	Fork-Join Stalls	T_{fj}	Cycles
7	Load-Imbalance	$T_{\mathrm{load-imb}}$	Cycles
8	Not Modeled	$T_{\mathrm{not-modeled}}$	Cycles

$\max(T_{\mathrm{body}})$ is the barrier time at the end of a parallel region. It is computed as the total time taken by the region excluding the fork time and the region time on the slowest processor, as illustrated in Fig. 2. $T_{\mathrm{fj}} = T_{\mathrm{fork}} + T_{\mathrm{join}}$. The load-imbalance is the time each processor waits for all other processors to reach the final barrier of the region: $T_{\mathrm{load-imb}} = T_{\mathrm{par}} - T_{\mathrm{body}} - T_{\mathrm{fj}}$. The fork and join overhead is the same on all processors, while the load-imbalance can differ.

Table 3 also shows several terms that do not represent overheads. *Speedup* is computed as $T_{\mathrm{ser}}/T_{\mathrm{par}}$. $T_{\mathrm{comp,s}}$ and $T_{\mathrm{comp,p}}$ represent the time taken by useful computation in the serial and parallel code, respectively. Subscripts s and p stand for serial and parallel, respectively. $T_{\mathrm{comp,s}} = T_{\mathrm{ser}} - T_{\mathrm{memory,s}} - T_{\mathrm{pipeline,s}}$. We cannot measure or compute $T_{\mathrm{comp,p}}$ precisely and, instead, estimate $\sum^{N} T_{\mathrm{comp,p}}$ over all processors (N) to equal $T_{\mathrm{comp,s}}$. The inaccuracy of this estimate factors into $T_{\mathrm{not-modeled}}$, which represents a time component that is not accounted for in our model. $T_{not-modeled}$ exhibits the inaccuracy of our model. Additional factors covered by this term include the items 6–8 listed in Section 4.1. We will discuss effects believed to be caused by these overheads in Section 6.

4.3. Computing the model values

The goal of our model is to split the difference between measured and ideal speedup into a number of *speedup components*. These components represent the overhead factors, each being proportional to the increase in overhead cycles from the serial to the parallel code. That is,

$$Speedup_{\mathrm{loss}} = Speedup_{\mathrm{ideal}} - Speedup_{\mathrm{measured}},$$

where $Speedup_{\mathrm{loss}}$ is split into its contributing overheads.

We first consider a fully parallel region. For this region, the parallel execution time equals the sum of the times spent in useful computation plus overheads:

$$\begin{aligned} T_{\mathrm{par}} &= T_{\mathrm{comp,p}} + T_{\mathrm{memory,p}} + T_{\mathrm{pipeline,p}} + T_{\mathrm{fj}} \\ &\quad + T_{\mathrm{load-imb}} \\ &= \sum^{K} T_{\mathrm{factor,p}} \end{aligned}$$

T_{par} is the same on all processors, while its constituents may differ. $\sum^{K} T_{\mathrm{factor,p}}$ is short for the sum of all K constituent times. Summing up this term across all N processors results in

$$\sum^{N,K} T_{\mathrm{factor,p}} = N \cdot T_{\mathrm{par}}$$

With this relationship and the definition of speedup, it follows that

$$\begin{aligned} Speedup_{\text{loss}} &= Speedup_{\text{ideal}} - Speedup_{\text{measured}} \\ &= N - \frac{T_{\text{ser}}}{T_{\text{par}}} \\ &= \frac{\sum^{N,K} T_{\text{factor}} - T_{\text{ser}}}{T_{\text{par}}} \\ &= \frac{1}{T_{\text{par}}} \\ &\quad \sum_{i=1}^{K} \left(\sum^{N} T_{\text{factor}_{\text{i}},\text{p}} - T_{\text{factor}_{\text{i}},\text{s}} \right) \end{aligned} \tag{1}$$

Hence, each overhead factor contributes a speedup component proportional to the sum of its cycles consumed on all processors in parallel minus the overhead time in the corresponding sequential program. For the factors T_{fj} and $T_{\text{load-imb}}$ the sequential overhead is zero. We have defined $\sum^{N} T_{\text{comp,p}} = T_{\text{comp,s}}$ as an estimate. With this estimate the speedup component of T_{comp} is zero, that is, the combined useful cycles in the parallel code equal the useful cycles of the sequential code. In its place we use the $T_{\text{not-modeled}}$ term, capturing model inaccuracies, as discussed in Section 4.1. Hence,

$$\begin{aligned} Speedup_{\text{loss}} = {} & SC_{\text{memory}} + SC_{\text{pipeline}} \\ & + SC_{\text{load-imb}} + SC_{\text{fj}} \\ & + SC_{\text{not-modeled}} \end{aligned} \tag{2}$$

where the first four components are computed by formula 1, and $SC_{\text{not-modeled}}$ fills the gap between $Speedup_{\text{ideal}} - Speedup_{\text{measured}}$ and $SC_{\text{memory}} + SC_{\text{pipeline}} + SC_{\text{fj}} + SC_{\text{load-imb}}$.

4.3.0.1. The model for an entire program is a small extension of the model for a fully parallel region. In addition to parallel regions, a whole program contains serial sections.[2]. According to Amdahl's law, the maximum speedup of such a program is limited to $Speedup_{\text{ideal}} - SC_{\text{amdahl}}$, where

$$SC_{\text{amdahl}} = N - \frac{1}{(p/N) + (1-p)}, 0 \leqslant p \leqslant 1,$$

and p is the parallel coverage. Parallel coverage is a fraction of the serial program that is enclosed by parallel regions. Hence, for parallel programs that include serial sections, the speedup components are,

$$\begin{aligned} Speedup_{\text{loss}} = {} & SC_{\text{memory}} + SC_{\text{pipeline}} \\ & + SC_{\text{load-imb}} + SC_{\text{fj}} \\ & + SC_{\text{not-modeled}} + SC_{\text{amdahl}} \end{aligned} \tag{3}$$

Note, that a speedup component can amount to a negative value if the overheads in the parallel code are less than the overheads in the serial code. This can occur when, for example, the compiler has applied locality-enhancement transformations to the parallel code but not to the serial code. In this case the negative SC_{memory} exhibits the source of potential superlinear speedup behavior. Negative components can also result from measurement artifacts, if two overheads present in the same cycle are counted differently in two program executions. For example, a cycle may include both a memory and a pipeline stall; in one run the hardware counters record a pipeline stall, in the second run a memory stall. In this case one can expect that, while one speedup component becomes negative, another component grows significantly.

In the following sections we will use the two model formulas, 2 and 3, to characterize the speedup loss of the SPEC OMPM2001 benchmarks and their important loops.

5. Overall performance

We have used the Quantitative Model as presented in the previous section to organize and summarize the timer and hardware counter measurements of the SPEC OMPM2001 benchmarks on our platform. Before presenting the speedup components of the programs, we will outline their most basic characteristics.

5.1. Basic characteristics

Table 4 shows the parallel coverage based on a sequential run, the sequential and 4-processor parallel execution times, and the overall speedup of the benchmarks. With the exception of *galgel*, all benchmarks show a parallel coverage of 97% or more in parallel and sequential runs. *Galgel* shows a parallel coverage of about 95%. Thus, the SPEC OMP codes are highly parallel. The table also shows that the SPEC OMP codes run for a long time even with the medium Reference data sets.

[2]The serial sections of code in parallel and sequential executions may take different amounts of time. We are assuming that they take a same amount of time. Our assumption is valid only because the serial sections are very small in all programs and do not contribute substantially to the overall execution time.

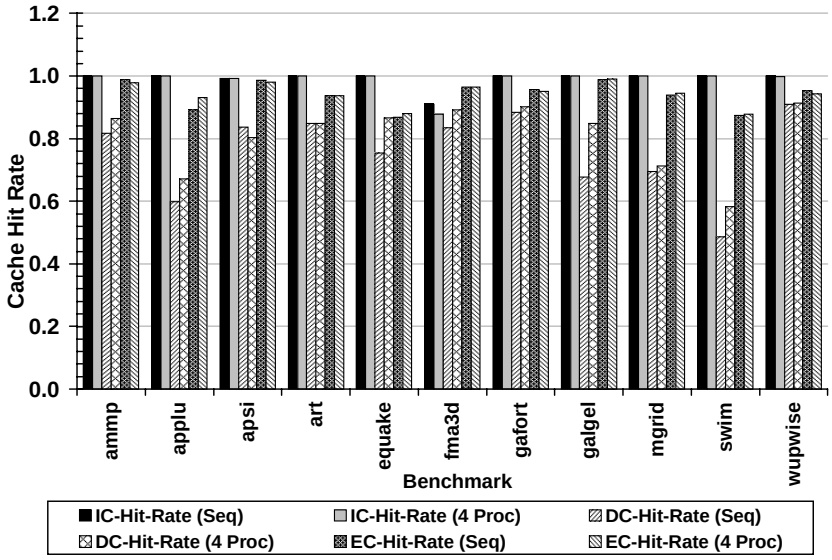


Fig. 3. Cache hit rates (IC=Instr. Cache, DC=Data Cache, EC=External (L2) Cache).

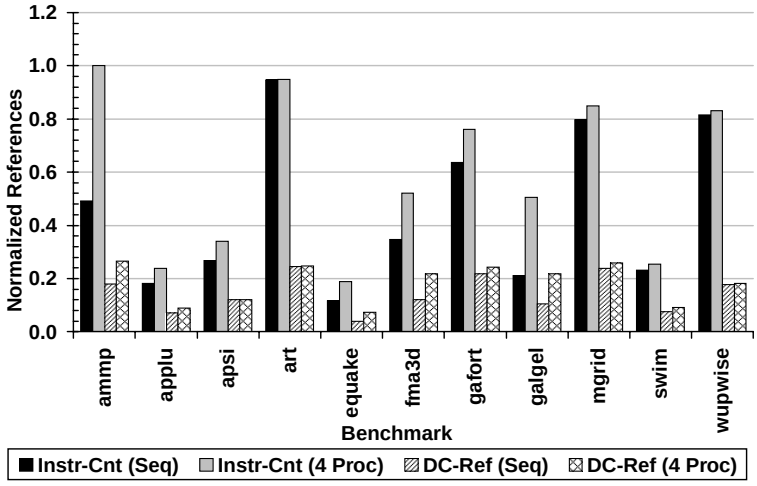


Fig. 4. Total and memory-related instructions normalized with respect to the 4-processor count of instructions in ammp.

Figures 3 and 4 show the overall cache hit rates, the number of executed instructions, and the number of memory access instructions. Figure 3 indicates that most benchmarks suffer from poor first-level data cache hit rate. Even the secondary cache hit rates are below 90% in some instances. The secondary cache hit rate is computed with respect to the total references going *only* to the secondary cache. With the exception of *apsi*, *art*, and *wupwise*, most programs report an increase in the cache hit rates. The instruction cache hit rates are almost 100% in all instances except *fma3d*.

Figure 4 shows the relative proportion of the memory access instructions to the overall instructions. It is between 20% and 45% in all benchmarks. Also, the figure shows that all benchmarks perform more work in the parallel version compared to the sequential version. *Ammp*, *equake*, *fma3d*, and *galgel* show a large increase in the number of memory access instructions.

Figure 5 shows the fork-join overhead scaling of the benchmarks. Relative to the execution times, the fork-join overhead is very small. Consequently, Fig. 6 shows negligible fork-join speedup components. *Galgel* is an exception; it has the largest fork-join overhead, mainly due to several million invocations of the LAPACK routines. At the other extreme, *art* has nearly zero fork-join overhead, because it has only one loop which is re-

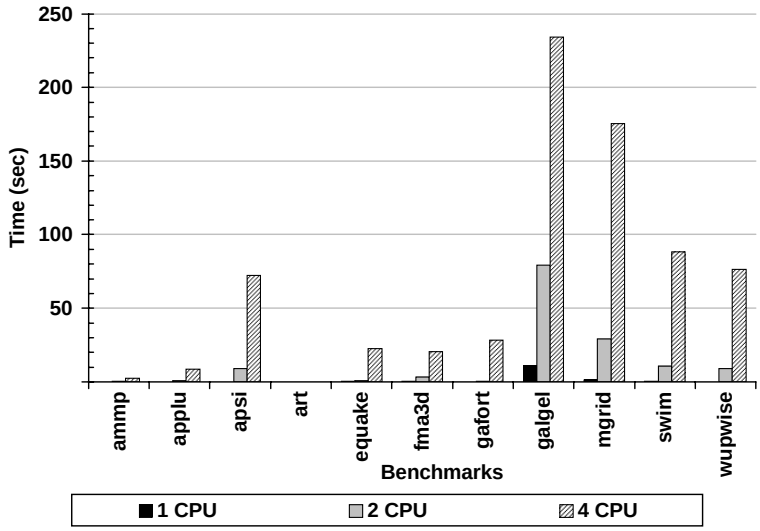


Fig. 5. Fork-join overhead.

Table 4
Basic runtime characteristics of the SPEC OMPM2001 benchmarks

Code	Parallel Coverage (%)	Execution Time(sec) Seq.	4	Speedup (4 CPU)	# of Parallel Regions
ammp	99.12	16841	5898	2.84	7
applu	99.99	11712	3677	3.14	22
apsi	99.84	8969	3311	2.72	24
art	99.83	28008	7698	3.62	3
equake	99.16	6953	2806	2.54	11
fma3d	99.46	14852	6050	2.50	$92/30^1$
gafort	99.94	19651	7613	2.56	6
galgel	95.58	4720	3992	1.19	$32/31^1$
mgrid	99.98	22725	8050	2.84	12
swim	99.44	12920	7613	1.70	8
wupwise	99.83	19250	5788	3.31	10

[1]Programmed regions / regions called at runtime.

sponsible for about 99% of the execution time and is invoked only once. Finally, even if all loops in *apsi* have a relatively small number of invocations (few tens to few hundred), the fork-join overhead is high compared to loops in some other benchmarks with a similar invocation count. The higher fork-join time in *apsi* comes from the `malloc` and `free` calls by the Fortran compiler to allocate a privatized array inside a parallel region. In fact, the local arrays in a subroutine are always allocated dynamically by our Fortran compiler. Since the Fortran compiler extracts each parallel region into a subroutine [8,11] and declares all `PRIVATE` variables as local variables in that subroutine, the `PRIVATE` arrays end up getting allocated and freed dynamically. The high overhead of the `malloc/free` calls comes from the fact that each thread gets a copy of the array from the same process heap. Therefore, these calls are serialized by the operating system. Also, allocating memory dynamically is usually costly. Such overhead is experienced every time an `OMP PARALLEL` construct is executed in *apsi*. There is a multithreaded malloc library (libmtmalloc) available on Solaris, which might alleviate serialization. However this library was not used with our compilers.

5.2. Overall Speedup Components

Figure 6 shows the overall speedup components of the SPEC OMPM2001 benchmarks. It breaks down the lost speedup into the responsible components. We have used $Speedup_{ideal}$ = 4 to compute the overall speedup components. We can make several observations from Fig. 6:

- The key reason for speedup loss is the memory stalls, which increase in the parallel versions.
- Fork-join and load-imbalance overheads are relatively minor reasons for the lost speedup.
- Pipeline stalls are important in *art* and *equake*.
- *Swim* has a negative speedup loss component, enabling potential superlinear performance. However, of all the benchmarks it also has the largest speedup loss due to memory stalls.
- Our model is the most accurate for *apsi*, *mgrid*, and *wupwise*, where it explain almost the entire speedup loss. It is fairly accurate for *gafort*. On the other hand, it is the least accurate for *galgel*, because it has a "Not Modeled" component of about

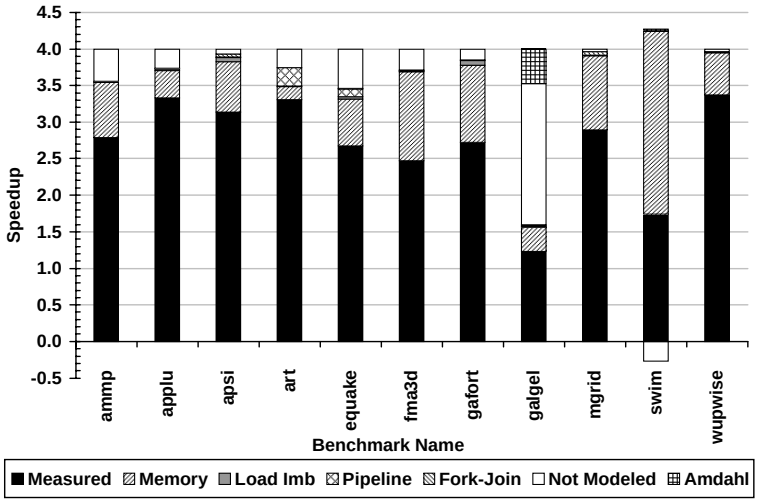


Fig. 6. Overall speedup components.

2.0. Nevertheless, a part of *galgel*'s speedup loss can be explained by the memory stalls component.

– All benchmarks, except *galgel*, have nearly zero "Amdahl" speedup component. In *galgel*, the "Amdahl" speedup component is about 0.5 due to only 95% parallel coverage.

Figure 7 further categorizes the speedup components related to memory and pipeline stalls. All benchmarks suffer from "Load Use" stalls. These stalls delay all instructions in the execute and the grouping stages of the UltraSPARC II pipeline [12]. "Store Buf." related stalls are important in *apsi*, *fma3d*, *galgel*, *mgrid*, *swim*, and *wupwise*. *Swim* has a very large speedup loss component due to the "Store Buf." stalls. The "Store Buf." stalls result from a full store buffer. *Fma3d* shows a speedup component because of the instruction cache miss stalls ("IC Miss"), and *gafort* loses speedup because of the read-after-write (RAW) dependence. Finally, *equake* experiences reduced scalability due to the stalls related to floating-point dependences. The "FP Dependence" stalls occurs when the first instruction in the group depends on the result from an earlier floating-point instruction. These stall cycles are counted only if the earlier floating-point instruction is *not* waiting on a load. Thus, "Load Use" and "FP Dependence" are mutually exclusive.

6. Loop-by-loop performance

In this section we discuss the performance of the individual benchmarks and their major loops. Table 5 shows the speedup components for these loops. We have computed speedup components for the loops using formula 2 for the fully parallel region (see Section 4.3 and $Speedup_{\text{ideal}} = 4$.

6.1. Ammp

Ammp is about 13,500 lines of C code in the area of chemistry/biology. Mmfvupdate_5 is the most important loop in *ammp*. It has an average execution time of 80 seconds on a single processor. Table 5 shows the speedup components of mmfvupdate_5, which are identical to the overall program. All loops show negligible load-imbalance, which is in part due to the chosen `guided` scheduling option.

From the hardware counter measurements, we found that the first-level data cache hit rate for mmfvupdate_5 increases from 79% to 85% for loads and from 92% to 93% for stores while running in parallel. However, there are 1.5 times more loads in the parallel version than in the sequential one. The number of stores remains nearly unchanged between the two versions. From Fig. 7, we can see that "Load Use" is the biggest reason for the speedup loss. Despite the increase in the cache hit rate, mmfvupdate_5 experiences more memory stalls on 4 processors. We attribute this effect to the increase in loads. Even though the cache hit rate improves, absolute number of cache misses goes up by 35% for loads and 8% for stores. It is the increase in cache misses that contributes to the increased memory stalls.

We discovered that one reason for the increase in loads is the `threadprivate` array `naybor`.

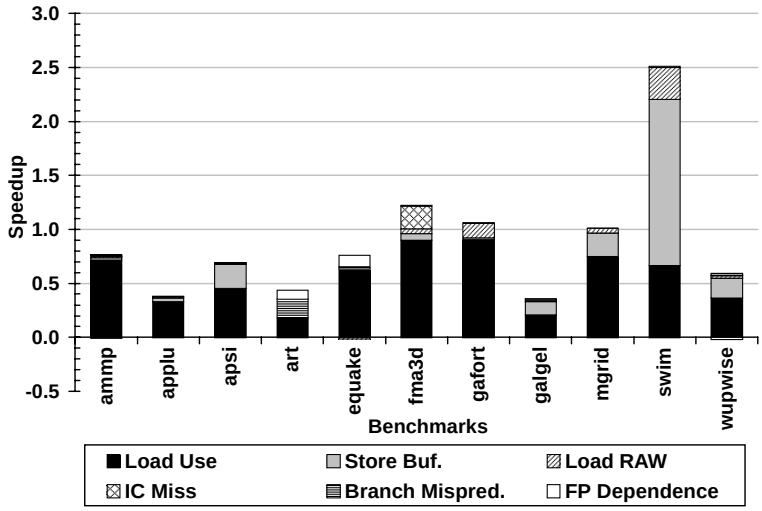


Fig. 7. Breakdown of the memory and pipeline components.

`naybor` is an array of 27 integers. It is responsible for about 18% increase in the memory references suggesting that accessing `threadprivate` variables involves more memory references than accessing `private` variables. As an experiment, we declared `naybor` as a `private` array. We found that the number of memory references went down, and the speedup increased to 2.92.

Since mmfvupdate-5 uses `omp_set_lock/omp_unset_lock` in 3 places, there are two overheads in the parallel version, that are not present in the sequential program: an overhead of executing extra instructions while spin waiting and delays due to lock contention. In order to quantify the overall contribution of the locks to the speedup, we removed them temporarily. In this loop, removing the locks is a programming error. However, we found that the program had executed correctly most of the time without the locks suggesting that the race conditions exist, but they are infrequent. After removing the locks, the speedup increased to 2.99 on 4 processors. Our model does not capture this speedup component explicitly. It shows up as a part of "Not Modeled" component.

Also, we discovered that the sharing of the lock variable leads to increased invalidations and copybacks boosting the secondary cache misses. See [14] for the detailed data on invalidations, copybacks, and secondary cache misses. Such sharing of the lock variable is an example of true-sharing in the SPEC OMP benchmarks.

6.2. Applu

Applu is about 4,000 lines of Fortran code in the area of fluid dynamics. From Table 5, we see that ssor_do#3 is the most important loop in *applu*. It has 177 seconds of average execution time on a single processor and is responsible for about 81% of the overall execution time. Rhs-do#1 to rhs-do#4 are also important collectively.

Ssor_do#3 has a speedup of 3.5, which is among the best in the benchmark suite. The remaining speedup component of 0.5 is due to "Load Use" stalls (0.3) and not-modeled effects (0.2), as shown in Figures 6 and 7. We attribute the increased memory stalls in the parallel code in part to code patterns that lead to increased demands on the memory bus. Code examination of ssor-do#3 shows that it contains two DO-loops. Each loop makes two subroutine calls. Each subroutine consists of a doubly nested loop that performs some matrix computations. In these doubly nested loops, we find references to several shared arrays. At runtime, the loop exhibits low computation-to-memory-access ratio. In general, the code that accesses large arrays with little or no computation leads to increased demands on the memory bus or interconnect resulting in longer memory access latencies. With little or no computation to perform, a processor issues loads and stores more frequently and waits for them to complete. This situation is worsened in parallel by frequent bus locking and queuing delays in the interconnect, leading to longer memory access latencies and, in turn, more memory stalls. While we cannot measure bus and memory bank contentions, we include such effects in the increased

Table 5
Loop-by-loop speedup components of the SPEC OMPM2001 benchmarks

Code	Loop Name	% Exec. (Seq)	Load-Imb	Pipeline	Memory	Fork-Join	Not Modeled	Measured
ammp	mmfvupdate_5	97.12	0.01	0.00	0.75	0.00	0.43	2.81
	mmfvupdate_2	2.10	0.02	0.02	0.20	0.01	0.97	2.78
applu	ssor_do#3	81.25	0.02	0.01	0.31	0.00	0.18	3.49
	rhs_do#3	4.54	0.17	−0.02	−0.17	0.03	0.32	3.68
	rhs_do#4	3.91	0.03	−0.04	−0.05	0.00	0.25	3.80
	rhs_do#2	3.54	0.03	0.01	0.90	0.01	0.12	2.93
	rhs_do#1	2.07	0.00	−0.01	0.43	0.01	1.80	1.76
apsi	RUN_DO#60	10.85	0.08	0.01	0.69	0.02	−0.04	3.24
	RUN_DO#40	10.83	0.06	0.01	0.69	0.02	−0.03	3.25
	RUN_DO#30	10.82	0.06	0.01	0.67	0.03	−0.02	3.25
	RUN_DO#20	10.82	0.08	0.01	0.67	0.03	−0.05	3.26
	RUN_DO#100	7.51	0.09	0.02	0.89	0.02	−0.03	3.01
	DVDTZ_DO#40	6.94	0.04	0.18	−0.14	0.03	−0.07	3.96
	DTDTZ_DO#40	5.21	0.15	0.07	0.43	0.02	−0.15	3.49
	DUDTZ_DO#40	5.06	0.15	0.37	0.34	0.09	−0.34	3.38
	DKZMH_DO#30	5.00	0.03	−0.67	−1.26	0.04	0.23	5.64
	DCDTZ_DO#40	4.26	0.04	0.02	0.07	0.04	0.03	3.81
	RUN_DO#50	4.17	0.09	0.01	0.83	0.04	−0.09	3.11
	RUN_DO#70	3.84	0.07	0.02	0.83	0.07	−0.07	3.08
	WCONT_DO#30	3.67	0.03	0.00	−0.41	0.19	0.32	3.87
art	scanreco_0	99.92	0.01	0.26	0.18	0.00	0.25	3.31
equake	smvp-#0	65.57	0.05	0.11	0.76	0.01	0.47	2.61
	main-#3	31.61	0.01	0.13	0.34	0.00	0.55	2.97
fma3d	platq_do#2	76.62	0.01	0.00	0.78	0.00	0.27	2.94
	solve_do#6	11.83	0.01	0.00	1.44	0.03	0.26	2.27
	solve_do#4	5.23	0.01	0.07	2.44	0.00	0.42	1.06
	solve_do#2	2.85	0.07	0.00	2.50	0.02	0.08	1.34
gafort	shuffle-do#10	34.89	0.06	0.00	1.70	0.01	−0.04	2.26
	gafortran-do#45	26.35	0.05	0.03	0.41	0.00	0.04	3.48
	mutate-jump	18.03	0.13	−0.01	0.88	0.00	0.40	2.61
	evalout-do#30	13.30	0.01	−0.01	0.16	0.00	0.05	3.79
	newgen-do#94	7.39	0.02	0.01	1.02	0.00	0.21	2.75
galgel	syshtN_do#1234	24.64	0.01	0.01	0.04	0.00	2.81	1.13
	sysnsn_do#123	23.74	0.01	0.03	0.20	0.00	3.16	0.61
	lapak_do#7	13.65	0.07	0.00	0.09	0.03	−0.10	3.91
	lapak_do#1	9.88	0.02	0.00	0.00	0.00	−0.02	4.00
	lapak_do#3	7.12	0.02	0.00	2.53	0.00	0.04	1.40
	lapak_do#5	5.69	0.10	0.01	1.36	0.00	0.02	2.50
	lapak_do#4	3.73	0.03	0.00	−1.33	0.00	−5.28	10.58
	lapak_do#10	3.23	0.03	0.00	0.15	0.00	−0.02	3.84
mgrid	RESID_do600	50.40	0.00	0.00	1.08	0.04	−0.07	2.94
	PSINV_do600	23.53	0.01	0.00	0.55	0.04	0.03	3.37
	RPRJ3_do100	10.24	0.02	0.00	0.67	0.05	−0.06	3.32
	INTERP_do400	5.30	0.05	0.00	1.24	0.05	0.11	2.55
	INTERP_do800	5.22	0.03	0.00	1.77	0.05	0.12	2.02
swim	CALC3_DO#300	35.03	0.00	0.00	2.47	0.01	−0.22	1.74
	CALC2_DO#200	30.68	0.00	0.00	2.36	0.01	−0.19	1.81
	CALC1_DO#100	28.59	0.01	0.00	2.75	0.01	−0.60	1.82
	SWIM_DO#400	5.61	0.01	0.12	2.50	0.00	0.09	1.29
wupwise	MULDOE_DO#1	43.01	0.00	0.00	0.13	0.01	0.07	3.79
	MULDEO_DO#1	42.83	0.00	0.00	0.08	0.01	0.06	3.85
	ZAXPY_DO#1	5.49	0.01	0.00	2.26	0.04	−0.13	1.82
	ZDOTC_DO#1	4.19	0.01	−0.01	1.84	0.01	0.00	2.15
	ZCOPY_DO#1	2.68	0.01	0.00	2.51	0.03	−0.36	1.81

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

demands on the memory bus. Therefore, we attribute the increase in memory stalls to such effects.

We found that ssor_do#3 had been hand optimized for this benchmark. In particular, loop scheduling

was applied explicitly, rather than by making use of `OMP DO` directive. In other benchmarks we will discuss experiments showing that replacing `OMP DO` constructs by explicitly scheduled code leads to improvements. Hence, the optimization applied to ssor_do#3 contributes to the good performance relative to other benchmarks.

Table 5 shows that rhs_do#3 and rhs_do#4 have negative "Load Use" components. Rhs_do#3 loses speedup due to load-imbalance and "Not Modeled" effects. Similarly rhs_do#4 also loses speedup because of "Not Modeled" component. Yet both loops speedup quite well on 4 processors.

We did not find any evidence of false-sharing in *applu*.

6.3. Apsi

Apsi is a 7,500 lines of air pollution analysis code written in Fortran. The execution time in *apsi* is distributed over several important loops as shown in Table 5. The most important loops from Table 5 are RUN_DO#60, RUN_DO#40, RUN_DO#30, RUN_DO#20, RUN_DO#100. We will refer to these loops as the RUN-loops.

The RUN-loops in *apsi* lose speedup due to the increase in memory stalls, and they are responsible for the majority of memory stalls in *apsi*. Figure 7 shows that "Load Use" and "Store Buf." are the key reasons for increase in the memory stalls. From the timer and hardware counter measurements, we found that all RUN-loops show an increase in the "Load Use" stalls along with several other important loops. The RUN-loops also show a large and almost a linear increase in the "Store Buf." stalls. We discovered that in each RUN-loop, the first-level data cache hit rate decreases from 94% to 91% for the loads, and from 90% to 85% for the stores. However, the numbers of loads and stores remain about the same between the serial and parallel versions of the program. In addition to the RUN-loops, several other loops in *apsi* also show lower cache hit rates in the parallel version. The increase in cache misses explains the increase in memory stalls.

All RUN-loops have almost 99% secondary cache hit rate. Yet there is a slight degradation of the cache hit rate in the parallel version leading to nearly 4% more secondary cache misses. The decline in the secondary cache hit rate is partly an artifact of increased invalidations and copybacks suggesting some false-sharing in these loops.

Dkzmh-do#30 is the only loop that shows superlinear speedup over 5.0. The main reasons, as found by the detailed hardware counter measurements [14], are decreases in "Load Use" stalls and "FP Dependence" stalls. We found that the first-level cache hit rate of loads decreases in the parallel version, the overall secondary cache hit rate also decreases in the parallel version, but the first-level store hit rate goes up. Overall, there is almost 12% less secondary cache misses in the parallel version. We attribute decreased "Load Use" stalls to the decline in the number of secondary cache misses.

6.4. Art

Art is the longest running code among all benchmarks on a 1-processor sequential execution, as shown in Table 4. It contains 1,300 lines of C code in the area of image recognition and neural networks. Scanreo_0 is the most important loop. It is invoked once and consumes 99% of the execution time. *Art* scales fairly well among all benchmarks. The cache hit rates, the number of instructions, and the number of memory access instructions remain nearly the same between the parallel and serial versions of the program as shown by Figs 3 and 4. However, *art* still shows some increase in the memory stalls. Figure 7 indicates that the memory stalls component is due to "Load Use" stalls. It also has a large pipeline stalls component due to the branch misprediction and the floating-point dependences.

6.5. Equake

Equake is about 1,500 lines of C code that simulates an earthquake. Smvp-#0 and main-#3 are the most important loops as shown in Table 5. The average execution time of smvp-#0 is about 1.4 seconds and of main-#3 is about 0.6 seconds on a single processor sequential execution.

Table 5 shows the speedup components of smvp-#0 and main-#3. It is evident from the table that the speedup loss in *equake* comes from memory and pipeline stalls. From Figure 4, we see that the program also has about 2.3-fold increase in the memory references, most of which comes from the loads in smvp-do#0. Main-#3 shows 1.7 times more loads in parallel. The first-level data cache hit rate in both loops goes up, but because the number of loads goes up, the absolute number of first-level cache misses increases by 56% in smvp-do#0 and 41% in main-do#3. The number of stores remains nearly unchanged between the parallel

and sequential versions. Figure 7 shows that floating-point dependences are the main reason for pipeline stalls. Finally, smvp-#0 shows slight load-imbalance.

We discovered that the increase in memory references is partly a result of using extra temporaries for address calculations by our OpenMP C compiler (GuideC). We have found that the OpenMP C compiler declares several local automatic pointer variables to hold the addresses of the multidimensional array elements. It then assigns values to the elements of the array by dereferencing the pointers. Also, because the pointers are declared per block of C code, they are not reused. Thus, there are more loads from the stack in parallel than there are in the sequential version. As an experiment we declared an explicit pointer to the `w1` array and reused it throughout the loop to access the array elements in the sequential and parallel codes. We found that the number of loads went down by nearly 47% in the parallel version. An explicit pointer eliminated many of the temporaries generated by the OpenMP C compiler.

We have observed that a code with long floating-point expressions typically has more floating-point dependence stalls in the parallel version than a code with many simpler expressions, suggesting potential improvements for compiler optimizations. As an experiment we simplified the math expressions in smvp-#0. After simplifying the math expressions, the floating-point dependence stalls declined by almost 70%. As a combined effect of removing the extra memory references and reducing the floating-point dependence stalls, the 4-processor execution time decreased by 3.5%.

Finally, smvp-#0 shows a 0.045 speedup loss because of load-imbalance. We determined that the load-imbalance is a result of uneven iteration space of the inner `while`-loop in smvp-#0. The `while` loop has 0 to 12 iterations every time it is invoked by the outer `for`-loop. Since the iterations of the outer loop are divided among multiple processors, each processor executes a different number of overall inner-loop iterations. Use of `guided` scheduling instead of simple block scheduling decreased the load-imbalance component to 0.004.

We did not find any evidence of significant false-sharing in *equake*.

6.6. *Fma3d*

Fma3d is a finite element method computer program designed to simulate the inelastic, transient dynamic response of three-dimensional solids and structures subjected to impulsively or suddenly applied loads. It contains over 60,000 lines of Fortran code. Platq_do#2, solve-do#6, solve-do#4, and solve-do#2 are the top four loops in *fma3d*. It is evident from Table 5 that all loops lose speedup due to memory stalls. We can see from Fig. 7 that the speedup loss due to "Load Use" is about 0.9 and due to "IC Miss" is about 0.3. Figures 3 and 4 reveal that the first-level data cache hit rate improves in the parallel version of the code, but the memory references and the number of instructions almost double on 4 processors. A closer examination of the loops reveals that all important loops in *fma3d* show an increase in the number of loads and stores, but platq_do#2 has by far the highest increase. The first-level data cache hit rate improves in all loops except in solve-do#4. In solve-do#4 the cache hit rate of stores drops from 99% to 91%. Due to the increased number of loads and stores, there is a 48% increase in the absolute number of load misses and a 9% increase in the absolute number of store misses in platq_do#2. Lastly, platq_do#2 experiences a reduction in the instruction cache hit rate from 90% to 84%.

A study of platq_do#2 shows that it contains 9 conditional subroutine calls. These calls inside the loop perform the majority of work. All subroutines inside the loop use a set of 70 variables that are declared in a `THREADPRIVATE` common block. We discovered that the Fortran compiler makes stores to the `THREADPRIVATE` variables "volatile." Since the volatile variables must be loaded from the memory each time they are needed, they cannot be allocated in registers. After we declared the `THREADPRIVATE` variables as `PRIVATE` variables, the number of memory references dropped by 18% and the overall speedup rose to 2.83. Also, in order to avoid the cost of `OMP DO`, we manually scheduled the loop along with the privatized common block variables. We found that the overall speedup climbed to 3.09 suggesting that the `OMP DO` construct is implemented rather inefficiently. As mentioned in Section 6.2, the main cost of `OMP DO` construct comes from the additional loop body subroutine call and the related stack activity. Finally, we attribute "IC Miss." stalls to the decrease in the instruction cache hit rate in platq_do#2.

We found evidence of significant data sharing in platq_do#2 and solve-do#4 by measuring the invalidations and copybacks. We measured that the invalidations and the copybacks follow each other closely and scale with the number of processors. Also, they are responsible for most of the misses in the secondary cache. We attribute this effect to the "+" `REDUCTION`

of `ENG1` and `ENG2` variables in solve-do#4 and MIN/MAX `REDUCTION` of `TIME_STEP_MIN` and `TIME_STEP_MAX` in plaq_do#2.

6.7. Gafort

Unlike other SPEC OMPM2001 applications, *gafort* is an integer application. It is written in Fortran and contains about 1,500 lines of code in the area of genetic algorithms. Shuffle-do#10, gafort-do#45, mutate-jump, evalout-do#30, and newgen-do#94 are the most important loops in *gafort*. From the overall speedup components in Figs 6 and 7 we see that the increase in memory stalls is the key reason for speedup loss in *gafort*. Specifically, the "Load Use" and "Load RAW" stalls contribute to the memory stalls. Figures 3 and 4 show that the data cache hit rates do not increase significantly in the parallel version, and there is about 17% increase in the total memory references, of which about 13% comes from mutate-jump and 3% from newgen-do#94. From Table 5, we also find that evalout-do#30 and gafort-do#45 speedup quite well, and the top three loops have slight load-imbalance.

We discovered that the primary reason for increase in memory references in mutate-jump and newgen-do#4 is the `OMP DO` construct, just as it was in *fma3d*. As an experiment, we removed `OMP DO` and manually scheduled the loop iterations in mutate-jump[3] and newgen-do#4. We found that the number of loads dropped by 20% in mutate-jump and by 40% in newgen-do#45 bringing them closer to the sequential count of loads. The number of instructions also went down proportionally, and the overall speedup rose from 2.56 to 2.88.

Shuffle-do#10 is the most time consuming loop in *gafort*. It is responsible for about 36% of the overall execution time. It is also responsible for the largest increase in the "Load Use" and "Store Buf." related memory stalls. From the hardware counter measurements, we determined that even though Shuffle-do#10 shows about 8% more memory references in the sequential version than the parallel one, the secondary cache misses decrease from 94% in sequential to 91.8% in parallel, leading to about 9% increase in the absolute number of secondary cache misses in the parallel version. Thus, even if the first-level data cache hit rate does not change significantly in shuffle-do#10, the more expensive secondary cache misses lead to the memory stalls. Also, shuffle-do#10 is memory bound, because it performs 3 array copies (shuffles), which result in about 1GB of loads and stores during every invocation of the loop. On the other hand, there are no computational steps. Thus, shuffle-do#10 has nearly zero computation-to-memory-access ratio. We find that the memory bandwidth requirement increases from 140 MB/s to 300 MB/s on our machine. As explained in Section 6.1, a loop with low computation-to-memory-access ratio puts more demands on the memory bus. Therefore, we attribute the limited scalability of shuffle-do#10 in part to the increased demands on the memory bus.

Shuffle-do#10 uses OpenMP locks inside the loop. We measured that about 5% of the overall execution time, is spent in the locks on our platform. Unlike the sharing of lock variables in *ammp*, we did not see excessive sharing of lock variables in *gafort*, mainly because there is one lock per row of the `iparent` array. The chance of two processors grabbing the same lock before swapping the rows is 1 in 400,000.

Load-imbalance in *gafort* is inherent in the algorithm because of the random number generator. In mutate-jump and gafort-do#45, for example, there are several conditional paths, which are either taken or not taken depending on a random number. Both of these loops use `guided` scheduling to mitigate the effects of load-imbalance. As a result, the load-imbalance speedup component is minor, as shown in Fig. 6.

We did not find any evidence of noteworthy false-sharing in *gafort*.

6.8. Galgel

Galgel is a fluid dynamics code written in Fortran. It contains about 15,300 lines. The most important parallel regions in *galgel* are syshtN_do#1234, sysnsn_do#123, and lapak_do#7. In addition to lapak_do#7, several other loops in the LAPACK routines are also important. However, given their small average execution times, their importance comes from the large invocation counts. Many of them scale quite well with respect to the number of processors. Table 5 shows that lapak_do#4 has a superlinear speedup, and the top two loops have large "Not Modeled" components.

SyshtN_do#1234 and sysnsn_do#123 are very similar in structure. There are 4 parallel loops in syshtN_do#1234 and 3 parallel loops in sysnsn_do#123, where the first loops are the most time-consuming ones. The real workhorses in both regions are the matrix multiply, transpose, and dot product intrinsics. We

[3] We used the simple `guided` scheduling algorithm described in [6].

measured that both regions have an average execution time of 10 seconds in a sequential run. However, in a 1-processor parallel execution the average execution time degrades to 32 seconds for syshtN_do#1234 and 55 seconds for sysnsn_do#123. Therefore, both loops demonstrate very poor scalability on our platform as shown in Table 5. We also discovered that the average execution time scales almost perfectly on 2 and 4 processors with respect to the 1-processor *parallel* execution time, suggesting a high parallelization overhead. The parallelization overhead here refers to the fork-join overhead. However, Fig. 5 shows that the fork-join time is not a significant part of the overall execution time. Instead we found that the matrix manipulation functions are the culprits for the increase in the execution time in parallel. The assembly code inspection of these functions revealed that the matrix manipulation routines (e.g. `MATMUL` or `dgemm`) are never called directly but are rather inlined in both the sequential and parallel versions. The difference in the parallel version is that we also found calls to `malloc` and `free` inside the loop body subroutines. These calls seem to allocate temporary arrays before the matrix manipulation occurs. As explained in Section 5.1, the dynamically allocated arrays introduce a significant overhead in the parallel version. Therefore, we attribute the increase in the average execution time to the dynamic allocation of arrays in the parallel version in syshtN_do#1234 and sysnsn_do#123. Since we do not have an explicit category for such an overhead, *galgel* has a large "Not Modeled" component.

Figures 6 and 7 show that the memory stalls due to "Load Use" and "Store Buf." are important reasons for the speedup loss in *galgel*. We found that lapak_do#3 and lapak_do#5 are responsible for the increase in "Load Use" stalls. Lapak_do#3 also shows a large increase in "Store Buf." stalls. Lapak_do#5 shows only 1% reduction in the secondary cache miss rate in parallel, almost all of which comes from the false-sharing of array `C`. Similarly, lapak_do#3 also shows about 3% reduction in the secondary cache miss rate. We attribute the increase in memory stalls in these loops to the reduction in secondary cache hit rate.

6.9. Mgrid

Mgrid is a 500 line multigrid solver code written in Fortran. RESID_do600, PSINV_do600, and RPRJ3_do100 are the three most important loops as shown by Table 5. We can see from Figs 6 and 7 that the memory stalls is the key reason for speedup loss in *mgrid*. Also, the increase in memory stalls comes from the increase in "Load Use," "Store Buf.," and "Load RAW" related stalls. From the hardware counter measurements, we discovered that RESID_do600 is responsible for the largest increase in all three areas, specifically in "Store Buf." stalls. PSINV_do600 and RPRJ3_do100 also show a substantial increase in "Load Use" stalls.

The first-level data cache hit rate of the loads in RESID_do600 and PSINV_do600 is 67% and is unchanged in the parallel version. For RPRJ3_do100, the hit rate is 77% and also remains constant between the sequential and parallel versions. While the hit rate of stores is over 80% for PSINV_do600 and RPRJ3_do100, it is only 58% for RESID_do600. These hit rates also remain constant between the serial and parallel versions. The number of loads and stores remain nearly the same from the sequential to the parallel versions for all three loops.

From the code inspection, we find that RESID_do600 and PSINV_do600 both perform stencil computations on an entire 3 dimensional array during each invocation. The array sizes range from $4 \times 4 \times 4$ to $256 \times 256 \times 256$ in the increments of powers of two. The array access patterns in the loops lead to good spatial and temporal locality in the sequential and parallel versions. Also, there is no data sharing among processors, because the arrays are an exact multiple of the number of processors (4) on our system. We found that the bandwidth requirement grows from 33 MB/sec in a sequential run to 95 MB/sec in a parallel run on 4 processors. Although the maximum bandwidth of the interconnect is significantly higher, this increase may contribute to the higher memory stalls in the parallel execution.

Finally, we did not find any evidence of false-sharing in *mgrid*.

6.10. Swim

Swim is a shallow water modeling program. It contains about 400 lines of Fortran code. CALC3_DO#300, CALC2_DO#200, and CALC1_DO#100 are the three most important loops in *swim*. We will refer to them as the CALC-loops from here on. The average execution time of each loop is about 3 seconds. From the performance point of view SWIM_DO#400 is also an important loop.

From Figs 6 and 7 we can see that almost all of the speedup loss in *swim* is due to the increase in memory stalls. "Store Buf." is the largest memory stalls component followed by the "Load Use" and

the "Load RAW" components. CALC3_DO#300 and SWIM_DO#400 show an increase in memory references. CALC3_DO#300 has 1.33 times more loads in the parallel version, but the number of stores is unchanged. In SWIM_DO#400, there are 2 times more loads and several hundred times more stores. While all four loops show substantial increase in the "Store Buf." and "Load RAW" stalls, only the CALC-loops are responsible for the majority of increase in "Load Use" stalls.

In general, *swim* suffers from poor cache performance. The first-level data cache hit rate is about 50% for the loads in all loops. The hit rate for the stores is almost 99% in CALC3_DO#300, about 50% for SWIM_DO#400, and below 1% in the remaining two CALC-loops. We found that the reason for the poor cache performance is lack of temporal locality in the `CU`, `CV`, `Z`, `H`, `PNEW`, `UNEW`, `VNEW`, `P`, `U`, `V`, `POLD`, `UOLD`, and `VOLD` arrays with respect to the outermost time-step loop. `CU`, `CV`, and `Z` act as intermediate arrays in *swim*. Every time-step the new arrays (e.g. `PNEW`) are written in two steps: first the intermediate arrays are updated, and then the new arrays are written. The size of each array is approximately 110 MB. An examination of the code reveals that these arrays display good spatial locality, but the use of intermediate arrays causes the final arrays to get replaced in the cache, before the final arrays could be reused. In the same way, the final arrays replace the intermediate ones before the next iteration of the time-step loop. Thus, *swim* experiences thrashing in the caches.

In order to prevent thrashing we conducted an experiment where we completely removed any references to the intermediate arrays by performing aggressive expression propagation on the CALC-loops. Also, we coalesced the CALC-loops in a single loop for a larger granularity of parallelism. We found that the overall data cache hit rate improved from 49% to 75% for the first-level cache and from 87% to 91% for the secondary cache when comparing the parallel versions with and without the transformation. The sequential code with the transformation, however, showed a 10% increase in execution time when compared with the sequential version without it. Also, we found more floating-point dependence related stalls with the parallel transformed version. The increase in the floating-point dependences is linked to the increased complexity of math expressions in the transformed loop. Nonetheless, the 4-processor execution time with the expression propagation improved by almost 25% and the overall speedup also rose to 2.45.

In addition to thrashing, *swim* also has a low computation-to-memory-access ratio in parallel. Therefore, it exerts more demands on the memory bus than other benchmarks. One indication of the increased demands is the increased bandwidth requirements in parallel. For example, in CALC1_DO#100 the bandwidth requirement increases from about 250 MB/sec to 447 MB/sec. The increased demand on the memory bus combined with poor temporal locality leads to more memory stall in parallel, which severely limit *swim*'s scalability on our platform.

6.11. Wupwise

Wupwise is a Fortran code of about 2,200 lines in the area of quantum chromodynamics. Even though MULDOE_DO#1 and MULDEO_DO#1 are the most important loops in *wupwise*, the speedup loss is mainly due to ZAXPY_DO#1, ZDOTC_DO#1, and ZCOPY_DO#1. These loops are similar to BLAS [2] routines. They operate on vectors and matrices of complex numbers. As shown in Table 5, MULDOE_DO#1 and MULDEO_DO#1 scale quite well on 4 processors but not ZAXPY_DO#1, ZDOTC_DO#1, and ZCOPY_DO#1.

Figures 6 and 7 show that the speedup loss in *wupwise* is a result of memory stalls, mainly "Load Use" and "Store Buf." stalls. MULDOE_DO#1 and MULDEO_DO#1 are responsible for only a small percentage of the memory stalls. A vast majority of the memory stalls result from ZAXPY_DO#1, ZDOTC_DO#1, and ZCOPY_DO#1. Although these loops demonstrate poor cache performance, ZDOTC_DO#1 is the only loop where the first-level data cache hit rate of stores decreases to 33% from 50%, and the number of loads increases by 23%. In the remaining two loops, the cache hit rates and the number of loads and stores remain unchanged. Hence, we attribute the increase in the memory stalls in ZDOTC_DO#1 to the combined effect of the lowered cache hit rate and the increased loads. We discovered that in ZCOPY_DO#1, the secondary cache hit rate drops from 79.7% to 76.3% and the number of references to the secondary cache rise up by almost 16%. Similarly, ZAXPY_DO#1 also reports a drop in the secondary cache hit rate from 85% to 81% and a rise in the number of references to the secondary cache by 7%. Therefore, we attribute the increased memory stalls in these two loops to the poorer performance of the secondary cache in the parallel version.

We did not see significant false-sharing in *wupwise*.

7. Summary

Most benchmarks in the SPEC OMP suite adhere to the golden rule that over 90% of program execution time is spent in less than 10% of code. Majority of the programs spend most of the execution time in 5 loops or less. Also, these programs are highly parallel, and they run for a long time, even in parallel.

We have presented a Quantitative Model that explains gap between ideal speedup and realized speedup into the speedup components. We also showed ways of measuring, deriving, and computing the speedup components. We derived two basic formulas that quantify the speedup loss: a formula for a fully parallel region and a formula for a partially parallel region that contains serial sections. Using the Quantitative Model, we explained the performance of SPEC OMP benchmarks. We presented the overall characteristics of benchmarks followed by loop-by-loop analysis. We focused our explanations on the most important loops in each benchmark and explained the reasons for their speedup loss.

Overall we discovered that memory system related stalls are the biggest reason for speedup loss. The memory stalls increase in parallel. The stalls due to loads are dominant compared to stores. However, stalls due to a full store buffer are also important in several codes. We found that there are three primary reasons for the memory stalls in the SPEC OMP benchmarks:

- Increased memory reference instructions: Increase in memory references results mainly due to an increase in the number of loads. In all benchmarks there are considerably more loads than stores. We found that even if the cache hit rate increases in many benchmarks, more memory references lead to a higher absolute number of cache misses. The benchmarks that exhibit such behaviors are *ammp*, *equake*, *fma3d*, and *gafort*. We found more memory references in parallel without counting references at the implicit barriers. We could not find any specific code patterns that lead to increased memory references in parallel. The reasons are mostly inefficient compilation and run-time management of some OpenMP directives such as `OMP DO` and `THREADPRIVATE`.
- Decreased data cache hit rate: *Apsi* reported a lower cache hit rate in parallel. False-sharing was partly responsible for the reduction in the cache hit rate. However, in general false-sharing is really a minor reason for the cache misses. *Wupwise* also reported slightly lower cache hit rates in the BLAS loops. The memory stalls due to a full store buffer is a secondary effect of increased cache misses.
- Increased demands on the memory bus: Increased demands on the memory bus in parallel is an artifact of lower computation-to-memory-access ratios in the benchmarks, such as *applu*, *gafort*, *mgrid*, and *swim*.

True-sharing is important in *ammp* and *fma3d*. In *ammp* the source of true-sharing is the locks, and in *fma3d* it is the `REDUCTION` clause.

Floating-point dependences is the most important reason for pipeline stalls in parallel in the SPEC OMP benchmarks. The higher complexity of floating-point math operations seems to lead to higher floating-point dependence stalls in parallel. Loops in *equake* and *swim*[4] exemplify such a pattern. Nevertheless, considering the fact that the SPEC OMP codes represent scientific applications' domain, whose execution times are dominated by the floating-point and memory access operations, we found that the efficiency of memory access operations is far more important than the efficiency of floating-point operations. The diminished importance of floating-point operations is a result of on-chip, efficiently pipelined floating-point unit. *Art* is the only benchmark whose scalability is limited by the pipeline stalls related to branch misprediction.

Because the SPEC OMP codes are highly parallel, serial sections of the codes do not impact the parallel performance on our 4-processor system. Finally, the performance of SPEC OMP benchmarks is not significantly limited by fork-join and load-imbalance overheads. The fork-join overhead, however, can become significant if entering a parallel region involves expensive operations such as memory allocation as illustrated by *apsi*. The overhead can be worsened by a large invocation count of the parallel region. Also, we expect the fork-join overhead to scale much more rapidly as the number of processors increases. While we found slight load-imbalance in some instances, in general it does not hamper the performance of SPEC OMP programs on our system.

8. Conclusions

We have presented detailed analysis of the SPEC OMP benchmarks in Section 6. Our goal was to study a set of modern scientific shared-address-space (SAS) parallel programs. The SPEC OMP benchmarks pre-

[4] In *Swim*, the math expressions become more complex after applying aggressive expression propagation optimization.

sented that opportunity to us. There are three distinguishing aspects to our study: We have chosen a modern multiprocessor platform on which to study the programs, the programs are parallelized with the OpenMP directives, which is a new standardized way of writing SAS programs, and the large size of data sets, which results in long execution times even on 4 processors. We have learned several lessons about the performance of such programs.

The parallelization overhead in terms of fork-join time is only a minor factor in determining the speedup of long running codes, because the cost of fork-join gets amortized over a long period of time. That is, the efficiency of `OMP PARALLEL/OMP END PARALLEL` directives without the data environment clauses is of little concern in terms of performance. However, we found that an efficient implementation of the `OMP DO` construct (worksharing construct) is important to the performance of loop-based programs. For example, we found differences between our Fortran and OpenMP C compilers in how each implements `OMP DO` directive: the Fortran compiler generates a subroutine from the `OMP DO` loop body, whereas the C compiler does not. Therefore, the Fortran compiler incurs more overhead at runtime. Also, our OpenMP C compiler automatically coalesces adjacent parallel regions into a single subroutine, thereby reducing subroutine calls.

The importance of `OMP PARALLEL` and `OMP DO` in terms of performance is linked to the efficiency of data environment clauses. Efficient code generation of `PRIVATE`, `SHARED`, and `THREADPRIVATE` is critical to the performance of OpenMP programs, since they are the most frequently used data environment clauses. In particular, efficient ways of allocating, accessing, and deallocating `THREADPRIVATE` and `SHARED` variables and `PRIVATE` arrays is important. Individual heaps for the threads can help improve the efficiency of dynamic memory allocation in parallel for the `PRIVATE` arrays, which is a part of the fork-join overhead. As the number of processors increases, serialization to allocate from the same process heap can drastically increase the fork-join overhead. Similarly, repeated address calculations to access `SHARED` and `THREADPRIVATE` variables can be avoided by using more registers and more efficient register allocation algorithms.

The OpenMP locks and `OMP CRITICAL` introduce two overheads in a parallel program, which are absent in a sequential one: an overhead of executing extra instructions while spin waiting and an overhead of acquiring the lock. The overhead of executing extra instructions is paid in a parallel program if the processors end up spin waiting for the lock. However, the overhead of acquiring a lock depends on the amount of lock contention and the implementation of locks. Therefore, an implementation that reduces contention for the locks is important [1,4]. The lock variables are also a source of true-sharing. True-sharing can lead to increased invalidations and copybacks, which in turn may lead to increased cache misses. For example, the lock variable in *ammp* experiences true-sharing. The `REDUCTION` clause is much more likely to be a source of false and true-sharing than any other OpenMP clause for three reasons: (1) all processors update the reduction variable, (2) some implementations of the `REDUCTION` clause use implicit locks, and (3) reduction operations are popular in parallel programming.

The presented study is one step towards understanding realistic, OpenMP shared-address-space parallel programs on modern SMP systems. While the used programs are highly parallel, their efficiency, even on four processors, is below expectation. In addition to a few intrinsic reasons, we have found room for improvements of compilers and libraries. In order to find and evaluate remedies, more such studies are necessary, quantifying application performance on other architectures, larger numbers of processors, and future generations of software systems.

Acknowledgement

This material is based upon work supported in part by the National Science Foundation under Grant No. 9974976-EIA and 0103582-EIA.

References

[1] A. Kagi, D. Burger and J. Goodman, Efficient Synchronization: Let Them Eat QOLB, *Proc. 24th International Symposium on Computer Architecture* (*ISCA 24*), June, 1997.

[2] C.L. Lawson, R.J. Hanson, D.R. Kincaid and F.T. Krogh, Basic Linear Algebra Subprograms for Fortran usage, *ACM Transactions on Mathematical Software* **5** (Sept, 1979), 308–323.

[3] E.L. Boyd, W. Azeem, H.-H. Lee, T.-P. Shih, S.-H. Hung and E.S. Davidson, A Hierarchical Approach to Modeling and Improving the Performance of Scientific Applications on the KSR1, in: *Proceedings of the 1994 International Conference on Parallel Processing* **3** (August, 1994), 188–192.

[4] J.M. Mellor-Crummey and M.L. Scott, Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, *ACM Transactions on Computer Systems*, 1991.

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

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

[7] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonal and R. Menon *Parallel Programming in OpenMP*. San Diago, California.: Academic Press, 2001.

[8] R. P. Garg and I. Sharapov, *Techniques for Optimizing Applications: High Performance Computing*. Prentice Hall, first ed., 2002.

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

[10] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh and A. Gupta, The SPLASH-2 Programs: Characterization and Methodological Considerations, in: *Proceedings of the 22nd International Symposium on Computer Architecture*, 1995, pp. 24–36.

[11] Sun Microsystems Inc., *Analyzing Program Performance with Sun Workshop*, July 2001.

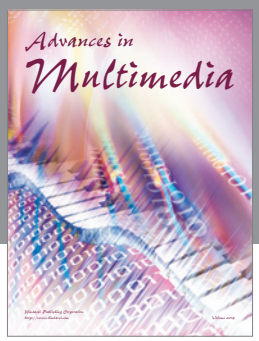
[12] Sun Microsystems Inc., *UltraSPARC User's Manual*, 1997.

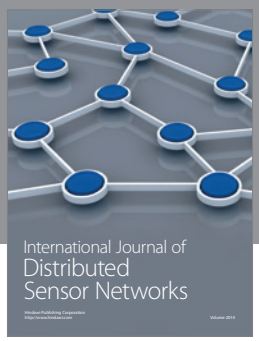
[13] S.W. Kim and R. Eigenmann, Where Does the Speedup Go: Quantitative Modeling of Performance Losses in Shared-Memory Programs, *Parallel Processing Letters* **10**(2 and 3), (2000), 227–238.

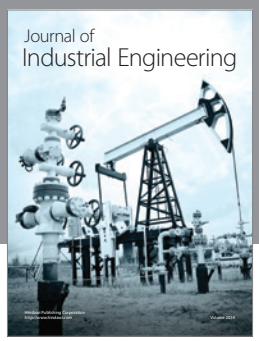
[14] Vishal Aslot, Performance Characterization of the SPEC OMP Benchmarks, Master's thesis, Purdue University, 2002.

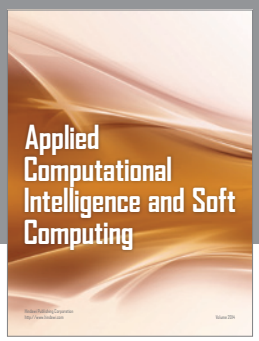
[15] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W.B. Jones and B. Parady, Specomp: A new benchmark suite for measuring parallel computer performance, in: *OpenMP Shared-Memory Parallel Programming*, Lecture Notes in Computer Science #2104, (Springer Verlag, Heidelberg, Germany), July, 2001, pp. 1–10.

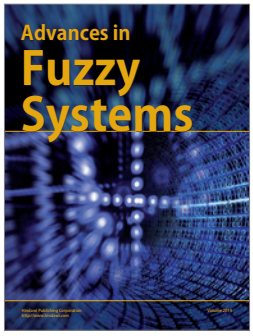
[16] V. Pai, P. Rangnathan, S. Adve and T. Harton, An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors, *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems* (*ASPLOS-VII*), October, 1996, pp. 12–23.

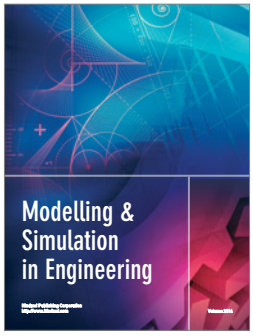


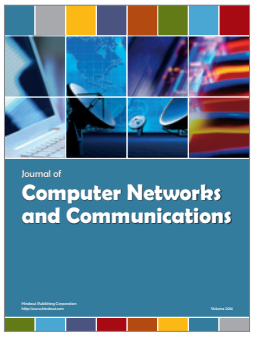


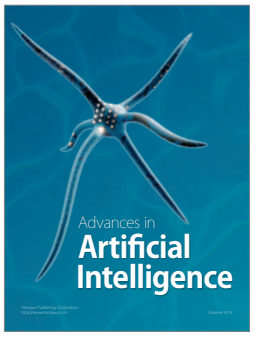
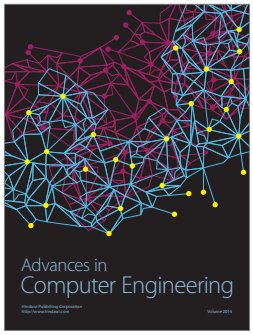
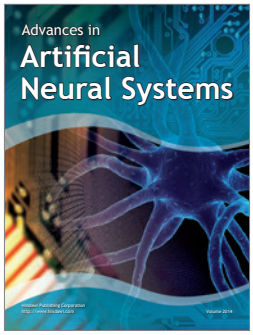
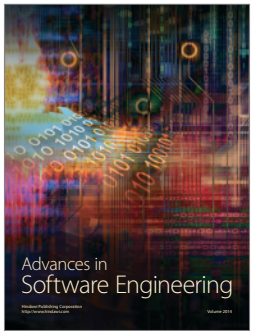
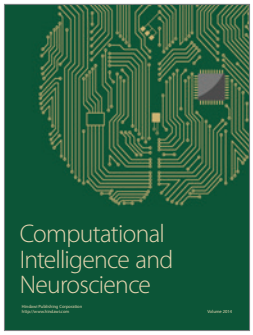










Hindawi

Submit your manuscripts at
http://www.hindawi.com