

# OMP AMMP analysis with Sun ONE Studio 8

Brian J. N. Wylie & Darryl J. Gove  
Sun Microsystems, Inc.

*The Sun ONE Studio 8 release of compilers, libraries and tools incorporate new and improved functionality to help understand the execution of multithreaded (and multiprocess) applications and the analysis of memory and cache performance. One of the applications from the SPEC OMP(M) benchmark suite, AMMP, is studied to demonstrate how data-oriented analysis and a customisable chart of profile and trace events complement existent functionality to direct the tuning of applications to execute more efficiently on modern multiprocessor systems.*

## Introduction

Sun WorkShop™ compilers' and performance tools' support for profiling OpenMP applications [1], has been improved in the latest release, Sun™ ONE Studio 8 [2,3,4]. The performance tools' Functions, Callers-Callees, annotated Source and Disassembly displays are largely unchanged from previous WorkShop versions, though the Analyzer GUI has been restyled and integrates new displays. A graphical Timeline has been added to provide customisable views of the collected profile and trace event data, charting events (including their callstacks) by their thread, lightweight process (LWP) or CPU versus time to reveal temporal distribution and relations. Data-oriented analyses of UltraSPARC® hardware counter events which relate to data references from memory and caches are also being developed to provide insight into this often significant aspect of application performance which complements traditional code-oriented analyses [5].

The AMMP application from the SPEC OMP(M) benchmark suite is studied to demonstrate use of the latest compilers and performance tools in the identification and remedying of execution performance inefficiencies.

## Experiment setup and collection

AMMP is a molecular mechanics, dynamics and modelling program developed by Robert W. Harrison [6] which has been explicitly parallelised using OpenMP directives and functions to run on any number of processors as part of the SPEC OMP(M) benchmark suite (332.ammmp\_m). It consists of 31 source files, which were compiled using the Sun ONE Studio 8 C compiler (v5.5) with the usual “base” (i.e., `-fast -xopenmp`) and 64-bit address (i.e., `-xarch=v9`) build options augmented with the build options for debugging and hardware counter profiling support, i.e., `-g -xhwcprof -xdebugformat=dwarf`.

An experiment was collected to investigate its memory performance on a Sun Fire™ 6800 system with 900 MHz UltraSPARC-III Cu processors (each with 8MB of external cache, E\$) and 48GB of RAM [2], running Solaris 9 update 2, having set environment variables specifying the execution should create and bind threads for 12 (of the systems' 24) processors:

```
setenv OMP_NUM_THREADS 12
setenv MT_BIND_PROCESSOR TRUE
collect -p on -h +ecstall,on,+ecrm,on -S off ammmp ...
```

This experiment includes both clock and hardware counter profile data for E\$ stall cycles and E\$ read misses (the former converted to seconds in the analyses based on the processor clock frequency) using the default resolution settings, and disabled periodic sampling. Of particular note, apropos backtracking search for event load instructions and their effective data addresses was requested (via the '+' preceding the counter names) for the hardware counter profiling. Binding OpenMP threads to processors [7] results in a one-to-one association between user-level threads and the LWPs scheduled by the Solaris™ Operating System, which improves performance through better cache locality and reduced remote memory access.

## Experiment analysis

Analysis of collected experiments can either be done interactively with the *analyzer* GUI or *er\_print* command-line report generator: examples of both are demonstrated in the following study.

### Functions

The functions analysis in *Figure 1* shows that while almost a quarter of the *ammp* User CPU time is associated with the MP doall from line 596 in *mm\_fv\_update\_nonbon*, this parallel function accounts for almost half of the E\$ read misses and nearly two-thirds of the E\$ stall time. The OpenMP library function *omp\_set\_lock\_* also has many E\$ read misses, with most of the rest attributed to the MP doall from line 422 in *f\_nonbon*. The E\$ stall time in these functions is also seen to be sizable fractions of their User CPU time, suggesting that memory and cache performance is a dominant factor in this application's execution. (The two functions with the most User CPU time, *\_\_mt\_WaitForWork\_* and *\_\_mt\_EndOfTask\_Barrier\_*, are part of the Sun compilers' implementation of multitasking support in a specialised library, *libmtnsk*, [2]: by default these functions “busy wait” in a spin loop, and therefore don't constitute useful work nor generate non-negligible numbers of cache misses.)

Excl. User CPU	Excl. E\$	Excl. E\$	Name			
sec.	Stall Cycles	Read Misses	sec.	%		
1189.942	100.0	203.294	100.0	741385789	100.0	<Total>
<b>274.402</b>	<b>23.1</b>	<b>129.951</b>	<b>63.9</b>	<b>359810812</b>	<b>48.5</b>	<b>mm_fv_update_nonbon -- MP doall from line 596</b>
<b>100.510</b>	<b>8.4</b>	<b>28.982</b>	<b>14.3</b>	<b>234507035</b>	<b>31.6</b>	<b>omp_set_lock_</b>
<b>34.504</b>	<b>2.9</b>	<b>21.767</b>	<b>10.7</b>	<b>68402052</b>	<b>9.2</b>	<b>f_nonbon -- MP doall from line 422</b>
9.226	0.8	5.168	2.5	17600528	2.4	tpac
5.464	0.5	3.560	1.8	14500435	2.0	a_next
2.972	0.2	2.054	1.0	8000240	1.1	f_bond
3.693	0.3	2.017	1.0	7900237	1.1	f_angle
2.382	0.2	1.697	0.8	6200186	0.8	fv_update_nonbon
1.401	0.1	1.106	0.5	3500105	0.5	a_inactive_f_zero
2.452	0.2	1.038	0.5	3100093	0.4	mm_fv_update_nonbon -- MP doall from line 356
3.232	0.3	0.887	0.4	3200096	0.4	f_torsion
1.211	0.1	0.710	0.3	3100093	0.4	f_box
435.014	36.6	0.551	0.3	500015	0.1	__mt_WaitForWork_
0.020	0.0	0.506	0.2	800024	0.1	__mt_run_my_job_
0.010	0.0	0.470	0.2	1600048	0.2	__mt_SlaveFunction_
278.255	23.4	0.410	0.2	1100044	0.1	__mt_EndOfTask_Barrier_
0.060	0.0	0.397	0.2	1000030	0.1	atom -- OMP parallel region from line 119
1.021	0.1	0.363	0.2	700021	0.1	a_f_zero
0.	0.	0.325	0.2	1127199	0.2	collector_record_counters
0.751	0.1	0.308	0.2	1100033	0.1	f_hybrid
1.781	0.1	0.206	0.1	600018	0.1	mm_fv_update_nonbon -- MP doall from line 561
0.	0.	0.162	0.1	500015	0.1	atom -- OMP parallel region from line 94
0.650	0.1	0.138	0.1	800024	0.1	__mt_MasterFunction_
0.140	0.0	0.104	0.1	400012	0.1	mm_fv_update_nonbon
...						

Figure 1: Functions ranked by E\$ stall time

### Callers-Callees

Analysis of the caller and callee functions of the most costly parallel function, that of the MP doall from line 596 of *mm\_fv\_update\_nonbon*, shows that it was only called from *mm\_fv\_update\_nonbon* (as expected) which is therefore attributed 100% of its metric values, which are also attributed partially to itself and the four functions it called. Of these callees, significant User CPU time is attributed to *\_\_mt\_EndOfTask\_Barrier\_*, and rather lesser amounts to *omp\_set\_lock\_* and *omp\_set\_unlock\_*, while only *omp\_set\_lock\_* is attributed a significant amount of E\$ stall time. (The other callee *\_\_mt\_get\_thread\_num\_* is inconsequential.)

Attr. User CPU	Excl. User CPU	Incl. User CPU	Attr. E\$ Stall Cycles	Name				
sec.	sec.	sec.	sec.					
604.883	100.0	0.140	0.0	610.657	51.3	150.516	100.0	mm_fv_update_nonbon
<b>274.402</b>	<b>45.4</b>	<b>274.402</b>	<b>23.1</b>	<b>604.883</b>	<b>50.8</b>	<b>129.951</b>	<b>86.3</b>	<b>*mm_fv_update_nonbon -- MP doall from line 596</b>
248.324	41.1	278.255	23.4	278.255	23.4	0.090	0.1	__mt_EndOfTask_Barrier_
76.243	12.6	100.510	8.4	100.510	8.4	20.466	13.6	omp_set_lock_
5.794	1.0	23.516	2.0	23.516	2.0	0.002	0.0	omp_unset_lock_
0.120	0.0	0.050	0.0	0.120	0.0	0.007	0.0	__mt_get_thread_num_

Figure 2: Callers-Callees of MP doall from line 596 of *mm\_fv\_update\_nonbon*

Re-focussing on *\_\_mt\_EndOfTask\_Barrier\_* shows that it calls no other functions, but that it's “called” from eight parallel functions, with almost 90% of its time attributed to the MP doall from line 596 of

mm\_fv\_update\_nonbon and some 6% attributed to the MP doall from line 422 of f\_nonbon. (Other MP doalls and OMP parallel regions in these functions and in atom are attributed the remaining 4% between them.)

Attr. User CPU	Excl. User CPU	Incl. User CPU	Name			
sec. %	sec. %	sec. %				
248.324	89.2	274.402	23.1	604.883	50.8	mm_fv_update_nonbon -- MP doall from line 596
18.263	6.6	34.504	2.9	94.756	8.0	f_nonbon -- MP doall from line 422
4.413	1.6	0.010	0.0	4.433	0.4	atom -- OMP parallel region from line 87
4.133	1.5	0.060	0.0	4.193	0.4	atom -- OMP parallel region from line 119
1.891	0.7	0.	0.	1.891	0.2	atom -- OMP parallel region from line 94
0.620	0.2	1.781	0.1	2.402	0.2	mm_fv_update_nonbon -- MP doall from line 561
0.500	0.2	2.452	0.2	2.952	0.2	mm_fv_update_nonbon -- MP doall from line 356
0.110	0.0	0.	0.	95.227	8.0	f_nonbon -- OMP parallel region from line 418
<b>278.255</b>	<b>100.0</b>	<b>278.255</b>	<b>23.4</b>	<b>278.255</b>	<b>23.4</b>	<b>*__mt_EndOfTask_Barrier_</b>

Figure 3: Callers-Callees of \_\_mt\_EndOfTask\_Barrier\_ ranked by attributed User CPU time

\_\_mt\_EndOfTaskBarrier\_ is not in fact called by any of these functions, but used internally by libmtsk for barrier synchronisations at the end of parallel loops and regions (Figure 4), however, its incorporation in recorded callstacks allows the barrier wait time to be attributed to parallel loops and regions as a means to determine their importance and efficiency: in the case of ammp, the MP doall from line 596 of mm\_fv\_update\_nonbon is particularly inefficient and contributes almost 250 seconds of time wasted by threads waiting at the final barrier.

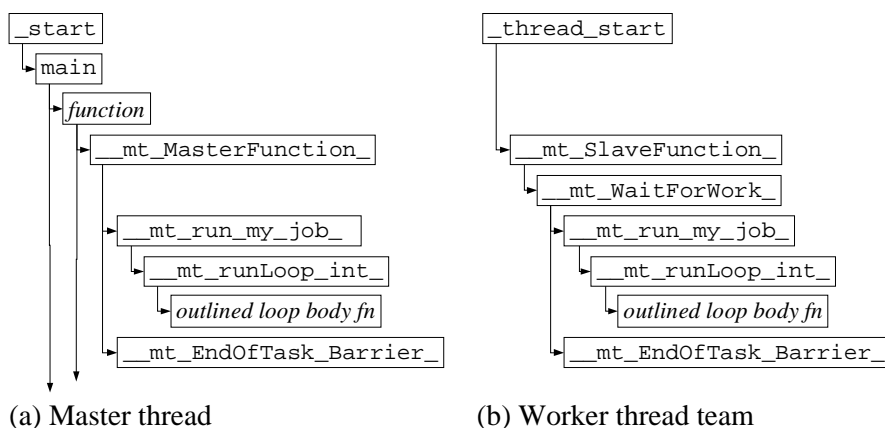


Figure 4: Multitasking library (libmtsk) implementation of parallel loops as outlined loop body functions

## Source

To understand the performance of the MP doall from line 596 of mm\_fv\_update\_nonbon, listings of source and disassembly annotated with metrics associated with each line or instruction can be examined (Figures 5 & 6).

Excl. User CPU	Excl. E\$	Excl. E\$	Excl. E\$		
sec. %	Stall	Cycles	Read	Misses	
sec. %	sec. %	%	%		
...					
0.280	0.0	0.001	0.0	0.	596. #pragma omp parallel for private (...) schedule(guided)
					597.
0.360	0.0	0.001	0.0	0.	598. for (ii=0; ii<jj; ii++)
					599. { /* if this is met update the expansion for this atom */
...					
0.010	0.0	0.003	0.0	0.	712. for (j=1; j< (*nodelistt)[inode].inode -1 ; j++)
					713. {
					714. i = (*atomlist)[i].next;
1.481	0.1	2.430	1.2	0.8	715. atomwho = (*atomlist)[i].who;
<b>76.654</b>	<b>6.4</b>	<b>51.595</b>	<b>25.4</b>	<b>19.7</b>	<b>716. if (atomwho-&gt;serial &gt; alserial)</b>
					717. {
1.941	0.2	0.211	0.1	0.	718. for (kk=0; kk<a1->dontuse; kk++)
					719. {
2.202	0.2	0.293	0.1	0.0	720. if (atomwho == a1->excluded[kk]) goto SKIP2;
					721. }
0.170	0.0	0.	0.	0.	722. (*atomall)[natoms*o+iang0++] = atomwho;
					723. }
0.841	0.1	0.178	0.1	0.	724. SKIP2: j=j;
					725. }
...					

Figure 5: Annotated source extract of MP doall from line 596 of mm\_fv\_update\_nonbon

## Disassembly

The key section includes four load instructions, the most costly of which accounts for just over 25% of the E\$ stall time (and almost 20% of the E\$ read misses) when referencing the `serial` member of the `ATOM` structure. In addition to providing data-object descriptors as annotations for memory-referencing instructions, closer examination of the annotated disassembled instructions in *Figure 6* reveals several features specific to data-oriented analysis. In particular, the E\$ stall time and E\$ read misses metrics are seen to correlate with memory-referencing instructions, whereas significant amounts of User CPU time are shown against unlikely instructions, such as the 69s associated with the compare instruction at `0x1000275F8`. This is explained by the fact that only the hardware counter overflow events have been (and generally can be) corrected with the apropos backtracking search: general profiling events are delivered with the program counter of the instruction next to be executed and are uncorrectable. (Additional anomalies, such as the load at `0x1000275F4` attributed 52s of E\$ stall time yet only 8s of User CPU time, are similarly explained.)

Excl. User CPU	Excl. E\$ Stall	Excl. E\$ Cycles	Excl. E\$ Read Misses		
sec.	%	sec.	%	%	
...					
0.030	0.0	0.329	0.2	0.0	[ 715] 1000275e4* <branch target> <=====<<<
					[ 715] 1000275e4: ld [%o1 + 8], %o1 {structure:MMATOM -}.{int next}
0.500	0.0	0.0	0.0	0.0	[ 715] 1000275e8: sra %o1, 0, %o7
0.450	0.0	0.0	0.0	0.0	[ 715] 1000275ec: sllx %o7, 4, %i0
0.060	0.0	2.101	1.0	0.8	[ 715] 1000275f0: ldx [%i0 + %i0], %o7 {structure:MMATOM -}.{pointer+structure:ATOM who}
<b>8.066</b>	<b>0.7</b>	<b>51.595</b>	<b>25.4</b>	<b>19.7</b>	[ 716] 1000275f4: ld [%o7 + 2148], %i4 {structure:ATOM -}.{int serial}
68.588	5.8	0.0	0.0	0.0	[ 716] 1000275f8: cmp %i4, %i3
0.0	0.0	0.0	0.0	0.0	[ 598] 1000275fc: nop
0.0	0.0	0.0	0.0	0.0	[ 716] 100027600: ble,pn %icc,0x10002764c
0.440	0.0	0.0	0.0	0.0	[ 715] 100027604: add %i0, %i0, %o1
0.130	0.0	0.211	0.1	0.0	[ 718] 100027608: ld [%i2 + 2196], %i4 {structure:ATOM -}.{int dontuse}
0.360	0.0	0.0	0.0	0.0	[ 718] 10002760c: cmp %i4, 0
0.0	0.0	0.0	0.0	0.0	[ 598] 100027610: nop
0.0	0.0	0.0	0.0	0.0	[ 718] 100027614: ble,pn %icc,0x100027640
0.120	0.0	0.0	0.0	0.0	[ 709] 100027618: clr %i7
0.0	0.0	0.0	0.0	0.0	[ 709] 10002761c: add %i2, 1880, %o3
...					

Figure 6: Annotated disassembly extract of MP doall from line 596 of `mm_fv_update_nonbon`

## PCs

Another perspective on the cost of individual instructions is provided by the PCs list, *Figure 7*, where the program counter (PC) for each instruction on recorded callstacks is tabulated with its associated metric data. Examination of the PCs ranked by their E\$ stall time reveals that all but two of the top 10 are in the MP doall from line 596 of `mm_fv_update_nonbon`; notably accessing the lock in `omp_set_lock_` is the second most costly PC.

Excl. User CPU	Excl. E\$ Stall	Excl. E\$ Cycles	Excl. E\$ Read Misses	Name
sec.	%	sec.	%	
1189.942	100.0	203.294	100.0	741385789 100.0 <Total>
<b>8.066</b>	<b>0.7</b>	<b>51.595</b>	<b>25.4</b>	<b>145804385 19.7</b> <code>mm_fv_update_nonbon</code> -- MP doall from line 596 + 0x000004F4 {structure:ATOM -}.{int serial}
<b>29.140</b>	<b>2.4</b>	<b>28.014</b>	<b>13.8</b>	<b>87302619 11.8</b> <code>omp_set_lock_ + 0x00000008</code> (Unascertainable)
<b>0.190</b>	<b>0.0</b>	<b>17.642</b>	<b>8.7</b>	<b>50201506 6.8</b> <code>mm_fv_update_nonbon</code> -- MP doall from line 596 + 0x000020AC {structure:ATOM -}.{double px}
12.389	1.0	9.268	4.6	25000750 3.4 <code>mm_fv_update_nonbon</code> -- MP doall from line 596 + 0x000020B0 {structure:ATOM -}.{double py}
16.762	1.4	9.070	4.5	20300609 2.7 <code>mm_fv_update_nonbon</code> -- MP doall from line 596 + 0x00002320 {structure:ATOM -}.{double qzz}
0.240	0.0	8.024	3.9	21500645 2.9 <code>mm_fv_update_nonbon</code> -- MP doall from line 596 + 0x0000231C {structure:ATOM -}.{double qyz}
0.170	0.0	7.995	3.9	20500615 2.8 <code>mm_fv_update_nonbon</code> -- MP doall from line 596 + 0x000022F8 {structure:ATOM -}.{double qxy}
10.297	0.9	7.631	3.8	24000720 3.2 <code>mm_fv_update_nonbon</code> -- MP doall from line 596 + 0x000022D0 {structure:ATOM -}.{double dpz}
0.0	0.0	7.496	3.7	22200673 3.0 <code>mm_fv_update_nonbon</code> -- MP doall from line 596 + 0x000022C4 {structure:ATOM -}.{double dpx}
0.0	0.0	6.636	3.3	19400582 2.6 <code>f_nonbon</code> -- MP doall from line 422 + 0x00000238 {structure:ATOM -}.{double x}
...				

Figure 7: PCs ranked by E\$ stall time

Data-object descriptors for each instruction are also shown where available, however, since `omp_set_lock_` is part of the *libmstk* library which was not built with symbolic debugging information, its data-object descriptors are Unascertainable. (Poor correlation of User CPU profile data with critical instructions is also evident.)

## DataObjects

The data-objects associated with memory-referencing instruction events also provide a basis for a fundamentally different set of metrics derived from the hardware counter profile event data. (Such data-derived metrics are not possible from other hardware counter events or other types of profile data.) Aggregation by data-object structure instead of code structure reveals that 78% of the E\$ stall time and 62% of the E\$ read misses relate to accesses to the ATOM structure (*Figure 8*). Around 1% of E\$ stall time and read misses are attributable to the MMATOM and MMNODE structures, and negligible amounts to other structures. Around one-third of the E\$ read misses have Unascertainable data-objects having occurred in libraries which weren't built with debug information (such as *libmstk*, *libc* and *libm*) and there are an additional small percentage which have unknown attribution for various other reasons: the <Unknown> entry is an aggregation of Unascertainable, Unspecified, Unidentified and Unresolvable unknowns.

Data. E\$ Stall Cycles sec.	Data. E\$ Read Misses %	Data. E\$ Stall Cycles sec.	Data. E\$ Read Misses %	Name
203.294	100.0	741385789	100.0	<Total>
<b>158.783</b>	<b>78.1</b>	<b>460813842</b>	<b>62.2</b>	{structure:ATOM -}
38.445	18.9	259271308	35.0	<Unknown>
31.594	15.5	241370771	32.6	(Unascertainable)
5.046	2.5	16100483	2.2	(Unspecified)
3.268	1.6	8600258	1.2	{structure:MMATOM -}
1.586	0.8	7400222	1.0	{structure:MMNODE -}
1.379	0.7	1800054	0.2	(Unidentified)
0.547	0.3	2200066	0.3	{structure:TORSION -}
0.426	0.2	0	0.	(Unresolvable)
0.334	0.2	1800054	0.2	{structure:BOND -}
0.277	0.1	1200036	0.2	{structure:ANGLE -}
0.047	0.0	100003	0.0	{structure:HYBRID -}
0.006	0.0	0	0.	<Scalars>

*Figure 8: DataObjects ranked by data-derived E\$ stall time*

Furthermore, the 78% of E\$ stall time for the ATOM structure can be attributed 25% to element `serial`, 9% to element `px` and a few percent to each of a dozen or so other elements (*Figure 9*). The `lock` element is shown with zero E\$ stall time since the only memory operations which reference it are in *libmstk*, and with those local references Unascertainable they furthermore can't be attributed to the ATOM structure. While these constitute the critical references, inducing the bulk of the E\$ read misses and stall time, the other miscellaneous references which show up in the profile also need to be considered in any possible optimisation: the current data is insufficient to quantify the number of such references which may be efficiently exploiting the E\$ but adversely impacted by code or data modifications. With references distributed throughout such a large and complex structure as ATOM, it may be more efficient to partition it into several smaller structures or re-order members such that the most referenced parts of the structure(s) are able to fit into a single cacheline.

## Segment/Page/Cacheline/Address

Event data addresses can themselves be a basis for further analysis, or aggregated by corresponding machine entities, such as the memory segment (of program loadobjects or allocated as stack, heap, shared or otherwise mapped memory), and broken down by page for those segments. Alternatively, data addresses can be aggregated by corresponding cacheline for analysis of cache mappings and utilisation.

Data. E\$ Stall Cycles sec.	Data. E\$ Read Misses %	Data. E\$ Read Misses	Data. E\$ %	Name +offset .element
158.783	78.1	460813842	62.2	{structure:ATOM -}
8.834	4.3	27800834	3.7	+0 .{double x}
0.871	0.4	2600078	0.4	+8 .{double y}
2.678	1.3	8400252	1.1	+16 .{double z}
0.721	0.4	2600078	0.4	+24 .{double fx}
0.464	0.2	1600048	0.2	+32 .{double fy}
0.020	0.0	100003	0.0	+40 .{double fz}
0.143	0.1	500015	0.1	+48 .{double q}
5.457	2.7	16900507	2.3	+56 .{double a}
1.839	0.9	5900177	0.8	+64 .{double b}
1.356	0.7	4600138	0.6	+72 .{double mass}
0.	0.	0	0.	+80 .{double chi}
0.	0.	0	0.	+88 .{double jaa}
0.755	0.4	2500075	0.3	+96 .{double vx}
0.008	0.0	0	0.	+104 .{double vy}
0.099	0.0	300009	0.0	+112 .{double vz}
0.	0.	0	0.	+120 .{double vw}
5.495	2.7	20400612	2.8	+128 .{double dx}
0.508	0.2	1600048	0.2	+136 .{double dy}
1.710	0.8	6100183	0.8	+144 .{double dz}
0.	0.	0	0.	+152 .{double gx}
0.	0.	0	0.	+160 .{double gy}
0.	0.	0	0.	+168 .{double gz}
0.024	0.0	0	0.	+176 .{double VP}
<b>18.320</b>	<b>9.0</b>	<b>52101563</b>	<b>7.0</b>	<b>+184 .{double px}</b>
9.629	4.7	26400792	3.6	+192 .{double py}
0.067	0.0	0	0.	+200 .{double pz}
7.530	3.7	22300676	3.0	+208 .{double dpx}
0.162	0.1	0	0.	+216 .{double dpy}
7.635	3.8	24000720	3.2	+224 .{double dpz}
0.120	0.1	0	0.	+232 .{double qxx}
8.461	4.2	22600678	3.0	+240 .{double qxy}
0.091	0.0	0	0.	+248 .{double qxz}
0.752	0.4	900027	0.1	+256 .{double qyy}
8.027	3.9	21500645	2.9	+264 .{double qyz}
9.269	4.6	21200636	2.9	+272 .{double qzz}
0.	0.	0	0.	+280 .{array:pointer+void close}
0.	0.	0	0.	+1880 .{array:pointer+void excluded}
3.605	1.8	14900447	2.0	+2136 .{pointer+void next}
0.	0.	0	0.	+2144 .{omp_lock_t=enumeration:<ANON> lock}
<b>52.135</b>	<b>25.6</b>	<b>146604409</b>	<b>19.8</b>	<b>+2148 .{int serial}</b>
1.767	0.9	6300189	0.8	+2152 .{char active}
0.	0.	0	0.	+2153 .{array:char name}
0.	0.	0	0.	+2162 .{array:char exkind}
0.229	0.1	100003	0.0	+2196 .{int dontuse}

Figure 9: DataObject structure ATOM and its elements with data-derived metrics

## Statistics

Process usage statistics can be examined for an overview of the 100 second duration experiment, and a breakdown of the 1208 seconds of total lightweight-process time by accounting state. 97% of the *ammp* execution is seen accounted as User CPU time, however, since by default OpenMP threads “busy wait” when they idle, this doesn't necessarily imply that effective use was made of the available processors. The negligible time accounted in User Lock state provides reassurance that actual locking, when required for the user process, was efficient.

```

Duration (sec.): 100.685
Process Times (sec.):
    User CPU: 1171.700 ( 97.0%)
    System CPU: 33.112 ( 2.7%)
    Wait CPU: 3.148 ( 0.3%)
    User Lock: 0.149 ( 0.0%)
Text Page Fault: 0.000 ( 0.0%)
Data Page Fault: 0.002 ( 0.0%)
Other Wait: 0.061 ( 0.0%)

```

Figure 10: Statistics for the entire experiment

## Timeline

The aggregate statistics for the entire experiment are also available for individual “samples” taken periodically during execution, made interactively during collection, or specified via the Collector API. To help understand the phases in the complex execution of *ammp*, the main iteration loop in the `tpac` function was annotated with a call to `collector_sample` to delimit each of its 333 steps.

Figure 11 shows these samples are charted at the top of the Timeline display in its first row (labelled with an experiment identifier, “Ex 1”) in a stacked bar with colour-coded breakdown of the proportion of time spent in the different accounting states. For this *ammp* experiment, since the majority of the time is solely accounted as User CPU, the samples are almost uniformly green. Zooming and scrolling the Timeline chart can focus on regions of particular interest. The lower part of Figure 11 shows the sample for `tpac` step 114 selected in the main panel with associated sample details shown in the side panel. Since each of the samples correspond to `tpac` steps, it's clear that some steps, such as 114 and 102, take almost 2 seconds, while intervening steps take around one-eighth of a second. The upper part of Figure 11 shows the selected sample for `tpac` step 114 in the context of the entire execution, where this repeated pattern of long steps separated by ten or so much shorter steps is less apparent but nonetheless still discernible throughout the execution and for all of the threads. Below the sample data, the main Timeline panel also shows event data for each of the 12 OpenMP threads, in this case configured to chart only the executing function at the top of each callstack for only the clock profile events.



Figure 11: Timeline views showing full experiment duration and zoomed with `tpac` step 114/333 selected

Functions in the Timeline have been colour-coded (as shown in the Legend) to emphasise key characteristics. The colour mapping applied used greens for the most costly parallelised function `mm_fv_update_nonbon` and magentas for its caller parallelised function `f_nonbon`, purple for the base stepping function `tpac`, with all of the other functions set to dark-grey. Additionally, functions in the `libmtsk` multitasking library were coloured light-grey, with key functions in contrasting colours red for `__mt_EndOfTask_Barrier_`, yellow for `__mt_WaitForWork_` and blue for `omp_set_lock_` and `omp_unset_lock_`.

While the master thread (“1.0”) works steadily through the short steps, the other worker threads are seen to spend the majority of their time idling, with only brief participation in the parallelised `f_nonbon`. More significantly, during the long steps involving the parallelised `mm_fv_update_nonbon`, imbalance results in most of the threads (including the master thread) spending more than half of their time waiting in the barrier at the end for the slowest thread to finish. Events showing `omp_set_lock_` and `omp_unset_lock_` are distributed throughout both of these parallelised functions.

The Timeline can also show additional experiment data, such as the E\$ stall (`ecstall`) hardware counter event data and full event callstacks, as shown in *Figure 12*, for a more comprehensive analysis. The upper part of *Figure 12* has a clock profile event selected from the master thread at the end of `tpac` step 114, with the callstack revealing an explicit locking with `omp_set_lock_` in an MP doall nested inside an OMP parallel region of `f_nonbon`.

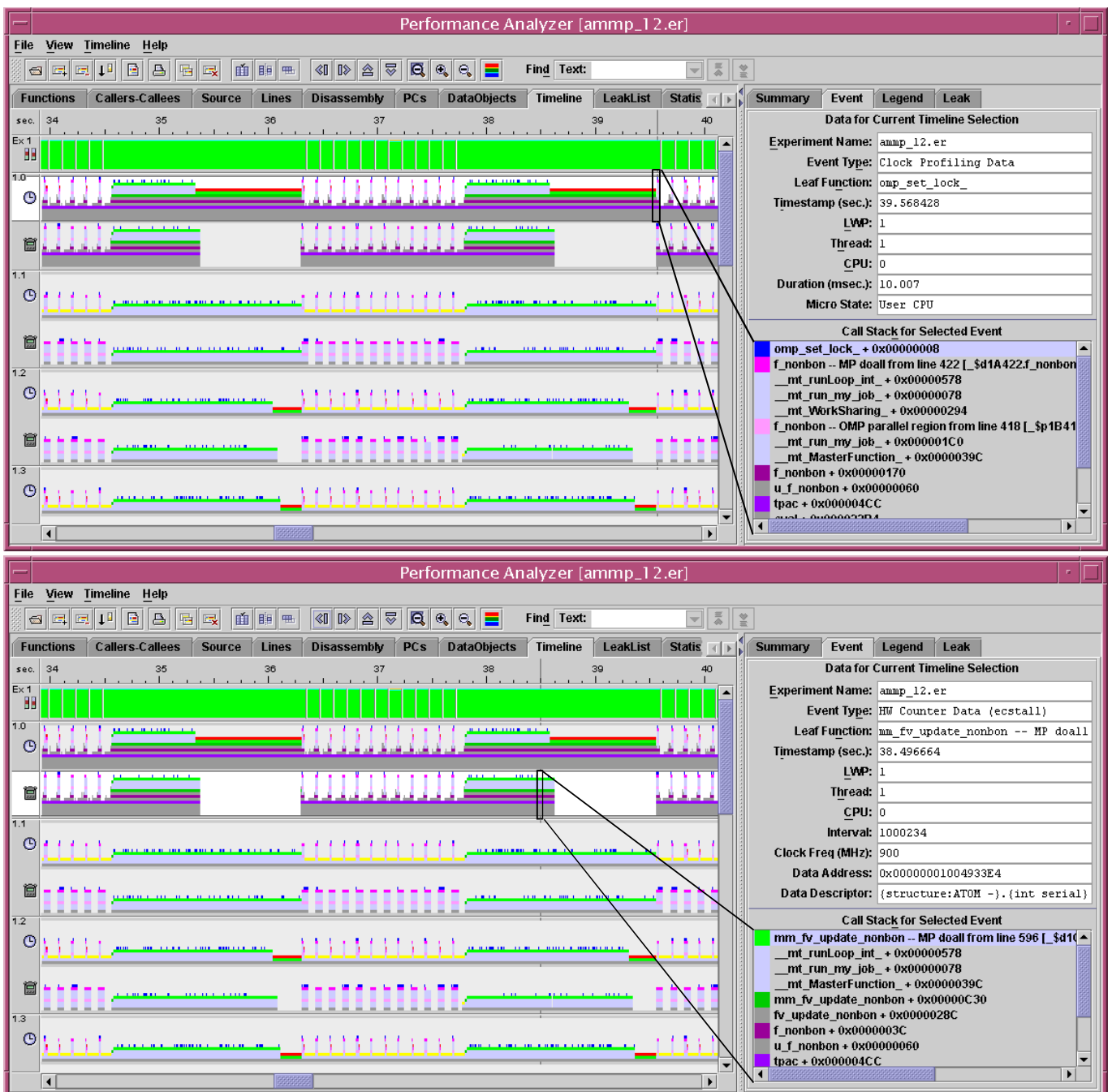


Figure 12: Timeline views zoomed showing full callstacks and with clock and ecstall counter events selected

In the lower part of *Figure 12*, one of the ecstall hardware counter overflow events has been selected at 38.5s from the master thread (which has its row's background highlight and time marked) and its callstack shows the parallel function for the MP doall from line 596 in `mm_fv_update_nonbon` executing. When charted E\$ read miss and stall events are selected, event detail includes the associated data address and dataobject descriptor (where available): in this case, the `serial` element of the ATOM structure at heap address `0x00000001004933E4`.

A clear difference in the profiles of clock and ecstall events is the absence of ecstall events during the periods when the `libmstk __mt_EndOfTask_Barrier_` and `__mt_WaitForWork_` functions are being executed, which is to be expected since they don't involve memory accesses. Both clock and ecstall event profiles show `omp_set_lock_` and `omp_unset_lock_` functions, as they do involve memory accesses to lock variables, and their proportion in the profiles for all threads highlights their significant impact on *ammp* performance.

Filtering the experiment data to get metrics calculated for each OpenMP thread and comparing those for the MP doall from line 596 of `mm_fv_update_nonbon` shows a clear correlation between the User CPU time and E\$ stall time (and associated E\$ read misses), i.e., the time that the OpenMP thread takes for its useful work appears to depend on the number of E\$ read misses and associated E\$ stall time handling those misses, and the fact that some threads encounter notably more misses referencing the data they require results in them lagging significantly behind the others. This imbalance ultimately leads to diminished effectiveness and scalability as the number of processors available for use increases.

Excl. User CPU		Excl. E\$ Stall Cycles		Excl. E\$ Read Misses		Name
sec.	%	sec.	%		%	
274.402	100.0	129.951	100.0	359810812	100.0	<Total>
18.063	6.6	8.566	6.6	23100693	6.4	Thread 0
<b>35.915</b>	<b>13.1</b>	<b>14.161</b>	<b>10.9</b>	<b>42301269</b>	<b>11.8</b>	<b>Thread 1</b>
<b>32.273</b>	<b>11.8</b>	<b>14.206</b>	<b>10.9</b>	<b>40901227</b>	<b>11.4</b>	<b>Thread 2</b>
<b>32.152</b>	<b>11.7</b>	<b>14.302</b>	<b>11.0</b>	<b>41701258</b>	<b>11.6</b>	<b>Thread 3</b>
26.118	9.5	12.781	9.8	33200996	9.2	Thread 4
22.526	8.2	11.272	8.7	28400852	7.8	Thread 5
20.504	7.5	11.010	8.5	28900867	8.0	Thread 6
17.562	6.4	8.826	6.7	24200726	6.7	Thread 7
17.322	6.3	8.687	6.7	24500735	6.8	Thread 8
17.382	6.3	8.755	6.7	25300770	7.0	Thread 9
17.192	6.3	8.619	6.6	22700681	6.3	Thread 10
17.392	6.3	8.766	6.7	24600738	6.8	Thread 11

*Figure 13: Breakdown of the MP doall from line 596 of mm\_fv\_update\_nonbon metrics by OpenMP thread*

## Tuning results

Having identified significant cache utilisation and thread scheduling inefficiencies from the analysis of the *ammp* performance experiment, algorithmic and data structure modifications can be investigated and their effectiveness similarly analysed by collecting further experiments. Re-layout of a couple of members of the ATOM structure to pack key elements for more effective cache utilisation provided a 5% improvement in execution time. To address the serious load imbalance in the MP doall from line 596 of `mm_fv_update_nonbon`, the loop iterations were reversed to better suit the guided schedule for an independent gain of over 12%. Compound benefit of both small modifications was over 17% improvement in total *ammp* execution time on 12 processors. Furthermore, the modest scalability of *ammp* is improved, with superior performance from all configurations up to the full 24 processors of the SF6800, and a best gain of 25% over the original version.

## Related work

This study focused on performance analysis and preliminary tuning of the AMMP application, which had been already parallelised using OpenMP directives as part of the SPEC OMP 2001 benchmark suite. Previous analysis [8] noted that parallel performance was primarily limited by memory system performance, partially attributed to true-sharing as a result of explicit locking (required for correctness), however, with additional insight into the performance of the application's data objects it was possible to refine this prognosis to particular structure elements and their layout. Furthermore, loop load-imbalance which had been considered to be negligible on small-scale systems, was found by visual representation and per-thread analysis to be a major impediment to efficient execution on even moderately larger systems. In both cases, the Sun ONE Studio 8 tools helped clearly identify performance opportunities and pursue effective remedies.

Sun ONE Studio 8 performance tools provide comprehensive support for analysis of a wide variety of applications, including those which are explicitly multithreaded and/or multiprocess, and indeed hybrids such as OpenMP combined with MPI, via configurable timer and hardware counter event sampling in conjunction with synchronisation and message-passing event tracing through dynamic library interposition. Symbolic information optionally provided by compilers facilitates source-level analyses, and doesn't materially reduce performance, for convenient use during development, as well as in production. The GuideView and VAMPIR tools, for OpenMP statistics and MPI traces respectively, are being integrated [9], however, they currently rely on compiler-inserted instrumentation. Similarly, the EXPERT tool environment [10] utilises source-code and compiler-inserted instrumentation combined with tracing libraries to collect OpenMP and MPI execution data for automated analysis directed from a catalogue of performance properties, but currently lacks source-level attribution. Notably, the Paraver [11] set of tools have used dynamic (binary) instrumentation for tracing OpenMP and MPI events and offer extensibility to derive additional metrics for visualisation and analysis. While memory and cache performance can be profiled as a whole by each of these tools, only the Sun ONE Studio tools support attribution to program data objects, which has previously only been possible from simulation and impractical for large-scale distributed applications.

## Conclusion

New functionality in Sun ONE Studio 8 compilers and performance tools extends the analyses previously provided for multithreaded, memory-sensitive applications. Complementary support for data-oriented analyses derived from hardware counter events provides insight into ineffective memory and cache use, and the graphical event chart helps identify work imbalance and when and where it significantly impacts execution performance. Both aspects help understand and investigate execution inefficiencies and in the development of more efficient and scalable OpenMP applications.

## References

- [1] Larry Meadows, "OpenMP on SPARC Solaris: Compilers, Tools and Performance," *Proceedings of the 2nd European Workshop on OpenMP (EWOMP 2000, Edinburgh, Scotland)*, September 2000.
- [2] Myungho Lee, Larry Meadows, Darryl Gove, Dominic Paulraj, Sanjay Goil, Brian Whitney, Nawal Copty, and Yonghong Song, "Compiler Support and Performance Tuning of OpenMP Programs on SunFire Servers," to appear in *Proceedings of EWOMP 2003 (Aachen, Germany)*, September 2003.
- [3] *Sun ONE Studio Compiler Collection*, Sun Microsystems Inc., <http://developers.sun.com/prodtech/cc/>
- [4] *Sun ONE Studio 8: Program Performance Analysis Tools*, Sun Microsystems, Inc., Part 817-0922-10, May 2003. <http://docs.sun.com/db/doc/817-0922>
- [5] Marty Itzkowitz, Brian J. N. Wylie, Christopher Aoki and Nicolai Kosche, "Memory Profiling using Hardware Counters," to appear in *Proceedings of SC2003 (Phoenix, AZ, USA)*, November 2003.
- [6] Robert W. Harrison, "AMMP home page," <http://www.cs.gsu.edu/~cscrwh/ammp/ammp.html>
- [7] *Sun ONE Studio 8: OpenMP API User's Guide*, Sun Microsystems, Inc., Part 817-0933-10, May 2003.
- [8] Vishal Aslot and Rudolf Eigenmann, "Quantitative Performance Analysis of the SPEC OMPM2001 Benchmarks," *Scientific Programming*, 11(2):105-124, 2003.
- [9] Jay Hoeflinger, Bob Kuhn, Wolfgang Nagel, Paul Petersen, Hrabı Rajic, Sanjiv Shah, Jeffrey Vetter, Michael Voss, and Renee Woo, "An Integrated Performance Visualizer for MPI/OpenMP Programs," *Proceedings of WOMPAT 2001 (Purdue, IN, USA), LNCS 2104*, pp. 40-52, July 2001.
- [10] Felix Wolf and Bernd Mohr, "Automatic Performance Analysis of Hybrid MPI/OpenMP Applications," *Proceedings of 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing (Euro PDP 2003, Genova, Italy)*, pp. 13-22, February 2003.
- [11] Jordi Caubet, Judit Gimenez, Jesús Labarta, Luiz DeRose, and Jeffrey Vetter, "A Dynamic Tracing Mechanism for Performance Analysis of OpenMP Applications," *Proceedings of WOMPAT 2001 (Purdue, IN, USA), LNCS 2104*, pp. 53-67, July 2001.