Recent Developments of LS-DYNA Performance Optimization

Authors:

Youn-Seo Roh and Henry Fong
Sun Microsystems, Inc.

Correspondence:

Youn-Seo Roh
260 Constitution Dr.
MS MPK24-201
Menlo Park, CA 94025
U.S.A.

Tel: 650-786-6093
Fax: 650-786-6530
e-mail: youn-seo.roh@sun.com

Keywords:
SunFire^{TM} servers, Sun ONE^{TM} Studio compilers, performance optimization, tuning, scalability, cluster performance

ABSTRACT

A recent effort of optimizing the performance of LS-DYNA running on SPARC(R)-Solaris™ servers is described. With new releases of compilers, generated executables benefit from the additional performance of latest UltraSPARC(R) CPU's for SMP servers. Also, new release of Sun HPC ClusterTools™ cluster environment includes tools that facilitates tuning of LS-DYNA-MPP executables and the MPI environment. A collection of development tools targeted for SPARC performance improvement results in faster simulations, fully benefiting continuously updated hardware performance. Those developments are exhibited with customer benchmark examples. With Sun ONE Grid Engine products, more efficient simulation environment is viable for LS-DYNA simulation.

1. Performance Measurement

Code performance improvement starts with tools that measure behavior of executables. This section describes some recent developments in Solaris-SPARC performance measurement tools.

1.1. **Hardware Counter Tools**

Sun's Solaris development stack includes a number of performance measurement tools. Of these, most relevant to measuring LS-DYNA performance is a set of hardware performance counters called `cpustat` and `cputrack` [1], as well as the application programming interface called cpc [2] that utilizes these performance counters. `cpustat` measures system-wide behavior of the counters and requires super user privilege to use, while `cputrack` measure process-wise statistics and does not require super user privilege. In a multi-user, multi-process environment of a server running different sorts of application, in many cases, cputrack gives accurate account of CPU-related behavior of LS-DYNA processes.

```
-------------------------est------------------------
Ultra-III                ticks          sec     %
----------------------------------------------------
D0_IC_miss         33325744728       36.306   12.6%
D0_br_targ_calc      459873112        0.501    0.2%
D0_2nd_br             51559648        0.056    0.0%
D0_mispred           451455776        0.492    0.2%
D_rs_mispred         203698744        0.222    0.1%
Rs_storeQ          43962920640       47.894   16.6%
Rs_FP_use           8410373720        9.162    3.2%
Rs_IU_use           3384755336        3.687    1.3%
Re_FPU_bypass            15152        0.000    0.0%
Re_RAW_miss         1041032320        1.134    0.4%
Re_DC_miss         38082830648       41.488   14.4%
Re_EC_miss          4686765408        5.106    1.8% (in DC miss)
Re_PC_miss           227774560        0.248    0.1%
DTLB_miss              2042800        0.189    0.1%
total             129775672384      141.381   49.1%
----------------------------------------------------
time              264376964512      288.019  100.0%
instr             222093630424
IPC                      0.840 (instr/time)
Grouping                 1.650 (instr/(time-total))
```

Figure 1. Example output of a multiplexed performance counter tool of an LS-DYNA job.

`Cputrack` utilizes two on-chip hardware performance counters which can measure several different hardware events, including instruction and data cache misses as well as other internal states of the processor. Only two event types can be measured simultaneously, but by repeating runs it is possible to gather useful run statistics of a user process. Especially if the application has a relatively constant run profile and the total run is long enough to average out the variations in the measurements, it is possible to obtain through multiplexing a meaningful run statistics in a single pass of the job.

Figure 1. shows the example output of an internal tool that uses such method. It reveals overall characteristics of LS-DYNA job including store queue misses, instruction and data cache misses, as well as TLB(Translation Lookaside Buffer) misses. With the output, the user will be able to have a clear concept of how the application is performing in terms of CPU statistics.

More detail of the SPARC performance counter can be achieved from [3]. With the SPARC architecture available in public domain, and with the information on the hardware counter along with the CPC API, it is possible for a Solaris user to develop a customized performance characterization tool for his/her own purpose.

1.2. Compiler Tools: Performance Collector and Analyzer

Starting from Sun ONE Studio 6, Sun compiler includes performance tools suite called collector and analyzer. Current release of Studio 7 and the version soon to be released of Studio 8 [4] have additional improvements including MPI profiling. Recent releases will also benefit from the CPU-specific information of the latest hardware.

There are both GUI-based and command line-based version of the tools. Analyzer is a GUI-based tools incorporate both data collection and analysis. Collect, er_print and er_src are command line version of analyzer. Collect tool is based on cputrack and can collect the run statistics in a experiment file and directory. After the experiments are recorded with collector, analyzer (or er_print for command line) tool can use the recorded experiment data. Experiment data to be analyzed includes regular function profile, source code annotation of various metrics, and disassembly listing.

With the previous results of the run statistics via hardware counters as exemplified in Figure 1, it is now possible to figure out which line (with the aid of source annotation) or which instruction (with the aid of disassembly listing) is contributing to the achieved statistics. Figure 2 shows an example of a MPP-LS-DYNA run. The run was collected with

```
% mprun -np $np collect $bin i=$data ncycle=$nc
```

This will create a default experiment directory and data under `test.N.er`, where `test.N.er` will be created as separate directories as many as the number of ranks of the MPI job. After the run, the experiment data will be analyzed by typing

```
% analyzer test.N.er      (GUI version)
% er_print test.N.er      (command line version)
```

Inside the analyzer (or er_print), it is possible to set various metrics including exclusive or inclusive user CPU time. By appending function name to the metrics, function profile is achieved. The example in Figure 2 shows exclusive user CPU time in seconds and in percent total time, inclusive user CPU time in seconds and percent total time, and function name, respectively.

```
% er_print test.0.er
(er_print) metrics e.user:e%user:i.user:i%user:name
(er_print) limit 10
(er_print) function
Functions sorted by metric: Exclusive User CPU Time
Excl. User     Incl. User      Name
CPU            CPU
   sec.     %     sec.     %
308.100 100.0  308.100 100.0   <Total>
 29.210   9.5   29.210   9.5   trnfbt_
 26.050   8.5   27.750   9.0   tranbt_
 20.130   6.5   20.130   6.5   shl3s_
 15.210   4.9  133.160  43.2   blytsy_
 14.110   4.6   14.110   4.6   mppcns13a_
 10.750   3.5  143.940  46.7   elem2d_
  9.200   3.0    9.200   3.0   stdspb_
  8.910   2.9    8.910   2.9   dfnls_
  8.870   2.9    8.870   2.9   tbsc1s_
  8.620   2.8   19.930   6.5   strgen_
```

Figure 2. Analyzer(er_print) function profile output from an MPI process.

With Studio 7 compiler and later, it is also possible to trace MPI function calls by including -m option to collect command :

```
% mprun -np $np collect -m on $bin i=$data ncycle=$nc
```

MPI trace metrics available are: MPI Time, MPI Sends, MPI Bytes Sent, MPI Receives, MPI Bytes Received, and Other MPI calls.

```
% er_print test.2.er
(er_print)metrics
        i.mpitime:i.mpibytessent:i.mpisend:i.mpibytesrcvd:name
(er_print) limit 20
(er_print) functions
Functions sorted by metric: Inclusive MPI Time
Incl.    Incl. MPI  Incl. MPI  Incl. MPI    Name
MPI      Bytes      Sends      Bytes
   sec.  Sent                  Received
305.021  60418348   164054     4898051124   MAIN_
305.021  60418348   164054     4898051124   main
305.021  60418348   164054     4898051124   _start
305.021  60418348   164054     4898051124   <Total>
305.019  60414524   163992     4898049192   overly_
304.628  55853468   163489     4895694108   fem3d_
304.628  55853468   163489     4895694108   soltn_
274.258    550872    41788         550872   pmpi_allreduce_
274.258    550872    41788         550872   PMPI_Allreduce
 29.921   6907480   111463     4440882696   mppcon_
 29.008   6743904    91462     4440718544   mppc13_
 29.008   6743904    91462     4440718544   mppc13a_
 19.282         0        0     4568334348   pmpi_recv_
 19.282         0        0     4568334348   PMPI_Recv
 14.467   2107112    19997     1765326864   snfsum_
  6.183         0        0              0   pmpi_waitall_
  6.183         0        0              0   PMPI_Waitall
  4.755   2733776    59978     1764250872   mppccpm_
  4.338    161216    40304         161216   pmpi_alltoall_
  4.338    161216    40304         161216   PMPI_Alltoall
```

Figure 3. Analyzer function profile output with MPI call tracing turned on.

After verifying the function profile, it is possible to view the source annotation to find out specific line that contribute to the timing. It is possible to view various metrics including hardware counter values associated with each line of source. It is also possible to view compiler commentaries generated during compilation process next to the source line. Figure 4 shows, for example, the process of collecting for measuring data cache read miss rate, annotated alongside the source line. `% collect` command without any argument or option will print out available hardware counters for collecting. The example below was invoked with

```
% mprun -np 1 collect -h dcr,,dcrm $bin i=$data ncycle=$nc
```

where `dcr` and `dcrm` are the counter names that are recognized inside `collect` command, and represent “Data Cache Read reference” and “Data Cache Read Misses” respectively. After the run, when `er_print` is invoked, the default metrics is automatically set for `e.dcr:i.dcr:e.dcrm:i.dcrm:name`, which stands for “exclusive D-cache read reference, inclusive D-cache read reference, exclusive D-cache read misses, inclusive D-cache read misses, function name.” This default metrics can be changed by `metrics` command inside `er_print`.

```
% er_print test.3.er
(er_print) metrics e.dcr:e.dcrm:e%dcrm:i%dcrm:name
current: e.dcr:e.dcrm:e%dcrm:i%dcrm:name
(er_print) sort e.dcrm
current sort metric: Exclusive D$ Read Misses
(er_print) limit 10
(er_print) functions
Functions sorted by metric: Exclusive D$ Read Misses
Excl. D$     Excl. D$ Read     Incl. D$   Name
Read Refs    Misses            Read
                          %    Misses %
71451878355  2116114464 100.0  100.0      <Total>
 8349086731   429813610  20.3   20.3      trnfbt_
 7176078267   252307951  11.9   11.9      tranbt_
 2732022528   176605570   8.3   57.3      blytsy_
 1387011338   150505499   7.1    7.1      tbsc1s_
 2601027221    92702994   4.4    4.4      mppcns13a_
 1016008219    90902729   4.3    4.3      updatec_
 1913017553    83002622   3.9    3.9      frcbt1_
  970006243    69002183   3.3   60.6      elem2d_
 1043007930    63501927   3.0   99.8      fem3d_
```

Figure 4. performance analyzer output based on hardware counter.

As can be seen in Figures 4 and 5, the user can immediately notice which part of code is becoming a performance bottleneck, and further investigation on how the system behavior is at the CPU register level is possible.

In addition to the source annotation, it is also possible from within the analyzer(er_print) tool to obtain the disassembly listing of a file or a function of interest. It will further enhance the understanding of code performance. Disassembly listing can be obtained as

```
(er_print) disasm <file name or function name>
```

with appropriate metrics settings.

```
      (er_print) <with previous settings>
      (er_print) src trnfbt_
...
   Excl. D$   Excl. D$
  Read Refs       Read
                Misses
     7000032   1400044       1.        subroutine trnfbt(e,...)
...
                          Loop below pipelined with steady-state cycle count ..
                          Loop below unrolled 1 times
                          Loop below has 11 loads, 6 stores, 0 prefetches,
                               6 FPadds, 6 FPmuls, and 0 FPdivs per iteration
                           219. c
           0         0     220.        if (icase(18).eq.0) then
                           221. c
     2000010         0     222.        do i=lbnd,ubnd
   106000945   9600289     223.        xft31(i)=-xft11(i)+gym3(i)*qzs1(i)
    38000383   2900088     224.        xft32(i)=-xft12(i)+gym3(i)*qzs2(i)
    45000431   4100125     225.        xft33(i)=-xft13(i)+gym3(i)*qzs3(i)
    55000535   4500147     226.        xft41(i)=-xft21(i)+gym4(i)*qzs1(i)
    42000362   2800085     227.        xft42(i)=-xft22(i)+gym4(i)*qzs2(i)
   164001550   6300192     228.        xft43(i)=-xft23(i)+gym4(i)*qzs3(i)
                           229.
           0         0     230.        enddo
```

Figure 5. Performance analyzer output with source annotation along with hardware counter (D-cache) information (Source code altered).

1.3. Solaris large page support

Although this cannot be categorized as performance measurement tools, but the large page support that became available as of Solaris 9 offers a run-time performance improvement opportunity. Even before Solaris 8, it was possible to utilize process memory pages larger than 8kByte default using such tools as Intimate Shared Memories (ISM). But with this operating system support, it is possible with just command line options to change the page sizes between 8k, 64k, 1M, and 4M. The relevant commands are [5] :

```
% ppgsz -o heap=4M ls970 i=$data ...                (for SMP)
% mprun -np $np ppgsz -o heap=4M mpp970 i=$data ... (for MPP)
```

After launching the process with a specified page size, the process page size can be verified with `% pmap -s <pid> | grep heap`.

Larger page sizes can be beneficial in improving a significant TLB misses, which is not uncommon for a large datasets. A hardware counter tools such as what is explained in Section 1.1. and Figure 1 can be used to measure the portion of TLB miss time out of the total run time. If the measurement shows a significant amount, then the job can be launched with large pages set.

2. OpenMP Performance Improvement

With the aid of performance measurement tools, it was possible to obtain an optimization of LS-DYNA binary, both -SMP and -MPP version. In this section, we describe an example improvement for a customer's stamping application where the performance of -SMP binary became an issue.

2.1. Loop splitting

It is found that most of time-consuming routines of LS-DYNA run spends time in loops with many floating point instructions. The instruction pipeline scheduler makes uses of floating point registers, but in many cases, sheer number of instructions poses excessive load on the scheduler, causing it to fail to schedule the loop. This is usually helped by splitting the loops into smaller ones. In many cases, this technique helps performance noticeably. In the current customer's case, loop splitting was used to improve the run-time performance.

2.2. Compiler prefetches

Sun compiler has been continuously improved in its prefetch capability. Prefetches can be generated automatically as well as through compiler directives.

```
% f90 -xprefetch=auto
```

will let the compiler insert prefetch instruction into the generated instructions. In many cases, turning on the automatic prefetch improves code performance noticeably. But for a certain functions or routines, automatic prefetch may even degrade the performance. In such cases, it is possible to specifically control the location, variable or stride of the prefetch instruction. Compiler option for this case is

```
% f90 -xprefetch=explicit
```

Inside the source, it takes a pragma statement to specify the explicit prefetch as:

```
      do i n=lbnd,ubnd
c$pragma sparc_prefetch_read_many (src(1,n-k+2))
c$pragma sparc_prefetch_write_many(dst(n+1,1))
      dst(n,1)=src(1,n-k)
      ...
      enddo
```

A right mix of these two kind of prefetch instruction has been applied. For that purpose, function profile and source annotation, as well as the disassembly listing from the collector/analyzer experiments was used to compare the timing difference before and after the application of prefetches.

2.3. Inlined Math Function

The function profile revealed that sign(.) function took an unnecessary portion of time. It was also found that the function inside a computationally expensive loop prevents the instruction scheduler from properly scheduling the loop. A fix was made to the math library, and a patched version of libmil.so library (inlined math library) proved to be improving the performance.

2.4 Performance Optimization Results

With above optimization aggregated, the resulting improvement was 1.8X on a UltraSPARC III+ CPU. Table 1. shows the results. The application was a metal stamping problem. The source code tuning including loop splitting and addition of prefetch pragma have been implemented into the source code of release 970.3280. Although the tuning for the current problem was measured with a specific stamping case, other similar stamping problems will probably benefit from the tuning. For other types of problems such as crash or drop test, same techniques will be used for tuning.

Splitting the loop manually within the source can be quite labor intensive. It also will make a code maintenance somewhat cumbersome. For this, compiler initiated automatic loop splitting, or loop distribution, is being pursued simultaneously.

No. of shell elements	42,000
Hardware	SunBlade 2000 900MHz US-III+ 2GB Memory
Reference Binary	ls960.1488
Tuned Binary	ls970.2630
Reference Elapsed time	2981 second, 1.0
Tuned Elapsed time	1633 second, **1.83X**

Table 1. An example performance improvement of LS-DYNA-SMP.

3. MPI Performance Improvement

The performance measurement tools and techniques of section 1 will benefit both SMP and MPP executables. Loop-splitting and prefetch tuning achieved for SMP binary will also improve the performance of MPI binary. In addition to this, it is also possible to set environment variables that affect the MPI jobs. The latest release of HPC ClusterTools (Release 5) [6] provides a useful tool called mpprof that generates a birds-eye view of the performance of an MPI job.

3.1. HPC ClusterTools 5: mpprof

Based on the performance tuning methods described above, LS-DYNA-MPP binary has been tuned. Along with profile-based tuning, MPI application tuning is facilitated with HPC cluster development stack built around ClusterTools. The latest release of ClusterTools 5 has an additional tool called mpprof. Mpprof gives overview of an MPI process. The tool starts with saving index files from an MPI job by setting

```
% setenv MPI_PROFILE 1
% mprun -np $np mpp940 i=$data
% unsetenv MPI_PROFILE
```

It creates an index file named mpprof.index.cre.<jid> where <jid> is a job id set by the cluster runtime environment. Then mpprof is launched by

```
% mpprof mpprof.index.cre.<jid>
```

The output of mpprof consists of suggestions on MPI environment variables for better MPI communication performance. Usually those suggestions involve MPI_SPIN, MPI_POLLALL, MPI_PROCBIND and many of shared memory related ones such as MPI_SHM_CYCLESTART when running on the SPARC cluster nodes.

3.2. Scalability results

Based on the aggregate of tuning techniques described above, a tuned MPP binary was generated and a customer benchmark was run for Mefos, Metallurgical Research Institute, AB [7]. The job was run on a cluster environment consisted of SunFire V480 (4-processor) [8] and V880 (8-processor) servers [9], linked with Myrinet interconnect. The new binary showed an excellent scalability of greater than 80% efficiency at processor counts bigger than 100 and node count of 32 (for V480). Table 2. shows the summary of results and achieved scalability.

V880 cluster that consisted of 8 nodes of 8-processor servers showed similar scalability. It scaled up to the full 64-processors running the same problem at 1227 seconds, which is a remarkably close number compared to the cluster of 4-processor nodes.

	SunFire V480 Cluster
Hardware	32 x SunFire V480 4 x UltraSPARC III+ @900MHz/8MB/150MHz 16 GB memory 1 M3F-PCI64C-2 Myrinet card (optical fibre) NFS server for Storage: SF V880 with 4 x T3's (9x36G) a shared UNIX file system through a private 1000BT network

Software	Solaris 8
	HPC ClusterTools 4.0
	Sun ONE Studio 7, Compiler Collection
	Myrinet driver for HPC CT 4.0:
	gm-1.6.4_rc0-sun4u-SunOS-5.8-8port
	LS-DYNA: mpp970.2779 with MPI environment settings of
	`export MPI_SHM_SBPOOLSIZE=8388608`
	`export MPI_SHM_NUMPOSTBOX=256`
	`export MPI_PROCBIND=1`

Problem	1.2M nodal points, 600K brick elements. Sheet metal rolling.

Table 2. Specification of SunFire 480 cluster.

NCPU	***Elapsed Time (second)***	***Scaling***	***Efficiency (%)***	***Best Config (Nodes x CPU)***
1	71790	1.00	100.0	1 x 1
2	36194	1.98	99.2	2 x 1
4	20419	3.52	87.9	2 x 2
8	10282	6.98	87.3	8 x 1
12	6077	11.8	98.4	12 x 1
16	5144	14.0	87.2	8 x 2
24	3356	21.4	89.1	12 x 2
32	2547	28.2	88.1	16 x 2
48	1650	43.5	90.6	16 x 3
64	1278	56.2	87.8	32 x 2
72	1154	62.2	86.4	24 x 3
96	889	80.8	84.1	32 x 3
128	753	95.3	74.5	32 x 4

Table 3. Scalability of a Sun Fire V480 cluster.

4. Summary and Conclusions

With the developments in the performance tools, Solaris development environment still benefits the continuous hardware performance improvements in UltraSPARC processor lines. Recently introduced entry level multi-processor servers perform well in a clustered environment running LS-DYNA-MPP executables. Also a portal environment that incorporates Sun ONE Grid Engine[10] will serve as an efficient computation platform as demonstrated in a previous report[11] of Technical Compute Portal.

5. Acknowledgment

The following people are acknowledge for their contributions:

The multiplexed performance counter tool is from Darryl Gove of Sun Microsystems.

The Mefos benchmark results were due to Eduardo Pavon, Jonas Edberg, Brian Whitney, Hugh Caffey, Borje Lindh, and Phil Pincus of Sun Microsystems.

6. References

1. Man pages of cpustat, cputrack: `% man -s 1M cpustat` `man -s 1 cputrack`

2. Man pages of cpc : `% man -s 3cpc cpc`

3. SPARC V9 JPS1 Implementation Supplement: Sun UltraSPARC-III, Sun Microsystems, 2000.

4. Sun ONE Studio website:http://wwws.sun.com/software/sundev/solde/index.html

5. Man page of ppgsz : % man ppgsz

6. HPC Cluster Tools website: http://www.sun.com/servers/hpc/software/

7. Metallurgical Research Institute, AB: http://www.mefos.se/

8. Sun Fire V480 Server: http://www.sun.com/servers/entry/v480/index.html

9. Sun Fire V880 Server: http://www.sun.com/servers/entry/880/index.html

10. Sun Grid Engine: http://wwws.sun.com/software/gridware/

11. Dan Fraser, Youn-Seo Roh, and Henry Fong, “Web-Centric LS-DYNA – development of a Technical Computing Portal”, 7th International LS-DYNA Users Conference, 2002.