



**Global Business Integration and
Telecommunications Solutions (Pty) Ltd
Trading As GlobeTOM**

Waterford Court Office Park
Blocks B7 & B8
234 Glover Ave
Lyttelton
South Africa

Postnet Suite 17
Private Bag X1015
Lyttelton
0140

Tel: +27 12 664 6205
Fax: +27 12 644 1200

TECHNOLOGY WHITE PAPER

*Performance comparison
between the HPUX 11i Multi-
arena Memory Management
Module and the Memory
Tuning System (MTS) from
NewCode Inc.*

Published in association with



A MEMBER ORGANISATION OF THE



DOCUMENT DETAILS

Document Version	1.1. 8
Date	18 January 2004
Document Name	Mts_ Whitepaper
Author	PHILIP STANDER, GlobeTOM

TABLE OF CONTENTS

1	Introduction	5
2	Experimental Conditions	5
3	The memtest utility.....	5
4	Replacement Memory Management Software Considerations	7
5	Motivation for this investigation.....	7
6	Extrapolation of results to customer production systems	9
7	Experimental results	10
7.1	Examination of Fragmentation Issues.....	10
7.1.1	Arena Growth as a function of allocation iterations.....	10
7.2	Apparent instability with sharing of HPUX arenas across threads	11
7.3	Performance Results	13
7.3.1	Performance as a function the number of threads per process	13
7.3.2	The effect of varying small blocks.....	14
8	TECHNICAL RECOMMENDATIONS	15
9	APPENDIX A – FRAGMENTATION TEST RUN RESULTS	16
10	APPENDIX B – PERFORMANCE TEST RUN RESULTS.....	17
11	APPENDIX C – MEMTEST SOURCE CODE LISTING.....	21
12	APPENDIX D – MEMTEST BUILD PROCEDURE.....	27

COPYRIGHT NOTICE

This document is jointly published by GlobeTOM (Pty) Ltd and NewCode Inc. It is disclosed to prospective customers interested in the Memory Tuning System (MTS) from NewCode. It may be distributed to any prospective customers of NewCode without obtaining prior permission from GlobeTOM.

DISCLAIMER

GlobeTOM certifies that the results presented in this White Paper are accurate, but shall assume no liability for any errors or omissions in this White Paper.

While this White Paper illustrates improved performance results achieved with MTS instead of the standard HPUX Multi-Arena allocator, this White Paper is not intended to damage the image of Hewlett Packard and, in particular, HPUX. MTS shows similar performance improvements over the allocators of other Operating Systems as it offers a specialised product for specialised applications. The standard HPUX memory allocators operate faultlessly in many customer environments.

TERMS

TERM	MEANING
MTS	Memory Tuning System. Formerly ATS fro, Rogue Wave
IBS	The Code Name of the specific real-time Telecommunications Industry Sector product investigated ins this White Paper for performance gains using MTS.
Multi-arena allocator	A memory allocation policy based on a collection of memory arenas that may be accessed by multi-threaded applications where one or more threads is assigned an arena to use.

1 Introduction

This technical communication reports on the findings from an investigation into the performance of the HPUX 11i Multi-Arena memory management implementation. An evaluation copy of the Application Tuning System (MTS) from NewCode was used as alternative to the HPUX multi arena allocator.

The subject matter of this investigation was focused on multi-threaded application performance only and therefore testing was only conducted with a multi-threaded application using the multi-arena allocator or the fast MTS allocator (libatsfm32.sl).

The MTS memory management library is fully supported on HPUX including warranty and support and was therefore considered as a viable replacement for the HPUX multi-arena allocator in the event that these findings prove that MTN will experience some performance and integrity gains from the use of such a replacement library.

2 Experimental Conditions

While the results presented in this technical communication are accurate given the experimental conditions used, GlobeTOM shall not be liable for the accuracy or ability to repeat these results. GlobeTOM declare that the results published in this technical communication will be repeatable using the memtest.cpp listing provided in Appendix A together with the build conditions and HPUX OS release and patch levels as listed below:

OS Release / Patch Level	2 Processor L-Class Server	8 Processor N Class Server
HPUX	11i	11i
libc.sl	PHCO_27434	PHCO_26124
libCsup.sl	A.03.33	A.03.33
libpthread.sl	CUP180_BL2000_1005_2 Thu Oct 5 16:30:41 PDT 2000	PHCO_27632 Wed Aug 28 15:47:47 PDT 2002

The HP aCC compiler (release A.03.35) was used to build the memtest binary versions. Two versions were generated:

```
memtest_hp:    Generated with aCC and linked only with libpthread.sl
memtest_ats:  Generated with aCC and linked with libpthread.sl and libatsfm32.sl
```

Both binaries were generated as PA RISC 32-bit binaries (i.e. compiled with the +DA2.0 compiler directive).

Tests were conducted on two HP PA RISC platforms as specified below:

```
N-Class 8-processor server equipped with 10Gb RAM
L-Class 2-processor server equipped with 2Gb RAM
```

3 The memtest utility

The memtest utility is a standard memory thrashing utility developed by GlobeTOM. In short its functionality is to start a number of threads and for each thread to contribute equally to allocation of a specified total memory arena. Each thread also completes a number of iterations during which it will allocate its total proportion of the total application memory and deallocate all of its allocated memory before starting another iteration. A small block size is also specified and memory chunks are allocated in 1Kb blocks followed by another 2Kb block fragmented into small block segments of the size specified as command line argument. The latter small block segments are allocated to encourage fragmentation. The exact value of the small block may also vary from run to run between iterations in order to ensure fragmentation.

After completion of a run, the following statistics are reported:

- CPU utilization to complete the run. This is computed by using the `getrusage()` call.
- Real time and CPU time to complete a run
- The total memory arena size after completion of the run. This is computed using the arena size returned by the `mallinfo()` call for the `memtest_hp` binary, while a MTS API call is used to obtain the MTS total heap size after completion of the run.

The `memtest` utility arguments are as follows:

```
Usage: memtestf_ats <threads> <it> <sb> <mem> <sleep> <interval>
```

where:

<threads> is the number of threads to create

<it> is the number of memory allocation iterations per thread

<sb> is the size of the small block to be allocated in bytes

<mem> is the total memory to be allocated across all threads in Mb

<sleep> is the time to sleep between ~3Kb allocations in micro second

<interval> is the number of malloc iterations between sleeps. 0 ==> sleep after every malloc

e.g. `memtest_ats 20 100 200 20 0 50000`

This example run will start 20 threads and will repeat 100 iterations in which each thread will contribute to the allocation of 10Mb (200Mb / 20 threads = 10Mb). The memory will be allocated in 1Kb and ~6-7 times small blocks of up to 200bytes. Each thread will therefore perform ~27000-28000 memory allocation operations per iteration.

Threads are created during start-up processing and these threads are never destroyed.

The test results described in this technical communication were conducted without overriding the default behaviour of the memory management products. In other words, the default arena options, small block and cache options of the HPUX multi-arena allocator were used and the MTS environment setting `MTS_INIT_THREAD_HEAPS` environment variable were not set. This implies that the L-Class server tests were conducted with 2 heaps and the N-Class server tests with 8 heaps in comparison with the default 16 arenas used by the HPUX allocator.

The `memtest` utility generates standardized comma-separated output with the columns as described in the table below:

COLUMN NAME	DESCRIPTION	FORMULA
PROG NAME	The test utility used : memtest_hp – memtest using the HPUX multi-arena allocator memtest_ats – memtest using MTS memory management	-
THREADS	The number of threads contributing to the memory allocation run	-
ITERATIONS	The number of allocation/de-allocation iterations performed	-
SLEEP INTERVAL	The number of 1Kb and small block allocation cycles completed between sleeps	-
SMALL BLOCK	The small allocation block size, e.g. if 200 bytes, ~10 allocations will be done per allocation to complete ~3Kb of allocations	-
SLEEP (us)	The time (in microseconds) to wait before the next 3kB allocation. This is executed at intervals specified by SLEEP INTERVAL.	-

REAL TIME	Real time (in seconds) taken to complete the memtest run.	Lapsed time from start of run to completion of run.
CPU TIME	The CPU time (in seconds) used to complete the memtest run.	From getrusage() call.
CPU	The CPU utilization for run.	From getrusage() call.
ARENA	Total arena size after run (taken after all threads completed their iterations).	MTS: ats_heap_total_size() call HPUX: mallinfo arena
MEM ALLOCATED 1	Total memory allocated across all threads, inclusive of space required to track allocated memory areas. This value is therefore more than the specified total memory allocation.	<mem> command argument value + THREADS X 2Mb
MEM ALLOCATED 2	Total allocated arena size after completion of all thread iterations.	MTS: ats_heap_size() HPUX: mallinfo1.uordblks/nThreads + mallinfo1.usmblks (Note 1)

NOTES:

1. The division of the ordinary blocks by the number of threads appears to yield the correct results (verified with glance and top). The incorrect reporting of unrodblks is documented and HPUX patches are available that resolves this problem (PHCO_27434). This patch was loaded, but does not appear to have resolved the problem.

4 Replacement Memory Management Software Considerations

While the results of this investigation measures the performance and handling of fragmentation of the standard HPUX 11i multi-arena memory management functions by using NewCode MTS as replacement alternative, other products were also considered.

The HOARD replacement library was considered (<http://www.hoard.org>), but was unfortunately not supported on HPUX 11i at the time of this investigation. The library source code was obtained and compiled with the aCC compiler, but would unfortunately not function when using multi-threading. From discussions on the libhoard news groups there seems to be a possibility that this product will be supported on HPUX 11.23 in future.

5 Motivation for this investigation

This study emanated from service degrading incidents experienced on the IBS platform caused by BEA Tuxedo deadlock situation occurring when IBS applications reached the HPUX per-process memory limit (maxdsiz) and dumps core. The memory growth patterns experienced in production could not be explained as both Rational Purify and Linux valgrind runs revealed no memory leaks in the application code and it was therefore suspected that the problems may relate to memory fragmentation issues in the new multi-arena allocator of HPUX introduced with HPUX 11i (11.11).

In order to further illustrate the problem, the HPUX mallinfo call was used to show the growth in ordinary block, small block and free memory in one of the critical IBS application modules. The outcome of this test (conducted on IBS Code Base 2 Build 77) with a mallinfo call and result printout added per transaction. The test was conducted with the TBALANCE business service (GSM subscriber balance enquiries) as implemented at MTN South Africa.

The key points to be highlighted are as follows:

- The application's small block to ordinary block allocation ration is approximately 1.7:1.
- The rate of growth of ordinary blocks allocated is ~54 bytes per transaction.
- The rate of growth of small blocks allocated is ~15 bytes per transaction.
- The rate of growth of free memory is ~155 bytes per transaction.

It must be noted that this application is multi-threaded (POSIX threads) and hence uses the HPUX 11i multi-arena allocation algorithm. None of the arena management environment variables were

set during this test run, i.e. the default values for `_M_ARENA_OPTS`, `_M_SBA_OPTS` and `_M_CACHE_OPTS` were used.

It is clear from the results illustrated in Figure 1, Figure 2 and Figure 3 that the rate of growth of free memory is higher than that of small and ordinary allocated blocks. The results therefore indicate that the HPUX multi-arena allocation algorithm is not ideally suited for the memory allocation and de-allocation patterns of the IBS module tested (the IBS Business Rule Engine).

The net result of the behaviour shown in this diagram is that the process memory consumption grows up to the defined per-process memory limit (`maxdsiz` for 32 bit applications) and then dumps core. The growth rates shown below is also a function of the incoming traffic rate, i.e. the higher the rate, the more aggressive the memory growth pattern.

What is also experienced is that this allocated memory is never released and made available to other processes, i.e. the memory utilization per process shows a high watermark pattern.

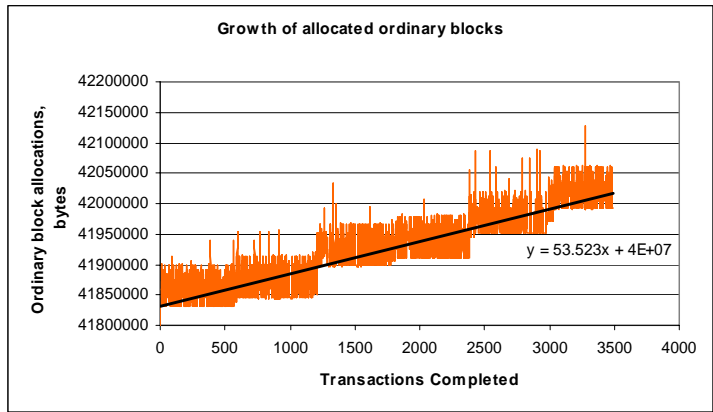


Figure 1 – Growth of allocated ordinary blocks (Rate ~54 bytes/transaction)

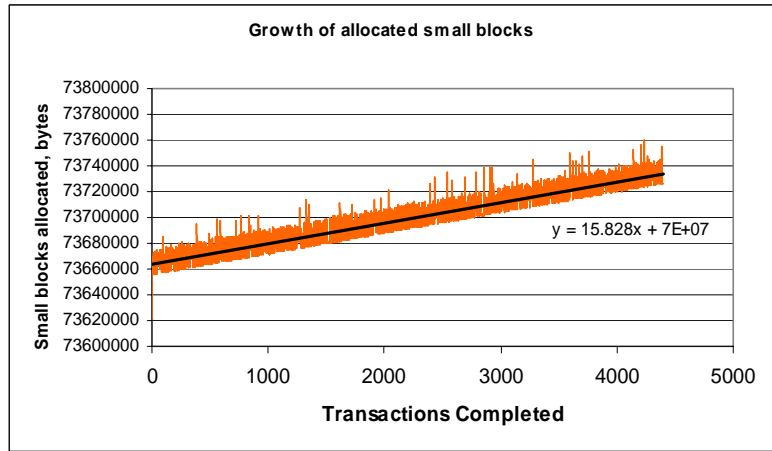


Figure 2 – Growth of small blocks (Rate ~ 15 bytes/transaction)

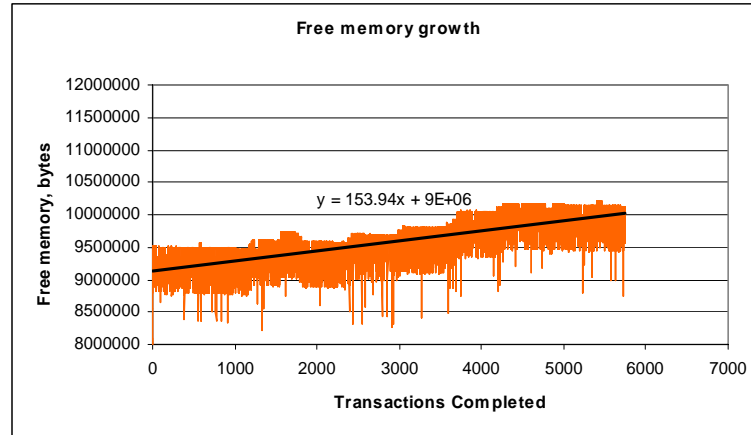


Figure 3 – Rate of free memory growth (Rate ~154 bytes/transaction)

This same application was used to generate a build using the MTS replacement library (libatsfm32.sl) and the same benchmark run was performed. Using this replacement, runs were performed on the same hardware platform (HP L-Class 2 processor server) using the application without the replacement functionality and with the replacement functionality. The BRE process reached the per-process limit of 268Mb after approximately 700 000 transactions and dumped core, while the run with the MTS library was terminated at 1.5 million transactions and showed no memory growth. After completion of the run, some memory (allocated during start-up configuration processing as well as transaction processing) was released by the BRE process running with the MTS library. This result conclusively proves that the module has no application memory leak and that the memory usage pattern experienced with the HPUX 11i multi-arena allocation algorithm is, in fact, as a direct result of the memory management implementation.

6 Extrapolation of results to customer production systems

While the results presented in this white paper provides compelling evidence that the MTS product provides significant memory management performance improvements over the HPUX Multi Arena allocator, careful consideration of the specific application bottlenecks must precede introduction of MTS into production applications.

Once customers are convinced that their application illustrates performance bottlenecks attributable to access to memory, then introduction of MTS may introduce significant performance improvements. Even then, more operating system level changes may have to be introduced before the benefits of MTN will result in production system performance improvements. For example, in this study the results were extrapolated to the IBS production system and no significant performance gains were experienced initially. The cause was that, once access to memory was removed as bottleneck, the Tuxedo based applications suffered from very high forced context switch rates by the default HPUX scheduler as the BEA Tuxedo IPC-based system calls became a bottleneck.

During experimentation it was found that the HPUX RTSCHED (Real-time Scheduler) yielded excellent performance improvements together with MTS for the specific application.

The moral of this story is therefore: Customer can be sure that MTS will ensure optimal use of system memory and remove access to memory and allocation of memory as bottleneck, but test and test again to ensure that the introduction of MTN does not merely lead to the next bottleneck as this may cause even worse performance.

7 Experimental results

7.1 Examination of Fragmentation Issues

A set of experiments were conducted on the L-class server to examine the fragmentation issues observed in production and as described in section 5.

The results are listed in 9 (Appendix) were generated using the memtest utility. The graphs shown in this section were all derived from this result set.

7.1.1 Arena Growth as a function of allocation iterations

The arena growth as a function of memtest iterations is shown for the results listed in Figure 4. It shows the same trend for both allocators, but more aggressive growth of the arena when using the HPUX allocator. The rate of growth when using the MTS allocator must also be seen in the context that the allocation rate is at least double that achieved with the HPUX allocator on a 2 processor HP L-class server.

The finding is therefore that the MTS memory allocator displays more conservative memory allocation scheme while, at the same time, providing superior performance.

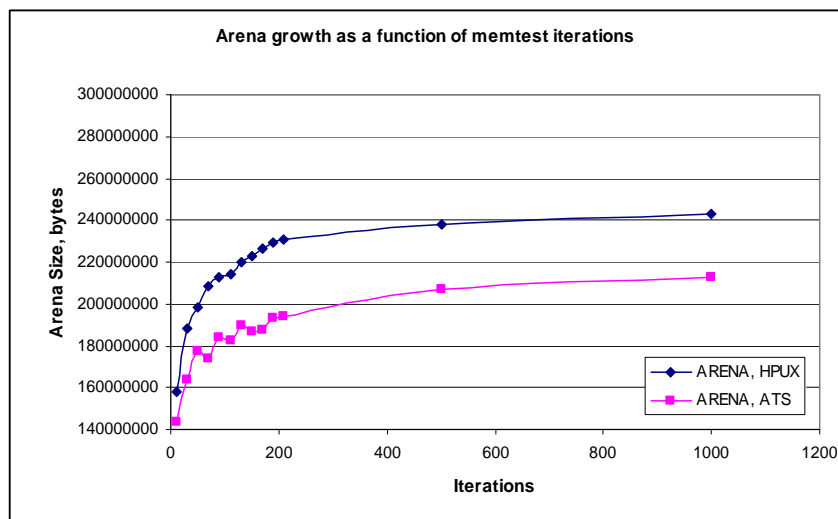


Figure 4 – Arena growth as a function of completed iterations (compiled from results listed in Appendix A)

```

philips@goodwill: /workspace/home/philips
File Edit Settings Help
System: blackbrd Thu Feb 20 13:07:55 2003
Load averages: 46.20, 20.05, 11.54
311 processes: 268 sleeping, 43 running
Cpu states:
CPU LOAD USER NICE SYS IDLE BLOCK SWAIT INTR SSYS
0 54.42 99.6% 0.2% 0.2% 0.0% 0.0% 0.0% 0.0% 0.0%
1 37.98 97.0% 0.0% 3.0% 0.0% 0.0% 0.0% 0.0% 0.0%
-----
avg 46.20 98.4% 0.0% 1.6% 0.0% 0.0% 0.0% 0.0% 0.0%

Memory: 2594928K (788860K) real, 5576924K (1638964K) virtual, 17736K free Page# 1/29

CPU TTY PID USERNAME PRI NI SIZE RES STATE TIME %WCPU %CPU COMMAND
0 pty/ttyp8 4675 philips 152 20 223M 145M run 2:00 181.20 175.45 memtestf_hp
0 pts/ta 3732 marne 152 20 2572K 3676K run 10:07 10.31 10.29 simulator
1 ? 1310 root -16 20 6476K 5848K run 1:50 1.51 1.50 midaemon
1 pty/ttyp1 2841 marne 152 25 11328K 3220K run 2:26 1.05 1.05 t2adapter
0 pty/ttyp1 2832 marne 152 25 6888K 4080K run 0:49 0.78 0.78 DBAdapter
0 pty/ttyp1 2827 marne 152 25 84616K 1144K run 1:18 0.64 0.64 bre
0 ? 4087 marne 152 35 5252K 4924K run 0:40 0.64 0.64 ReadandFwd
0 ? 34 root 152 20 0K 1216K run 0:08 0.64 0.64 vxfsd
1 pty/ttyp1 3863 marne 152 25 117M 115M run 1:21 0.63 0.63 bre
1 pty/ttyp4 4144 marne 152 39 8720K 9336K run 3:12 0.60 0.60 t1adapter
0 pty/ttyp1 2839 marne 152 25 11072K 3132K run 0:43 0.60 0.60 t2adapter
    
```

Figure 5 – memtest_hp run memory usage sample

```

philips@goodwill: /workspace/home/philips
File Edit Settings Help
System: blackbrd Thu Feb 20 13:08:40 2003
Load averages: 60.79, 27.65, 14.63
311 processes: 268 sleeping, 43 running
Cpu states:
CPU LOAD USER NICE SYS IDLE BLOCK SWAIT INTR SSYS
0 68.30 99.8% 0.2% 0.0% 0.0% 0.0% 0.0% 0.0% 0.0%
1 53.28 94.0% 0.6% 5.4% 0.0% 0.0% 0.0% 0.0% 0.0%
-----
avg 60.79 97.0% 0.4% 2.6% 0.0% 0.0% 0.0% 0.0% 0.0%

Memory: 2599464K (724008K) real, 5581436K (1507732K) virtual, 12916K free Page# 1/29

CPU TTY PID USERNAME PRI NI SIZE RES STATE TIME %WCPU %CPU COMMAND
0 pty/ttyp8 4675 philips 152 20 228M 150M run 3:23 185.99 185.37 memtestf_hp
0 pts/ta 3732 marne 152 20 2572K 3572K run 10:11 9.98 9.97 simulator
1 ? 1310 root -16 20 6476K 5848K run 1:52 1.36 1.36 midaemon
1 pty/ttyp1 2829 marne 152 25 10432K 2888K run 0:47 1.05 1.05 t2adapter
1 pty/ttyp1 3863 marne 152 25 117M 115M run 1:21 0.66 0.66 bre
0 ? 34 root 152 20 0K 1216K run 0:08 0.63 0.62 vxfsd
1 pty/ttyp1 2841 marne 152 25 11328K 3220K run 2:26 0.58 0.58 t2adapter
0 pty/ttyp1 2839 marne 152 25 11072K 3132K run 0:44 0.57 0.57 t2adapter
1 pts/1 4490 philips 178 20 5092K 736K run 0:01 0.56 0.56 top
0 pty/ttyp1 2823 marne 152 25 144M 1336K run 1:34 0.52 0.52 bre
0 pty/ttyp1 2827 marne 152 25 84616K 1144K run 1:18 0.52 0.52 bre
    
```

Figure 6 – memtestf_hp memory usage sample (one minute later than that showed in Figure 5 to highlight memory growth)

7.2 Apparent instability with sharing of HPUX arenas across threads

A peculiarity that was observed during the testing on the L-Class platform is described in this section. This seems to point to a serious error in the HPUX multi arena allocator sharing of arenas across threads.

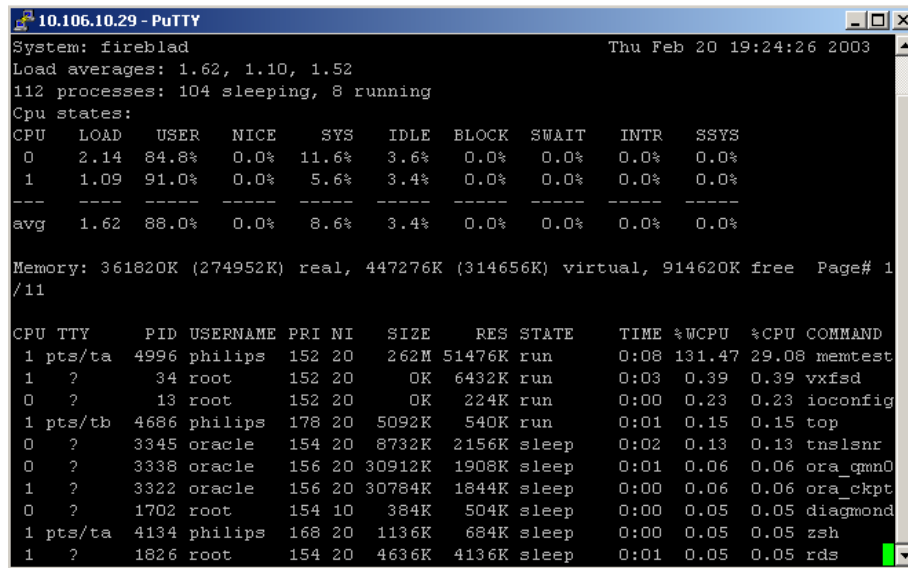
Using the default arena settings (i.e. `_M_ARENA_OPTS=8:32` – 8 arenas, 32 expansion factor) on HPUX, two memtest runs were conducted:

```
memtestf_hp 28 10 200 10 20 0 50000
memtestf_hp 32 10 200 10 20 0 50000
```

The output from top is shown in Figure 7 and Figure 8 respectively. The 28 thread run completes with 51Mb allocated which represents approximately 25% overhead. The 32 thread run does not complete and dumps core. The output in Figure 8 shows that the memory consumption grew to 262Mb, the configured per process limit on the test platform and the application dumps core. This test was repeated several times and the findings were confirmed to be repeatable. It was also confirmed that this did not occur with the `memtest_at`s utility, which is 100% similar code with the only variance being the MTS allocator usage.

The test was then repeated by exporting a `_M_ARENA_OPTS` with more arenas than the number of threads (export `_M_ARENA_OPTS=40:1` was used). The 32-thread run succeeded and the same behaviour was not observed.

The conclusion drawn from this is that the HPUX multi-arena allocator does not handle sharing of arenas across threads correctly. This finding should be further investigated by HP labs.



```

10.106.10.29 - PuTTY
System: fireblad                               Thu Feb 20 19:24:26 2003
Load averages: 1.62, 1.10, 1.52
112 processes: 104 sleeping, 8 running
Cpu states:
CPU  LOAD   USER   NICE   SYS  IDLE  BLOCK  SWAIT  INTR  SSYS
 0    2.14  84.8%  0.0%  11.6% 3.6%  0.0%  0.0%  0.0%  0.0%
 1    1.09  91.0%  0.0%   5.6% 3.4%  0.0%  0.0%  0.0%  0.0%
---  ---
avg   1.62  88.0%  0.0%   8.6% 3.4%  0.0%  0.0%  0.0%  0.0%

Memory: 361820K (274952K) real, 447276K (314656K) virtual, 914620K free Page# 1
/11

CPU TTY      PID USERNAME PRI NI   SIZE  RES STATE  TIME %WCPU  %CPU COMMAND
 1 pts/ta    4996 philips  152 20  262M 51476K run    0:08 131.47 29.08 memtest
 1 ?         34 root      152 20   OK  6432K run    0:03  0.39  0.39 vxfsd
 0 ?         13 root      152 20   OK  224K  run    0:00  0.23  0.23 ioconfig
 1 pts/tb    4686 philips  178 20  5092K 540K  run    0:01  0.15  0.15 top
 0 ?         3345 oracle   154 20  8732K 2156K sleep  0:02  0.13  0.13 tnslsnr
 0 ?         3338 oracle   156 20 30912K 1908K sleep  0:01  0.06  0.06 ora_qmn0
 1 ?         3322 oracle   156 20 30784K 1844K sleep  0:00  0.06  0.06 ora_ckpt
 0 ?         1702 root      154 10   384K  504K  sleep  0:00  0.05  0.05 diagmond
 1 pts/ta    4134 philips  168 20  1136K  684K  sleep  0:00  0.05  0.05 zsh
 1 ?         1826 root      154 20  4636K 4136K  sleep  0:01  0.05  0.05 rds

```

Figure 7 - Top result for memtest_hp run with 28 threads (memtest_hp 28 10 200 10 20 0 50000)

```

10.106.10.29 - PuTTY
System: fireblad                               Thu Feb 20 19:42:59 2003
Load averages: 4.22, 3.18, 2.64
113 processes: 105 sleeping, 8 running
Cpu states:
CPU  LOAD   USER   NICE   SYS   IDLE   BLOCK  SWAIT   INTR   SSYS
0    3.82   1.8%   0.0%   1.2%  97.0%  0.0%   0.0%   0.0%   0.0%
1    4.62   0.4%   0.0%   23.1% 76.5%  0.0%   0.0%   0.0%   0.0%
---  ---
avg  4.22   1.2%   0.0%   12.2% 86.7%  0.0%   0.0%   0.0%   0.0%

Memory: 537224K (463776K) real, 716016K (609888K) virtual, 712164K free Page# 1
/11

CPU TTY  PID USERNAME PRI NI  SIZE  RES STATE  TIME %WCPU %CPU COMMAND
0 pts/tb 5346 philips 152 20 262M 263M run   0:04 45.27 19.15 memtestf
0 ?      34 root      152 20  OK 6432K run   0:04 0.45 0.45 vxfsd
0 ?      5350 oracle   154 20 30784K 1844K sleep 0:00 9.01 0.44 oracleIB
0 ?      13 root      152 20  OK 224K run   0:00 0.23 0.23 iocconfig
0 pts/ta 5265 philips 178 20 5092K 540K run   0:00 0.16 0.16 top
1 ?      3345 oracle   154 20 8732K 2156K sleep 0:03 0.15 0.15 tnslnsr
0 ?      20 root      147 20  OK 32K sleep 0:00 0.10 0.10 lvmkd
1 ?      1826 root     154 20 4636K 4136K sleep 0:01 0.08 0.08 rds
0 ?      23 root      147 20  OK 32K sleep 0:00 0.06 0.06 lvmkd
0 ?      2230 root     154 10 740K 952K sleep 0:01 0.06 0.06 psmctd
    
```

Figure 8 – Top result for memtest_hp run with 32 threads (memtest_hp 32 10 200 10 20 0 50000)

7.3 Performance Results

7.3.1 Performance as a function the number of threads per process

Several memtest runs were conducted on an idle 8-way N-Class server equipped with 10Gb RAM as well as a 2-way L-Class server equipped with 2Gb RAM and both servers configured with a per-process memory limit of 268Mb (maxdsize). The key findings from these tests are summarized below.

A memtest run was performed with the default HPUX memory allocator and the MTS replacement library. The results are illustrated in Figure 9 and Figure 10.

The results clearly indicate that there is no degradation of performance in the range of threads tested for the MTS allocator while the HPUX allocator results in degradation of performance as the thread count is increased. This is even true for scenarios where 1 arena is made available per thread (Figure 9).

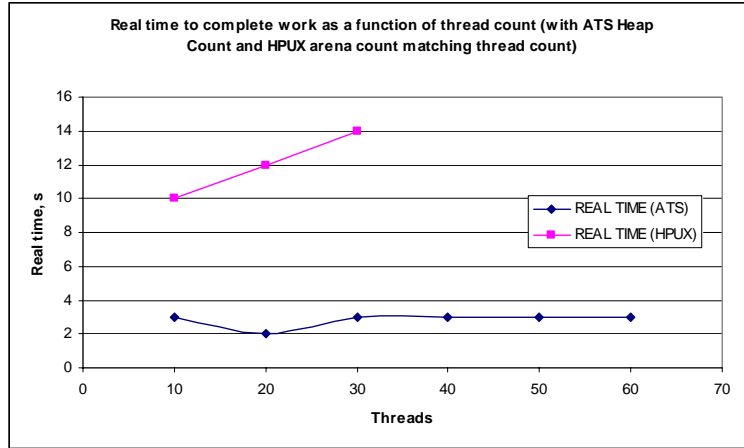


Figure 9 – Time to complete memtest runs for 10-60 threads and 200byte small blocks (L-Class - Compiled from Table 5)

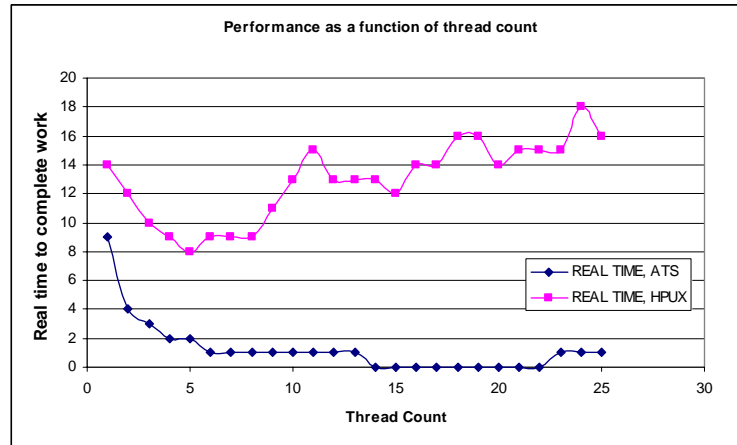


Figure 10 - Time to complete memtest runs for 1-25 threads and 200byte small blocks (N-Class Compiled from Table 5)

7.3.2 The effect of varying small blocks

A memtest run was conducted with small blocks sizes ranging from 200 bytes to 2Kb. The results conducted on the N-Class server clearly indicates that the number and size of allocations has a marked effect on the performance of the HPUX allocator, while the MTS library shows no significant degradation unless the block size is less than 200 bytes. Also observed was the fact that there is significant degradation of performance at high small block counts when using the HPUX allocator, while the same behaviour is not observed for the MTS allocator. As a general rule, performance gains of ~800% are observed throughout and therefore illustrates that the MTS claims to reduce heap contention with the MTS library are well founded.

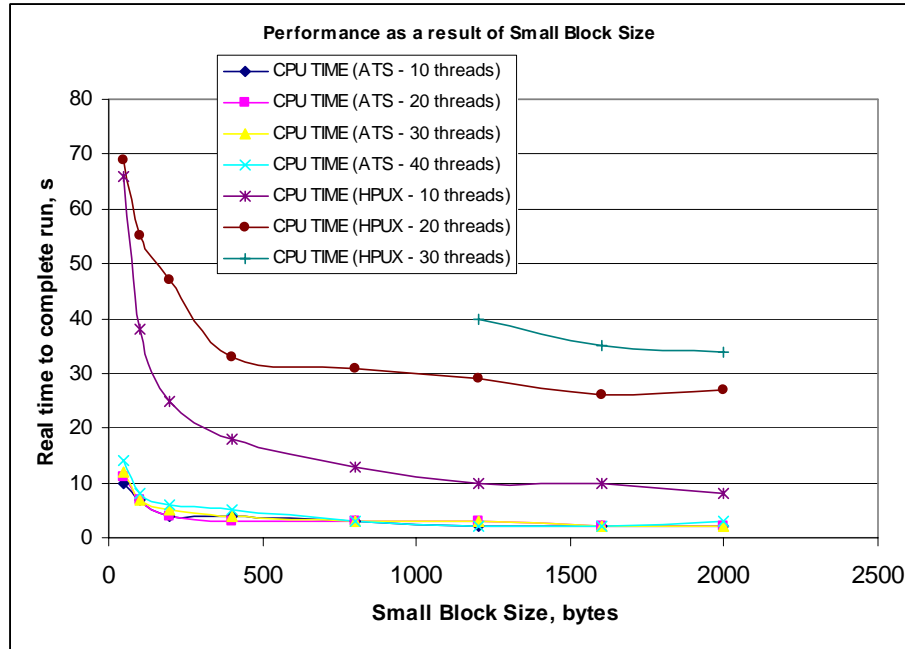


Figure 11 – Performance as a function of the small block size (Compiled from Table 1)

8 TECHNICAL RECOMMENDATIONS

From the results presented the following is evident:

- The NewCode MTS memory management library performs factors better than the HPUX allocator (Performance gains of roughly the CPU count can be expected, i.e. 200% on a 2 processor sever and 800% on a 8 processor server);
- The NewCode MTS memory management library scales better with increased thread counts;
- The NewCode MTS memory management library deals satisfactorily with fragmentation issues;
- The HPUX memory allocator displayed some instability at the HPUX level tested with as the thread count increased. This relates to unexpected memory consumption once certain thread count thresholds are exceeded.

With the real-time processing demands placed on the IBS system it is recommended that the feasibility of MTS introduction into IBS as an option be considered and the actual performance gains be assessed. It is anticipated that huge performance gains may be achieved by replacing the HPUX multi-arena allocator functions with the NewCode MTS allocator. The MTS library is fully supported on the HPUX operating system and its cost should be measured at the cost of buying additional memory required to reduce the per-process-memory limit reached when using the standard HPUX allocator for the IBS application.

9 APPENDIX A – FRAGMENTATION TEST RUN RESULTS

Conditions:

HPUX L-Class Server Runs

memtest_<allocator> 10 n 200 20 0 50000 where n is a varying number of iterations

Summary:

This run examines the memory management behaviour under conditions of extremely aggressive memory allocation rates using a medium sized small block.

Table 1 – Fragmentation Test Run 1 output

PROG NAME	THREADS	ITERATIONS	SLEEP INTERV	SMALLBLOCK	SLEEP(us)	REAL TIME	CPU TIME	CPU	ARENA	MEM ALLOCATED1	MEM ALLOCATED2
memtestf_hp	10	10	50000	200	0	4	7	184.5	157793404	40971520	73729434
memtestf_hp	10	30	50000	200	0	13	24	187.54	188202108	40971520	95263833
memtestf_hp	10	50	50000	200	0	21	40	190.67	198163580	40971520	95410003
memtestf_hp	10	70	50000	200	0	30	59	198.6	208125052	40971520	104611356
memtestf_hp	10	90	50000	200	0	40	80	200.07	212581500	40971520	98633262
memtestf_hp	10	110	50000	200	0	50	98	196.7	213892220	40971520	114295578
memtestf_hp	10	130	50000	200	0	55	107	195.89	220314748	40971520	111282668
memtestf_hp	10	150	50000	200	0	64	126	197.16	223067260	40971520	115880726
memtestf_hp	10	170	50000	200	0	80	158	197.56	226212988	40971520	112985909
memtestf_hp	10	190	50000	200	0	80	159	198.93	229620860	40971520	110352292
memtestf_hp	10	210	50000	200	0	91	179	196.86	230931580	40971520	114926348
memtestf_at	10	10	50000	200	0	3	6	219	143802368	40971520	34832384
memtestf_at	10	30	50000	200	0	11	16	150.73	164044800	40971520	34635776
memtestf_at	10	50	50000	200	0	9	18	205.67	177356800	40971520	40177664
memtestf_at	10	70	50000	200	0	13	26	202.38	174080000	40971520	35901440
memtestf_at	10	90	50000	200	0	19	36	192.05	184311808	40971520	38916096
memtestf_at	10	110	50000	200	0	22	41	188.86	182378496	40971520	40198144
memtestf_at	10	130	50000	200	0	25	49	199.64	189530112	40971520	41455616
memtestf_at	10	150	50000	200	0	29	57	196.66	186548224	40971520	37928960
memtestf_at	10	170	50000	200	0	33	64	195.06	187359232	40971520	40300544
memtestf_at	10	190	50000	200	0	38	73	193.42	193536000	40971520	40251392
memtestf_at	10	210	50000	200	0	41	81	198.44	194297856	40971520	43454464
memtestf_at	10	500	50000	200	0	100	197	197.72	206970880	40971520	43380736
memtestf_hp	10	500	50000	200	0	256	500	195.43	238140540	40971520	118649987
memtestf_at	10	1000	50000	200	0	194	378	195.15	213073920	40971520	46120960
memtestf_hp	10	1000	50000	200	0	544	1031	189.65	243383420	40971520	130735798

10 APPENDIX B – PERFORMANCE TEST RUN RESULTS

Conditions:

HPUX N-Class Server Runs

Summary:

This run examines the performance of the HPUX multi-arena allocator and the MTS allocator

Table 2 – Performance Test Runs Conducted on 8-processor N-Class Server with default multi-arena allocator and MTS settings

PROG NAME	THREADS	ITERATIONS	SLEEP INTERVAL	SMALLBLOCK	SLEEP(us)	REAL TIME	CPU TIME	CPU	ARENA	MEM ALLOCATED1	MEM ALLOCATED2
memtestf_ats	10	100	500000	20	0	23	135	588.22	131104768	40971520	14151680
memtestf_ats	10	100	500000	50	0	10	65	650.7	151298048	40971520	15466496
memtestf_ats	10	100	500000	100	0	7	43	623.29	190287872	40971520	19578880
memtestf_ats	10	100	500000	200	0	4	28	715.25	257232896	40971520	26767360
memtestf_ats	10	100	500000	400	0	4	25	638.5	176869376	40971520	18337792
memtestf_ats	10	100	500000	800	0	3	18	609.33	110907392	40971520	11591680
memtestf_ats	10	100	500000	1200	0	2	16	823	113790976	40971520	11587584
memtestf_ats	10	100	500000	1600	0	2	15	768.5	113840128	40971520	11608064
memtestf_ats	10	100	500000	2000	0	2	13	667	112242688	40971520	11616256
memtestf_ats	20	100	500000	50	0	11	79	727.09	223789056	60971520	23887872
memtestf_ats	20	100	500000	100	0	7	53	768.29	252215296	60971520	25477120
memtestf_ats	20	100	500000	200	0	4	32	803.75	295677952	60971520	31830016
memtestf_ats	20	100	500000	400	0	3	26	877	238231552	60971520	24113152
memtestf_ats	20	100	500000	800	0	3	22	752	193880064	60971520	19648512
memtestf_ats	20	100	500000	1200	0	3	18	616.33	193933312	60971520	19648512
memtestf_ats	20	100	500000	1600	0	2	17	889	193937408	60971520	19648512
memtestf_ats	20	100	500000	2000	0	2	15	781	193896448	60971520	19648512
memtestf_ats	30	100	500000	50	0	12	94	786.75	299044864	80971520	31002624
memtestf_ats	30	100	500000	100	0	7	55	798	316141568	80971520	32370688
memtestf_ats	30	100	500000	200	0	5	38	773.2	351920128	80971520	37625856
memtestf_ats	30	100	500000	400	0	4	32	802.75	312537088	80971520	32428032
memtestf_ats	30	100	500000	800	0	3	23	785.67	289366016	80971520	29122560
memtestf_ats	30	100	500000	1200	0	3	21	718.67	289357824	80971520	29106176
memtestf_ats	30	100	500000	1600	0	2	18	906.5	289275904	80971520	29085696
memtestf_ats	30	100	500000	2000	0	2	19	959	289386496	80971520	29143040
memtestf_ats	40	100	500000	50	0	14	108	778.21	386596864	100971520	48271360
memtestf_ats	40	100	500000	100	0	8	56	706.5	386723840	100971520	48553984
memtestf_ats	40	100	500000	200	0	6	51	866.17	409346048	100971520	53891072
memtestf_ats	40	100	500000	400	0	5	37	758.6	388362240	100971520	48607232
memtestf_ats	40	100	500000	800	0	3	23	767.67	386625536	100971520	48574464
memtestf_ats	40	100	500000	1200	0	2	17	877.5	386691072	100971520	48570368
memtestf_ats	40	100	500000	1600	0	2	16	832.5	386629632	100971520	48549888
memtestf_ats	40	100	500000	2000	0	3	19	645.67	386543616	100971520	48627712
memtestf_hp	10	100	500000	20	0	151	784	519.86	129219708	40971520	63620514
memtestf_hp	10	100	500000	50	0	66	333	505.35	147438716	40971520	71256372
memtestf_hp	10	100	500000	100	0	38	185	488.08	168672380	40971520	86718056
memtestf_hp	10	100	500000	200	0	25	112	448.2	199343228	40971520	95958712
memtestf_hp	10	100	500000	400	0	18	76	425.94	237354108	40971520	111359840
memtestf_hp	10	100	500000	800	0	13	55	425.85	170245244	40971520	95056712
memtestf_hp	10	100	500000	1200	0	10	46	464.1	108117116	40971520	55892964
memtestf_hp	10	100	500000	1600	0	10	42	422.9	110476412	40971520	58565222
memtestf_hp	10	100	500000	2000	0	8	38	483.38	108641404	40971520	56599086
memtestf_hp	20	100	500000	50	0	69	253	367.16	248892540	60971520	-80388025
memtestf_hp	20	100	500000	100	0	55	180	327.44	268422268	60971520	-82938284
memtestf_hp	20	100	500000	200	0	47	142	303.85	268422268	60971520	-72820877
memtestf_hp	20	100	500000	400	0	33	104	317.85	268422268	60971520	-83412923
memtestf_hp	20	100	500000	800	0	31	87	283.39	246971516	60971520	-66209521
memtestf_hp	20	100	500000	1200	0	29	82	286.17	188705916	60971520	-104987101
memtestf_hp	20	100	500000	1600	0	26	76	293.46	189791356	60971520	-92023849
memtestf_hp	20	100	500000	2000	0	27	76	283.33	191487100	60971520	-94082850
memtestf_hp	30	100	500000	50	0						
memtestf_hp	30	100	500000	100	0						
memtestf_hp	30	100	500000	200	0						
memtestf_hp	30	100	500000	400	0						
memtestf_hp	30	100	500000	800	0						
memtestf_hp	30	100	500000	1200	0	40	106	266.15	268422268	80971520	10873250
memtestf_hp	30	100	500000	1600	0	35	96	276.6	267971708	80971520	6235646
memtestf_hp	30	100	500000	2000	0	34	94	277.35	268422268	80971520	16133598
memtestf_hp	40	100	500000	50	0						
memtestf_hp	40	100	500000	100	0						
memtestf_hp	40	100	500000	200	0						
memtestf_hp	40	100	500000	400	0						
memtestf_hp	40	100	500000	800	0						
memtestf_hp	40	100	500000	1200	0						
memtestf_hp	40	100	500000	1600	0						
memtestf_hp	40	100	500000	2000	0						
memtestf_hp	40	100	500000	2000	0	Core dump					

Table 3 – Detailed Performance Analysis as a function of the number of threads (N-Class Server with default multi-arena allocator and MTS settings)

PROG NAME	THREADS	ITERATIONS	SLEEP INTERVAL	SMALLBLOCK	SLEEP(us)	REAL TIME	CPU TIME	CPU	ARENA	MEM ALLOCATED1	MEM ALLOCATED2
memtestf_ats	1	100	500000	200	0	9	17	191.22	69193728	12485760	1060864
memtestf_ats	2	100	500000	200	0	4	13	330	93945856	14485760	1060864
memtestf_ats	3	100	500000	200	0	3	11	394.67	118210560	16485760	1060864
memtestf_ats	4	100	500000	200	0	2	10	548.5	126042112	18485760	1060864
memtestf_ats	5	100	500000	200	0	2	9	489	134049792	20485760	1060864
memtestf_ats	6	100	500000	200	0	1	9	931	146898944	22485760	1060864
memtestf_ats	7	100	500000	200	0	1	8	870	154660864	24485760	1060864
memtestf_ats	8	100	500000	200	0	1	8	851	166526976	26485760	1060864
memtestf_ats	9	100	500000	200	0	1	7	786	174108672	28485760	1060864
memtestf_ats	10	100	500000	200	0	1	7	769	184299520	30485760	1060864
memtestf_ats	11	100	500000	200	0	1	7	752	190459904	32485760	1060864
memtestf_ats	12	100	500000	200	0	1	7	704	199348224	34485760	1060864
memtestf_ats	13	100	500000	200	0	1	7	704	206450688	36485760	1060864
memtestf_ats	14	100	500000	200	0	0	6	inf	212258816	38485760	1060864
memtestf_ats	15	100	500000	200	0	0	6	inf	221102080	40485760	1060864
memtestf_ats	16	100	500000	200	0	0	6	inf	235311104	42485760	1060864
memtestf_ats	17	100	500000	200	0	0	6	inf	241528832	44485760	1060864
memtestf_ats	18	100	500000	200	0	0	6	inf	248623104	46485760	1060864
memtestf_ats	19	100	500000	200	0	0	6	inf	260399104	48485760	1060864
memtestf_ats	20	100	500000	200	0	0	6	inf	268730368	50485760	1060864
memtestf_ats	21	100	500000	200	0	0	6	inf	285253632	52485760	1060864
memtestf_ats	22	100	500000	200	0	0	6	inf	291147776	54485760	1060864
memtestf_ats	23	100	500000	200	0	1	6	646	304386048	56485760	1060864
memtestf_ats	24	100	500000	200	0	1	6	639	306536448	58485760	1060864
memtestf_ats	25	100	500000	200	0	1	6	641	314781696	60485760	1060864
memtestf_hp	1	100	500000	200	0	14	27	196.36	52804732	12485760	32216
memtestf_hp	2	100	500000	200	0	12	32	267.75	75742332	14485760	22890156
memtestf_hp	3	100	500000	200	0	10	33	332.7	98286716	16485760	48949669
memtestf_hp	4	100	500000	200	0	9	32	363.44	112573564	18485760	56177484
memtestf_hp	5	100	500000	200	0	8	35	437.75	123321468	20485760	56008782
memtestf_hp	6	100	500000	200	0	9	37	413.56	131841148	22485760	61944862
memtestf_hp	7	100	500000	200	0	9	40	446.44	137346172	24485760	69500545
memtestf_hp	8	100	500000	200	0	9	39	439.22	149535868	26485760	82603649
memtestf_hp	9	100	500000	200	0	11	47	428	155434108	28485760	74753541
memtestf_hp	10	100	500000	200	0	13	59	460.15	162249852	30485760	75943088
memtestf_hp	11	100	500000	200	0	15	74	493.8	163429500	32485760	70601050
memtestf_hp	12	100	500000	200	0	13	67	520.31	168803452	34485760	72440272
memtestf_hp	13	100	500000	200	0	13	65	501.54	173128828	36485760	66849358
memtestf_hp	14	100	500000	200	0	13	66	512.08	179420284	38485760	68160876
memtestf_hp	15	100	500000	200	0	12	62	521.25	190561404	40485760	67531952
memtestf_hp	16	100	500000	200	0	14	68	491.71	195673212	42485760	63435046
memtestf_hp	17	100	500000	200	0	14	74	532	202751100	44485760	64546981
memtestf_hp	18	100	500000	200	0	16	82	514.69	206027900	46485760	63299451
memtestf_hp	19	100	500000	200	0	16	82	515.69	212057212	48485760	62058701
memtestf_hp	20	100	500000	200	0	14	78	557.21	222018684	50485760	63453808
memtestf_hp	21	100	500000	200	0	15	80	537.87	225819772	52485760	62760411
memtestf_hp	22	100	500000	200	0	15	76	509.53	233946236	54485760	62500294
memtestf_hp	23	100	500000	200	0	15	78	520.6	241548412	56485760	61715394
memtestf_hp	24	100	500000	200	0	18	88	494.33	249019516	58485760	61089470
memtestf_hp	25	100	500000	200	0	16	86	543.44	258063484	60485760	61802739

Table 4 – Performance as a function of thread count (L-Class Server – MTS: 2 Heaps, HPUX – 8 Arenas)

PROG NAME	THREADS	ITERATIONS	SLEEP INTERVAL	SMALLBLOCK	SLEEP(us)	REAL TIME	CPU TIME	CPU	ARENA	MEM ALLOCATED1	MEM ALLOCATED2
/memtestf_atl	10	100	50000	200	0	10	17	179.9	139898880	30485760	28385280
/memtestf_atl	20	100	50000	200	0	8	16	202.62	195698688	50485760	49704960
/memtestf_atl	30	100	50000	200	0	8	15	194.12	267943936	70485760	57827328
/memtestf_atl	40	100	50000	200	0	15	28	188	372109312	90485760	95014912
/memtestf_hp	10	100	50000	200	0	21	41	196.9	160545916	30485760	77766471
/memtestf_hp	20	100	50000	200	0	26	50	193.04	217169020	50485760	56724708
/memtestf_hp	30	100	50000	200	0	27	51	192.52	268418172	70485760	55603444

Table 5 - Performance as a function of thread count (L-Class Server – MTS: Heap count matching thread count, HPUX – Arena Count matching thread count)

PROG NAME	THREADS	ITERATIONS	SLEEP INTERVAL	SMALLBLOCK	SLEEP(us)	REAL TIME	CPU TIME	CPU	ARENA	MEM ALLOCATED1	MEM ALLOCATED2
/memtestf_atl	10	100	50000	200	0	3	6	202	131772416	25242880	12935168
/memtestf_atl	20	100	50000	200	0	2	4	245	196476928	45242880	19767296
/memtestf_atl	30	100	50000	200	0	3	4	148	282345472	65242880	29343744
/memtestf_atl	40	100	50000	200	0	3	5	195.33	369647616	85242880	39124992
/memtestf_atl	50	100	50000	200	0	3	6	222	488640512	105242880	49242112
/memtestf_atl	60	100	50000	200	0	3	5	195.33	428843008	125242880	48902144
/memtestf_hp	10	100	50000	200	0	10	18	187.1	130530428	25242880	67137568
/memtestf_hp	20	100	50000	200	0	12	22	191.33	195148924	45242880	66450859
/memtestf_hp	30	100	50000	200	0	14	26	192.14	268418172	65242880	66409723

11 APPENDIX C – MEMTEST SOURCE CODE LISTING

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <malloc.h>
#include <pthread.h>
#include <sys/resource.h>
#include <time.h>
#include <sys/mpctl.h>

#ifdef __MTS__
#include <MTS2/ats.h>
#endif

// Some globals
int nThreads = 0; // Number of threads to run
int nIterations = 10; // Number of memory allocations/deallocation cycles to run
int smallBlock = 100; // Small block size
int id[1000]; // Thread IDs
int done[1000]; // Thread completion indicators
#define true 1
#define false 0
long cpuUsage[1000]; // CPU usage reporting using getrusage
int totalMem = 100;
long sleepTime = 0;
long sleepIv = 0;
long wantedMem = 100*1024*1024;

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER; // Mutex

/*-----*\
 * METHOD/FUNCTION : Thread
 * DESCRIPTION : Runs for n = nIterations iterations and in each
 * iteration does the following:
 * - Contributes to allocating of memory
 * - De-allocates this memory
 * - Reports on time taken & CPU usage
 *-----*/
void* Thread(void *arg)
{
    char *chPtr;
    char *chPtr1;
    long rnd;
    struct mallinfo mallinfo1;
    struct mallinfo mallinfo2;
    struct timespec interval, remainder;
    long arena1 = 0;
    long arena2 = 0;
    long memUsed1 = 0;
    long memUsed2 = 0;
    long totalAllocated = 0;
    long differentialArena = 0;
    long differentialMem = 0;
    int malCount = 0;
    int malCount1 = 0;
    int j,i;
    int k;
    time_t time2;
    unsigned long val1, val2, val3;
    struct rusage rusg1;
    struct rusage rusg2;
```

```

        struct timeval timeVal;
        time_t time1 = time(NULL);
        int id = *((int *) arg);
        char ** malPtrs;
        done[id] = false;
        int sBlock = smallBlock;
        cpuUsage[id] = 0;
        long slTime = sleepTime * 1000; // Sleep time in nanoseconds
        //fprintf(stdout, "In Thread %d\n", id);
        wantedMem = totalMem * 1024 * 1024;
        malPtrs = (char **) malloc(2000000*sizeof(char *));

        for (j=0; j<nIterations; j++) {
            //fprintf(stdout, "Start of iteration %d\n", j+1);
            mallinfo1 = mallinfo();
            FILE *fPtr = fopen("./memstats.csv", "a");
            fprintf(fPtr, "%03d,B,%ld,%ld,%ld,%ld,%ld,%ld\n",
                    id, mallinfo1.arena,
                    mallinfo1.uordblks, mallinfo1.usmblks,
                    mallinfo1.fordblks, mallinfo1.fsmblocks,
                    mallinfo1.hblkhd);
            fclose(fPtr);

            i =0;
            totalAllocated = 0;
            while (totalAllocated < wantedMem/nThreads) {
                //mallinfo1 = mallinfo();

                chPtr = (char *) malloc(1024*sizeof(char));
                memset(chPtr, ' ', 1024);
                malPtrs[malCount] = chPtr;
                malCount++;
                malCount1++;
                totalAllocated = totalAllocated + 1024;
                // We wanted to use random - but its to expensive on HPUX - so
                // rather used gettimeofday() to get slight variance in the
                // small block size
                // rnd = random();
                gettimeofday(&timeVal,NULL);
                rnd= timeVal.tv_usec;
                while (rnd > sBlock) rnd =rnd/2;
                if (rnd == 0) rnd = sBlock;
                for (k=0; k<(2048/sBlock); k++) {
                    chPtr1 = (char *) malloc(rnd*sizeof(char));
                    memset(chPtr1, ' ', rnd);
                    malPtrs[malCount] = chPtr1;
                    malCount++;
                    malCount1++;
                    totalAllocated = totalAllocated + rnd;
                } // End while
            #ifdef __TIME_OF_DAY__
                gettimeofday(&timeVal, NULL);
            #endif
            #endif

            if (slTime > 0 && malCount1 >= sleepIv) {
                interval.tv_sec = 0;
                interval.tv_nsec = slTime;
                nanosleep(&interval, &remainder);
                malCount1 = 0;
            } // End if
            i++;
        } // End for

        mallinfo2 = mallinfo();
        fPtr = fopen("./memstats.csv", "a");
        fprintf(fPtr, "%03d,A,%ld,%ld,%ld,%ld,%ld,%ld\n",
                id, totalAllocated, mallinfo2.arena,
                mallinfo2.uordblks, mallinfo2.usmblks,
                mallinfo2.fordblks, mallinfo2.fsmblocks,

```

```

        mallinfo2.hblkhd);
fclose(fPtr);
for (k=0; k< malCount; k++) free(malPtrs[k]);
    malCount=0;
    getrusage(RUSAGE_CHILDREN, &rusg2);
    val1 = + rusg2.ru_utime.tv_sec*1000000 + rusg2.ru_utime.tv_usec;
    val2 = rusg2.ru_stime.tv_sec*1000000 + rusg2.ru_stime.tv_usec;
    val3 = val1+val2;
    cpuUsage[id] = val3;
    time2 = time(NULL);
//      fprintf(stdout, "THREAD=%d ITER=%d USER=%ld SYS=%ld TOTAL=%ld
CTX(V)=%ld, CTX(F)=%ld ABS TIME=%ld\n", id, j+1, val1, val2, val3,
rusg1.ru_nivcsw+rusg2.ru_nivcsw, rusg1.ru_nivcsw+rusg2.ru_nivcsw, time2-time1);
    } // End for

    // Free the array of pointers to allocated areas
free(malPtrs);

// Indicate that I am done
pthread_mutex_lock(&mutex1);
done[id] = true;
pthread_mutex_unlock(&mutex1);

return (NULL);

} // Thread

/*-----*\
 * METHOD/FUNCTION : Exec
 * DESCRIPTION    : For single-threaded runs - not used for white paper
 *
\*-----*/
void Exec()
{
    char *chPtr;
    char *chPtr1;
    long rnd;
    struct mallinfo mallinfo1;
    struct mallinfo mallinfo2;
    long arena1= 0;
    long arena2= 0;
    long memUsed1= 0;
    long memUsed2= 0;
    long totalAllocated = 0;
    long differentialArena = 0;
    long differentialMem = 0;
    int id = 1;
    int i;
    for (i=0; i<100000; i++) {
//      mallinfo1 = mallinfo();
//      arena1 = mallinfo1.arena;
//      memUsed1 = mallinfo1.uordblks + mallinfo1.usmblks;
        chPtr = (char *) malloc(1024*sizeof(char));
        totalAllocated = totalAllocated + 1024;
        rnd = random();
        while (rnd > 2048) rnd =rnd/10;
        chPtr1 = (char *) malloc(rnd*sizeof(char));
        totalAllocated = totalAllocated + rnd;
//      mallinfo2 = mallinfo();
//      arena2 = mallinfo2.arena;
//      memUsed2 = mallinfo2.uordblks + mallinfo2.usmblks;
//      differentialArena = differentialArena + (arena2-arena1);
//      differentialMem = differentialMem + (memUsed2-memUsed1);
        fprintf(stdout, "%02d,%010ld,%010ld,%010ld\n",
//      id, totalAllocated, differentialArena, differentialMem);
        usleep(100);
    }
    while (1) usleep(10000);
}

```

```
        return;
    } // Exec

/*-----*\
 * METHOD/FUNCTION : main
 * DESCRIPTION    : Multi-threaded memory test
 *
 *-----*/

int main(int argc, char **argv)
{
    pthread_t thrd[1000];
    time_t time1 = time(NULL);
    time_t time2;
    int allDone;
    struct rusage rusg1;
    struct rusage rusg2;
    long val1, val2, val3;
    int i;
    malloc(1);

    if (argc != 7) {
        fprintf(stdout,
            "Usage: %s <threads> <it> <sb> <mem> <sleep> <interval>\n",
argv[0]);
        fprintf(stdout, "   where:\n");
        fprintf(stdout, "       <threads> is the number of threads to
create\n");
        fprintf(stdout, "       <it> is the number of memory allocation
iterations per thread\n");
        fprintf(stdout, "       <sb> is the size of the small block to be
allocated in bytes\n");
        fprintf(stdout, "       <mem> is the total memory to be allocated
across all threads in Mb\n");
        fprintf(stdout, "       <sleep> is the time to sleep between ~3Kb
allocations in micro second\n");
        fprintf(stdout, "       <interval> is the number of malloc
iterations between sleeps. 0 ==> sleep after every malloc\n");
        exit(1);
    } // End if invalid args

    nThreads = atoi(argv[1]);
    nIterations = atoi(argv[2]);
    smallBlock = atoi(argv[3]);
    totalMem = atoi(argv[4]);
    sleepTime = atoi(argv[5]);
    sleepIv = atol(argv[6]);
    if (smallBlock > 2048) {
        fprintf(stdout,
            "Small block can not be larger than 2Kb for this test
application\n");
        exit(1);
    }

    if (sleepTime > 999999) {
        fprintf(stdout,
            "Only sleep times < 1 second allowed\n");
        exit(1);
    }

    // HPUX Multi-processor control - if you like to try this
#ifdef MPCTL
    mpctl(MPC_SETPROCESS_FORCE, 1, MPC_SELFPID);
#endif
}
```

```
    if (nThreads > 0) {
        int i;
        for (i=0; i< nThreads; i++) {
            id[i] = i;
            pthread_create(
                &thrd[i],
                NULL,
                &Thread,
                &id[i]);
        }
    }
    else {
        Exec();
    }
    while (1) {
        float cpuPerc ;
        allDone = true;
        pthread_mutex_lock(&mutex1);
        for (i=0; i<nThreads; i++) {
            if (done[i] == false) {
                allDone = false;
            }
        }
        pthread_mutex_unlock(&mutex1);

// If all threads are done proceed to calculating stats
        if (allDone) {
            unsigned long totCPU = 0;
            for (i=0; i<nThreads; i++) {
                totCPU = totCPU + cpuUsage[i];
            }
            getrusage(RUSAGE_SELF, &rusgl);
            vall = rusgl.ru_utime.tv_sec*1000000 + rusgl.ru_utime.tv_usec;
            val2 = rusgl.ru_stime.tv_sec*1000000 + rusgl.ru_stime.tv_usec;
            totCPU = vall + val2;
            time2 = time(NULL);
            struct mallinfo mallinfo1;
            mallinfo1 = mallinfo();
            cpuPerc = (float) totCPU * 100;
            cpuPerc = cpuPerc / (float) ((time2-time1)*1000000);
            FILE *fPtr1 = fopen("./memtestresult.csv", "r");
            if (!fPtr1) {
                FILE *fPtr1 = fopen("./memtestresult.csv", "a");
                fprintf(fPtr1,
                    "PROG          NAME,THREADS,ITERATIONS,SLEEP
INTERVAL,SMALLBLOCK,SLEEP(us),REAL TIME,CPU TIME,CPU,ARENA,MEM ALLOCATED1, MEM
ALLOCATED2\n");
                fclose(fPtr1);
            }
            fPtr1 = fopen("./memtestresult.csv", "a");
            char tmpBuf1[200];
#ifdef __MTS__
            fprintf(fPtr1,
                "%s,%d,%d,%ld,%d,%ld,%ld,%ld,%5.2f,%ld,%ld,%ld\n",
                argv[0], nThreads, nIterations, sleepIv, smallBlock, sleepTime, time2-time1,
                totCPU/1000000, cpuPerc, ats_heap_total_size(), wantedMem+nThreads*2000000,
                ats_heap_size());
#else
            fprintf(fPtr1,
                "%s,%d,%d,%ld,%d,%ld,%ld,%ld,%5.2f,%ld,%ld,%ld\n",
                argv[0], nThreads, nIterations, sleepIv, smallBlock, sleepTime, time2-time1,
                totCPU/1000000, cpuPerc, mallinfo1.arena, wantedMem+nThreads*2000000,
                mallinfo1.uordblks/nThreads + mallinfo1.usmblks);
#endif

            fclose(fPtr1);

            exit(0);
            while (1) {
                sleep(10);
            }
        }
    }
}
```

```
FILE *fPtr = fopen("./memstats.csv", "a");
fprintf(fPtr, "-,%ld,%ld,%ld,%ld,%ld,%ld\n",
        mallinfo1.arena,
        mallinfo1.uordblks, mallinfo1.usmblks,
        mallinfo1.fordblks, mallinfo1.fsmbks,
        mallinfo1.hblkhd);
fclose(fPtr);
}
}
} // main
```

12 APPENDIX D – MEMTEST BUILD PROCEDURE

The following aCC compiler options were used to generate the memtestf_ats and memtestf_hp binaries. It requires the MTS library to be deployed in directory MTS, in this case under /home/philips.

```
aCC -g -DHPUX +DA2.0 -D_REENTRANT -L/usr/local/lib memtest.cpp -o memtestf_hp
-lpthread -lc
aCC -g -DHPUX +DA2.0 -I/home/philips -D__MTS__ -D_REENTRANT -L/usr/local/lib
memtest.cpp -o memtestf_ats -lpthread -lats
```