Autonomic-based Persistent Memory Management in AIX

- Improving System Performance and Reliability -

Scott Trent *

Determining the most efficient tuning strategy for AIX persistent memory is challenging since system memory and file usage characteristics can change significantly over time making it impossible for any single set of values to be universally applicable. Yet, incorrect tuning can result in highly visible performance incidents. After analyzing commonly used tuning strategies, this paper introduces an original but broadly applicable autonomic program developed by the author to solve real-world persistent memory performance problems through analyzing memory usage characteristics and dynamically modifying tuning parameters.

Key Words & Phrases: Autonomic Computing, MAPE, AIX, Performance Tuning, File Cache

1. Introduction

Although the amount of persistent memory available on an AIX system for caching disk files in memory is tunable, an improper setting can cause serious performance problems and even contribute to system instability through thrashing. Yet, the correct setting may vary over time as the usage characteristics of the system change. Even worse, certain operations such as backups may have a residual negative impact on future performance. There are also limitations with commonly utilized techniques such as setting extreme limits to persistent memory, adding physical memory, pinning programs and data in physical memory, reducing and spacing out load, etc.

Even if a specific system has a constant unchanging load, determining the ideal tuning strategy is often a time consuming process of trial and error. Thus, the use of an autonomic Monitor, Analyze, Plan, Execute (MAPE) loop is desirable to determine the correct setting to use for a changing environment. [1] Since this specific dynamic functionality is not yet built into AIX, this paper introduces and analyzes the benefits of a custom autonomic program that maintains AIX persistent memory tuning attributes. The techniques introduced in this paper address a common need to maintain acceptable system performance through effectively managing AIX file cache usage.

2. The AIX File Cache Mechanism

2.1 The Benefit and the Challenge

Data caching is a common performance improving technique used in all levels of computer hardware and software architecture. [2, 3] The AIX file cache contains recently used file system pages which can be used to avoid repetitive and time consuming physical disk access. [4, 5, 6]

```
# time fgrep testsearchstring BIGFILE

real      0m25.70s
user      0m0.71s
sys       0m4.01s

# time fgrep testsearchstring BIGFILE

real      0m1.76s
user      0m0.92s
sys       0m0.84s
```

Figure 1: Benefit of File Cache

In Figure 1, the second search command (which accesses the file cache) runs in less than 7% of the time consumed by the first search command which must access the physical disk. However, the benefits of the file cache can be nullified with an overly restrictive setting. In Figure 2, file caching is highly restricted, resulting in a search time that is almost twice as long as the worst case in Figure 1.

```
# vmo -o minperm%=1 -maxclient%=2 -o maxperm%=2
-o strict_maxperm=1

Setting minperm% to 1
Setting maxperm% to 2
Setting maxclient% to 2
Setting strict_maxperm to 1

# time fgrep testsearchstring BIGFILE

real      0m44.13s
user      0m0.82s
sys       0m3.77s
```

Figure 2: Negative Effects of Restrictive Tuning

Given these two examples, one might conclude that the file cache should be made very large. However,

Submission Date: August 31, 2004

* trent@jp.ibm.com, Server Systems Department Number 1

an unlimited file cache can be risky. AIX uses a Least Recently Used (LRU) algorithm to determine which pages to keep in memory. Thus during periods of memory stress, required (but relatively less used) computational pages for a process may be paged out. [5, 6] This effect is further reinforced when the process needs to block on a page-in from paging space, allowing other required computational pages in the process to go unaccessed. These pages then become candidates themselves for being paged out, further impeding the process, and further allowing other pages to become paged out.

This is not merely a theoretical risk. The author has assisted many Customers with paging and performance issues related to file cache. In one particular case, users were concerned with very slow system response in the morning. Analysis revealed that the processes used for on-line transactions during the day were largely paged out when backups were performed during the night. The slow response in the morning was caused by massive page-in activity. Although performance was improved by severely restricting the size of the file cache, an excessively small value has the performance risks identified above.

2.2 File Cache Tuning Parameters

Most memory on an AIX system can be classified as "Persistent" (also known as "File Cache") or "Computational". [3, 4, 6] Each page in persistent memory directly maps to a page in a file on disk. Computational memory pages such as heap or stack contain dynamically generated data. Unix systems have traditionally been designed to support a virtual memory space that can be much larger than available physical memory. In order to realize this, during periods of high memory demand the Unix kernel must determine which pages to keep in memory, which to erase, which to write back to disk, and which to write into a special paging area on disk for future use. [2]

In AIX, there are four primary tuning parameters which affect this file cache usage. [4, 5, 6, 7] These parameters can be modified with the vmtune command in the bos.adt.samples fileset prior to AIX 5L 5.2, or the vmo command starting in version 5.2.

Traditionally, the most important parameters are minperm and maxperm. AIX documentation states that when the percentage of physical memory used for file cache exceeds maxperm percent, that only file cache pages will be released or "stolen" when additional physical memory is required. If less than minperm percent of physical memory is used for file cache, then when pages must be "stolen" (or released) then both file cache and computational pages will be chosen. If the percentage of physical memory used by file cache is between minperm and maxperm then file pages will generally be chosen unless the repage rate for file pages is higher than that for computational pages. [6, 7] The documentation does not emphasize that the Virtual Memory Manager (VMM) is one of the most complex areas of the AIX kernel, and that exceptions to these documented rules do exist. For example, given this standard explanation, one would not expect file cache usage to exceed maxperm percent of physical memory, yet this is not an uncommon occurrence.

Of the remaining parameters, strict_maxperm was added in AIX 4.3.3 to enforce the maxperm limitation. By default, strict_maxperm is 0, permitting file cache usage to exceed maxperm. When strict_maxperm is 1, file cache usage is not allowed to exceed maxperm percent of physical memory. The maxclient parameter was added in AIX 5L to control the use of file cache for JFS2 files which are classified as "client memory", along with pages cached from non-JFS file sources such as NFS. [6]

Actual file cache use percentage can be determined from the numperm value displayed by vmtune on pre-5.2 systems. "vmtune –A" and "vmstat –v" can be used on 5.2 or later systems (Figure 3). topas and "svmon –G" also display useful information about file cache and other memory usage. [5, 6]

```
# /usr/samples/kernel/vmtune -A | grep perm

        20.0 minperm percentage
        80.0 maxperm percentage
         5.9 numperm percentage
```

Figure 3: Viewing numperm on AIX 5.2

3. Common Persistent Memory Tuning Strategies

The ideal file cache tuning strategy will enable as much memory as possible for file cache while not causing process computational memory pages to be paged out which will impede future computation.

3.1 Default File Cache Tuning

Although, by default, minperm is 20% and maxperm is 80%, it is not uncommon to see actual file cache size (numperm) ranging from several percent to over

90% depending on the characteristics of the processes running on that system. If the “working set”, the amount of memory that the system requires to make progress, is sufficiently less than the physical memory size, and if computational/file cache memory demands made by processes on the system do not conflict, then the default values may be fine. However, if vital computational data pages are paged-out, causing performance problems, then some other strategy may be required. The pi and po columns displayed by the vmstat command are commonly used to diagnose paging activity. (Figure 4) The topas command also displays useful paging and memory statistics.

3.2 Highly Restricted File Cache Tuning

A highly restricted file cache size can be effective on systems that do not require a larger file cache. Yet, the correct setting which both prevents paging, and does not otherwise impact performance must be determined by trial and error. These values are likely to change over time as either system load increases or as system configuration evolves. Values which are too low can cause processes to experience the performance impact demonstrated in Figure 2.

3.3 Work Load Manager

If categorization and strict prioritization among processes by user name, group name, program name, or WLM tag can be made, it is possible to ensure that certain groups of processes receive a predefined amount of memory or CPU. [8] Starting in AIX 5L it is also possible to control disk I/O bandwidth usage. Yet, it is not always possible to separate resource utilization into separate classes. These usage characteristics also change over time, making it hard to predict the perfect work load manager configuration.

3.4 Pinning Pages in Physical Memory

It is common for vital kernel code and data to be pinned in physical memory to prevent paging. It is also possible to use the AIX plock() function from a user process to pin user process text and/or data in memory. This can be useful for a small vital process. Yet, pinning too much data in memory can directly cause memory contention. (It is also important to use either the ulimit command or API to limit the amount of accessible memory to the process before calling the plock() function, otherwise, hundreds of megabytes of data will be pinned in memory, reducing the usable physical memory on the system for other processes. (See plock() man page.)) Memory pinning should only be used when there is sufficient unused physical memory.

Although this approach can prevent a specific process from having its pages swapped out to paging space, it does not protect the process from other effects of paging or thrashing. For example, a system which is thrashing uses significant CPU shuttling pages back and forth from paging space, thus less CPU time is available for all processes, including any pinned processes even they may be immune from actual paging.

3.5 Reduced Load

Since memory demand is usually a direct result of application load, reducing the number of transactions, users, programs, connections, etc, on a system can also help resolve memory problems. The exact technique used, and the applicability of this approach clearly depends on user requirements.

3.6 Additional Physical Memory

Extended page-in and page-out activity, shown through the vmstat columns pi and po (see Figure 4), when file cache usage is low, is an indication that the system may require additional physical memory. Increasing physical memory when extensive file cache space is being used may not improve performance. For example, when file backup causes processes to page out, performance will continue to suffer unless a file cache larger than the sum of all files being backed up can co-exist with computational memory. Assuming that average memory and disk sizes increase in parallel, a more reasonable solution would be to terminate processes that will page before the backup and restart them afterwards, or to limit file cache so that it does not cause computational pages to be paged out.

```
# vmstat
kthr     memory             page              faults        cpu
r  b    avm    fre  re   pi   po  fr   sr  cy   in    sy  cs us sy id wa
2  1 333173   1079   0 239 320 182 3438   0 1344 101684 2410 26 26 25 23
3  1 333173   1150   0 180 257 257 7944   0 1225 99789 2252 32 26 14 28
2  2 333920   1022   0 270 840 1024 1763  0 1386 100585 2402 30 28 16 26
2  2 333945   1107   0 331 454 329 5992   0 1439 103971 2559 31 29 12 27
3  0 333208    725   0 1318 199 199 3886  0 2377 98920 4488 30 30 18 21
2  1 333173    475   0 527 236 252 26843  1 1583 99783 3082 28 27 24 22
3  1 333173    474   0 316 315 315 2841   0 1368 102119 2650 27 25 25 24
2  1 333173    475   0 242 243 243 6844   0 1299 101807 2439 28 25 24 22
2  1 333173   1131   0 121 761 769 55780  0 1260 101195 2300 27 26 25 22
2  1 333173   1132   0 174 156 180 14073  0 1257 101946 2295 26 26 24 24
```

Figure 4: Excessive Paging seen through vmstat

4. An Original Autonomic Solution

An autonomic MAPE control loop is an excellent technique to dynamically analyze system characteristics which change over time and modify the necessary tuning values on-the-fly. As the acronym indicates, a MAPE loop uses sensors to monitor an element, analyze this data, plan to improve or maintain the situation, execute the plan using effectors, and then repeats. [1, 9]

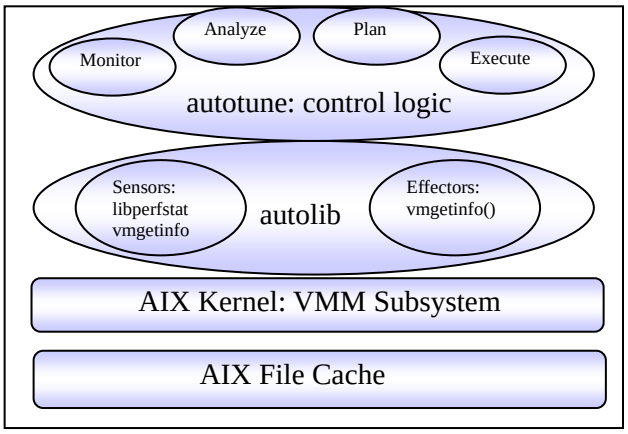

Figure 5: autotune: MAPE Based Architecture

4.1 autotune: Implementation Details

The author developed autotune, a unique AIX program, to monitor and control the AIX file cache. The MAPE control loop, logic, and rules are all contained within autotune.c which uses an independent module, autolib, that contains sensors and effectors (see Figure 5). Although the program is written entirely in C, an object oriented approach is used within autolib to hide the sensor/effector implementation details from the calling program. The sensor gathers performance statistics about the current file cache tuning parameters with the vmgetinfo() API, and current running system statistics including memory usage and paging through API's provided in libperfstat. [7] The effector modifies file cache related tuning parameters with the misleadingly named AIX internal system call vmgetinfo(). (This API is used to both get and set VM tuning parameters) Early versions of autolib called the vmtune commands using fork()/exec() to enhance portability among different versions of AIX. However, during high memory stress, the time required to call an external command to change parameters made the program entirely ineffective. Thus, autolib was enhanced to control VM tuning parameters via the vmgetinfo() API. The drawback is that the structures passed to and from vmgetinfo() are only declared in unshipped header files, so a different version of autolib is needed for each major version of AIX.

To assure that autotune is always runable, even during periods of very high stress, setpri() was used to increase the priority, and plock() was used to ensure that it would not be paged out. Additionally, autotune was developed to consume minimal resources. Even while monitoring and coordinating memory on a highly memory stressed environment, autotune was observed to have a CPU time to real time ratio of less than 0.1%.

4.3 Original Automatic Tuning Rules and Policies

As a proof-of-concept project, autotune was designed with the following general tuning rules and policies.

Decrease File Cache Rule: If pages are being read and written from paging space simultaniously, then reduce minperm, maxperm, and maxclient, taking into account the degree of paging and the minimum allowable values. If this continues, turn strict_maxperm on.

Increase File Cache Rule: If paging space activity has not occurred "for a while", and there is sufficient available memory, gradually increase maxperm and maxclient.

Table 1: autotune Policy Values

Policy Description	**Default Value**
Monitor Interval	4 seconds
Maximum maxperm	90%
Minimum maxperm	6%
Maximum minperm	20%
Minimum minperm	2%
Maximum maxclient	90%
Minimum maxclient	6%
Rearm interval for increasing file cache	60 seconds
File cache increase delta	varies
File cache decrease delta	varies
Numeric definition of sufficient available memory for increase	varies
Paging threshold for modifying file cache	varies

4.4 Test Environment

Tests were performed on a 375 Mhz 4-way SP node running a 64-bit AIX 5L 5.2 ML3 kernel with 2GB of physical memory. Files were stored in a dedicated JFS file system. (A JFS2 file system would likely have resulted in better results, since

unlike maxperm, maxclient is strictly enforced by default.)

Two 32-bit large-memory model test programs were used as CPU/memory bound and disk I/O bound processes. The CPU bound program allocates a user specified amount of memory and repeatedly writes to all of this memory, reporting the number of megabytes of memory that were accessed every 10 seconds. The disk I/O bound program creates a file of user specified size, and then repeatedly reads the entire file, reporting the number of megabytes that were read every 10 seconds. The test machine was also concurrently running a large db2 (version 8.01) database to further duplicate real world conditions.

Performance data was gathered from output generated by the test programs, the autotune program, and Nigel Griffiths' performance monitoring tool, nmon. [10]

4.5 Base Line: Test Results with Sufficient Memory

The test programs were executed with and without autotune with sufficient memory to determine ideal performance of the test programs, and verify that the autotune tool does not negatively impact performance in this case. Both with and without the autotune tool, the CPU bound process is consistently able to process around 6,000 megabytes every 10 seconds, and the disk I/O bound process is consistently able to process around 3,000 megabytes every 10 seconds. The disk I/O bound process is started and terminated in the middle of the test to determine the effect it has on the CPU bound process. When there are sufficient memory and CPU resources, this impact is less than 3%, and can be seen in Figures 6 and 7 as a very slight dip in the CPU bound process's performance. This data demonstrates that autotune does not have a significant negative impact on the normal sufficient memory case.

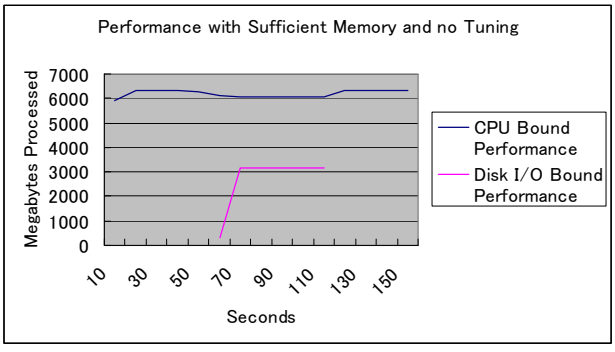

Figure 6: Sufficient Memory without autotune

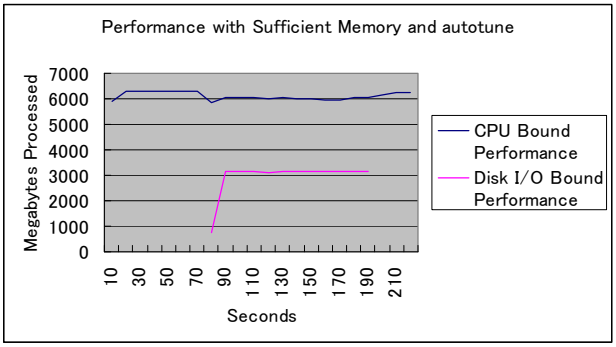

Figure 7: Sufficient Memory with autotune

4.6 Benefit: Test Results with Insufficient Memory

Next, the test programs were executed with and without autotune in an environment without sufficient physical memory, to determine the effect that autotune has on their performance.

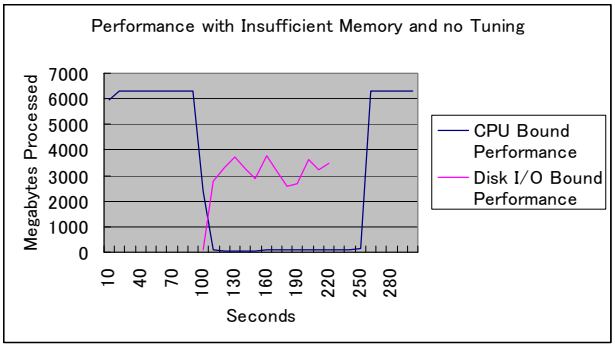

Figure 8: Insufficient Memory without Tuning

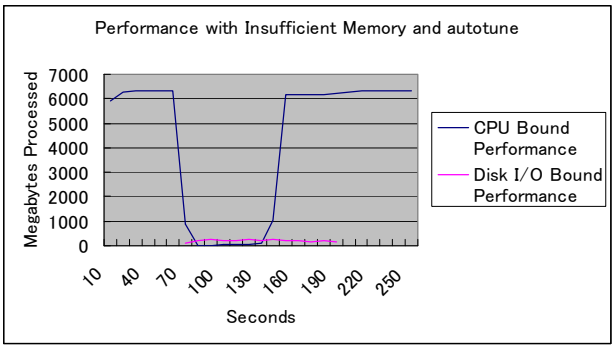

Figure 9: Insufficient Memory with autotune

Without autotune, the disk I/O bound process prevents the CPU bound process from making progress (Figure 8). In fact, it takes the CPU bound process over 30 seconds to resume AFTER the disk I/O bound process terminates (Figure 8). With autotune, immediate tuning prevented the disk I/O bound process using excessive memory, but it still took about a minute for the CPU bound process to recover (Figure 9). However, the advantage is that **the CPU bound process recovers while the Disk I/O bound process is still running!** (Figure 9) This is desirable when priority should be placed on

CPU bound processes providing, for instance, user oriented real time services. Additionally, unlike using a statically limited file cache, when the CPU bound process terminates, the disk I/O bound process will run at full speed.

These results are enforced by the data on paging-space activity measured and graphed with the nmon tool. [10] Both high memory demand and an initial page-out spike are present with and without autotune. However with autotune, this spike is 30% smaller and further page-out activity is suppressed. While as without autotune, page-out activity (which contributes directly to CPU and memory bound process performance degradation) continue throughout the test. (Figures 10 and 11.)

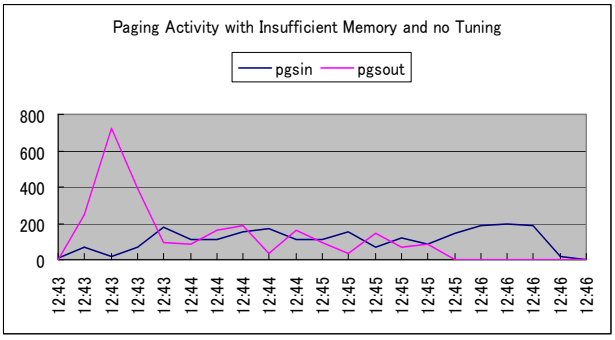

Figure 10:Paging:Insufficient Memory, No autotune

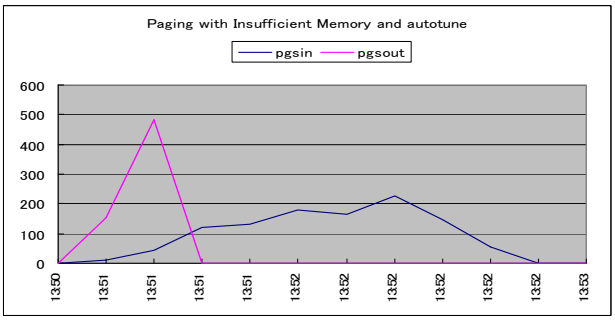

Figure 11:Paging:Insufficient Memory with autotune

5. In Conclusion

This paper has presented a successful and broadly applicable proof-of-concept in the unique application of an autonomic MAPE control loop to solve the AIX file cache tuning problem. The original autonomic autotune program demonstrates the validity of this dynamic approach by improving performance in highly memory constrained environments that would ordinarily lead to severe performance degradation caused by heavy paging or thrashing. This technology is directly applicable to solving performance problems in the field **before they happen**.

The next logical step is to extend and apply the approach and tool described in this paper to monitor and tune other operating system variables that affect reliability and serviceability as well as performance.

Bibliography

[1] IBM, “An architectural blueprint for autonomic computing”, www.ibm.com/autonomic/pdfs/ACwpFinal.pdf, April 2003

[2] Tanenbaum, Andrew, *Modern Operating Systems*, Prentice Hall, ISBN 0136386776, 2001

[3] Cannon, Jones, Trent, *Simply AIX 4.3, Edition 2*, Prentice Hall, ISBN 0130213446, 1999.

[4] Furutera, et al, *AIX Operating Systems Concepts and Advanced System Administration*, ASCII, ISBN 4756139124, 2001.

[5] Chukran, *Accelerating AIX:Performance Tuning for Programmers and System Administrators*, Addison-Wesley, ISBN 0201633825, 1998

[6] IBM Corporation, *AIX 5L Version 5.2 Performance Management Guide*, IBM, SC23-4876-00, May 2004.

[7] IBM, *AIX 5L Version 5.2 Performance Tools and Guide and Reference*, SC23-4859-01, May 2003

[8] Gfroerer, Castro, Tezulas, Yu, Berg, Kim, *AIX 5L Workload Manager (WLM)*, IBM, ISBN 0738422436, 2001

[9] Stojanovic, Schneider, Maedche, “The Role of Ontologies in Autonomic Computing Systems”, IBM Systems Journal, Volume 43, Number 3, 2004.

[10] Griffiths, N, “nmon performance -- free tool to analyze AIX”, www.ibm.com/developerworks/eserver/articles/analyze_aix, November 4, 2003