Operating System Support for Database Management Systems Revisited

Daniel Fellig, Olga Tikhonova*

December 18, 2000

Abstract

In the early 1980s Michael Stonebraker [9] wrote a paper titled "Operating System Support for Database Management", outlining where and how operating systems at the time failed to meet the needs of database management systems in terms of some typical services that operating systems provide to applications. Stonebraker points out these deficiencies in the areas of buffer pool management, file system, scheduling, process management, and interprocess communication, and consistency control. The examples drawn are mostly from UNIX and INGRES. This paper twenty years or so later, investigates the operating system support of one of these services - buffer pool management provided to database management systems by a modern commodity operating system, Microsoft Windows NT 4.0.

1 Introduction

Buffer management can be broken down to three types of policies: a fetch policy, a placement policy, and a replacement policy [8]. In order to develop an optimized database management system, it is crucial to implement these type of policies such that having to go to disk for data or instructions is avoided as much as possible. In terms of performance, roughly, a conservative estimate is that a typical disk access is about 100,000 times slower than a main memory access. Even though memory prices have certainly come down on a per MB basis, RAM is still about 100 times the cost of disk space for the same space [7]. Also, as memory sizes become larger in systems, the amount of data that database management systems have to process also becomes larger.

In addition, one has to realize that when making a decision on how to optimize DBMSs memory management, the following property has to be accounted for: DBMS can often predict the order in which pages will be accessed, or page reference patterns accurately. This is based on the fact that most page references are generated by higher-level operations with a known pattern of page accesses.

*Department of Computer Science, University of Wisconsin, Madison WI 53706; email: olga@cs.wisc.edu, dfellig@students.wisc.edu. We thank Professor Arpaci-Dusseau for advice and encouragement.

Another consideration with respect to buffer management is the issue of consistency. This paper will not address issues of consistency, these have been dealt extensively leading to the formulation of WAL, write-ahead logging, which is used ubiquitously across the database world. This paper will focus on issues of speed of access to data by database management systems. Even more specifically, this paper will focus on how a database management system may optimize buffer management on top of Microsoft Windows NT 4.0. Note that anytime a reference to NT is made in this paper, this reference is to be taken as NT 4.0 specifically.

When thinking about buffer management in terms of the needs of database management systems, one of the first thoughts that come to mind is why not let the operating system perform this management, after all it does precisely this for most applications. Isn't a database management system just another application? This is the first question that this paper will attempt to answer, putting NT under scrutiny. This question will be investigated by supplying simple stimuli to NT, using a "gray-box" approach [1], and capturing its responses to establish empirically what its buffer management policy is. Dabak, Phadke, and Borate [2], point out that the policy used by NT for page replacement when under available physical memory pressure is FIFO. Solomon [8] specifies that in uniprocessor x86 systems, a policy known as "clock algorithm" is implemented by NT. This constitutes the a priori information that we have of NT's page-replacement policy that we intend to either support or contradict with data obtained by probing into NT's buffer management with a custom micro-benchmark built by us specifically for that purpose.

Once NT's buffer management policy has been exposed as reflected by the data obtained, this paper will consider the interfaces that NT provides to applications for imposing their own buffer policies on top of NT's. This is precisely what Stonebraker described twenty years ago in terms of database management systems having to manage their own buffer pool on top of UNIX, citing Ingres as a specific example [9]. In essence, this paper simply asks whether Microsoft Windows NT 4.0, a commodity operating system, is any closer to satisfy buffer requirements of RDBMS than UNIX was twenty years ago; and if the answer is in the negative, we investigate what appropriate interfaces NT exposes for the DBMS' buffer manager to impose its own optimized buffer management policy on top of NT's.

2 Related Work

Interestingly enough, given that the major vendors of relational database management systems have engines that run on NT, there is really no work that we were able to find that probes NT's buffer management system in detail. This surprised us, but not entirely given that NT is not an open system.

3 Documented Relevant Aspects of the NT Memory Model

Independently of the available physical memory in the system, and of how much physical memory is assigned to each process, each process is assigned a virtual memory space of

2GB, with the system using the remaining also 2GB. In Windows NT Server, Enterprise Edition, there is the additional option of choosing to configure NT such that 3GB of virtual space is assigned to the application with 1GB assigned to the system. This feature was added as a short term solution to accommodate resource intensive applications such as database management systems [8]. All processes use the same range of linear 32-bit addresses from 0000000016-7FFFFFFF16 [4]. Each process has its own address space independently of other processes. Shared memory between processes is implemented through memory mapped files. Each application maps a view of the same portion of the pagefile, making the same pages of memory available to each process [4].

Virtual addresses in a process can be in one of three states: free, reserved, or committed. Processes can only allocate space that is free. Space can be reserved or committed, the difference being that reserved space ensures that the reserved address range will be available to be used at a later time without actually binding resources to it, ensuring a large contiguous space if required by the application. Reserved space cannot be used without committing it. Committing an address range means that resources have been bound to that space in terms of physical pages of memory and reserved pages in the pagefile [4].

The number of frames in physical memory allocated to a process at any given instant is called the working set for this process. Each process has its own working set. The most important property of the working set is its size. The essence of any buffer management policy is how the working set changes in size as a function of the different states of a page in memory. Two very important aspects of a page are the length of time it is in memory since it was paged in, and the frequency with which it is accessed once in memory. NT monitors this and chooses a victim. If a page frame allocated to a process is evicted due to physical memory pressures, the virtual memory manager doesn't immediately reuse this page frame, instead it places it on a stand-by list, thus delaying the reuse and giving the process the opportunity to regain the page frame by referencing its address within a certain time. The algorithm to determine which pages to evict from a working set is different on single-processor x86 systems from multiprocessor x86 or Alpha systems. In the case of single-processor x86 systems how often a page has been referenced determines whether it will "survive" or not. NT knows how often a page has been referenced by checking a reference bit in the hardware PTE part of the processor. This page replacement algorithm is known as the clock algorithm. On Alpha or multi-processor x86 systems the working set manager doesn't check the accessed bit; clearing it would affect other processors. This results in the replacement policy in those systems having more FIFO components.

Given the physical memory available, NT will provide the process with an ever increasing working set, as more pages are brought in from the disk. There is documentation that claims that the working set size of a process can be bounded through an API. The API mentioned is:

BOOL SetProcessWorkingSetSize ((1)
HANDLEh Process, // handle to process (2)
SIZE_T dwMinimumWorkingSetSize, // minimum working set size (3)
SIZE_T dwMaximumWorkingSetSize // maximum working set size (4)
). (5)

We tried using this API in our micro-benchmark code, but were unsuccessful in keeping the working set bounded. The working set of the process on which we applied the API kept increasing going beyond the upper limit set in the API , as pages were brought in from the disk. The only way we were able to stabilize the size of the working set of this process was to engage more physical memory, thereby creating pressure on NT's buffer management system to start evicting pages.

4 Causing Page Replacements

To be able to make any type of measurements we first had to be able to cause pertinent events. The most obvious one being the fetching of pages from disk, and the eviction of victim pages from memory. To cause the fetching of a page from disk a segment of virtual memory was allocated. The interface to do this exposed by NT in the API

LPVOID VirtualAlloc ((1)
LPVOID lpAddress, // region to reserve or commit (2)
SIZE_T dwSize, // size of region (3)
DWORD flAllocation // type of allocation (4)
DWORD flProtect // type of access protection (5)
). (6)

This API call returns the base address of the segment allocated if successful, otherwise a NULL value is returned. This call failed in our testing when the requested virtual memory segment was to large with respect to the paging file size. This file can be enlarged if desired. The size of the virtual memory segment requested was based on the total physical pages of the host machine. The idea is that our micro-benchmark should stimulate any host x86 system running NT with different hardware in a similar way by taking into account its hardware resources. To achieve this we base the sizing of the virtual memory request as a proportion of the physical pages of the machine running the micro-benchmark. The memory address returned as the base address for the block is aligned with the beginning of a page. In the case of all x86 systems, a page has 4096 bytes. The byte page value, as well as all the system parameters were extracted with the Windows Native API

NTSTATUS ZwQuerySystemInformation (		(1)
IN SYSTEMINFOCLASS SysInfoClass,	// type of system	(2)
IN OUT PVOID SysInfo,	// caller allocated buffer	(3)
IN ULONG SysInfoLength	// size in bytes	(4)
OUT PULONG ReturnLength	// # of bytes returned	(5)
).		(6)

The Native APIs are system services provided by NT to user and kernel mode programs [6]. These NT interfaces tend to provide a lower level window to the system.

Once the page-aligned segment of virtual memory was allocated, bringing a particular page from disk to memory simply meant de-referencing a virtual memory address in the allocated space. This is of course is just one side of the buffer management equation, the other being the eviction of pages from memory as a result of a tight memory budget. Being able to trace specific pages being evicted from memory was a much more difficult task. We created a workload that would use enough memory such that NT would evict pages resident in memory often enough so as to be able to trace them. This workload load was created by a small utility we created called memEater. This utility allocates virtual memory, and then starts reading pages from disk to memory to use enough memory such that NT has to evict pages most of the times a new page is read in from disk. This utility has the lowest possible priority both at the process and at the thread level. The significance of this lies in that the program that will do the probing needs to run along side memEater. So when the probing program ntinfo starts running, the memEater program looses CPU cycles to ntinfo, since it has a higher priority than memEater. This forces memEater to consume less memory per unit time, thus ensuring that ntinfo will have enough memory to run, at the same time that memory will be in short supply forcing pages in memory to be evicted most of the times that new pages are brought in from disk.

5 Tracing Pages Across Their Lifetime

Causing pages to be brought in from disk to memory and evicted from memory to disk at will was essential to our experiments, but just a beginning. The next issue is how, when and where should probing occur. A key window into the NT paging action was the API:

BOOL QueryWorkingSet (		(1)
HANDLEh Process,	// handle to process	(2)
PVOID pv,	// information buffer	(3)
DWORD cb	// size of buffer	(4)
).		(5)

This API provided the address and the attributes of each of the pages in the working set existent at the instant of the call. This enabled us to look inside the paging action by taking a snapshot of the working set, then reading in a page to memory , and then taking another snapshot of the working set, finally compare both working sets and thus detecting page evictions immediately after bringing a data page into memory.

There were two general types of pages in the working sets we were working with: executable pages with op codes, target and destination memory registers, and data pages. Given that the code we were running was highly repetitive; i.e., loops, the expectation was that the victim pages would not be these pages, but the data pages. Differentiating between page types involved filtering on the basis of virtual address intervals.

Looking at the working set differences every time a new page was brought in from disk was not a practical option, given that our experiments involved more than 12,000 data pages. Our approach was to make our data gathering manageable but representative. We divided the data pages into ten page groups, establishing a sampling interval in each of the ten groups. Our sample size was ten pages per group. In other words, for the first ten pages of each page group the paging action was monitored, the remaining time pages were brought into memory without any tracing. This mode of sampling enabled to distribute our monitoring window across each of the page groups making our tracing more representative, and at the same time maintaining the volume of experimental data manageable.

6 Page Reference Frequency

To determine the buffer management policy that NT uses, different page reference frequencies were imposed on the page groups. A page reference group multiplier was created that reflected the relative reference frequency that a specific page group was being subjected to as compared with the other page groups. It should be noted that at the very least each page was referenced once when read into memory. After this, pages of the target group were referenced multiple times more as controlled by the reference multiplier that is a variable parameter. The point of this was to determine what LRU components are existent in NT's buffer management policy.

7 Data Gathering Associated with Page States

In order to only influence paging in the context of our experimentation with the data pages, the information gathered on each of the data pages traced in memory was saved in blocks, each 40 bytes long, using an array of pointers to each of these blocks. These blocks were kept in memory at all times, never being allowed to be paged, thus providing us with more control independent of NT's paging. The API used for this is:

BOOL VirtualLock (		(1)
LPVOID lpAdress,	// first byte in range	(2)
SIZE_T dwsize,	// number of bytes in range	(3)
).		(4)

This API provided the means to lock pages into memory such that when any of the locked pages were referenced for the purpose of saving or retrieving data of a particular page, page faults would not be induced directly. This did mean that the available memory in the working set with the possibility of paging decreased, thus increasing the eviction rate of the data pages. We believe that even though this is a stimuli to the system, we were able to account for the state of those pages at all times, thus providing less distortion to our measurements than had we used pages that could be either in the paging file or in memory in a given instant.

The data gathered for each page as it moves through the states of being paged-in, referenced, and finally being paged-out is as follows: at the moment of "birth", when a page is brought into memory, the NT performance counter is queried and captured as the "date of birth" of the page and as its last access (last reference) counter value; both values at that point are identical. A sequence number is generated to denote the relative order of arrival into memory, the group number that the page belongs to is captured as well. All this data is saved in the corresponding memory-locked 40 byte block for that particular page. Every time the same page is referenced during its lifetime, the NT counter value is captured, this value replacing the last value of the last access value for that page. Finally, when the page is paged out (instant of "death" of the page), the NT performance counter is queried; the age of the page is computed, taking the difference between the counter value just queried and the "date of birth" counter value. The last access value is also computed, taking the difference of the last counter value (instant of "death" value) and the last access counter value. These values are used to generate statistical data, specifically the average age and average last access value of all the data pages paged in.

8 Pages That Survive Replacement

Part of our test methodology was to pick a page group and reference its set of pages by more than the rest of the page groups, by setting the page group reference multiplier to a specified value. For example, when this value is unity, the page reference level of each group is the same. When the page group reference value is two for example, the target group gets referenced twice as much as the rest of the other page groups, etc. The idea is to observe if there are any changes in the way NT treats pages that are referenced at a greater rate.

Given that sampling takes place once per each page group, constituting the comparison of two instances of the working set, before and after a given data page gets brought in to memory -this occurring for ten consecutive times- there are plenty of opportunities for evictions of pages of the target group to occur that could go undetected. To be able to monitor the extent of the eviction of pages in the target group, at the end of the run of ntinfo, the working set is queried, and filtered on the virtual addresses of the pages that belonged to the target group, thus determining how many of these pages constitute the make up of the working set at that instant. These pages are the "survivors" of NT's page eviction policy. The number of these survivor pages as a function of how many times these were referenced is the means to find LRU components in NT's buffer policy decisions.

Figure 1: Test 1 Results

Reference Multiplier (Num. of times target group is referenced more than the other groups)	**Avg. Last Ref. Time** (us)	**Num. of Survivor Pages in The Target Group Out of 1490**
1x	26066186.51	6
2x	24631398.71	9
3x	25659722.23	973
4x	26547592.32	1093
5x	26531243.82	1336
10x	1014103.13	1336
20x	373849.42	1335
40x	751986.64	1332
80x	442458.5	1333
160x	216303.80	1332
320x	90526.44	1336
640x	137186.54	1255
1080x	74513.25	1043

9 Testing and Results

Test 1:
The values that will be obtained in this test are the number of pages that survive as the page multiplier is changed.
Goal:
To detect LRU components in the buffer management policy of NT.
Procedure:
(a) Increment the page reference multiplier associated with page group 1 as follows: 1x, 2x, 3x, 4x, 5x, 10x, 20x, 40x, 80x, 160x, 640x, 1080x.
(b) Repeat each run 5 times and get the average values. (Refer to Figure 1).
Comment:
The response behavior of the buffer manager to increased reference of the pages in group 1 is clearly asymptotic in the number of survivor pages as a function of the reference multiplier. There is a drastic increase of survivor pages when the reference multiplier is increased from 2x to 3x. Once the multiplier is 3x the number of survivor pages varies only from 1093 to 1047. That percentage variation can be expected as a result of noise in the testing set up rather than as a response from the buffer manager. The average last access time does not appear to give a good reflection of the average time intervals between page references. The average time between page references is a difficult quantity to extract from this testing set up, given that context switches are occurring as the test process is executing

Figure 2: Page Survivors as a function of reference frequency

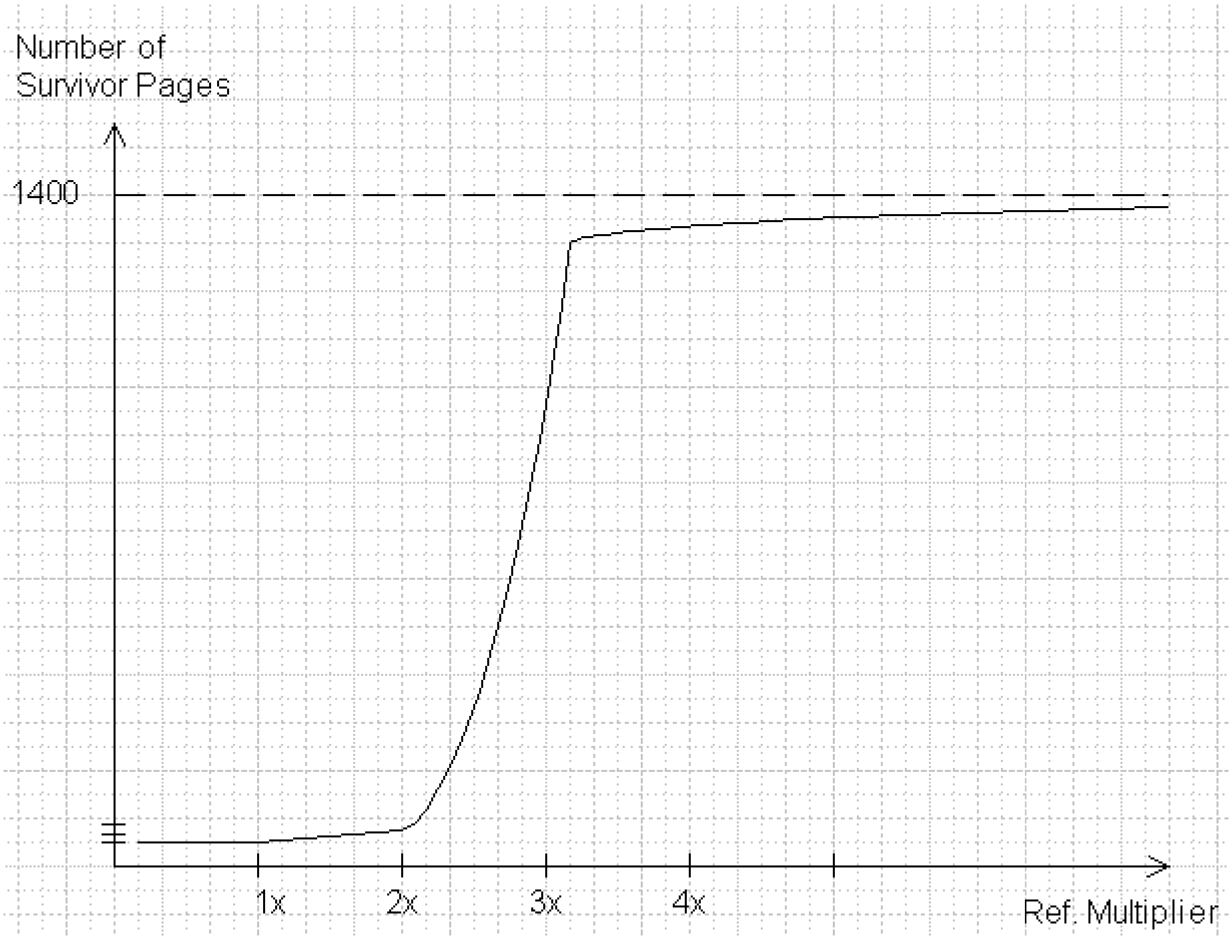


Figure 3: Test 2 Results

	Total Pages Referenced	Time(us)
App. Buffer Policy	10037	1446909.11
NT Buffer Policy	10037	2911277.29

thus consuming time that is not accounted for. We tried increasing the priority of the test process to minimize context switches, but found that by taking CPU cycles from the memeater process the availability of physical memory increased, thereby eliminating page outs in the test process.

Test 2

The value that will be obtained in this test is the response time difference that exists when pages are accessed in a looping sequential pattern LS [3], letting NT use its native buffer management policy, and when we superimpose our own buffer management policy.

Goal:

To determine the performance gains of application buffer management as contrasted with NT's buffer management.

Procedure:

(a) Run the looping sequential pattern 10 times for each of the buffer management policies, and average the time differences.

Comment:

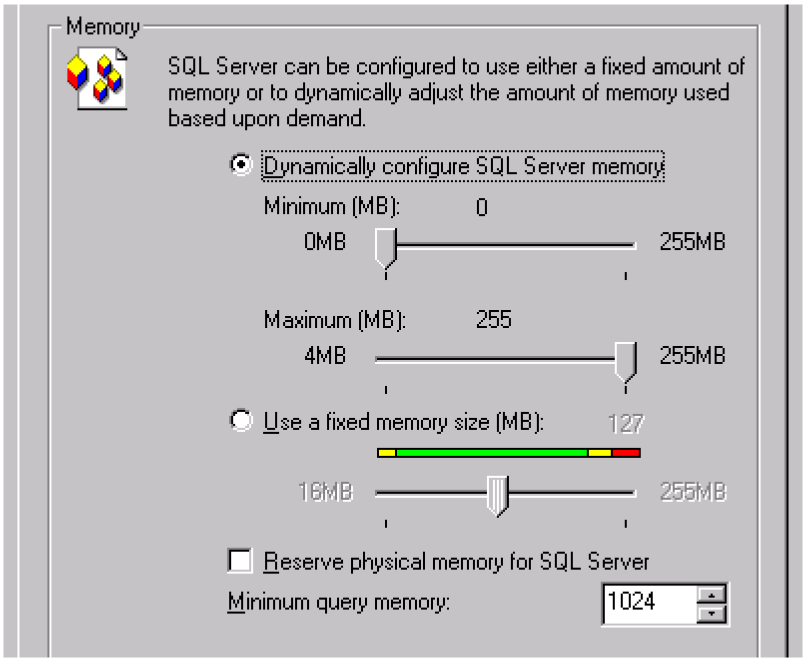
The time response decrease by managing the buffer on top of NT was by about 50%. Even though this test was applied to a very specific access pattern, it clearly demonstrates the potential for buffer policy to be a critical component of performance.

10 Real Life Solutions: Microsoft SQL Server 7.0

SQL Server allocates space from physical memory for 2 main components: executable code and the memory pool. Executable code includes SQL Server code and other dynamic-link libraries and executable files. The memory pool is an area of memory from which the following objects are allocated: system-level data structures, buffer cache, procedure cache, log cache, and connection context. System-level data structures hold data that is global to SQL Server, such as the lock table and database descriptors. The buffer cache is used to store data and index pages using LRU replacement policy. The procedure cache holds the execution plans for stored procedures. The log cache is used to read and write log pages. The connection context is a set of data structures that record the current state of a connection, such as the parameters used for a stored procedure or the tables being accessed. SQL Server automatically determines the size needed for each of these sections of the memory pool.

SQL Server can manage system memory dynamically. It automatically allocates and deallocates memory for its memory pool as necessary. If SQL Server needs more memory for the memory pool, it will allocate it from the available physical memory in the system.

Figure 4: SQL Server 7.0 Memory Configuration Utility



If there are unused pages in the memory pool, SQL Server ca deallocate them, thus freeing memory for other applications to use. However, if no other applications are requesting memory, SQL Server will maintain its memory pool at the current size; it deallocates memory only if there is a need for it.

To avoid excessive paging, SQL Server automatically maintains its virtual memory space to be at least 5MB less than available physical memory. This allows SQL Server to have the largest memory pool possible while preventing SQL Server pages from swapping. If the 5MB of available memory begins to be consumed by some other application, SQL Server deallocates some more of its memory pool in order to keep 5MB free physical memory on the system at all times.

SQL Server implements prefetching mechanism with the Read-Ahead Manager. When SQL Server detects that a SQL query is going to fetch a substantial amount of data from the database (table scan, for instance) the Read-Ahead Manager starts a read-ahead thread up to fetch the data in 16-KB extents from disk and read them into the data cache [5]. The Read-Ahead Manager follows the page chain to read pages before the application discovers it needs them (if it is processing sequentially). This way, the data is in the cache for the query engine to operate on.

For better performance, SQL Server can lock memory as a working set (see Figure 4).

11 Conclusion

Our goal when we started with this investigation was to empirically determine if twenty years or so later, Michael Stonebraker's paper, "Operating System Support for Database Management", as related specifically to buffer management still applies to a modern commodity operating system Microsoft Windows NT 4.0. Our measurements have shown that in terms of NT, database management systems have ample justification in managing their own buffers on top of the operating system's management policy. Michael Stonebreaker's paper twenty years or so later is still valid in terms of buffer management as provided by NT. This is not surprising given that database management systems have very specific needs, and a commodity operating system such as NT is designed for a generalized application type.

The other interesting question with respect to operating system support for database management systems is that of interfaces that an operating system exposes for low-level buffer management. We realized that at the end of our research we might conclude that NT does not optimize buffer management for database engines. In that case we were interested in determining if NT provided the necessary interfaces such that the database management system could manage it's own buffer space on top of the operating systems'. We determined that using the "VirtualLock" API, an application was able to override NT's buffer policies implementing its own. Another API that permits buffer control on top of NT's is "FlushViewOfFile". This API causes any byte range within the mapped view of a file to be written to disk. Given a memory mapped file operated on by a database engine, it is able to commit to disk pages in memory thus once again given the database the possibility of implementing its own buffer policy on top of NT's. Both of these interfaces are examples of the type of access that database code has on top of NT, thus providing a glimpse into how vendors must build their database engines on top of the NT operating system.

References

[1] R. Arpaci, D. Culler, A. Krishnamurthy, S. Steinberg, and K. Yelick. "Empirical Evaluation of the CRAY-T3D: A Compiler Perspective", ISCA, 1995.

[2] P. Dabak, S. Phadke, and M. Borate, "Undocumented Windows NT", MtT Books, 1999.

[3] H.-T. Chou, D. DeWitt, "An Evaluation of Buffer Management Strategies for Relational Database Systems", 1985.

[4] R. Kath, "Managing Memory-Map Files in Win32", Microsoft Corp., 1993.

[5] H. Lau, "Microsoft SQL Server 7.0 Performance Tuning Guide", Microsoft Corp., 1998.

[6] G. Nebbett, "Windows NT/2000 Native API Reference", MTP, 2000.

[7] R. Ramakrishnan, J. Gehrke, "Database Management Systems", 2nd Edition, McGraw-Hill Higher Education, 2000.

[8] D. Solomon. "Inside Windows NT", 2nd Edition, Microsoft Press, 1998

[9] M. Stonebraker. "Operating System Support for Database Management", Communications of the ACM, vol.24, no. 7, pp. 412-418, 1981.