Performance Evaluation of Extended Storage Architectures for Transaction Processing

Erhard Rahm

University Kaiserslautern, Germany
E-mail: rahm@informatik.uni-kl.de

Abstract:

The use of non-volatile semiconductor memory within an extended storage hierarchy promises significant performance improvements for transaction processing. Although page-addressable semiconductor memories like extended memory, solid-state disks and disk caches are commercially available since several years, no detailed investigation of their use for transaction processing has been performed so far. We present a comprehensive simulation study that compares the performance of these storage types and of different usage forms. The following usage forms are considered: allocation of entire log and database files in non-volatile semiconductor memory, using a so-called write buffer to perform disk writes asynchronously, and caching of database pages at intermediate storage levels (in addition to main memory caching). Simulation results will be presented for the debit-credit workload frequently used in transaction processing benchmarks.

1. Introduction

Disk I/O is a significant performance factor for transaction processing. Typically, a large portion of a transaction's response time is determined by synchronous disk I/O, e.g., for reading in a database page or writing log data. Furthermore, the overhead for disk I/Os (process switches, etc.) reduces the effective CPU utilization and thus throughput. What is more, long I/O delays may prevent full utilization of the available CPU capacity. This danger increasingly becomes a reality since CPU speed is improving at a high rate while only modest improvements in disk latency could be achieved so far [PGK88]. A consequence of this growing speed mismatch is that faster CPUs require much higher multiprogramming levels to overlap I/O deactivations. High multiprogramming levels, however, cause increased data contention and potentially lock thrashing that may prevent full CPU utilization [BHR91].

There are numerous approaches to improve I/O performance. Database management systems (DBMS) typically offer a variety of access methods like index structures, hashing schemes or clustering to optimize the physical database structure according to the application's access characteristics. DBMS also cache database pages in main memory to limit the number of disk accesses. Increasing the size of the main memory database buffer together with the CPU speed is a simple means to improve I/O performance since hit ratios may be increased (fewer disk reads). On the other hand, the number of disk writes (logging, database writes) is not improved by a larger main memory buffer. In addition, it is unlikely that the I/O delay per transaction can be reduced by an increased main memory buffer as much as the CPU speed improves. This is also because the database size on disk grows constantly and the database buffer must cache pages for more concurrent transactions.

Main memory databases (e.g., [GLV84, De84, Le86, Ei89]) promise a complete solution to the I/O problem by storing the entire database in main memory. One problem of main memory databases is cost. While the cost per megabyte declines faster for main memory than for disks, disks still have a significant cost advantage particularly for mainframe architectures. Apart from technical problems, keeping large databases of hundreds of gigabytes memory-resident is simply not cost-effective for the foreseeable future [GP87, CKS91]. Mixed solutions where only some databases are kept memory-resident while others reside on disk incur a high DBMS complexity to support both access modes (e.g., different types of access paths, different query optimization strategies, etc.).

Another approach to improve I/O performance is the use of *disk arrays* [PGK88, GHW90]. The main idea is to replace a single large disk drive by an array of many smaller drives to improve I/O bandwidth and I/O rates. On the other hand, access to a single page (which is the dominating access type in transaction processing) is not improved, but likely to be slower. In proposals like RAID (redundant arrays of inexpensive disks) [PGK88] up to four disk accesses are needed to update a single page because parity information stored on separate disks must be accessed and updated (for fault tolerance reasons). Higher I/O latency, however, increases transaction response time and therefore data contention (longer lock holding times).

In this paper, we consider the use of extended storage hierarchies with intermediate storage levels between main memory and disk to improve I/O performance for transaction processing. Non-volatile semiconductor memories are particularly attractive as they provide not only fast access times but can also reduce the number of disk writes. In [CKKS89], the use of a so-called "safe RAM" has been proposed to improve transaction processing performance. Safe RAM is supposed to be a DRAM memory with enough backup power to copy the memory contents to a disk after a power failure. All write I/Os (database and log writes) should be directed to this store so that database reads remain the only I/O delays for transactions. The authors argue that a comparatively small store is sufficient to significantly improve performance compared to a disk-based architecture. They also provide cost estimates to demonstrate the cost-effectiveness of such an approach.

There have been some performance studies on the use of disk caches, but these studies were not specifically concerned with transaction processing applications. In [Sm85], for instance, the use of disk caches was investigated for three I/O traces from large IBM installations for which the disk caches were found to be very effective. This study used the cache miss ratios as the primary performance metric and did not consider caching at multiple levels of the storage hierarchy.

We present a detailed performance study that analyses the usefulness of three different types of intermediate storage for

1992 ACM SIGMOD - 6/92/CA, USA
© 1992 ACM 0-89791-522-4/92/0005/0308...$1.50

transaction processing: disk caches, solid-state disks and extended memory. We are not aware of any other performance analysis that compares these storage types side by side. We consider caching of database pages in main memory, in extended memory and in volatile or non-volatile disk caches. Furthemore, our simulation sytem supports the use of a write buffer in extended memory or in disk caches; portitions of the database may be kept resident in main memory or can be allocated to extended memory, solid-state disks or regular disks. Our study is not limited to internal performance metrics like miss ratios but directly determines throughput and response time results.

Some of the questions we try to answer with our simulation study are:

- What is the relative performance improvement for each type of intermediate storage compared to disk-based configurations ?
- Can less expensive storage types (e.g., disk caches) achieve comparable performance than expensive ones (e.g., extended memory) ?
- Does it make sense to use two or even three of the intermediate storage types together ?
- How does caching of database pages at more than one storage level affect performance ?
- Is a FORCE update strategy [HR83] where all modified pages are written from main memory to the permanent database at commit time affordable in the presence of non-volatile semiconductor memory[1] ?

The rest of this paper is structured as follows. The next section discusses the use of extended storage hierarchies in more detail. In section 3, we describe our simulation model. Section 4 presents the experiments conducted and analyses the simulation results. Finally, we summarize our main findings in section 5.

2. Extended Storage Architectures

In this section, we focus on the use of an extended storage hierarchy to improve I/O performance for transaction processing. For this purpose, we consider three types of page-addressable semiconductor memories: disk caches, solid-state disks (SSD) and extended main memory (Fig. 2.1). They are based on semiconductor memory thus permitting substantially better access times and I/O rates than disks. In contrast to main memory, these memories cannot directly be addressed by machine instructions but are page-addressable similar to disks. This means that in order to read data from such an intermediate memory, the corresponding page must be read into main memory. Similarly, data cannot directly be modified in the intermediate memory but pages are altered in main memory and written back at a later point in time. This page-oriented access interface offers better failure isolation than main memory against processor failures and software errors. In addition, the simpler access interface permits a lower cost per megabyte than for main memory. SSDs are always non-volatile (as the name implies) while disk caches and extended memories are currently mostly volatile. However, non-volatility can be achieved for all three memory types by using a battery backup or uninterruptable power supply.

Approximate values for cost per megabyte and access latency

[1] FORCE permits simpler logging and recovery procedures compared to the NOFORCE alternative requiring special checkpointing techniques and redo recovery after a system crash [HR83]. In disk-based DBMS, FORCE is generally not acceptable for high-volume applications since it can incur a significant increase in response time, data contention and I/O overhead. Meanwhile, most DBMS adopt the NOFORCE approach, but FORCE is still used in several existing DBMS including IMS (Full Function).

	price per MB *(for large systems)*	avg. access time per page (4 KB)
extended memory	500 - 1500 $	10 - 100 microsec
SSD	200 - 800 $	1 - 3 ms
disk cache	?	1 - 3 ms
disk	3 - 20 $	10 - 20 ms

Table 2.1: Storage costs and access times

(as of 1991) are given in Table 2.1. The storage costs refer to mainframe systems and are therefore much higher than for PCs or workstations. Solid-state disks improve the access time per page by about a factor 10 compared to disks, however at a 20- to 50-fold cost per MB. Extended memory is about twice as expensive than solid-state disks [Ku87], but about 50- to 100-times faster. Typically, main memory is twice as expensive as extended memory (per MB).

Disk caches [Sm85, Gro85, Gro89] are completely managed by the disk controllers and their existence is thus transparent to the accessing systems. That is, data in the disk cache is accessed via the conventional channel-oriented disk interface with access times largely determined by the speed of the channel and disk controller. While volatile disk caches can only improve read performance, non-volatile caches also speed up disk writes. *Solid-state disks* are functionally equivalent to disks but keep the entire data (all files) in non-volatile semiconductor memory [Ku87]. The channel-oriented interface results in about the same access time than for disk caches. However, disk caches keep only the 'active' data in semiconductor memory so that for some fraction of accesses the slow disk accesses remain. Thus, the average access time for a SSD is better than for disks with a disk cache. On the other hand, a comparatively small disk cache may already be sufficient to save many disk accesses thereby reducing cost compared to solid-state disks.

Extended memory is used in IBM 3090 mainframe computers as a volatile main memory extension [CKB89]. In contrast to disk caches and SSDs, this so-called expanded storage (ES) has no channel-oriented interface but is largely managed by software in the operating system (MVS, VM). Special machine instructions are provided to move pages between main memory and ES. Currently, access times are two to three orders of magnitudes faster than for SSDs and disk caches (about 75 microsec per 4 KB page including OS overhead). Since a process switch (typically costing several thousand instructions) would be more expensive than this delay, accesses to ES are *synchronous*, i.e. the CPU is not released during the page transfer. While conceptually the ES sits between main memory and the disk subsystem in the storage hierarchy, pages cannot directly

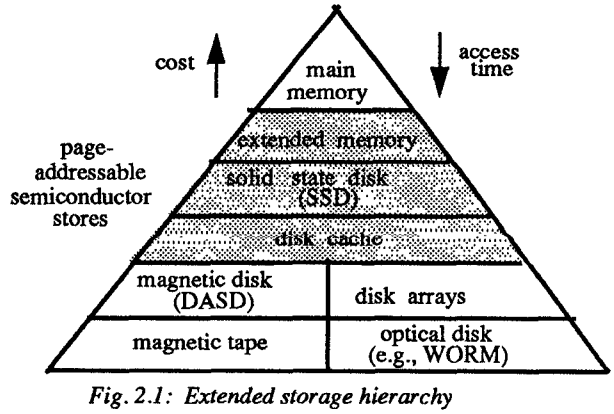

Fig. 2.1: Extended storage hierarchy

migrate from ES to disk. Rather all data transfers between ES and disk must go through main memory since page transfers are controlled by the accessing system rather than by a separate ES controller.

Originally, the ES has only been used as a fast paging and swapping device controlled by the operating system (LRU replacement of pages in ES). Meanwhile more flexible OS services have been provided to permit programs (in particular, the DBMS) to maintain data in ES [Ru89]. Fujitsu offers an ES-like store called SSU (System Storage Unit) which is non-volatile, has a capacity of up to 2 GB and supports a transfer rate of 300 MB/s between main memory and SSU. In [BHR91, Ra91a], a special type of non-volatile extended memory has been considered for use in centralized and locally distributed transaction systems. In our performance study here, we will only consider *non-volatile extended memory (NVEM)*.

	NVEM	SSD	disk cache non-vol.	disk cache volatile
resident files (database, log)	+	+	-	-
write buffer (database, log)	+	-	+	-
database buffer	+	-	+	+

Table 2.2: Usage forms of intermediate storage types

As shown in Table 2.2, there are three basic usage forms of the storage types for transaction processing. The first one is to keep entire (database or log) files resident in non-volatile semiconductor memory (NVEM or SSD) thereby eliminating all disk I/Os for the respective files. The second possibility is to keep a write buffer in non-volatile semiconductor memory (NVEM or disk cache). This approach fastens page writes since the respective transaction can continue processing as soon as the page has been written to the write buffer in semiconductor memory. The disk copy of the corresponding page is updated asynchronously, i.e. without increasing response time. Finally, the number of disk reads can be reduced by caching database pages in a second-level database buffer (extended memory, disk cache) which may be volatile. Database reads could also be reduced by an increased main memory buffer, but at a higher storage cost. Table 2.2 illustrates that only NVEM supports all three usage forms, while SSDs are limited to keep entire files and disk caches may be used as a write buffer and/or for caching database pages.

3. Simulation model

We developed a comprehensive simulation system called TPSIM for studying a variety of storage architectures for transaction processing. TPSIM has been implemented using the DeNet simulation language [Li89]. While TPSIM supports centralized and distributed transaction systems, we concentrate on the central case in this paper. In our model, a transaction system consists of three major parts (Fig. 3.1): a SOURCE which generates the workload of the system, a computing module (CM) that is responsible for processing the transactions, and a set of peripheral devices for storing database and log files. In 3.1, we describe the SOURCE component as well as our database model. Subsections 3.2 and 3.3 cover the CM model and external storage model, respectively.

3.1 Database and load model

For database performance evaluation, the database and workload model is of great importance since it largely determines the performance results and the value of a study. To cover a wide range of applications, we have built three workload generators supporting synthetic workloads and the use of database traces. One SOURCE modul creates general synthetic transaction loads with a high flexibility for studying different load profiles. In particular, our synthetic model supports flexible definition of non-uniform access pattern by means of a so-called relative reference matrix. Dedicated SOURCE modules support the use of database traces and the generation of Debit-Credit transactions according to the benchmark definition in [An85, Gr91]. Unfortunately, we can only present simulation results for Debit-Credit in this paper due to space constraints. As a result we restrict the describtion of the workload and database model to this case. The reduced set of parameters for the Debit-Credit workload is shown in Table 3.1. Workload generation and simulation results for the other load types are described in an extended version of this paper [Ra91b].

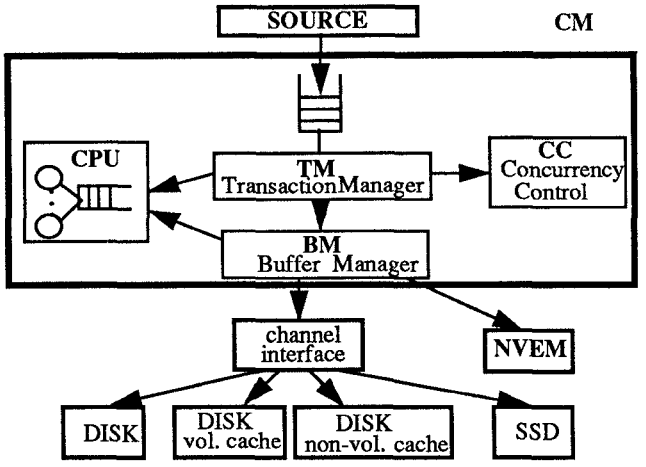

Fig. 3.1: Gross structure of the TPSIM system

The Debit-Credit workload consists of a single transaction type that accesses/updates four record types (ACCOUNT, BRANCH, TELLER and HISTORY). The arrival rate is provided as a simulation parameter. Each record type is stored in a separate database *partition*. Partitions are used to allocate the database to external devices and to specify a concurrency control strategy (see below). A partition consists of a number of database pages which in turn consist of a specific number of objects (e.g., records). The number of objects per page is determined by the blocking factor which can be specified on a per-partition basis. Differentiating between objects and pages is important in order to study the effect of clustering which aims at reducing the number of page accesses (disk I/Os) by storing related objects into the same page. Furthermore, concurrency control may now be performed on the page or object level.

There is a many-to-one relationship between ACCOUNT and BRANCH records and between TELLER and BRANCH records. The number of objects for these partitions determine how many ACCOUNT and TELLER records belong to the same BRANCH record. While the BRANCH record is randomly selected for a transaction, the TELLER record is (randomly) selected from the set of TELLER records associated with the

Parameter	Meaning
ArrRate	arrival rate
K	fraction of local ACCOUNT accesses
Clustering	clustering of BRANCH/TELLER records
Per-Partition Parameters	
NumObjects	number of objects in the partition
BlockFactor	blocking factor for the partition

Table 3.1: SOURCE parameters for Debit-Credit workload

selected BRANCH record. K% of the ACCOUNT accesses are to an account associated with the selected branch, while the remaining accesses go to an account of another branch (in [An85], K=85). The HISTORY partition is sequentially accessed by all transactions. A separate parameter permits clustering of BRANCH and TELLER records. In this case, TELLER records are stored in the same page where their associated BRANCH record is stored. This reduces the number of page accesses per transaction to three and is likely to improve hit ratios; in the case of page-level concurrency control data contention is also reduced.

Every transaction references the four record types in the same order so that no deadlocks can occur. The small TELLER and BRANCH record types are accessed last to keep lock holding times for them as short as possible.

3.2 CM model

The CM is responsible for processing the transactions assigned to it by the SOURCE component. As indicated in Fig. 3.1, a CM is represented by a transaction manager (TM), a buffer manager (BM), a concurrency control component (CC) and CPU servers. The main parameters of these components are shown in Table 3.2.

The *transaction manager* TM controls execution of the transactions. Its multiprogramming level (MPL) only determines the maximal number of concurrently active transactions as we use an open system. In the case that all MPL 'processing slots' are occupied, newly arriving transactions must wait in an input queue until they can be served. To account for the execution cost of a transaction, TM requests CPU service at the begin of a transaction (BOT), for every object access and at the end of a transaction (EOT). The actual number of instructions for each of these services is exponentially distributed over a mean specified as a parameter. Processing an object access also entails requesting an appropriate (read or write) lock from the CC component and asking the buffer manager to bring the corresponding database page into the main memory buffer (if not there already). Commit processing consists of two phases. In phase 1, the BM is requested to write log data and possibly to force modified database pages to non-volatile storage. In phase 2, the CC is requested to release the transaction's locks.

Parameter	Meaning
MPL	multiprogramming level
InstrBOT	average number of instructions for BOT
InstrOR	avg. no. of instructions per object reference
InstrEOT	avg. no. of instructions for EOT
$CCmode_i$	no CC, page-level CC, or object-level CC for partition i
NumCPU	number of CPUs
MIPS	MIPS rate per CPU
BufferSize	size of main memory database buffer
UpdateStrategy	FORCE or NOFORCE
Logging	yes / no
InstrIO	avg. no. of instructions per I/O
InstrNVEM	avg. no. of instructions per NVEM access
$MemResident_i$	memory residence of partition i (yes/no)
$AccessMode_i$	synchr. or asynchr. access to partition i
CacheSizeNVEM	size of NVEM cache
$CachingNVEM_i$	NVEM caching mode for partition i
$WriteBufferNVEM_i$	Use of NVEM write buffer for partition i (y/n)
WrBufferSizeNVEM	Size of write buffer in NVEM

Table 3.2: CM parameters

For *concurrency control*, we use strict two-phase locking (long read and write locks) together with a deadlock detection scheme. Deadlock checks are performed for every denied lock request; the transaction causing the deadlock is aborted to break the cycle. Our simulation system provides a choice between page- and object-level locking. For comparison purposes, it is also possible to switch off concurrency control (no lock conflicts). These choices are offered on a per-partition basis. This flexibility is desirable since real DBMS also use different locking strategies for different object types. For instance, we can now use page-level locking for 'normal' database objects, object-level locking for frequently accessed administration data, and no locking for objects for which accesses are synchronized by using latches or tailored protocols (e.g., HISTORY accesses for Debit-Credit).

CPU requests are served by a single CPU or multiple CPUs (multiprocessor). The number of CPUs and the capacity per CPU in MIPS are provided as simulation parameters. Modelling synchronous accesses to storage devices required a special CPU interface to keep the CPU busy until after an access has been completed.

The *buffer manager* (BM) is responsible for caching of database pages in main memory, for logging and for managing a write buffer and/or database cache in extended memory (NVEM). The database buffers in main memory and extended memory are managed according to a global LRU (least recently used) replacement strategy. Logging is modelled by writing a single page per update transaction to the log file[2]. In the case of a FORCE update strategy, all pages modified by a transaction are also written out at commit time. In the case of NOFORCE, we have ignored the checkpointing overhead assuming a fuzzy checkpointing scheme [HR83] which incurs little overhead during normal processing.

Database partitions can be kept memory-resident (to simulate main memory databases) or they can be allocated to a number of different storage devices (see below). For memory-resident partitions, obviously no caching is necessary (100% hit ratio) and a NOFORCE scheme for update propagation is assumed (i.e. only logging is performed at commit time). If a database partition resides on an external (non-volatile) storage medium, it is accessed either synchronously or asynchronously. In both cases the buffer manager requests CPU service to account for the I/O overhead. For asynchronous accesses the CPU is released before the I/O is actually performed, while synchronous accesses keep the CPU busy until the read or write access is completed.

The use of a write buffer and/or a 2nd-level database cache in extended memory is also managed by the buffer manager as it could be perfomed by the DBMS buffer manager in a real implementation. In TPSIM, the use of the NVEM write buffer and of the extended database buffer can be selected on a per-partition basis. Different modes of NVEM caching can be chosen depending on which pages should migrate to the extended database buffer when being replaced from the main memory cache (only modified pages, only unmodified pages or all pages). Management of the NVEM cache also depends on the chosen update strategy (NOFORCE or FORCE). In the case of NOFORCE, we ensure that every page is cached at most once either in main memory or in NVEM. Therefore, whenever a page migrates from main memory to NVEM because of a replacement decision (or from NVEM to main memory because of a main memory miss and a NVEM hit), the page copy in

[2] Possible optimizations like group commit or asynchronous buffer replacement from main memory are not yet supported. Although they are important for disk-based DBMS, they would reduce the performance differences for the new I/O devices. One conclusion we will draw from our performance study is that the use of non-volatile semiconductor memory reduces the need for such optimizations thereby simplifying buffer management.

main memory (NVEM) is deleted. As a result, the NVEM cache corresponds to a real extension of the main memory cache with the most frequently accessed pages in main memory. With FORCE such an approach is not appropriate since all page updates are written to the NVEM cache at EOT. If pages written to NVEM would be eliminated from main memory at EOT, we could get a very low buffer utilization and poor hit ratios in main memory. Hence, we leave pages that are written to the NVEM cache in main memory resulting in some replication of pages.

For both update strategies (NOFORCE and FORCE), we did not model a deferred propagation of modified pages from NVEM to disk. Rather, whenever a modified page is written from main memory to NVEM we directly start an asynchronous disk write for the respective page. The main advantage of this simple approach is that volatile memory can be used for the cache thereby reducing overall cost. Non-volatility is only needed for a small write buffer. With such an implementation, modified pages are written to both the cache and the write buffer. Writes occur at the speed of extended memory since the disk is updated asynchronously from the write buffer. All pages in the NVEM cache can therefore be considered unmodified so that they can be replaced from the cache without delay.

A deferred update strategy could reduce the I/O overhead and frequency of disk writes if a modified page in NVEM is updated multiple times before being replaced from NVEM. On the other hand, if the page is not modified again extra overhead is introduced since the page must be read from NVEM to main memory before it can be written to disk. For NOFORCE, the chosen approach seems reasonable since when a modified page is written to NVEM (replaced from main memory) this indicates that it has not been referenced for some time so that the likelihood that the page will be modified again in the near future is small. For FORCE, on the other hand, a deferred update strategy is clearly desirable for frequently modified pages. On the other hand, the write traffic to NVEM is expected to be much higher than for NOFORCE permitting only a comparatively short residence time of pages in NVEM before a replacement becomes necessary to make room for new pages. Hence, for the majority of pages the simple update strategy may also be a good choice for FORCE.

3.3 External devices

Database and log files can be allocated to a variety of external storage devices. Currently we support the use of conventional disks, disks with volatile or non-volatile disk caches, solid-state disks and the use of non-volatile extended memory (NVEM).There are numerous possibilities for allocating a database partition using up to four levels of the storage hierarchy (main memory, NVEM, disk cache / SSD, disk)[3]. A database partition is stored either on a regular disk, a solid-state disk, in NVEM or in main memory. Caching of database pages is supported at three levels, namely in main memory, in extended memory and in volatile or non-volatile disk caches. Furthermore, a write buffer may be used either in NVEM or in a non-volatile disk cache. The log file can be allocated in one of the following ways: NVEM-resident, SSD, disk with a write buffer either in NVEM or in disk cache, or on disk without using a write buffer.

[3] Not all combinations that could be chosen are meaningful. For instance, a write buffer for a partition should be used either in NVEM or in a volatile disk cache, but not in both storage types. Similarly, when NVEM caching is employed for a partition there is no further need for a write buffer in the disk controller.

Table 3.3 shows the major parameters for defining the external storage configuration. There can be at most one NVEM and an arbitrary number of so-called disk-units. *Disk-unit* is used as a generic term for devices that offer a disk interface such as solid-state-disks, and disks with or without cache. The parameter "DBallocation" specifies for every partition whether it is stored in NVEM or, if not, to which disk-unit it is assigned. Similarly, the log file is assigned to NVEM or to one of the disk-units.

A NVEM access is modelled by keeping a NVEM server busy for a specified service time. This access time includes the time to transfer the page between main memory and NVEM (NVEM is directly accessed by the CM). Multiple NVEM servers may be selected to permit concurrent NVEM access by different transactions (in the case of synchronous NVEM access, the number of CPUs determines the maximal concurrency).

Disk-units have in common that they are managed by one or more disk controller(s) and that there is a transmission delay for exchanging pages between main memory and disk-units. The number of controllers per disk-unit and the average page service time of the controller are provided as parameters. We did not explicitly model a channel subsystem, but assumed sufficient capacity so that page transfers do not cause a bottleneck.

If a disk-unit is used as a SSD, the I/O delay is determined by the transmission time and the queuing and service time at the controller assuming that the entire partition or log file is kept in semiconductor storage. For the other disk-unit types, one or more disk server(s) are modelled to account for the disk access time. The use of multiple disk servers represents the case where a partition is (uniformly) spread across multiple disks. In the case of regular disk-units (no SSD or disk cache), every I/O results in a disk access in addition to the controller delay and transmission time.

For the mangement of disk caches we followed the realization of IBM's disk caches. We employ a LRU replacement scheme for both volatile and non-volatile disk caches. For disk-units with volatile cache, every write I/O results in a disk access as in the case without cache. If the page to be written is found in the disk cache ('write hit'), the copy in the cache is refreshed (conceptually) and the LRU information is updated; on a write miss the cache contents remains unaffected. For read I/Os the disk access can be avoided, if the respective page is found in the disk cache ('read hit'). If a read miss occurs, the page is read from disk, stored in the disk cache and transferred to the requesting CM.

Parameter	Meaning
NumDiskUnits	number of disk units
$DBallocation_i$	allocation of database partition i
LogAllocation	allocation of log file
NumNVEMservers	number of NVEM servers (controllers)
NVEMdelay	average NVEM access time per page
Per-Disk-Unit Parameters	
DiskUnitType	regular, vol. cache, non-vol. cache, SSD
NumControllers	number of disk controllers
ContrDelay	average controller service time
TransDelay	average transmission time per page
NumDisks	number of disks
DiskDelay	average disk access time per page
CacheSize	size of disk cache / write buffer

Table 3.3: Parameters for external storage devices

In the case of a non-volatile disk cache, it is tried to satisfy all write I/Os in the disk cache and to update the disk copy of a modified page asynchronously, i.e. after the 'I/O done' signal has been returned to the CM. This is always possible for a write hit since no other page needs to be replaced from the cache in

this case. If a write miss occurs, we select the least recently accessed unmodified page from the cache as the replacement candidate (a page is considered as unmodified as soon as its disk copy has been updated). When there is no unmodified page in the cache, i.e. for all cached pages the disk update is not yet completed, we cannot satisfy the write I/O in the cache but directly go to the disk. To reduce the likelihood of this case, we immediately start the disk update when a modified page is stored in the disk cache. As for volatile disk caches, read I/Os are satisfied in the cache if possible (read hit) and a page is stored in the cache after a read miss.

If a disk-unit with non-volatile cache is solely used for logging, we do not employ LRU replacement, but simply use the disk cache as a write buffer to avoid synchronous disk writes if possible.

The described use of disk caches corresponds to the management of currently available caches, specifically the IBM 3990 disk cache [MH88]. To reduce cost, however, the 3990 cache uses non-volatile memory only for a write buffer (called non-volatile store, NVS) while the cache itself is volatile. The performance should be the same than with our method because they also bring every modified page (write hit or write miss) into the cache [MH88].

4. Experiments and Results

In this section, we present our performance results for a variety of storage configurations. Response time will be the primary performance metric in this study since our simulation system uses an open queuing model. (TPSIM also computes detailed statistics on the composition of response time and device utilization, waiting times, queue lengths, lock behavior, hit ratios, etc. in order to explain the results). In 4.1, the main parameter settings for the experiments are described. We study different allocation schemes for the log file (4.2) and database partitions (4.3). In addition we investigate the impact of the update strategy (FORCE vs. NOFORCE, 4.4) and of caching at different levels (4.5). Additional experiments using real-life database traces and other synthetic workloads are described in [Ra91b].

4.1 Parameter settings

Table 4.1 shows the default parameter settings for the Debit-Credit experiments. In all experiments, we used clustering of BRANCH and TELLER records (see 3.1) so that BRANCH and TELLER records reside in the same partition and only three different pages are accessed by a transaction. The database consists of 500 BRANCH/TELLER pages and 5 million ACCOUNT pages. The size of the HISTORY partition is immaterial here since every transaction adds a new record at the end of this sequential file. We did not set locks for HISTORY assuming an implementation that synchronizes accesses to the current end of this file by latches. The average pathlength of a transaction is 250.000 instructions (BOT, four object references, EOT) excluding I/O overhead. Given an aggregate CPU capacity of 200 MIPS, a theoretical maximum of 800 TPS (transactions per second) can be processed. CPU processing accounts for 5 ms per transaction in the case of 50 MIPS CPUs. The multiprogramming level has been chosen high enough to avoid queuing delays at the TM. Without I/O queuing delays, the average access time per page is 50 microseconds for NVEM, 1.4 ms for SSD and disk cache, 6.4 ms for log disks and 16.4 ms for disks storing database partitions. For log disks, a reduced access time has been assumed since the log file is sequentially accessed shortening disk seek times. The default access mode is synchronous for NVEM-resident data, and asynchronous for data stored on disk-units.

Parameter	Settings
NumObjects	500 (BRANCH partition), 5.000 (TELLER), 50.000.000 (ACCOUNT)
BlockFactor	1 (BRANCH), 10 (TELLER), 10 (ACCOUNT), 20 (HISTORY)
K	85
Clustering	True
InstrBOT	40.000
InstrOR	40.000
InstrEOT	50.000
CCmode	page-level CC (BRANCH, TELLER, ACCOUNT), no CC (HISTORY)
NumCPU	4
MIPS	50
BufferSize	2000 pages
Logging	yes
InstrIO	3000
InstrNVEM	300
AccessMode	synchronous for NVEM-resident files, asynchronous otherwise
NumNVEMservers	1
NVEMdelay	50 microseconds
ContrDelay	1 ms
TransDelay	0.4 ms
DiskDelay	15 ms for DB disks, 5 ms for log disks

Table 4.1: Parameter settings for Debit-Credit

Parameters that are changed include the arrival rate, the allocation of log and database files, the update strategy (FORCE, NOFORCE), cache sizes, and the number of controllers and disk servers per disk-unit.

4.2 Allocation of log file

In our first experiment, we considered four alternatives for allocation of the log file: 1) the log file resides on a single disk, 2) log file is on a single disk with non-volatile cache used as a write buffer (cache size: 500 pages), 3) the log is kept in solid-state disk, and 4) the log is stored in non-volatile extended memory. In all cases, the database partitions are stored on a sufficient number of regular disks so that no bottlenecks are introduced. NOFORCE was employed as the update strategy.

Fig. 4.1 shows the average transaction response time for the four log file allocations. Arrival rates from 10 to 700 transactions per second (TPS) have been used, resulting in a CPU utilization of about 90% for 700 TPS. As expected, a single log disk creates a bottleneck and limits the maximal transaction rate to about 180 to 200 TPS for our parameter settings (due to the chosen disk service time of 5 ms). In the case of a single log disk without cache, queuing delays at the log disk cause a steep response time increase for arrival rates of more than 100 TPS. The use of a non-volatile disk cache (write buffer) helps to keep response time low and almost constant over the entire range from 10 to 200 TPS! This is because in this range all log writes could be satisfied in the cache while the disk was asynchronously updated. For 200 TPS, the log disk is fully utilized and the disk writes for all cached pages are queued so that no more cache writes were possible. Still, the value of non-volatile disk cache is quite impressing since even for a higher disk utilization asynchronous I/Os are possible supporting better transaction rates and significantly shorter response times than without such a cache.

The two other log allocations did not have a log bottleneck so that 700 TPS could be processed. The best response times were observed for the NVEM-resident log file which incurred an almost negligible log delay. Slightly higher response times were achieved for the SSD-based log. The response time increase for 700 TPS is mainly because of increased CPU waits.

The simulation results show that a write buffer primarily improves response times since the log writes occur at the speed of

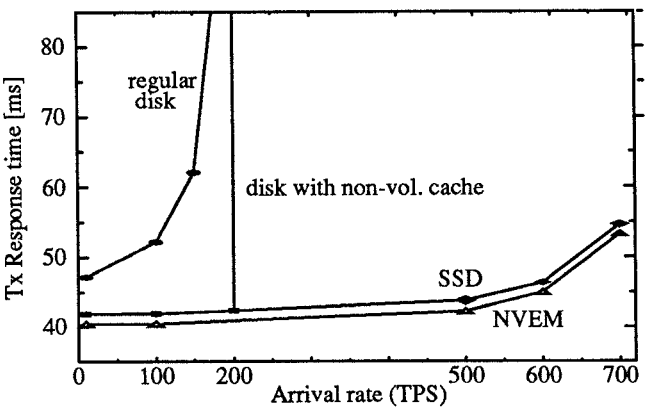

Fig. 4.1: Influence of log file allocation

the respective type of semiconductor memory. The maximal throughput is still limited by the disk I/O rate, although a higher disk utilization can be supported than without write buffer. Group commit would permit significantly higher transaction rates since the log data of multiple transactions can be written in one I/O. However, such transaction rates can also be achieved without group commit if the log is completely allocated to SSD or NVEM. Hence, these storage types supporting high I/O rates reduce the need for optimizations like group commit and permit simpler logging strategies.

Higher transaction rates than with a single log disk could also be achieved by using a disk-array with a declustering of the log file across several disks [De84]. In a RAID-like implementation [PGK88], however, the updating of parity information would result in higher response times and could also lead to bottlenecks. The use of non-volatile disk caches or write buffers could largely improve the write performance of disk arrays.

4.3 Allocation of database partitions

We studied the following six alternatives for allocating the database partitions: 1) all partitions (and the log) on disks without cache, 2) all partitions and log on disks with non-volatile cache used as a write buffer, 3) like 2 but with the write buffer in NVEM, 4) all partitions and log on SSD, 5) all partitions and log in NVEM, 6) all partitions main memory-resident, log on disk. Database partitions and the log have been assigned to the same device type to emphasize the relative differences. In all cases we used a sufficiently high number of disk servers and controllers to avoid bottlenecks. Again, the update strategy was NOFORCE.

Fig. 4.2 shows the response time results for the above listed configurations. Although the absolute values are small in all cases, the relative differences are significant. All configurations are CPU-bound since we eliminated potential I/O bottlenecks and the amount of lock contention was modest. The best results were again reached in the case of NVEM-resident data; in this case response time is almost exclusively determined by the queuing and service times at the CPU. The SSD-based configurations also achieved very short response times. For memory-resident partitions response times are higher than for NVEM-resident partitions because of the disk I/O for logging. If the log had been allocated to NVEM in this case, about the same response times than for NVEM-resident partitions were achieved. Memory-resident partitions have an advantage at higher transaction rates since they do not incur I/O overhead for database accesses but only for logging permitting reduced CPU waiting time and slightly higher throughput. This is also the reason why response time for main memory-resident partitions is better than for SSD-based partitions at 700 TPS in Fig. 4.2. Still, one can conclude that keeping the database in NVEM or SSD brings performance comparable to main memory databases, but at a lower cost. In addition, NVEM- and SSD-resident files can be supported by the operating system without affecting the DBMS, while memory-resident databases require explicit DBMS support[4].

A significant response time improvement could already be obtained by the use of a write buffer either in NVEM or with non-volatile disk caches. Since a small write buffer is already sufficient to achieve these improvements, such an approach is clearly more cost-effective than keeping entire files (in particular, the ACCOUNT and HISTORY relations) resident in semiconductor memory. The NVEM write buffer is only slightly better than a disk cache write buffer so that the latter would be sufficient. On the other hand, a single NVEM write buffer can be used for multiple disks and disk controllers so that less non-volatile memory may be needed than with a separate write buffer in each disk controller.

The response time values can largely be explained by the I/O behavior. The average hit ratio in main memory was about 72.5%[5] for all arrival rates and configurations (except for memory-resident partitions, of course) resulting in slightly more than 1 miss per transaction (on ACCOUNT). Since all pages are modified for Debit-Credit, every buffer miss resulted in an additional I/O to write back the page to be replaced. As a consequence, about 2 database I/Os and 1 log I/O occur per transaction. In the disk-based configuration, all three I/Os occur at disk speed accounting for about 40 ms. The use of a write buffer largely eliminated the delays for the two writes so that response times could be cut by a factor 2. If the ACCOUNT partition is also kept resident in semiconductor memory, the remaining read disk I/O can also be eliminated.

A more sophisticated buffer manager than the one used in TPSIM would have achieved better response times for the disk-based configuration by asynchronously writing modified pages to disk (before their replacement). In this case, only two synchronous I/Os would have remained per transaction (read I/O for ACCOUNT and the log write) thus considerably reducing the difference to the configurations using a write buffer. On the other hand, one can argue that there is no real need any more to support asynchronous writes in the DBMS buffer manager since the same performance improvements can be achieved by a write buffer in non-volatile semiconductor memory. The write buffer can be managed outside the DBMS, e.g., by the operating system's file manager in the case of a NVEM write buffer or by the disk controllers, so that not only log and database writes benefit from it but also other applications than transaction processing. Hence, using non-volatile semiconductor storage in this way permits simpler DBMS buffer management without sacrificing performance.

Our results suggest that it may be good idea to use more than one type of the intermediate memories together. For instance, the log and the small BRANCH/TELLER partition could be kept resident in non-volatile memory (SSD or NVEM), while the ACCOUNT and HISTORY relations may be stored on regular disks with a write buffer.

[4] However, main memory DBMS would achieve better performance if they could significantly cut transaction pathlengths. In particular, higher transaction rates per MIPS would then be possible.

[5] For a main memory buffer size of 2000 pages, the hit ratio was about 0% for ACCOUNT, 95% for HISTORY (due to the blocking factor 20), 95% for BRANCH and 100% for TELLER (due to the clustering with BRANCH records).

4.4 FORCE vs. NOFORCE update strategy

To study the impact of the update strategy, we used the storage allocations from the last experiment for the case of a FORCE update strategy. We obtained the same order of the different allocation alternatives than for NOFORCE, but the relative differences changed significantly. This is illustrated in Fig. 4.3, where the response time results for three storage allocations are compared with each other.

Response times for FORCE are generally higher than for NOFORCE since there are more I/Os per transaction due to forcing modified pages to the database at commit[6]. While this causes a considerable response time penalty for the disk-based configuration, the differences shrink with increasing speed of the used storage devices (Fig. 4.3). So even with a limited amount of non-volatile memory used as a write buffer, response times for FORCE are almost as good than for NOFORCE. This indicates that high performance is achievable even for a FORCE strategy since FORCE gains more from non-volatile semiconductor memory than the more optimized NOFORCE alternative. It can also be seen from Fig. 4.3 that FORCE using a write buffer supports even better response times than NOFORCE without using non-volatile semiconductor memory.

However, FORCE still causes more disk I/Os so that the I/O overhead is higher and I/O bottlenecks are more likely than for NOFORCE. The increased I/O overhead caused a steeper response time increase for FORCE in the case of 700 TPS since CPU utilization was higher than for NOFORCE. In addition, we had allocated the small BRANCH/TELLER partition to multiple disks to avoid an I/O bottleneck. If this partition were stored on a single disk, throughput for FORCE would be limited to less than 70 TPS in the disk-based configuration or when a write buffer is used. Keeping the BRANCH/TELLER partition resident in SSD or NVEM also avoids this bottleneck for FORCE.

4.5 Influence of caching for Debit-Credit

In addition to main memory caching, we considered buffering of database pages in NVEM and in volatile or non-volatile disk caches . In a first experiment, we varied the main memory buffer size for the different configurations indicated in Fig. 4.4. These simulation runs were conducted for the NOFORCE strategy and an arrival rate of 500 TPS. Results for FORCE will be discussed later in this subsection.

The response time results in Fig. 4.4 refer to main memory buffer sizes from 200 to 5000 pages. In addition to the main memory buffer, we studied the use of a 1000 pages second-level buffer in a volatile and non-volatile disk cache and in NVEM. Furthermore, the results for using a disk cache write buffer and a NVEM cache of 500 pages are shown in Fig. 4.4. Since the main memory buffer is used for all partitions of the database, the second-level cache was also shared for the four partitions. In the configurations using non-volatile disk caches or NVEM, these storage types were also used for logging.

Increasing the main memory buffer is most effective for a size of less than 2000 pages since in this range many misses occurred for the frequently accessed BRANCH/TELLER pages. A buffer size of 2000 pages was needed to keep the 500 BRANCH/TELLER pages in main memory; a larger main memory buffer (5000 pages) did not permit any significant response time improvements any more. The use of a volatile disk cache was only helpful for small main memory buffers where some misses on BRANCH/TELLER could be satisfied in the disk cache. As soon as the main memory buffer had reached the size of the volatile disk cache (1000 pages), no further hits occurred in the disk cache (Table 4.2) so that the same response times than without disk cache resulted. The use of non-volatile semiconductor memory permits substantially more I/O savings since all synchronous disk writes can be eliminated. So the use of a write buffer alone (no read hits) accounted already for the largest improvements compared to the disk-based configuration. The difference from the results with a non-volatile disk cache of 1000 pages to the results for a write buffer correspond to the I/O savings due to read hits in the non-volatile disk cache. Most effective was the use of a NVEM cache. Even a NVEM cache of 500 pages permitted better response times than with a non-volatile disk cache of 1000 pages.

		main memory buffer size			
		200	500	1000	2000
	main memory	53.7	59.6	66.7	72.5
NO-FORCE	vol. disk cache 1000	12.8	5.6	0	0
	nv disk cache 1000	13.0	7.4	3.8	0.8
	NVEM cache 1000	14.8	11.0	5.7	1.1
	NVEM cache 500	9.2	7.1	3.9	0.8
FORCE	vol. disk cache 1000	12.4	6.9	0.1	0
	nv disk cache 1000	12.8	7.0	0.1	0
	NVEM cache 1000	13.1	7.2	3.4	0.6

Table 4.2: Main memory and 2nd-level cache hit ratios (in %)

To analyse the effectiveness of the different cache types in more detail, Table 4.2 summarizes the hit ratios for the simulation runs of Fig. 4.4 (NOFORCE). The main memory hit ratios increase with growing buffer size, while the number of additional hits in the second-level caches decreases (for a main memory buffer size of 5000 pages, there were no more hits in the second-level caches). The table shows that from the three types of second-level caches, the NVEM cache supports the best hit ratios, followed by the use of a non-volatile disk cache. With a volatile disk cache lower read hit ratios than for both non-volatile disk caches and NVEM caches were obtained! Disk caches were less effective than the NVEM cache since they are managed independently from the DBMS buffer in main memory. A consequence of this was that the same pages were frequently cached in main memory and in the disk caches. This was particularly the case for the volatile disk caches: as soon as the main memory buffer size reached the size of the disk cache no more hits occurred in the disk cache holding merely a subset of the main memory cache. The double caching of pages comes from the fact that after a miss in main memory and in the disk cache, the page is cached in the disk cache as well as in main memory, although the hits will occur in main memory in the first place. If the disk cache is larger than the main memory buffer, more pages can be cached there so that some hits in the disk cache can be achieved despite the double caching of the most frequently accessed pages.

NVEM caching achieved better hit ratios than with disk caches primarily because a double caching of pages could completely be avoided for NOFORCE (see section 3.2). In particular, after a main memory miss the respective page is only cached in main memory and not in the NVEM cache. Only pages that are replaced from main memory migrate to the NVEM cache. A result of this technique is that the combined hit ratio for the main memory and NVEM caches was the same than for a main memory buffer of the same aggregate size. For instance, the same

[6] There are three write I/Os to force out the modifications at commit. On the other hand, no write I/O was necessary on a buffer miss because there were always unmodified pages to replace. Since we had the same hit ratios than for NOFORCE, there are about two disk writes more per transaction than in the NOFORCE configurations.

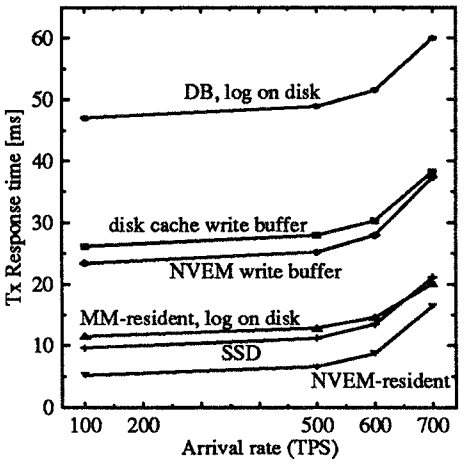

Fig. 4.2: Impact of database allocation

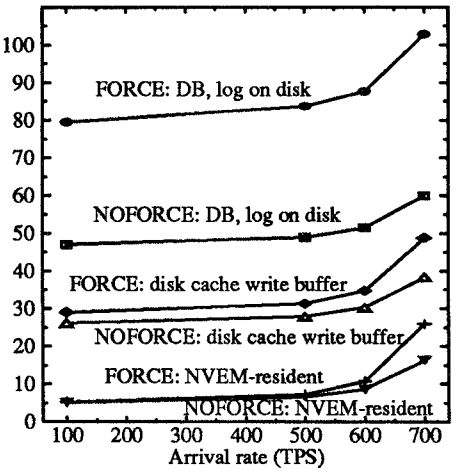

Fig. 4.3: FORCE vs. NOFORCE

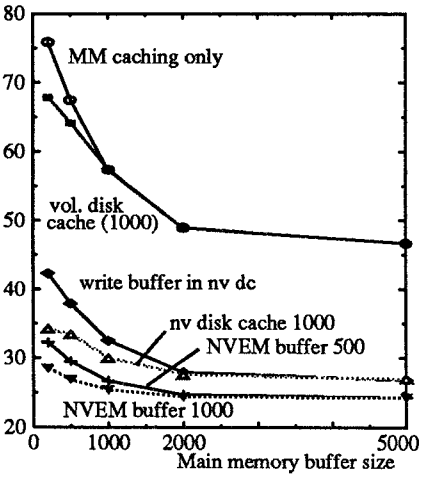

Fig. 4.4: Impact of caching for different main memory buffer sizes (500 TPS)

combined hit ratios are obtained for the combinations of main memory/NVEM cache sizes of 1000/0 and 500/500, 2000/0 and 1000/1000 or 1000/500 and 500/1000 (Table 4.2). Furthermore, since NVEM accesses are very fast basically the same response times can be achieved for NVEM hits than for main memory hits (e.g., in Fig. 4.4 we had the same response times for the combinations 500/1000 and 1000/500). This is an important observation since it indicates that for NOFORCE performance only depends on the aggregate buffer size of main memory and NVEM cache. In particular, more cost-effective solutions can be obtained by choosing a small main memory and a larger NVEM cache size than only having a main memory cache of the same aggregate size. Since this result refers to read hits, it can also be achieved for volatile caches in extended memory.

Non-volatile disk caches reached higher read hit ratios than volatile disk caches not because of the non-volatility but because of the different handling of write misses. For a non-volatile disk cache, a modified page replaced from main memory is inserted into the disk cache for a write miss as well as for a write hit. For volatile disk caches, on the other hand, the page is not cached upon a write miss. Due to the NOFORCE strategy, however, there were many write misses so that in contrast to non-volatile disk caches only few pages migrated from main memory to the volatile disk cache. This result suggests that the effectiveness of (IBM's) volatile disk caches can easily be improved by also caching pages on a write miss for files for which an additional caching is performed in main memory. Similarly, the effectiveness of disk caches could further be improved by not caching a page after a read miss if it is known that the page will be cached in main memory[7]. However, the applicability of such an approach is limited since typically only modified pages are written back from main memory to the disk controller (this is no problem for Debit-Credit where all pages are modified).

When using a FORCE strategy, the effectiveness of the 2nd-level caches is generally lower since more pages are written from main memory to the 2nd-level cache than for NOFORCE. As a result, the average cache residence time per page is reduced thus lowering the probability of a re-reference. This is reflected in Table 4.2 showing that the hit ratios in the 2nd-level cache are generally lower for FORCE than for NOFORCE. It can be seen from the table that the hit ratios for volatile disk caches are now very close to the values for non-volatile disk caches. This is due to the fact that FORCE results in a high write hit ratio in the disk cache since a page is written back (at EOT) shortly after it has been read. The highest read hit ratios were still obtained for a NVEM cache, although here the hit ratios decreased most compared to NOFORCE. This was because for FORCE a double caching of pages in main memory and NVEM could not be avoided (see section 3.2).

We did not explicitly study caching at three levels for the same partition, but the results can easily be predicted based on the already presented findings. Disk caches used in addition to NVEM and main memory caching would be similarly (in-)effective than their use in combination with an increased main memory buffer. Since the NVEM already caches modified pages, non-volatility would no longer be necessary for the disk caches. On the other hand, the performance of a NVEM cache could be approached by a database cache in *volatile* extended memory used in combination with disk cache write buffers to avoid synchronous disk writes.

Simulation experiments using traces from real-life applications with a high share of read-only transactions confirmed the improved effectiveness of NVEM caching over disk caches. In these applications, the differences between using volatile and non-volatile disk caches became very small [Ra91b].

5. Conclusions

We have presented a performance evaluation of extended storage hierarchies to improve transaction processing performance. We considered three types of page-addressable semiconductor memory (disk caches, solid-state disks (SSD) and extended memory) that offer substantially lower I/O latency and higher I/O rates than disks. Compared to main memory, they are less expensive and provide better failure isolation due to the page-oriented interface. Non-volatile semiconductor memories can be used to keep entire files resident in them thereby eliminating all (synchronous) disk I/Os for log or database files. A more space-efficient usage of the new memory types results if they are used as a write buffer or for caching database pages at an additional level of the storage hierarchy. A write buffer permits log and database writes to be satisfied in non-volatile semiconductor memory and performing the disk write asynchronously. Caching database pages at an intermediate storage level may reduce the number of disk reads at a lower cost than by increasing the main memory buffer size.

[7] Caching pages after a miss in the disk cache would still be appropriate for sequential files for which prefetching can be utilized.

Our performance study has shown that the use of non-volatile extended memory, SSD and non-volatile disk caches significantly improves response times compared to disk-based configurations in almost all cases. Transaction rates are increased in cases with otherwise low effective CPU utilization because of I/O bottlenecks (e.g., for logging) or lock contention.

We found that the use of a limited amount of non-volatile semiconductor memory reduces the need to employ sophisticated buffer management strategies. This was illustrated by comparing the performance of the FORCE and NOFORCE alternatives for propagating modified database pages to the permanent database. While the simpler FORCE strategy requires more I/Os than NOFORCE, the resulting performance impact often becomes insignificant when all force-writes go to non-volatile semiconductor memory (in fact, performance can be improved compared to NOFORCE configurations without non-volatile semiconductor memory). Similar conclusions apply for other software techniques to limit the number of synchronous disk I/Os like asynchronous page replacement and group commit. On the other hand, if a DBMS already supports these optimizations high transaction rates and sufficiently short response times may be achievable with little or no non-volatile semiconductor memory.

From the intermediate storage types considered here, non-volatile extended memory (NVEM) supports the best performance for transaction processing albeit at the highest cost. If the log and entire database are kept NVEM-resident, the performance is comparable to main memory database systems with a non-volatile log buffer. The use of solid-state disks is a less expensive alternative for keeping entire files resident in semiconductor memory and reduces I/O latency almost to the same degree than NVEM. Similarly, a disk cache write buffer is almost as effective than a NVEM write buffer. The main advantage of NVEM is that it can be used in a more flexible way since it is directly accessible by special machine instructions. So NVEM can be used for storing entire files, but also for caching database pages or as a write buffer (e.g., log buffer). In locally distributed systems, NVEM can be further utilized to speed-up inter-system communication and to hold globally shared data structures [Ra91a]. These extended usage forms require special support by the DBMS or/and operating system, while SSDs and disk caches offer a disk-oriented interface so that their use remains transparent to the DBMS (device independence).

Caching of database pages in a second-level buffer in addition to main memory buffering is most effectively supported by an extended database buffer in NVEM. For NOFORCE, NVEM caching was optimal in the sense that main memory and NVEM caching together achieved the same combined hit ratios than with a main memory buffer of the same aggregate buffer size alone. Since extended memory is less expensive than main memory, the cost-effectiveness of caching can be improved by choosing a small main memory and a large extended memory buffer. NVEM caching supported significantly better hit ratios than the use of volatile or non-volatile disk caches. Current disk caches are optimized for one-level caching so that their use in combination with main memory caching results in a double caching of the most frequently accessed pages. Our results suggest that all pages replaced from the DBMS buffer in main memory should be kept in the second-level database cache for future re-references. This can easily be achieved for the NVEM cache if it is managed by the DBMS. The use of disk caches, however, is transparent to the DBMS so that unmodified pages do not migrate from main memory to the disk cache. Furthermore, modified pages replaced from main memory will not be cached by current volatile disk caches if a write miss occurs. Caching of pages in a second-level cache was found to be less effective for FORCE than for NOFORCE because the high write traffic resulted in short cache residence times per page. In addition, the pages forced out of main memory and stored in the second-level cache, also remained buffered in main memory causing a double caching for modified pages.

While NVEM alone supports all usage forms of intermediate semiconductor memory to reduce the number of synchronous disk I/Os, the reduced cost of disk caches and SSD can make the combined use of two or even three of these storage types desirable. For instance, one could use non-volatile disk caches to implement write buffers and SSD to keep entire files resident in semiconductor memory. Extended memory can then be used to hold a second-level database cache.

References

An85 Anon et al.: A Measure of Transaction Processing Power. Datamation, 112-118, April 1985.

BHR91 Bohn, V.; Härder, T.; Rahm, E.: Extended Memory Support for High Performance Transaction Processing. Proc. 6th (German) Conf. on Measurement, Modelling and Evaluation of Computer Systems, Informatik-Fachberichte 286, Springer-Verlag, 1991.

CKB89 Cohen, E.I.; King, G.M.; Brady, J.T.: Storage Hierarchies. IBM Systems Journal 28 (1), 62-76, 1989.

CKKS89 Copeland, G.; Keller, T.; Krishnamurthy, R.; Smith, M.: The Case for Safe RAM. Proc. 15th VLDB, 1989.

CKS91 Copeland, G.; Keller, T.; Smith, M.: Database Buffer and Disk Configuring and the Battle of the Bottlenecks. Proc. 4th Int. Workshop on High Performance Transaction Systems, Asilomar, CA, Sep. 1991

De84 DeWitt, D. et al.: Implementation Techniques for Main Memory Database Systems. Proc. ACM SIGMOD conf., 1-8, 1984.

Ei89 Eich, M.: Main Memory Database Research Directions. Proc. 6th Int. Workshop on Database Machines, LNCS 368, Springer-Verlag, 251-268, 1989.

GHW90 Gray, J.; Horst, B.; Walker, M.: Parity Striping of Disc Arrays: Low-Cost Reliable Storage with Acceptable Throughput. Proc. 16th VLDB, 148-161, 1990.

GLV84 Garcia-Molina, H.; Lipton, R.J.; Valdes, J.: A Massive Memory Machine. IEEE Trans. on Computers 33 (5), 391-399, 1984.

GP87 Gray, J.; Putzolu, F.: The 5 Minute Rule for Trading Memory for Disk Accesses and the 10 Byte Rule for Trading Memory for CPU Time. Proc. ACM SIGMOD conf., 395-398, 1987.

Gr91 Gray, J. (ed.): The Benchmark Handbook for Database and Transaction Processing Systems. Morgan Kaufmann 1991.

Gro85 Grossman, C.P.: Cache-DASD Storage Design for Improving System Performance. IBM Sys. Journal 24 (3/4), 316-334, 1985.

Gro89 Grossman, C.P.: Evolution of the DASD Storage Control. IBM Systems Journal 28 (2), 196-226, 1989.

HR83 Härder, T.; Reuter, A.: Principles of Transaction-Oriented Database Recovery. ACM Comp. Surveys 15 (4), 287-317, 1983.

Ku87 Kull, D.: Busting the I/O Bottleneck. Computer & Communications Decisions, 101-109, May 1987.

Le86 Lehman, T.J.: Design and Performance Evaluation of a Main Memory Relational Database System. Ph.D. Thesis, Comp. Science Dept., Univ. of Wisconsin, Madison, 1986.

Li89 Livny, M.: DeNet User's Guide. Version 1.6, Comp. Science Dept., Univ. of Wisconsin, Madison, 1989.

MH88 Menon, J.; Hartung, M..: The IBM 3990 Disk Cache. Proc. IEEE Spring CompCon, 146-151, 1988.

PGK88 Patterson, D.A.; Gibson, G.; Katz, R.H.: A Case for Redundant Arrays of Inexpensive Disks (RAID). Proc. ACM SIGMOD conf., 109-116, 1988.

Ra91a Rahm, E.: Use of Global Extended Memory for Distributed Transaction Processing. Proc. of the 4th Int. Workshop on High Performance Transaction Systems, Asilomar, Sep. 1991.

Ra91b Rahm, E.: Performance Evaluation of Extended Storage Architectures for Transaction Processing. TR 216/91, Dept. of Comp. Science, Univ. Kaiserslautern (extended version of this paper)

Ru89 Rubsam, K.G.: MVS Data Services. IBM Systems Journal 28 (1), 151-164, 1989.

Sm85 Smith, A.J.: Disk Cache - Miss Ratio Analysis and Design Considerations. ACM Trans. Comp. Systems 3 (3), 161-203, 1985.