TANDEM

Varieties of Concurrency Control in IMS/VS Fast Path

Dieter Gawlick
David Kinkade

Technical Report 85.6
June 1985
Part Number: 87613

VARIETIES OF CONCURRENCY CONTROL IN IMS/VS FAST PATH

Dieter Gawlick, David Kinkade

June 1985

Tandem Technical Report 85.6

Tandem TR 85.6

VARIETIES OF CONCURRENCY CONTROL IN IMS/VS FAST PATH*

Dieter Gawlick
Amdahl Corporation
1250 E. Arques Avenue, Sunnyvale, CA 94088-3470

David Kinkade
Tandem Computers, Inc.
19333 Vallco Parkway, Cupertino, CA 95014

Abstract

Fast Path supports a variety of methods of concurrency control for a variety of different uses of data. This arose naturally out of efforts to meet the typical needs of large on-line transaction processing businesses. The elements of "optimistic locking" apply to one type of data, and "group commit" applies to all types of data. Fast Path development experience supports the idea that the use of a variety of methods of concurrency control by a single transaction is reasonable and is not as difficult as might be expected, at least when there is fast access to shared memory.

* This article is intended to be published in the June 1985 issue of the **IEEE** bulletin on **Database Engeneering**.

Table of Contents

1. Introduction

IBM introduced the Fast Path feature of IMS/VS in 1976, to support performance-critical on-line transaction processing.

Originally, Fast Path transactions shared the communication network and the system journal with standard IMS transactions, but little else was shared. Fast Path transactions could not access standard IMS/VS data bases and standard IMS/VS transactions could not access Fast Path data bases. Now Fast Path is integrated into IMS/VS, and a single transaction can access both kinds of data bases with consistency and atomicity across all of them. This integration in itself meant that the different methods of concurrency control common to Fast Path and to standard IMS/VS had to be supported within a single environment. In addition to this variety, Fast Path itself has always supported a variety of methods of concurrency control, although this was less apparent in 1976 than it is today, because the methods were simpler then.

Fast Path supports three kinds of data, each with its own methods of concurrency control. Two of these methods were novel when developed:

- Very active data items ("hot spots") are supported by Main Storage Data Bases (MSDBs). Very active data can include branch summaries, total cash on hand, accounting information (the number of transactions of a particular type processed), or any other kinds of information likely to cause bottlenecks because of locking conflicts between simultaneous transactions. Also, MSDBs are useful for terminal-related data, data likely to be accessed by a high percentage of all transactions from a particular terminal.
- Less frequently accessed data is stored in the main portions of Data Entry Data Bases (DEDBs). The main portion of a DEDB is hierarchical, with randomized (hashed) access to the roots. The significant improvement over standard IMS/VS hierarchical randomized data bases is in data availability, not in concurrency control. Because the amount of data is very large, access conflicts are rare and straightforward methods of concurrency control are adequate.
- Application journal data (application historical data) is also supported by DEDBs, in special "sequential dependent" portions of DEDBs. The implementation of sequential dependents avoids a bottleneck that could otherwise result from simultaneous transactions adding records at the end of an application journal.

Some Fast Path innovations (such as group commit and the elimination of "before" images from the system journal) are general, not for the benefit of particular types of data. Still, we will start by considering the special handling for highly active data "hot spots" and for application journal data.

2. Main-Storage Data Bases (MSDBs)

What innovations help with "hot spots"? Consider, for example, an ultra-hot spot: a counter updated by every transaction. To achieve a high transaction rate, we want a transaction to be able to access the counter without waiting for any other transaction. We also want to guarantee the integrity of the counter, consistency with other parts of the data base. etcetera. If we are to have high performance (> 100 transactions per second), then there are three things we cannot afford to do:

a. We cannot afford to lock the record and keep other transactions from accessing it until our transaction in complete.

b. We cannot afford to read the record from disk and write the record to disk.

c. We cannot afford to wait for the previous transaction's update log records to be written to disk (this is not unique to MSDBs, and is handled by group commit, to be described later).

To avoid the bottleneck that would occur from all transactions needing exclusive control of the counter, we must be a little tricky. In our example, the transaction wants to add 1 to the counter. The transaction doesn't care about the old or new values of the counter, except that the old value of the counter must be less than the maximum value the counter can hold. Accordingly, the transaction program does not "read" and "write" the counter value. Instead, it uses two new operations:

a)	*VERIFY counter*	<	*(maximum counter value)*
b)	*CHANGE counter*	+	*1*

The VERIFY operation checks a field and makes sure the field is <, <=. ==, >=, or > a specified value. Multiple VERIFY operations can be used to specify upper and lower limits.

The CHANGE operation updates a field by adding or subtracting a specified value, or by setting a field to a specified value. It could have been extended to include multiplication and division if these had seemed useful.

Fast Path does VERIFY tests both at the time of the VERIFY operation and again during commit time. The first test gives the application program a chance to do something different if the value is already out of range. The second test, the "real" test, is performed as part of a high-speed replay of the VERIFYs and CHANGEs for this transaction. During commit processing the data is locked, but only for a few instructions.

CHANGE operations are performed only during commit processing.

Besides VERIFY and CHANGE operations, Fast Path also allows the more usual operations (reading and writing records) against MSDB data. This is a complication for the developers, but not relevant to this paper.

The VERIFY and CHANGE operations implemented for MSDBs, publicly introduced in 1976, contain the essential elements of "optimistic locking," although the term was then unknown. This is readily apparent if you consider the use of "VERIFY = ."

3. Data-Entry Data Bases (DEDBs)

DEDBs are for large data bases. Except for sequential dependents, we are willing to wait for data to be read from disk (Fast Path doesn't even use buffer look-aside, except within a transaction). We are willing to write each updated record to disk (multiple read-write heads can be assumed), and we are willing for one transaction to wait for the other if both transactions want to access the same record.

Since many readers of this paper will be unfamiliar with Fast Path, it seems reasonable to mention the innovations that improved availability, even though this is not relevant to the main topic of the paper. It will at least make you aware of DEDB "AREAs." Originally there were two big availability improvements over standard IMS/VS:

1. Fast Path allows you to divide a DEDB into up to 240 different partitions called "AREAs." Each AREA is a separate file, and can be taken off-line while leaving the rest of the data base on-line. The internal data structures are such that pointers never point from one AREA to another. (The use of multiple AREAs also allows the construction of very large databases. The limit is 960 gigabytes.)
2. Since Fast Path has no way of telling which transactions will access which AREAs, it lets a transaction try to access data in an unavailable AREA. Fast Path introduced a new IMS/VS status code to tell the application program that the data is unavailable and let the application program take alternate action. Until Fast Path, IMS/VS refused to run a transaction that might try to access unavailable data.

In 1984 another data availability enhancement was provided: data replication, the ability to have from 0 to 7 copies of an AREA. Zero copies, of course, amounts to having the AREA off-line. The data replication facilities allow a new copy to be added and brought into use or allow a copy to be removed and taken away for safe storage without interrupting access to the AREA.

4. DEDB Sequential Dependents

Sequential dependents allow the construction of an application journal. They are to contain historical information. For example, a root record with your account number as its key may show how much money you currently have in your account, but your bank must also keep a history of how much money you used to have, and what happened to change your account balance. The bank would like to be able to get this information without having to scan through all updates to all accounts. At other times the bank would like to be able to scan all the updates for a particular period of time very rapidly; this is typically part of a daily batch run.

These considerations lead to the following strategy:

1. Sequential dependents, once written, can neither be modified nor individually deleted. They are eventually discarded as obsolete.
2. Sequential dependents are stored in chronological order according to time of commit processing (time of completion of the transaction).
3. To support retrieval of sequential dependents related to some particular thing (such as your bank account), without having to scan a lot of unrelated data, sequential dependents are chained in LIFO order from a chain anchor in the root record. This is very efficient; it can be done without accessing previously written sequential dependents.

There are many ways this strategy could have been implemented. The method actually chosen imposes many restrictions. At the time, these restrictions were appropriate, since the general strategy had not yet been tested by use. Here are the rules:

1. A sequential dependent segment (record) must be added to the DEDB as a child of one and only one non-sequential DEDB segment. Only one sequential-dependent segment type is allowed, and this segment type must be a child of the root segment.
2. Once inserted, sequential dependent segments can be retrieved as children of their root segment. Retrieved this way, they are presented in LIFO order. For example, the root segment for your bank account would point to the most recent update for your account, the most recent update record would point to the update before that, and so on.
3. Sequential dependents can also be scanned sequentially, for high-speed bulk processing. In this case, all the sequential dependents for a chosen period of time are presented in chronological order by commit time.
4. Sequential dependent segments are stored in a portion of disk storage reserved for this purpose at the end of each DEDB AREA. This portion of disk storage is used and reused as a large circular buffer.

5. There is no facility for deleting individual sequential dependents. Instead, there is a pointer to the oldest sequential dependent not yet deleted. When this pointer is moved forward, all previous records are logically deleted, and the space they occupied is made available for reuse. Because the space is circularly reused, the most significant bits of all pointers are a "cycle" counter: the first use of the disk storage is cycle 1, the second use is cycle 2, etcetera. Accordingly, a later record always has a larger pointer.

For sequential dependents. the "hot spot" is not a data item; it is the next piece of unused space. To handle this "hot spot," a Sequential Dependent insertion goes through three stages:

1. When the application program tells Fast Path to add a new sequential dependent, the data provided is NOT assigned a location on disk: this would not be a good idea because other transactions could insert sequential dependents and complete before this transaction completes. Instead. the data provided is placed in a holding area, the same as for MSDB VERIFY or CHANGE data. Also. the added data is chained LIFO from the main-storage copy of the record it is a child of. The pointer from the root is recognizable as a main-storage pointer (not a disk pointer) by the fact that its most significant bits (the "cycle" number) are 0.

2. During commit processing, Fast Path allocates the available space to the sequential dependents. After allocating the space, Fast Path revisits the root and all sequential dependents added by this transaction, converting main-storage pointers into disk storage pointers to the newly allocated locations. Also, Fast Path copies the data to the current buffer being filled with sequential dependents for this DEDB AREA.

3. The buffer the sequential dependents are copied to at commit time isn't written to disk until after the buffer is filled and all the commit records for all the transactions that added data to the buffer have been written to the system journal. This delay is not a problem; nobody has to wait for the buffer to be written to disk unless there is a shortage of buffers.

5. Resource Control

The design of Fast Path distinguishes between contention control (making sure this transaction and another transaction aren't incorrectly accessing the same data) and contention resolution (deadlock detection). Contention control is constantly needed. Accordingly, it is handled within Fast Path, using a variety of efficient mechanisms. Contention resolution is needed only when a conflict has actually occurred, and is much more expensive. It uses the IMS/VS Resource Lock Manager (IRLM). This is necessary because a deadlock could involve both Fast Path and standard IMS/VS (non-Fast Path) resources.

An historical note: Since the first release of Fast Path did not allow a single transaction to access data from both Fast Path and IMS/VS data bases, a common contention resolution mechanism was not required, and Fast Path originally had its own deadlock detection. At that time, contention control and contention resolution were not separated the way they are now.

Contention control in Fast Path is optimized towards processing simple cases with a minimum of CPU time. To do this, Fast Path uses hashing in the following way:

1. Fast Path creates a large number of hash anchors for each data base or DEDB AREA (file). The number of hash anchors is large compared to the number of resources likely to be in use at one time.
2. When a transaction requests a particular resource (requests a lock), one of the hash anchors is selected as a pseudorandom function of the resource name, in such a way that a single resource always hashes to the same hash anchor. The hash anchor is used to anchor a chain of all lock requests for resources that hash to the anchor point.

Since the number of hash anchors is normally much larger than the number of resources in use, most requests encounter unused hash anchors. Therefore, since there can be no resource conflict without a hash anchor conflict, the resource can usually be locked quickly. If we ignore the CPU time needed to create the resource information, such a lock/unlock pair typically costs about 20 instructions.

If the hash anchor is already in use, then there must be a check to see whether or not there is actually a conflict. These checks have to be made in such a way that only one process is checking and/or manipulating the chain off one anchor. This requires additional synchronization, but it still costs less than 100 instructions unless a conflict is found to actually exist.

If a conflict is found to actually exist, then a request must be forwarded to the contention resolution mechanism (the IRLM).

Fast Path could do some deadlock detection. However, neither Fast Path nor standard IMS/VS could detect all the deadlocks without information from the other, since cycles might involve both kinds of resources. With a full resource control system available, the natural choice is to use it for all deadlock detection. However, there are two challenging technical problems:

1. To avoid consuming an excessive amount of CPU time, Fast Path must avoid unnecessary interactions with the common resource manager (IRLM).

   The common resource manager should only know about those resources that are involved in a conflict. However, a conflict is only recognized when a resource is requested for the second time. Therefore, at the time a conflict is detected, two requests have to be sent to the common resource manager: the request for the current owner, which has to be granted: and a request for the competitor. This leads to the second problem.

2. When a conflict does arise, Fast Path must share all relevant information with the common resource manager.

   The difficulty is related to the fact that resources have to be obtained on behalf of other processes. That is, the IRLM has to give the lock to the current owner, not to the process that detected the conflict. Fast Path's solution to this problem depends on the details of IRLM, and is too complicated to describe here.

With all the difficulties to be solved in the case of actual conflict resolution, 2.000 instructions is a reasonable number for the cost of conflict resolution using the IRLM.

Although actual results depend on the characteristics of the data bases and application programs, in a typical case 98% of the requests might encounter unused hash anchors, and 98% of the other requests might be instances of two different resources hashing to the same hash anchor. Doing the arithmetic, we see that contention resolution is needed only 0.04% of the time, and a typical Fast Path lock/unlock pair winds up still costing only about 20 instructions. Again, this is not counting the instructions required to create the resource information.

For Fast Path, the type of resource information (the name, owner, chain fields, etc.) depends on the type of data, but creating the resource information typically takes about 30 instructions. Locking should not be charged with all this cost, since the resource information is also used for other purposes, including buffer look-aside within a transaction and group commit.

The logic just described works only as long as no data bases are shared between different Fast Path systems. If DEDBs are shared between systems, then all locking has to be done by the common resource manager (IRLM), and locking becomes much more expensive.

The lock granularity in Fast Path varies depending on the data base structure and the use of the data.

- Record granularity is always used for MSDBs.

- Page granularity is used for on-line non-sequential-dependent processing and sequential-dependent retrieval.
- AREA granularity is used for sequential-dependent insertion.
- Page clusters (named "Units of Work") are used for on-line utilities.

6. Commit Processing

Fast Path commit processing satisfies the following needs:

- To work together with standard IMS/VS data base handling, and to make a common GO/NOGO (commit/abort) decision.
- To do system journaling in a way that minimizes the amount of system journal data and that allows asynchronous checkpoints of data bases.
- To minimize the amount of time that critical resources such as MSDB records and/or application journal tails are held.

The commit process is structured in the following way:

1. Do phase one processing for standard IMS/VS data bases.
2. Get all locks that are required to do phase one of Fast Path. These locks represent the MSDB records that have to be verified and/or modified and the journal tails of DEDB AREAs for which inserts are pending. Locks are acquired in a specific order, minimizing the potential for deadlocks.
3. Journal Fast Path data base modifications. This includes building after images and copying them to the next available space in the system journal's main storage buffers; it does NOT include waiting for system journal records to be written to external media. All the Fast Path data base updates for a single transaction are grouped together. The use of after images allows easy implementation of asynchronous checkpoints.
4. Do phase two of commit processing. This includes updating MSDBs and DEDB application journal tails, and unlocking the resources related to MSDBs and application journals.
5. Do phase two processing for standard IMS/VS data bases.

Even after all these steps, the results of the transaction cannot yet be externalized (data base changes cannot yet go to DASD and the response cannot go to the terminal) until after the system journal records have been written. In step three, Fast Path does not wait for system journal records to be written; this is to reduce the time between step two and step four, when critical resources are held. This technique reduces the period for which critical resources are held to the millisecond range, allowing extremely high concurrency for these resources.

Even after step five, Fast Path does not force the system journal records out to external media. In this way, Fast Path attempts to minimize CPU time and disk/tape space for the system journal. The system journal records will eventually be forced out by a buffer becoming full, a time-limit expiring, or standard IMS/VS processing. Since resource ownership is recorded on an operating-system region (job) basis, Fast Path switches resource ownership to a dummy region. This lets the region start work on a new transaction.

When a system journal buffer is physically written out, it is likely to contain commit records for more than one transaction. Accordingly, a group of transactions all become committed at one time. This was named "Group Commit" long after the development of IMS/VS Fast Path.

7. Acknowledgments

We thank and acknowledge the work of the following people: Keith Huff, Wally Iimura, Cliff Mellow, Tom Rankin, and Gerhard Schweikert, who all contributed greatly to the Fast Path implementation and design; Jim Gray, who helped clarify and improve the design; Don Hyde and George Vogel, who managed the project.

8. Bibliography

Although this paper is based on personal experience and private discussions, the following literature would be helpful to anyone interested in further exploration of these ideas:

Anon., et al., *A Measure of Transaction Processing Power*, Tandem Technical Report 85.1, also (a shorter version) in: Datamation, April, 1985.

Galtieri, C. A., *Architecture for a Consistent Decentralized System*, IBM Research Report RJ2846, 1980.

Gawlick, D., *Processing "Hot Spots" in High Performance Systems*, Proceedings of COMPCON '85, 1985.

IBM Corp., *IMS/VS Version 1 Data Base Administration Guide*, Form No. SH20-9025-9, 1984.

IBM Corp., *IMS/VS Version 1 Application Programming*, Form No. SH20-9026-9, 1984.

Lampson, B., *Hints for Computer System Design*, in: Operating Systems Review, Volume 17, Number 5, Proceedings of the Ninth ACM Symposium on Operating Systems Principles, pp. 33-48, 1983.

Reuter, A., *Concurrency on High-Traffic Data Elements*, in: Proceedings of the ACM Symposium on Principles of Database Systems (SIGACT, SIGMOD), pp. 83-92, 1982.

Strickland, J., Uhrowczik, P., and Watts, V., *IMS/VS: An Evolving System*. in: IBM Systems Journal, Vol. 21, No. 4, 1982.

Distributed by
TANDEM
Corporate Information Center
10400 N. Tantau Ave., LOC 248-07
Cupertino, CA 95014-0708