

Building Multithreaded Applications With CMTL

Technical White Paper

ObjectStore 6.0.SP7

Date: June 2002



14 Oak Park
Bedford, MA 01730 USA
Phone: +1-781-280-4000
www.objectstore.net

Table of Contents

1	Introducing CMTL	3
1.1	What Type of Applications are Suited to CMTL?	3
1.2	CMTL Concepts and Features	4
1.2.1	<i>Cache Pools</i>	4
1.2.2	<i>Virtual Transactions</i>	4
1.2.3	<i>Transaction Context</i>	5
2	Using the CMTL Interfaces	6
2.1	Declarative Configuration	6
2.1.1	<i>Declaring Cache Pools Using XML</i>	7
2.1.2	<i>Declaring Logical Database and Root Names</i>	8
2.2	Using the CMTL C++ Programming Interface	9
2.2.1	<i>Initializing CMTL</i>	9
2.2.2	<i>Terminating CMTL</i>	10
2.2.3	<i>Using Virtual Transactions</i>	10
2.2.4	<i>Accessing Persistent Objects by Logical Name</i>	11
3	Transaction Routing and Scheduling	14
3.1	Routing Transactions to Caches	14
3.1.1	<i>Inter-process Routing</i>	15
3.2	Isolation Levels	15
3.3	Virtual Transaction Scheduling and Batching	16
4	CMTL Performance Tips	18
4.1	Keep Virtual Transactions Short	18
4.2	Take Advantage of Batching	18
4.3	Use MVCC Caches	19
4.4	Partition Data Across Cache Pools	19
4.5	Don't Assume that More is Better	20
5	Conclusion	20

List of Figures

Figure 1 CMTL Concepts	5
Figure 2 Constructing a Transaction Context.....	6
Figure 3 XML Configuration File.....	6
Figure 4 Declaring a Cache Pool	7
Figure 5 Declaring Logical Database and Root Names	8
Figure 6 Setting Up to Use CMTL	9
Figure 7 Initializing CMTL From An XML Configuration File	10
Figure 8 Demarcating a Virtual Transaction.....	10
Figure 9 Accessing a Root by Logical Name	12
Figure 10 Retaining References across Transactions.....	13
Figure 11 Data-dependent Virtual Transaction Routing	14
Figure 12 Scheduling and Batching in an Update Cache.....	17

1 Introducing CMTL

ObjectStore 6.0 introduced the sessions facility giving applications developers the ability to use multiple ObjectStore client sessions running in a single client process. In a multithreaded application, you can achieve greater concurrency by using more than one ObjectStore session. Sessions are like independent ObjectStore clients sharing a process. Each session has its own cache and its own persistent address space (PSR) partition.

Typically an application server has many threads (perhaps 20 or 30). It is not an efficient use of system resources to have one session per thread. It's more efficient for threads to share sessions, thereby sharing cache and address space. The ObjectStore sessions facility allows multiple threads to share a session, and share transactions associated with a session. Each session can have only one active transaction at a time.

With the sessions facility, threads sharing a session can participate in transactions by:

- Assigning each thread to a local transaction, or
- Assigning multiple threads to the same global transaction.

With local transactions, the transactions of different threads in the same session are serialized. When one thread begins a transaction, other threads that subsequently begin transactions in the same session are blocked until the first thread completes its transaction. At that point, the next thread's transaction can proceed, and so on. While this approach is easy to code and debug, it does not provide the highest possible degree of concurrency.

With global transactions, multiple threads can concurrently participate in the same transaction. However, the developer is responsible for synchronizing the threads' access to persistent data and for ensuring that the threads perform no persistent access during a commit or checkpoint. This approach provides greater concurrency, but is more difficult to code and debug.

The C++ Middle Tier Library (CMTL) was introduced in ObjectStore 6.0.SP6. CMTL handles all the scheduling and thread synchronization required in order to ensure data integrity while achieving a high degree of resource sharing and concurrency. Developers are free to focus on business logic rather than threading and resource management issues.

1.1 What Type of Applications are Suited to CMTL?

CMTL is appropriate for any application where a large number of short duration requests are handled by multithreaded web or application servers. This pattern is typical of e-business and OLTP (Online Transaction Processing) applications. Given the large number of user interactions (millions for customer-facing applications), the finite server-side resources need to be optimally shared. CMTL optimizes the sharing of sessions, address space and transactions across multiple threads.

1.2 CMTL Concepts and Features

In order to implement server-side resource sharing, CMTL introduces some new concepts and features, including:

- Cache pools
- Virtual transactions
- Transaction contexts

The following sections discuss key CMTL concepts that are used throughout this document.

1.2.1 Cache Pools

CMTL uses the term *cache* instead of the term *session*. A CMTL cache is implemented using an ObjectStore session. There is always a 1:1 relationship between a CMTL cache and an ObjectStore session.

With CMTL, you configure a pool of caches (sessions). Cache pooling provides sharing of sessions across multiple threads in an application. As threads service user requests, CMTL routes each thread's business transaction to a cache instance from a pool of caches. Multiple threads can share a cache instance, either concurrently or serially, depending on the compatibility of their business transactions.

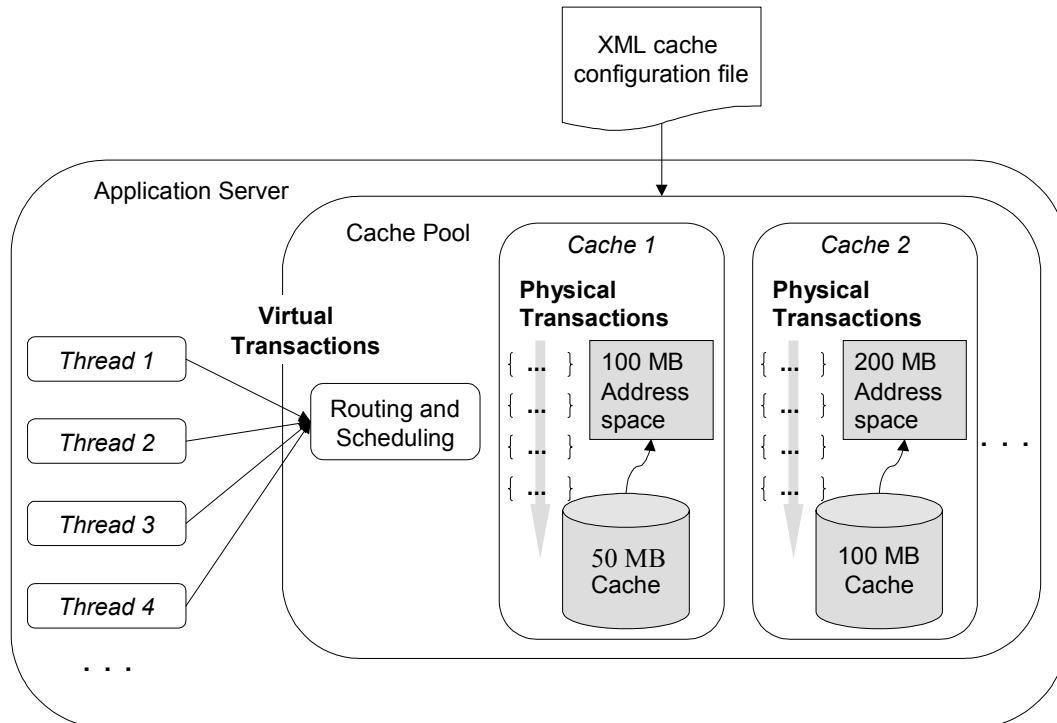
Cache pooling is similar in concept to database connection pooling. Connection pooling is a technique that was pioneered by database vendors to allow multiple threads to share a set of connection objects that provide access to a database resource. However, there is an important distinction between a CMTL cache pool and the typical database connection pool: An ObjectStore cache is heavier weight than the typical database connection. Therefore, it is even more imperative with ObjectStore caches to maximize sharing across threads. The choice of the term *cache pool* (instead of *session pool* or *connection pool*) is intended to help emphasize this distinction.

Cache pools can be configured declaratively using an XML configuration file. At program initialization, the XML configuration file is read by CMTL and the specified number of caches is created. Declarative configuration makes it easier to tune your application, since you can change the number and types of caches, and the resources allocated to caches (address space, cache size) without changing your C++ application code.

1.2.2 Virtual Transactions

With CMTL, threads process user requests as *virtual transactions*. CMTL routes virtual transactions to a *cache pool* and from there to a *cache* instance in the cache pool. The cache can be either update or read-only. CMTL selects an appropriate cache based on run-time information available in a *transaction context* that is associated with each virtual transaction. This information includes the *transaction type* and *isolation level* of the virtual transaction.

Figure 1 CMTL Concepts



After routing, CMTL *schedules* virtual transactions to execute as a group within a single *physical transaction*. When the physical transaction commits, any changes to data become permanent and visible in the caches, ensuring that the application has a transactionally consistent view of the data.

A virtual transaction is associated with *only one application thread*. Multiple application threads cannot participate in a single virtual transaction. However, multiple virtual transactions can share the same cache and the same underlying ObjectStore transaction. An application thread can run many virtual transactions (one at a time, in a series) and each virtual transaction is routed to a cache independently. They can route to the same cache, or different caches.

1.2.3 Transaction Context

Every virtual transaction must be associated with a transaction context. This context provides the following run-time information about the transaction:

- transaction type (update or readonly)
- isolation level
- cache pool to which the transaction is to be routed (optional)

Your application specifies this information by creating an object of the class `os_transaction_context` and by passing a pointer to this object to the virtual transaction to which it applies. CMTL uses the transaction context when routing a virtual transaction to a cache and when scheduling virtual transactions to execute compatibly within a physical transaction.

Figure 2 Constructing a Transaction Context

```
os_transaction_context* read_ctx =
    os_basic_transaction_context::create(books_cache_pool,
        os_transaction_context::read_only,
        os_transaction_context::repeatable_read);
```

See the sections "Routing Transactions to Caches" and "Isolation Levels" for additional information.

2 Using the CMTL Interfaces

With CMTL, developers can use an XML file to declaratively configure:

- cache pools
- logical names for databases and roots

Developers use the C++ programming interface to:

- initialize CMTL from an XML configuration file
- create transaction contexts and demarcate virtual transactions
- access databases and roots
- shutdown CMTL

2.1 Declarative Configuration

CMTL provides a declarative, XML-based interface for defining cache pools, as well as a programmatic C++ interface. Declarative configuration of cache pools in an external XML file makes it easier to deploy, tune and maintain your application. The top-level elements in the XML configuration file are shown in Figure 3.

Figure 3 XML Configuration File

```
<cache_pool_manager_configuration>
  <attributes>
    <attribute_name> cache_size </attribute_name>
    <string_value> 0x8000000 </string_value>
    ...
  </attributes>

  <db_name_mappings>
    ...
  </db_name_mappings>

  <logical_roots>
    ...
  </logical_roots>
```

```

</logical_roots>
<cache_pool_configurations>
...
</cache_pool_configurations>
</cache_pool_manager_configuration>

```

There are a number of attributes that can be declared at the top-level of the cache configuration file, and apply process-wide. One of these attributes is `cache_size`, which specifies the default size in bytes of CMTL caches in the process. As of ObjectStore 6.0.SP8, the `cache_size` attribute can also be specified at the cache pool level, overriding the default value set at the top-level.

CMTL makes it easier to develop multithreaded applications using ObjectStore. However, CMTL can also be used in single-threaded applications. Single-threaded applications benefit from CMTL's declarative configuration.

2.1.1 Declaring Cache Pools Using XML

Cache pools are *logical data partitions*. Cache pools allow you to define different data sets for different purposes; however, CMTL does not enforce that only objects from a particular segment or database are fetched into the cache pool. Your application must enforce this partitioning by insuring that the virtual transactions routed to a particular cache fetch only the desired objects. Figure 4 shows a sample cache pool declaration.

Figure 4 Declaring a Cache Pool

```

<cache_pool_configurations>
  <cache_pool_configuration>
    <cache_pool_name> BooksCachePool </cache_pool_name>
    <attributes>
      <attribute_name> update_caches </attribute_name>
      <string_value> 1 </string_value>
      <attribute_name> mvcc_caches </attribute_name>
      <string_value> 1 </string_value>
      <attribute_name> address_space_size </attribute_name>
      <string_value> 0x8000000 </string_value>
    </attributes>
  </cache_pool_configuration>
</cache_pool_configurations>

```

Often, user requests are routed to a cache pool by *behavior*; a particular application function always routes to a particular cache pool. However, requests can also be routed by *content*. For example, all requests for "NY" data route to the "NortheastCachePool". In both cases the cache pool is a *logical data partition*. That is, a cache pool is a logical name for the data set associated with a set of user requests.

Due to the potential for lock contention, the number of update caches should be limited to *one per cache pool*. See the section "Partition Data by Cache Pool" for additional information.

Cache Pool Resources

The cache pool attribute `address_space_size` must be at least as large as the process-wide attribute `cache_size`. The `cache_size` attribute applies to each cache in the process. The `address_space_size` attribute applies to each cache in the current cache pool.

The environment variable `OS_AS_SIZE` determines the total amount of address spaces available to the process. If you set both `address_space_size` and the environment variable `OS_AS_SIZE`, the latter must be set to a value at least as large as:

$$6\text{MB} + \sum (\text{address_space_size} * \text{number of caches})$$

Address space and cache are limited resources so it's important not to 'go overboard' in defining cache pools and cache instances. CMTL is intended to help you maximize the sharing of these resources across threads.

2.1.2 Declaring Logical Database and Root Names

You can configure CMTL to map the logical name of a database to its physical name, and to map the logical name of a root to its physical name and its logical database as shown in Figure 5.

Figure 5 Declaring Logical Database and Root Names

```
<db_name_mappings>
  <db_name_mapping>
    <logical_db_name> BooksDatabase </logical_db_name>
    <physical_db_name> myserver:/data/books.db
    </physical_db_name>
  </db_name_mapping>
</db_name_mappings>

<logical_roots>
  <db_root>
    <logical_root_name> BookExtent </logical_root_name>
    <physical_root_name> novels </physical_root_name>
    <logical_db_name> BooksDatabase </logical_db_name>
  </db_root>
</logical_roots>
```

The use of logical names makes an application more flexible and reconfigurable. For example, if you should decide at some point to partition a database extent into two sub-extends, you can do this without changing the application code. Identical application code can be deployed twice, where each deployed process works with a different physical root even though processes use the same logical root name in the application code.

Your application can access a root object without explicitly opening the database. CMTL will automatically open the database where the root is located, based on the information in the XML configuration file.

2.2 Using the CMTL C++ Programming Interface

CMTL layers on top of ObjectStore. The ObjectStore interfaces `os_session` and `os_transaction` are not used in a CMTL application. Higher-level interfaces, including `os_cache` and `os_virtual_transaction` are used in their place. Table 1 shows the CMTL equivalents for core ObjectStore classes.

Table 1 CMTL Programming Interface

ObjectStore class and macros	CMTL equivalents
<code>os_session</code>	<code>os_cache</code>
<code>os_transaction</code>	<code>os_virtual_transaction</code>
<code>OS_BEGIN_TXN()</code>	<code>OS_BEGIN_VT()</code>
<code>OS_END_TXN()</code>	<code>OS_END_VT()</code>

CMTL applications should not use the ObjectStore C++ `os_session` and `os_transaction` interfaces. However, CMTL applications can and should use all of the other of the ObjectStore C++ classes, such as the collections library.

2.2.1 Initializing CMTL

To use CMTL in your application, you need to include `<ostore/cmtl.h>`. Each thread that uses CMTL must call the `OS_ENTER_CMTL` macro in the application code, as shown in Figure 6.

Figure 6 Setting Up to Use CMTL

```
#include <ostore/ostore.hh>
#include <ostore/cmtl.hh>

int main(int argc, char** argv)
{
    OS_ESTABLISH_FAULT_HANDLER {
        OS_ENTER_CMTL {

            // initialize cache pools and collections
            // . . .

            // application logic here
            // . . .

        } OS_LEAVE_CMTL
    } OS_END_FAULT_HANDLER
    return 0;
}
```

CMTL is initialized by loading the XML configuration file and creating an instance of the class `os_cache_pool_manager` as shown in Figure 7. If your application uses the collections

facility, you must call `os_cache_pool_manager::initialize_collections()` after creating the cache pool manager and before using the collections API. In CMTL applications, this call replaces `os_collection::initialize()`.

Figure 7 Initializing CMTL From An XML Configuration File

```
os_cache_pool_manager_config* cpm_cfg =
    os_cache_pool_manager_config::create_from_xml_file(
        "config.xml");

os_cache_pool_manager* cache_pool_manager =
    os_cache_pool_manager::create(*cpm_cfg);

cache_pool_manager->initialize_collections();
```

You can call `os_cache_pool_manager::create()` multiple times per process, but you can only have one cache pool manager instance at any time. The cache pool manager is a singleton object that can be created, shutdown/destroyed, and re-created as many times as you want. Typically, your application will create the cache pool manager only once per process.

2.2.2 Terminating CMTL

The `os_cache_pool_manager::shutdown()` function stops all caches in all cache pools controlled by the current cache pool manager. If any virtual transactions are currently scheduled in a cache, CMTL allows them to complete before removing the cache. This function should be called in any application that uses CMTL in order to perform an orderly shutdown.

CMTL applications must *not* call `objectstore::initialize()`, however they *can* call `objectstore::shutdown()` after CMTL has shutdown. If you omit this call, client counters won't be printed even when `OS_PRINT_CLIENT_COUNTERS` is enabled.

2.2.3 Using Virtual Transactions

In order to access persistent data, an application thread must be associated with a virtual transaction. An application thread can demarcate virtual transactions using the CMTL transaction macros, as shown in Figure 8.

Figure 8 Demarcating a Virtual Transaction

```
OS_BEGIN_VT (print_root, *read_ctx) {
    os_cache *cache = os_cache::get_current();
    char *hello = (char*)cache->get_root_value("Hello Root");
    cout << hello;
} OS_END_VT (print_root)
```

If a lexical virtual transaction aborts because of a deadlock or because the underlying physical transaction aborted, CMTL automatically retries the transaction. The default number of retries is ten. If CMTL cannot restart the transaction after reaching this maximum, the exception `err_too_many_retries` is signaled. You can catch this and other exceptions using ObjectStore TIX exception handling.

In this example, the application is accessing a root object without opening the database first. This is possible with CMTL because the XML configuration file maps logical root names to logical database names. When the application accesses a root by logical name, CMTL will automatically open the physical database associated with the logical root.

Lexical or Dynamic Transactions

A virtual transaction can be lexical or dynamic, depending on whether you use the `OS_BEGIN_VT()` and `OS_END_VT()` macros or the member functions of the `os_virtual_transaction` class to demarcate the transaction.

In general, you should use the macros to define a virtual transaction because of the advantage of having CMTL automatically retry the transaction. The `os_virtual_transaction` class is mainly useful when you cannot define a virtual transaction within a scoping unit. Virtual transactions can only be associated with a single thread and they should be short in duration, so defining them within a scoping unit using the macros is usually the best approach.

Access to Persistent Data

CMTL applications should only access persistent data when in a virtual transaction. However, just as in non-CMTL ObjectStore C++ applications, ObjectStore cannot always detect when an application attempts to illegally access persistent data outside of a transaction.

If an application dereferences a pointer to a persistent object outside of a transaction, and the pointer is already mapped into virtual memory at the necessary access level, the operating system will allow access without any intervention by ObjectStore. This can result in unpredictable program failures.

On the other hand, if ObjectStore gets an opportunity to mediate access to the referenced memory, an exception will be signaled. In CMTL applications, threads are always disassociated from any ObjectStore session when they are not in a virtual transaction, so the exception `err_no_session` will be signaled.

Since accessing persistent data outside a transaction can result in unpredictable program failures, thorough testing is important. CMTL applications are no different from other ObjectStore C++ applications in this respect.

2.2.4 Accessing Persistent Objects by Logical Name

Many of the functions in the CMTL API take logical names as arguments that reference databases and roots. CMTL provides functions for the following:

- Creating a database by logical name
- Opening a database by logical name
- Creating a root by logical name
- Accessing a root by logical name (which automatically opens the database)

For example, you can use `os_cache::get_root_value()` to retrieve a pointer to the root object for any logical root that has been mapped in the XML configuration file. The pointer is valid only for *this* cache.

Figure 9 Accessing a Root by Logical Name

```
OS_BEGIN_VT (read_books, *read_ctx) {
    os_cache cache = os_cache->get_current();
    os_set* books_set = (os_set*)
        cache->get_root_value("BooksExtent");
    // access books ...
} OS_END_VT (read_books)
```

When a virtual transaction is active, the application has been routed to a cache (session) and can use the `os_cache` API to access persistent objects. The `os_cache::get_current()` function returns the current cache, and the `os_cache::get_root_value()` returns the root object associated with a logical root name. To call this function, the logical root name must have been mapped in the XML configuration file. The database in which the root value is stored does not have to be opened explicitly; CMTL will automatically open the database for you in the current cache.

Note that in CMTL applications you won't need to use the `os_database` or `os_root` APIs as frequently as in non-CMTL applications, since the `os_cache` API provides access to databases and roots by logical name. However, if you have existing components that you wish to use in a CMTL application, the `os_database` or `os_root` APIs will work the same as in a non-CMTL application.

Retaining References across Virtual Transactions

To retain references to persistent objects across virtual transactions, your application needs to keep track of which cache a reference is associated with, since a function may not always route to the same cache instance. If a function dereferences a pointer to a persistent object in a cache other than the one in which it was retrieved, `err_wrong_session` is signaled. This could occur, for example, if your function uses data-dependent logic to select a cache pool for the virtual transaction (see "Routing Transactions to Caches").

To avoid the `err_wrong_session` exception, you can create a transient collection or array of references to persistent objects keyed by cache name. For example, create an array of cache name/reference pairs as shown in Figure 10.

Figure 10 Retaining References across Transactions

```
struct BooksRef {
    char* cache_name;
    os_soft_pointer<os_set> books_set;
};
BooksRef books_ref_table[MAX_CACHES];

...
OS_BEGIN_VT (read_books, *read_ctx) {
    //
    // Lookup the persistent reference in the books_ref_table
    // using the current cache's name
    //
    os_cache* cache = os_cache->get_current();
    char* cache_name = cache->get_name();
    os_set* books_set = lookup_books_ref(cache_name);

    //
    // If the reference isn't in the table, then access the
    // database to get the reference and save it in the table
    // for next time.
    //
    if (books_set == NULL) {
        books_set = (os_set*)
            cache->get_root_value("BooksExtent");
        add_books_ref(cache_name, books_set);
    }
    // access books ...
} OS_END_VT (read_books)
```

In this example, the `books_ref_table` is a transient table of references keyed by cache name. The table is used to look up the appropriate `os_set` object for the current cache. This logic allows the same function to route to multiple cache instances; for example the function could route to a cache in the "NonFictionCachePool" or to a cache in the "FictionCachePool".

Caution: In any ObjectStore application, you need to be careful about retaining references to non-exported objects across transactions, as these objects can be deleted or moved by intervening transactions. References to root objects, and other exported objects, are 'delete safe' and thus it is always safe to use them across transactions.

3 Transaction Routing and Scheduling

CMTL optimizes the sharing of cache, address space and transactions across multiple threads by dynamically routing virtual transactions to caches, and scheduling virtual transactions to run in a physical ObjectStore transaction. This section describes how routing and scheduling are implemented in CMTL.

3.1 Routing Transactions to Caches

When a virtual transaction begins, CMTL routes it first to a cache pool and then to a cache instance (session), based on information in the transaction context that is associated with the transaction.

With CMTL, the transaction context is constructed programmatically at runtime, so your application can use attributes of the user request to select an appropriate cache pool. For example, consider a catalog browsing application with two cache pools: "BooksCachePool" and "VideosCachePool". If a user requests books data, your application can inspect attributes of the request and then route it to the "BooksCachePool" as shown in Figure 11.

Figure 11 Data-dependent Virtual Transaction Routing

```
If (user_request->type == BOOK_REQUEST)
    cache_pool_name = "BooksCachePool";
else
    cache_pool_name = "VideosCachePool";

os_transaction_context* read_ctx =
    os_basic_transaction_context::create(
        get_cache_pool(cache_pool_name),
        os_transaction_context::read_only,
        os_transaction_context::repeatable_read);

OS_BEGIN_VT (read_catalog, *read_ctx) {
    os_cache cache = os_cache->get_current();
    os_set* catalog = (os_set*)
        cache->get_root_value("CatalogExtent");
    // access catalog ...
} OS_END_VT (read_catalog)
```

In this example, the `get_cache_pool()` function would be implemented using the function `os_cache_pool_manager::get_cache_pools()` to get an array of `os_cache_pool` objects for the current process, and then using `os_cache_pool::get_name()` to look up the specified cache pool by name.

If you have defined more than one cache pool in an application process, it is important to route user requests to the cache pool containing the appropriate data, since *CMTL does not enforce data*

partitioning. CMTL has no knowledge of what data should be fetched into each cache pool. If requests are routed randomly, each cache pool will eventually contain all the data. Assuming each cache pool has one update cache, this will result in lock contention and deadlocking. See the section "Partition Data Across Cache Pools" for additional information.

3.1.1 Inter-process Routing

CMTL routing only applies to the caches within a single application process. CMTL routing is *intra-process routing*. As your application's workload increases, you may need to run multiple processes (web or application servers) across multiple host machines in order to scale up. Routing user requests across multiple process or hosts requires that your client application provide the routing logic.

There are a number of options you can use to implement *inter-process routing*, such as:

- HTTP request redirection in a web servlet
- Application server request routing, such as Tuxedo's data-dependent routing
- CORBA object naming for data partitions

Regardless of the method(s) you select, the important thing is that you use *intelligent routing* to route requests to the correct data partition. Round robin routing will result in a random distribution of data across caches, duplicating objects in memory and causing lock contention and deadlock.

3.2 Isolation Levels

Isolation levels are specified in the transaction context when your application starts a virtual transaction. Isolation levels influence how CMTL routes virtual transactions to caches and schedules virtual transactions to run in caches. Table 2 summarizes the effects of isolation levels.

	Update Serializable	Repeatable Read	Read Committed	Read Uncommitted
Can route to an MVCC cache	No	Yes	Yes	Yes
Returns before physical commit	No	If no update has run in the cache	If no update has run in the cache	Yes
Runs concurrently	No	Yes	Yes	Yes

Table 2 Effects of Isolation Levels on Virtual Transactions

Update transactions are always treated as `serializable`, even if you specify a different isolation level. Hence, update transactions are always routed to an update cache.

Read-only transactions will be routed to an update cache if their isolation level is `serializable` (the default), or if no MVCC cache exists in the cache pool. They can also be routed to an update cache if the MVCC cache is busy and the update cache is not busy. All read-only transactions, even those with isolation level of `serializable`, can run concurrently

You may note that `repeatable_read` and `read_committed` appear to have identical behavior. This is true in CMTL 1.0. In a future version of CMTL that supports XA (two phase commit), it will be possible to have virtual transactions controlled by an XA transaction. When this is the case, all virtual transactions controlled by an XA transaction with isolation level of `repeatable_read` (or stricter) will always route to the *same cache*. In contrast, virtual transactions controlled by an XA transaction with isolation level of `read_committed` (or less strict) will be able to route to *multiple caches*.

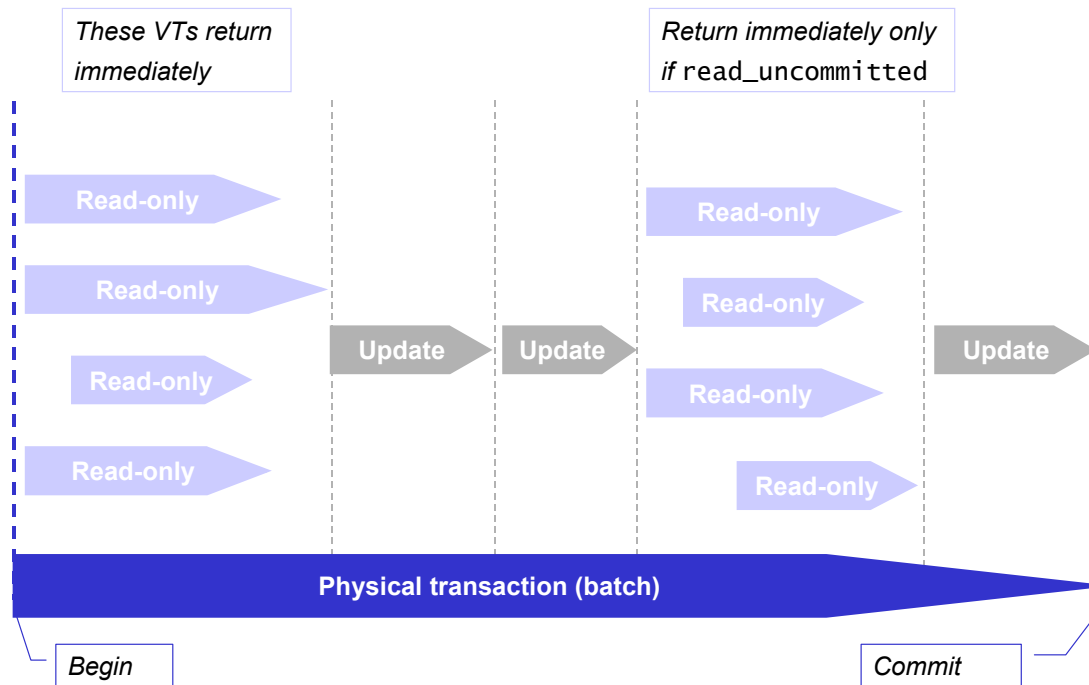
3.3 Virtual Transaction Scheduling and Batching

CMTL boosts the performance of multithreaded applications by scheduling compatible virtual transactions to run concurrently in a cache and batching virtual transactions into a physical transaction.

- Read-only virtual transactions run concurrently
- Update transactions are serialized and always wait for the physical transaction commit
- Both types of virtual transactions can be batched together into a physical ObjectStore transaction

Figure 12 shows an example of scheduling and batching in an update cache.

Figure 12 Scheduling and Batching in an Update Cache



The cache pool attribute `group_open_interval` specifies the interval during which CMTL schedules virtual transactions to execute within a physical transaction. When the interval expires, the physical transaction will commit if the physical transaction is idle - that is, if there are no active virtual transactions running in the physical transaction. The default value for `group_open_interval` is 500 milliseconds.

Read-only transactions can run concurrently with other read-only transactions, and hence provide greater concurrency. Read-only transactions will be routed to an update cache if their isolation level is `serializable` (the default), or if no MVCC cache exists in the cache pool. They can also be routed to an update cache if the MVCC cache is busy and the update cache is not busy.

As you can see, update transactions can slow down read-only transactions running in the same cache. Update transactions are always serialized with respect to all other transactions. Read-only transactions always return immediately in an MVCC cache, but they will wait for the physical commit when they run in a cache where update transactions have executed and their isolation level is `read_committed` or stricter.

4 CMTL Performance Tips

This section discusses a few performance tips for designing CMTL applications. Overall, there are two themes to keep in mind:

- Maximize sharing of resources
- Minimize overlap of caches

4.1 Keep Virtual Transactions Short

As Figure 12 demonstrates, update transactions running in a cache will block all other transactions routed to that cache, even those that access a completely different set of objects. Therefore, it's important to keep update transactions as short as possible.

Long running readonly transactions also present a problem; they can increase the time for a physical transaction to commit. Remember that a transaction batch is only open to new transactions until `group_open_interval` elapses. After that point, the physical transaction will commit as soon as all virtual transactions have completed. A long running virtual transaction will block those virtual transactions that arrived after `group_open_interval` elapsed.

If you must use long running transactions, route them to a separate cache from short-running transactions.

4.2 Take Advantage of Batching

Batching many small client requests into a smaller number of physical ObjectStore transactions provides a major performance boost for multithreaded applications, such as web or application servers. Two virtual transactions will execute faster if scheduled in a single physical transaction, than if scheduled in two physical transactions.

Two cache pool attributes influence virtual transaction batching:

- `group_open_interval`
- `commit_if_idle`

The attribute `commit_if_idle` specifies whether CMTL should commit a physical transaction before the value specified by the `group_open_interval` attribute expires, if there are no virtual transactions pending.

Setting `commit_if_idle=true` can have a negative impact on performance under some circumstances. One such circumstance is in an MVCC cache which has a small `group_open_interval` and where no virtual transactions are pending to be scheduled. The CMTL scheduler sees that the cache is idle and commits the underlying ObjectStore transaction. This reduces transaction batching. In the worst case scenario, this could result in a 1:1 ratio of virtual to ObjectStore transactions (no batching) -- where the CMTL cache would be slower than just using a plain ObjectStore transaction.

Setting `commit_if_idle=true` can, in some cases, improve performance in update caches. Remember that update virtual transactions always wait for the physical commit to complete before returning to the calling application. The `commit_if_idle` attribute will cause the physical transaction to commit before `group_open_interval` elapses, when there are no virtual transactions waiting to run in the cache. This allows update virtual transactions to return faster when no virtual transactions are pending.

Currently, the default setting for `commit_if_idle` is `true` for both update and MVCC caches. A future version of CMTL may change the default value to `false` for MVCC caches.

4.3 Use MVCC Caches

Route read-only transactions to an MVCC cache. Read-only transactions in an MVCC cache are never blocked by update transactions, and never need to wait for the physical commit before returning.

Read-only caches use Multi-Version Concurrency Control (MVCC). Reads to MVCC caches never conflict with (block) writes to update caches. Using MVCC can help achieve greater concurrency of readers and writers, however the MVCC readers may read slightly out-of-date snapshots of the data.

Read-only transactions with isolation level of `repeatable_read`, or less strict, can be routed to an MVCC cache.

4.4 Partition Data Across Cache Pools

If your application workload requires more than one update cache, *partition the data across cache pools*. Do not define more than one update cache instance in each cache pool, since CMTL provides no way to control which requests are routed to which update cache within the same cache pool. If the same data is fetched into more than one update cache, you are likely to have lock contention and deadlocking between the caches.

Lock contention and deadlocking are especially expensive in CMTL applications, due to the following:

- Lock wait blocks all other virtual transactions waiting to run in an update cache, since updaters are serialized at the *cache level*.
- Deadlock aborts the physical ObjectStore transaction *and all virtual transactions currently scheduled in that cache*. This includes virtual transactions that have completed and are waiting for the physical commit. The deadlock exception must be caught and retried by all of the virtual transactions.

If you plan to use more than one update cache, be sure that they access distinct data sets. For example, route all requests for data in `BooksDatabase` to `BooksCachePool`, and all requests for data in `VideosDatabase` to `VideosCachePool`. You can also partition data based on database segments and clusters, as long as you can guarantee that each update cache will contain a unique data set.

It's acceptable to define one update cache and one MVCC cache in each cache pool, since MVCC caches do not contend for locks. In this case, you will have two copies of the data in memory, one copy for each cache. However, request concurrency will be improved since read-only MVCC transactions are never blocked by update transactions.

4.5 Don't Assume that More is Better

Don't assume that adding more caches will improve performance. Always keep in mind that caches and transactions are heavyweight resources and the more you can *share* these resources across multiple client requests, the better.

Read-only virtual transactions always run concurrently when routed to an MVCC cache, and always return immediately, without waiting for the physical transaction commit. MVCC caches can provide a high throughput rate. So, unless your application workload is very high, it is not unusual to see performance actually *decline* when adding another MVCC cache.

Adding more caches can diminish the performance boost that you get from transaction batching. The effect is similar to shortening the `group_open_interval` as described above. Two virtual transactions will execute slower if routed to separate caches and scheduled in two physical transactions, than if routed to a single cache and scheduled in a single physical transaction.

Update virtual transactions are always serialized at the cache level, and always wait for the physical transaction commit. If update throughput is not sufficient, you might need to add another update cache. If you do add another update cache, be sure to follow the guidelines described earlier in the section "Partition Data Across Cache Pools", and use *only one update cache in each cache pool*.

5 Conclusion

One of ObjectStore's great advantages over other databases is its patented Cache Forward Architecture™. Customer-facing e-business applications serve hundreds or thousands of users at a time, and interaction with a database server can quickly become a bottleneck. But, with ObjectStore's patented Cache Forward Architecture™ queries can execute in a cache without any interaction with the server.

CMTL takes ObjectStore performance to new heights in multithreaded environments, such as web or application servers, by giving developers the tools to optimize cache sharing and concurrency. CMTL handles all the scheduling and thread synchronization required to ensure data integrity and frees the developer to focus on business logic. At the same time, CMTL provides transaction batching and relaxed isolation levels that turbo-charge application performance. Test-drive it today!

ObjectStore is a registered trademark, and Cache-Forward Architecture is a trademark, of Progress Software Corporation in the U.S. and other countries. Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. Any other trademarks or service marks contained herein are the property of their respective owners.