



*Introducing Cache-Forward  
Architecture  
White Paper*

**PROGRESS**  
SOFTWARE

***Real Time Division***

# Table of Contents

Introduction .....	3
The Value of Cache-Forward Architecture (CFA) .....	3
What CFA Is .....	4
Why CFA Makes Applications Run Fast: Virtual Memory Mapping .....	4
Where CFA Fits In and the History of CFA .....	5
How to Implement Applications on CFA .....	6
Description of Cache-Forward Architecture .....	6
Virtual Memory Mapping .....	7
What VMM Does .....	7
Why VMM Improves Performance .....	8
How VMM Manages Very Large Datasets .....	9
Single Level Storage .....	9
Distributed Data Processing .....	10
Client Caching .....	11
Client Page Cache Features .....	12
Client Page Cache Performance .....	12
About Locks on Pages in the Client Page Cache .....	14
Transaction Management .....	14
Standard Transaction Behaviors .....	14
Callback Locking .....	14
Multiversion Concurrency Control (MVCC) .....	16
Middle Tier Concurrency Control .....	17
Scalable and Integrated Deployment Options .....	17
Enterprise-Class Object Databases .....	17
Caching Applications That Access Relational Databases .....	18
Small-Footprint Object Databases .....	18
Glossary .....	21



## *Introduction*

Are you developing C++ or Java® applications that perform operations such as:

- Managing data for rich C++ or Java object models?
- Caching enterprise data in the middle tier?

Building real-time, high-performance event processing systems, such as financial trading systems, fraud detection, or radio frequency identification (RFID)?

If you are, then Progress® ObjectStore® Enterprise's Cache-Forward™ Architecture (CFA) can help you achieve the performance required by such applications.

ObjectStore's CFA lets applications process very large datasets at in-memory speeds. This approach to managing data ensures that the data your application operates on is in memory. The obvious benefit is that your applications run fast.

An understanding of CFA crystallizes the value of the ObjectStore approach to managing data for applications that use C++ or Java. To this end, *Introducing Cache-Forward Architecture* describes CFA features and discusses how CFA makes applications run so fast. The *italicized* terms that you will find throughout this paper are defined in the glossary.

This document is for potential ObjectStore customers who are experienced C++ or Java developers or development managers. No knowledge of ObjectStore or CFA is assumed. New ObjectStore developers will also find this information useful.

## *The Value of Cache-Forward Architecture (CFA)*

ObjectStore's Cache-Forward Architecture (CFA) lets applications process large datasets at in-memory speeds, with the full ACID properties (atomicity, consistency, isolation and durability) of a database. That's because the data these applications operate on is in memory, which minimizes the overhead of object dereferencing, locking, network access, data caching, and disk access. Even when a single *transaction* needs to touch more data than can fit in a process's address space, CFA's patented technologies ensure that the data an application needs is in memory. With CFA, applications can:

⇒ Apply business logic in real time to complex, inter-related data models.

- ⇒ Provide subsecond responsiveness from queries that compare newly collected data with historic data by executing all operations in memory.
- ⇒ Scale to handle a deluge of data. For example, 50,000 events per second in a stock trading application.

CFA is the foundation for ObjectStore and a key technology component of the Progress® Apama® EventStore. ObjectStore is a multiuser, enterprise-class, object database that stores objects in their native format. Within Apama EventStore, ObjectStore delivers real-time, high performance data management services for event-driven applications. To illustrate the value of the CFA approach to managing data, this section discusses the following topics:

- What CFA Is
- Why CFA Makes Applications Run Fast: Virtual Memory Mapping
- Where CFA Fits In and the History of CFA
- How to Implement Applications On CFA

## What CFA Is

CFA lets your application manage data in memory with the reliability of a database. A number of architectural elements work together to make this happen:

- ⇒ **Virtual Memory Mapping (VMM)** — When your application dereferences a pointer whose target is not already in memory, *VMM* dynamically maps the target object into memory.
- ⇒ **Single Level Storage** — Objects are stored on the server with the same representation as objects in memory. Mapping between formats is not required, which drastically reduces both the amount of code to store objects and the overhead of manipulating those objects.
- ⇒ **Distributed Data Processing** — *CFA servers* and *clients* share data management and data processing responsibilities. Whereas most traditional DBMS approaches optimize the server side of processing, CFA is optimized to bring data management capabilities into the memory space of the middle tier application servers.

Because CFA is a distributed architecture, it is ideally suited for service-oriented architecture (SOA) applications.

- ⇒ **Client Caching** — Lets the client keep recently used objects in case they are needed again. The CFA server keeps track of which clients hold which objects, which eliminates unnecessary client-to-server interaction and the associated network and processing overhead.
- ⇒ **Transaction Management** — CFA implements a unique transaction management approach by managing lock state on the client as well as on the server. Transactions automatically handle simultaneous access to objects to ensure that an update by one process does not interfere with an update by another process.
- ⇒ **Scalable and Integrated Deployment Options** — CFA can be the foundation for an enterprise-class object database, a cache application for relational databases, or a small-footprint object database that is a replacement for file-based storage.

## Why CFA Makes Applications Run Fast: Virtual Memory Mapping

The speed of pointer dereferencing is an important factor in the performance of applications that operate on complex object models. The fastest applications keep the data they are operating on in memory as much as possible. For maximum performance, dereferencing pointers to objects should occur in a single instruction.

CFA's unique virtual memory mapping (VMM) dynamically maps objects directly from the server into the clients' virtual memory. In addition, CFA caches lock state on the client to ensure that the *client page cache* is "warm" with the most relevant data. Together, VMM's direct mapping and CFA's client lock caching ensure that objects are accessed at in-memory speed.

VMM facilitates access to stored objects in much the same way as virtual memory systems provide access to data stored in virtual memory in the operating system's paging area on disk. Virtual memory systems support uniform memory references to data, whether that data is located in RAM or in the paging area on disk. CFA expands on this concept to apply it to all data that is managed by the CFA server. To accomplish this transparently, CFA takes advantage of the operating system's programming interfaces that allow software systems to control virtual memory. For example, CFA uses the memory-mapping interfaces (such as `mmap` on Linux) to implement both VMM and locking.

When a CFA application dereferences a pointer whose target has not already been retrieved by the client, the operating system detects an access violation. The operating system signals this as a memory fault, which the VMM system intercepts. The VMM system retrieves the target object from the CFA server, maps the object into the client's virtual memory, and triggers continuation of the dereference operation. Subsequent references to the same object, and to objects that were mapped into virtual memory at the same time as the target object, will run in a single instruction without causing a memory fault.

## Where CFA Fits In and the History of CFA

Although CFA is a client/server architecture, the applications that ultimately access its data are often thin clients. Typically, CFA applications are embedded in an application server in the middle tier. In this configuration, CFA clients participate as servers to thin clients, and as clients to CFA servers.

Some background about the development of the processing capabilities of CFA clients shows why CFA is a perfect fit for the middle tier of multi-tier applications.

CFA was developed by the leading visionaries of object-oriented data management software (OODMS). They designed CFA to meet the needs of computer aided design (CAD) and computer aided software engineering (CASE) applications, among others. CAD and CASE applications manage extremely complex object models, and require extremely high performance. It is typical for these applications to access hundreds of thousands of objects with subsecond response times.

As a result, OODMS architectures required a more client-centric architecture than, for example, relational databases. This client-centric approach let OODMS leverage workstation resources to provide the high performance required by these sophisticated applications. To achieve maximum performance, the inventors of CFA moved the data, as well as the processing, to the clients.

In the late 1990s, two-tier, client/server configurations evolved into multi-tier configurations, with application servers between end users and enterprise resources. Along with the data processing, much of the business logic moved from the edges into the middle.

It soon became clear that CFA's client-centric data cache management was ideal for these types of multi-tier, data processing infrastructures. Middle-tier application servers had the same demands that drove those early CAD applications to an in-memory approach to managing their data. That is, subsecond responsiveness was required for operating on complex object models. While the source of the data being managed in each of these cases might be different:

⇒ Middle-tier applications often need to combine data coming from real-time feeds with data cached from enterprise information systems.

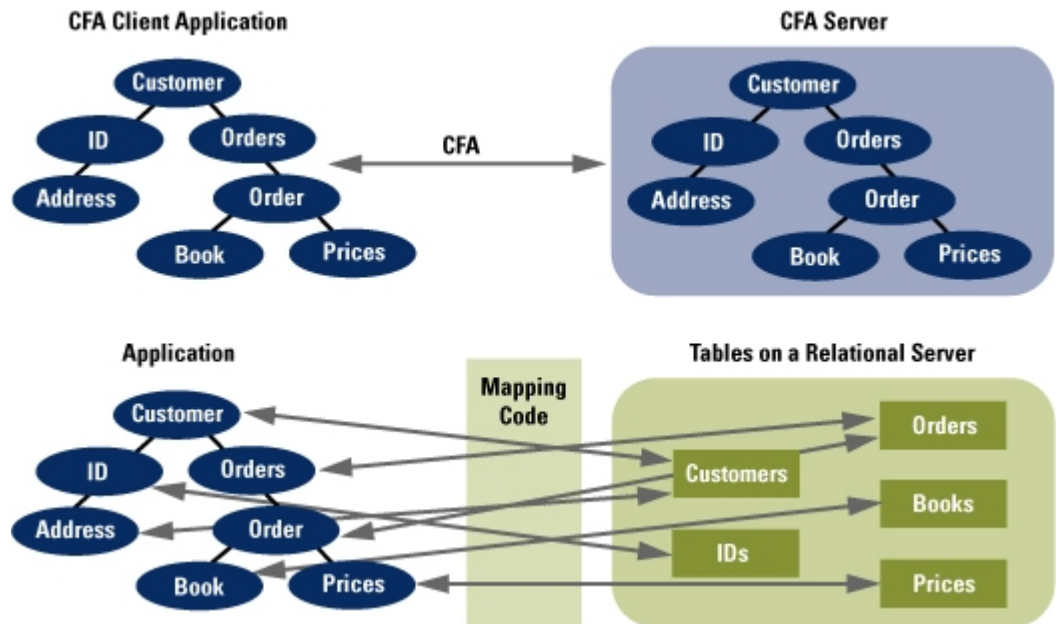
⇒ CAD systems largely created their own data or imported it from other engineering systems.

The fundamental requirements for managing the data were the same, and both cases matched very well with the CFA approach.

## How to Implement Applications on CFA

When you are writing applications that take advantage of CFA, ObjectStore's single-level storage model simplifies the way you approach stored data because:

- ⇒ Data you operate on is in memory.
- ⇒ There is no mapping from one data format to another, as shown in the figure below.
- ⇒ You can work with object models that are as complex as you need them to be.
- ⇒ The interface for writing applications is standard C++ or standard Java.



There is a tight integration between CFA and C++, and between CFA and Java. This lets developers who understand the language easily learn how to write applications that harness CFA. Writing CFA applications is just like programming with standard C++ or Java. With CFA, there is no separate data definition language or data manipulation language — the programming language is the interface.

The best way to develop an application that leverages CFA is to:

- ⇒ Obtain ObjectStore training.
- ⇒ Read ObjectStore white papers about applications that are similar to yours.
- ⇒ Apply the appropriate design principles for your goals.

## Description of Cache-Forward Architecture

CFA features let client processes transparently access data in a transactionally consistent way in the operation of application servers in the middle tier. Consequently, CFA clients play a significant role in accessing and manipulating data. The architecture features that support this are:

- ⇒ Virtual memory mapping

- ⇒ Single level storage
- ⇒ Distributed data processing
- ⇒ Client caching
- ⇒ Transaction management
- ⇒ Scalable and integrated deployment options

CFA provides a persistence model that is transparent in both C++ and Java. Java applications leverage CFA in a manner that works seamlessly with standard JVMs. In the Java binding, the Java interface to CFA communicates with the CFA run-time library to bring objects from the client page cache into the Java VM.

## Virtual Memory Mapping

The most common operation in an object application involves object dereferencing. The speed of object dereferencing is an important factor in the performance of complex object applications. CFA's patented virtual memory mapping (VMM) ensures the highest possible performance for object dereferencing.

The details about how VMM does this are discussed in the following topics:

- ⇒ What VMM Does
- ⇒ Why VMM Improves Performance
- ⇒ How VMM Manages Very Large Datasets

*Note: In a CFA Java application, the CFA client communicates with the CFA run-time library to bring the objects into the Java VM. VMM underlies this Java binding.*

### What VMM Does

When your application dereferences a pointer whose target is not already in memory, VMM dynamically maps the target object into memory. That is, VMM

1. Handles the *page fault* that occurs when there is a reference to an object that is not in memory.
2. Retrieves the page that contains the target object from the CFA server.
3. Maps the retrieved page into memory.
4. Triggers resumption of the referencing operation.

Subsequent references to the target object, or to objects that are on the same page as the target object, are direct memory references without causing communication with the CFA server.

Other architectures perform object dereferencing in software. A drawback of this approach is that it can take hundreds of times more instructions to perform pointer dereferencing even when the object is in memory. This is one of the reasons that CFA far outperforms other architectures. VMM's hardware dereferencing avoids the need for many lines of code. Once an object is in memory, object access is at direct in-memory speed.

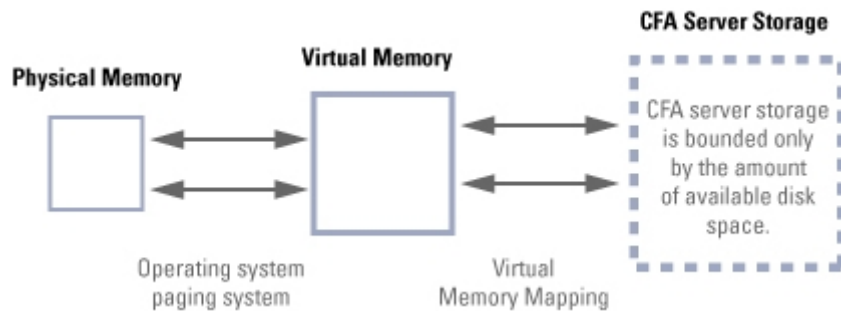
With VMM, your application is faster because VMM automatically maps objects into memory. Your application is also faster because it is smaller. A comparison of a CFA application with a flat file database application that does the same thing reveals that the CFA application is typically one-third the size or less than the flat-file version. The following graph compares the number of lines of code in two applications that store information about customers and their book orders. To examine the applications in this comparison, download them at

[http://www.progress.com/realtime/publications/cfa\\_intro\\_thanks](http://www.progress.com/realtime/publications/cfa_intro_thanks).



### Why VMM Improves Performance

After the CFA server sends an object to the client, VMM ensures that access to your data is exactly as fast as access to transient data. VMM takes advantage of the CPU's virtual memory hardware, as well as the operating system's interfaces that allow ordinary software to utilize that hardware. When VMM retrieves a page from the CFA server and maps it into virtual memory, it is similar to the operating system process of mapping objects from virtual memory into physical memory. As illustrated in the following figure, VMM extends the operating system process to map your data into virtual memory.



Distributed object applications typically interleave computation very tightly with data access. They do some computation, navigate through some object references, change a few values, issue a *query*, and then do some more computation, and so on. If it were necessary to communicate with a remote data server for each of these operations, the overhead would be enormous. By making the data directly available to the application and allowing ordinary instructions to manipulate the data, CFA applications avoid this overhead.

Many applications tend to reference large numbers of small objects. However, one operation on a group of objects is far more efficient than multiple operations on discrete objects. When VMM retrieves an object from the CFA server, it retrieves the page that the object is on. In other words, it retrieves all objects that are stored on the same page as the target object. If the application needs to operate on the other objects on the target page, those objects are already in memory. In a Java application, the CFA client brings objects from the in-memory client page cache into the Java VM.

In addition, object *clustering* works with VMM to improve performance. Object clustering lets you specify where to store an object. For example, if there is a group of objects that you frequently use together, you can store them near each other. This increases *locality*, which is the degree to which objects that are used together are stored near each other. The higher the locality, the greater the chance that an application can retrieve frequently used objects by accessing a single page. Also, high locality minimizes the data transfers between CFA servers and clients.

Clustering for increased locality not only groups related data (the working data set) on a minimal number of pages but also stores objects with the pointers that reference them.

When such a page is mapped into virtual memory, the amount of virtual memory that must be reserved for pages that contain pointed-to objects is minimal.

Because clustering lets you specify where to store an object, you can also use it to distribute data:

⇒ Among multiple distributed CFA servers for scalability

⇒ Across multiple files or databases or *storage caches* for efficient management

### How VMM Manages Very Large Datasets

CFA is the foundation for deployed applications that manage datasets that scale into the terabytes. With VMM, in one transaction, an application can address a dataset that is larger than the address space of the operating system. The application can do this because VMM dynamically assigns and reassigns portions of the dataset to address space.

Without VMM, the amount of data that an application can map is limited by the amount of virtual memory available to the application. With VMM, the amount of data that an application can map is limited only by available disk space.

CFA maintains a virtual address map that shows which pages are represented by which addresses. As the application references more objects, VMM assigns additional address space and maps the pages containing the new objects into these new addresses. When more address space is needed but no more address space is available, VMM checks the application's address space for pages that are no longer in-use. VMM swaps out the unneeded page mappings to make room for needed page mappings. At the end of each transaction, VMM makes all of the client's address space available. However, VMM keeps the objects from the previous transaction in the cache. If the new transaction operates on the same data as the previous transaction, the pages are already mapped into virtual memory.

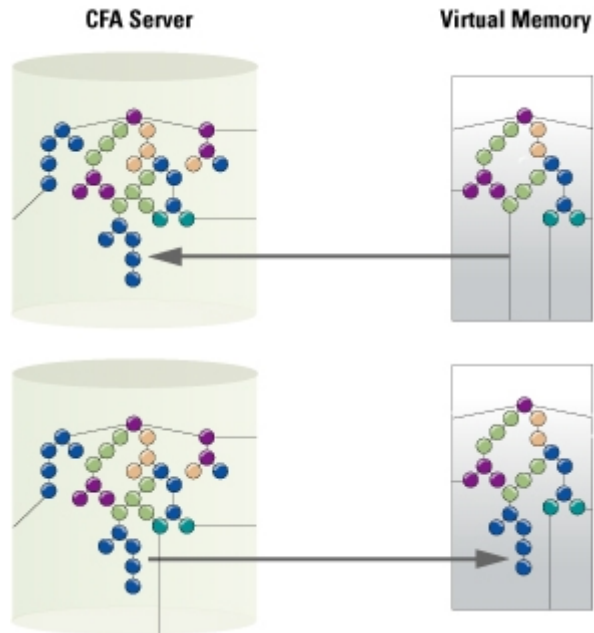
### Single Level Storage

In the C++ binding, objects are stored on the CFA server in the same format in which they are represented in virtual memory by the compiler. This avoids potential per-object overhead, such as reformatting the object. However, when operating in a *heterogeneous* environment some reformatting might be required, for example, byte order switching. (CFA provides full support for heterogeneous platform operations.)

With the Java binding, the CFA client manages the data in memory exactly as the language represents it, thus providing in-memory performance. The Java binding also takes advantage of the paging architecture. After pages are in the in-memory client page cache, the Java CFA client then brings these objects into the Java VM.

The following figure shows VMM dereferencing a pointer. The object in virtual memory is in the same format as the object on the CFA server. On the CFA server, the lines that go to the edge represent pointers to objects managed by another CFA server. In virtual memory, the lines that go to the edge represent pointers to objects that are not mapped into virtual memory. Contiguous objects of the same color are on the same page.

It is important to note that all the data you want to access in a single transaction does not need to fit into memory at the same time.



Dereferencing a pointer to a target not in memory signals a page fault. The client contacts the server to obtain the page that contains the target object.

The dark blue disks at the bottom of the object tree represent the objects on the page that contains the target object. One of these objects is the target object.

The server sends the page to the client. The client maps the page into memory in the same format in which it is stored.

## Distributed Data Processing

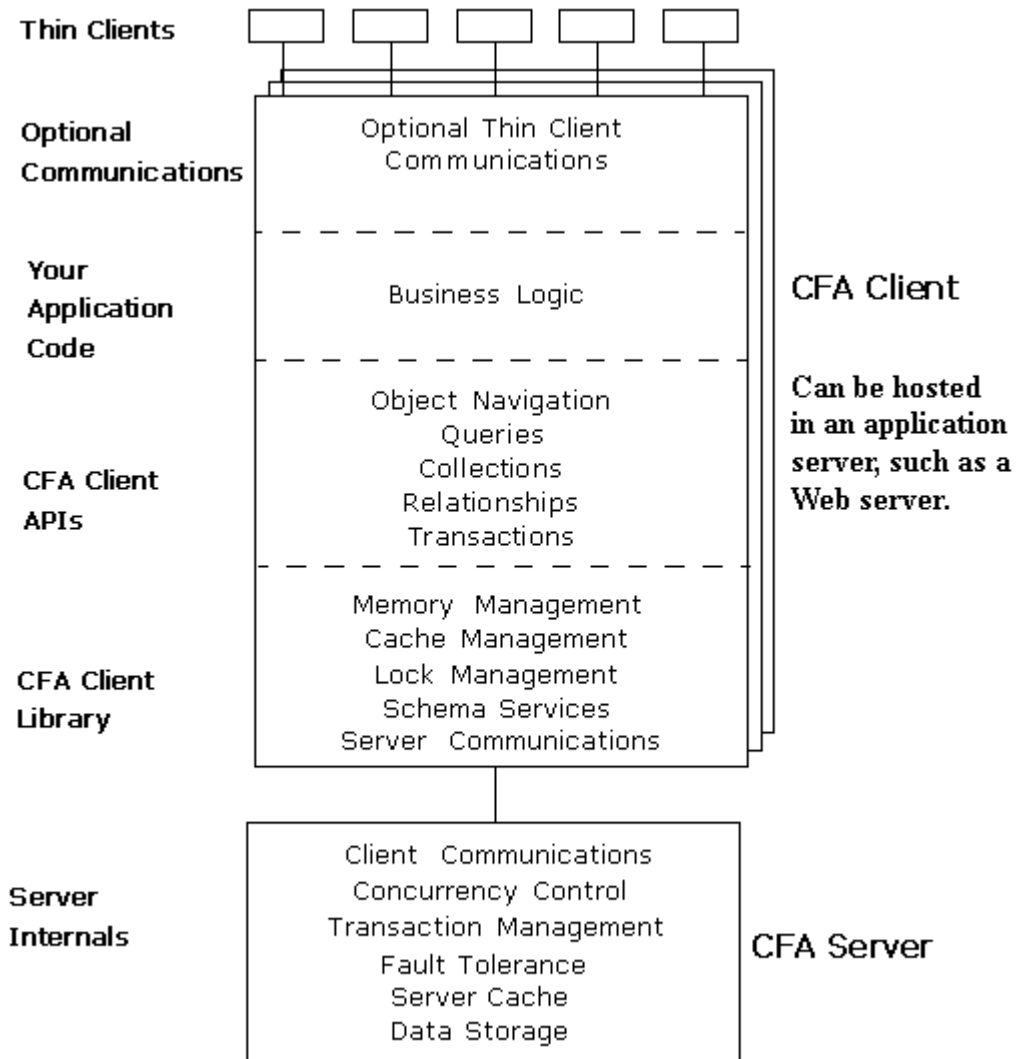
CFA is a distributed architecture in which clients and servers share data processing. In a multi-tier configuration, CFA clients typically reside in the application server tier where they significantly improve the performance of the application server.

The CFA server and the CFA client communicate by means of a local area network when they are running on different machines. They communicate by operating system facilities, such as shared memory or local sockets, when they are running on the same machine.

CFA servers manage physical data on disk, arbitrate among client processes that are requesting objects, and ensure that all clients have consistent views of data.

CFA clients provide the interface between your application and the CFA server. Each CFA client has a view of data that corresponds to its view of memory. In effect, the database is an extension of the application's heap. However, the access to this memory is transactional with all the ACID properties of normal database access.

The following figure represents the process architecture of CFA clients and servers in a distributed, service-oriented architecture. Lines represent communication. The larger boxes delineate the roles of each process.



## Client Caching

On each client host, CFA maintains an internal data page cache for each client. The client page cache contains objects that your application has recently used. By caching pages of objects on CFA clients, CFA makes the data accessible where the client needs it. Since CFA clients typically contribute to the operation of application servers, client page caching lets CFA deliver in-memory response times when your application is fulfilling service requests.

Applications that are based on service-oriented architectures tend to employ complex data models. To fulfill service requests, applications often traverse relationships between objects in these complex models. Consequently, a client page cache greatly reduces processing overhead because it contains groups of related objects that have been mapped into memory together in a format ready to be used by the client. (Remember that in the Java binding, the CFA client brings the object from the client page cache to the Java VM.)

The following topics provide the details about client page caching:

- ⇒ Client Page Cache Features
- ⇒ Client Page Cache Performance

⇒ About Locks on Pages in the Client Page Cache

## Client Page Cache Features

CFA's client page cache provides:

- ⇒ **Consistency** — When your application uses data in the page cache, that data is always up to date and in a coherent state. Your application can query and update data in the page cache.
- ⇒ **In-memory access speed** — Your application can read and write data in the page cache at in-memory speeds.
- ⇒ **Scalability** — When the page cache is full and cache space is needed, the page cache makes room for new objects by evicting the least recently used objects. This reuse of cache space helps your application scale to accommodate larger amounts of data without sacrificing performance. If there are modifications in the evicted pages, the CFA server logs them. When your application commits the current transaction, the update includes the logged modifications as well as any modifications in the page cache.
- ⇒ **Protection from failure** — At the end of a transaction, the page cache passes new or amended data to the CFA server. At all times, the CFA server has the latest committed version of the data. If there is an application failure, CFA removes the changes made by the uncommitted transactions.

## Client Page Cache Performance

For an application to access an object, the client must map the page that contains the object into the application's address space. To map the page into address space, the client must have the appropriate lock for the page and must relocate the pointers on the page. Consequently, in addition to caching pages, each client application maintains lock information and can maintain relocation information about each cached page. The CFA client automatically takes care of saving lock information. The way you end a transaction determines whether the client also saves relocation information.

Relocation refers to a set of tasks that the CFA client performs on a page, preparatory to making it accessible to the application or to storing the page on the CFA server. These tasks require the client to examine and sometimes modify pointer members of an object, while preserving the logical content of the data in the field. During relocation that makes the page accessible, the CFA client converts the pointers on the page from the encoded values they have on the CFA server into address values that are meaningful to the application. This means that multiple processes can access the same logical data page. Each process maps the data into its own address space.

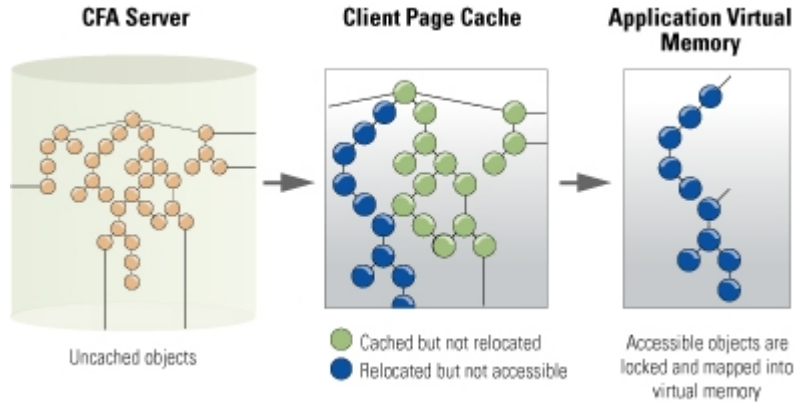
Each page in the cache has a corresponding lock state. Pages that are currently being read or written in a transaction have a read or write lock, respectively. When the transaction commits, the client caches these locks unless another client is making a conflicting lock request. In this way, the client optimizes the reuse of cached pages because it avoids network communication with the server.

Each object goes from an uncached state to an accessible state:

1. An object changes from an uncached state to a cached state when the page it is on is fetched from the server.
2. An object changes from a cached state to a relocated state when the client relocates the pointers on the page.
3. An object changes from a relocated state to an accessible state when the client locks the page and maps the objects into the application's virtual memory.

When an application accesses a previously accessed object, the object might be in the cache and it might already be mapped into virtual memory. The figure below shows the

states that an object goes through. Each disk represents an object. A page contains multiple objects.



When your application tries to access an object, the state of the page that contains the object impacts the speed at which the application can access that object, as shown in the following table. In addition, the client page cache incorporates the most important CFA elements.

<i>Page State</i>	<i>Page Temperature</i>	<i>Description</i>	<i>CFA Element</i>
Not cached	Cold	A page fault requires network and disk activity. The client fetches the page from the CFA server, performs relocation, and grabs the lock.	Distributed data processing
Cached	Lukewarm	The page is in the client page cache but the client needs to relocate the pointers in the cached objects. A page fault requires the client to grab the lock and perform relocation. The required CPU activity depends on the complexity of the objects. Communication with the CFA server is not required for the client to regain a previously held lock.	VMM
Relocated	Warm	The page is in the client page cache but it is not accessible to the application. A page fault requires the client to grab the lock. Communication with the CFA server is not required for the client to regain a previously held lock.	Transaction management
Accessible	Hot	The page is mapped into the application's address space. The client has the appropriate lock. There is no page fault and no overhead.	Single-level storage

CFA applications with the best performance keep the pages in the cache as hot as possible. In applications that have a large enough page cache, good locality, and low lock contention, pages rarely get cold. In applications that can take advantage of the page cache's relocation optimization feature, pages stay warm. Relocation optimization lets the client relocate the page once and keep it relocated. Applications that take advantage of transaction management, such as ObjectStore's C++ Middle Tier Library (*CMTL*) and Java Middle Tier Library (*JMTL*) have excellent performance because they maximize the application's use of pages in the hottest state.

You reap the benefits of the page cache automatically. There is nothing you need to code to achieve these advantages. The default configuration of the page cache is suitable for

many applications. Should you need to fine-tune client page cache resources, you can do so because the size of the page cache is configurable and is independent of the application code. Your application requirements and system resources govern the actual size.

### About Locks on Pages in the Client Page Cache

All locks are in place until the transaction ends. Ending a transaction releases all locks. That is, all objects that were in use during the transaction become available to other clients when the transaction ends. Pages that are in the client's page cache at the end of the transaction remain in the client's page cache until another process requests to update them. Although the client releases the locks, it maintains information about the locks it held. For pages that remain in the page cache, communication with the CFA server is not required to regain a lock that was previously held.

CFA avoids the network overhead of reacquiring locks for each new transaction. Data that is not being modified by another client can stay in the page cache and is immediately available to the application. The only communication with the CFA server is to commit modified or newly created data.

### Transaction Management

CFA transactions automatically handle concurrent access to objects to ensure that an update to one object does not interfere with an update to another object. A transaction is a unit of work that you determine in your application. An operation that reads or writes data must occur inside a transaction.

CFA transactions transparently provide the expected behaviors. What distinguishes CFA's *concurrency control* from traditional OODMS is the performance gain derived from distributing concurrency control to clients, including clients that participate in application servers in the middle tier. CFA transaction management features include:

- ⇒ Standard Transaction Behaviors
- ⇒ Callback Locking
- ⇒ Multiversion Concurrency Control
- ⇒ Middle Tier Concurrency Control

#### Standard Transaction Behaviors

CFA transactions provide all the usual features you would expect. That is, they are:

- ⇒ **Atomic** — Either all changes in a transaction are successful or no changes in that transaction are made.
- ⇒ **Consistent** — Your data moves from one consistent state to another consistent state. When a transaction commits, CFA writes all changes to stable storage.
- ⇒ **Isolated** — Each transaction is independent of any other transaction, including those that occur at the same time.
- ⇒ **Durable** — Updates that were made in completed transactions remain permanent, even if the system fails.

Of course, CFA detects transaction *deadlocks* and takes action to end them.

#### Callback Locking

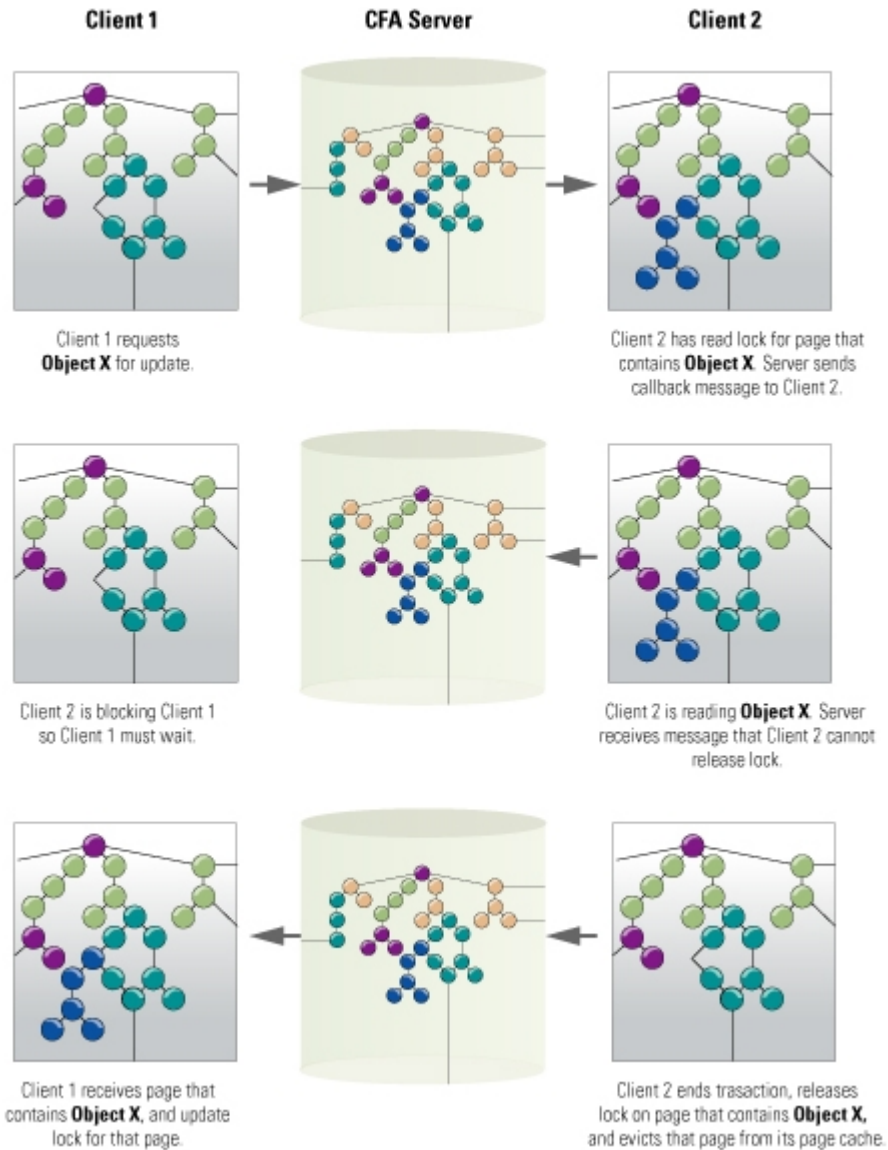
CFA caches lock information on CFA clients. This minimizes the need for network communication when the same process performs consecutive transactions on the same data. The CFA server keeps track of which pages are in the page cache of each client, and whether the client has permission to obtain a read or write lock for each cached page.

When a client requests an object from the CFA server and that object is in the page cache of some other client, the CFA server checks whether the lock requested conflicts with the lock already granted. For example, suppose there is a request for a read lock on an object that is already in another client's page cache and that client has permission for a read lock. Since multiple clients can read the same object simultaneously, these locks do not conflict. The CFA server sends the page that contains the desired object to the requesting client.

However, suppose the request is for an update lock. Since only one client can have an update lock or multiple clients can have read locks, the requested lock would conflict with the granted lock. In this scenario, the CFA server sends a message that asks the client to remove the page that contains the requested object from its page cache. Since this message is an attempt to call back permission for the lock, it is referred to as a callback message. (There is an exception to the one writer or multiple readers rule. See Multiversion Concurrency Control (MVCC) on page 16.) The result of the callback message depends on the current situation:

- ⇒ If the requested object is not in use in the current transaction, the client immediately removes from its page cache the page that contains the object and gives up permission to grab the lock. The CFA server grants the lock to the requesting client.
- ⇒ If the requested object is in use in the current transaction, the CFA server receives a message that indicates that the object is in use. The CFA server forces the requesting client to wait until the lock holder ends the transaction. When the holding client commits or aborts its transaction, it relinquishes the lock, and removes from its page cache the page that contains the target object. The CFA server allows the requesting client to proceed.

The following figure illustrates callback locking. Each disk represents an object. A page can contain one or more objects, and locks are on pages. Although the request is for a single object, the lock is on the page that contains Object X as well as other objects. Thus, a page (more than one object) is evicted from Client 2 and received by Client 1. Object X is one of the dark blue disks.



To minimize the need for callback locking, you can use clustering to distribute objects. You can manage and eliminate database concurrency hot spots without the performance degradation associated with per-object locking techniques. When appropriate, you can even isolate objects on individual pages.

### Multiversion Concurrency Control (MVCC)

The basic CFA locking model withholds an update lock when there is an in-use read lock on the data to be updated. This locking model can be too restrictive for applications that require very high throughput for read operations. With CFA's Multiversion Concurrency Control (MVCC), read operations do not block concurrent update operations, and vice versa. Instead, read-only transactions run in parallel, unaffected by update transactions. All write transactions follow the normal locking protocol and compete for locks with each other. When applications read data in MVCC mode, the data might be slightly out of date – another application might have recently updated that data—but the data is always transactionally consistent. The application is always reading committed data. An application never reads uncommitted or transactionally inconsistent data. When an application reads data in MVCC mode, the client brings its snapshot of the data up to date each time it starts a new transaction. Thus, the application controls how old the data snapshot can be.

Consider the figure on the previous page. If Client 2 is using MVCC, then Client 2 does not block Client 1. Even though Client 2 is reading Object X, Client 1 can obtain a write lock for Object X while Client 2 continues reading Object X.

Concurrent (nonblocking) read and update operations are at in-memory speed, and they provide transactionally consistent views of the data. Activities such as report writing can benefit from nonblocking read access to your data. Nonblocking read access can also be useful in applications with live data feeds such as stock tickers. One transaction might capture stock quotes and update your data, while several other transactions read the data and apply business logic.

### **Middle Tier Concurrency Control**

ObjectStore's Middle-Tier Libraries for C++ (CMTL) and Java (JMTL) build on CFA's concurrency control capabilities. For applications in the middle tier of distributed multi-tier architectures, CMTL and JMTL let you add *transactional caches* to increase concurrency and improve performance.

CMTL and JMTL include cache pool managers that oversee the transactional caches and keep them transactionally consistent. To minimize traffic to the back-end server, cache pool managers use virtual transactions to group and schedule requests. Only virtual transactions can access data in transactional caches. CMTL and JMTL efficiently map all virtual transactions into a minimal set of physical transactions.

Each transactional cache is for updates or MVCC access. A virtual update transaction or a virtual read-only transaction can access data in an update cache. However, only a virtual read-only transaction can access data in an MVCC cache. MVCC caches do not block concurrent virtual update transactions that operate on the same data that is in the MVCC cache.

Each transactional cache belongs to a cache pool, which is a collection of one or more transactional caches. As the size of your data grows, or as the number of requests increases, you can add transactional caches and/or transactional cache pools to partition data or the requests for data into mutually exclusive aggregates.

Each CFA client that participates in the operation of a middle-tier application server works on behalf of multiple thin clients. Consequently, CFA clients have capabilities built in that let them manage transactions and threads in the middle tier.

## **Scalable and Integrated Deployment Options**

CFA is the foundation for applications that provide a variety of deployment options:

- ⇒ Enterprise-Class Object Databases
- ⇒ Caching Applications That Access Relational Databases
- ⇒ Small-Footprint Object Databases

### **Enterprise-Class Object Databases**

ObjectStore is a multiuser, enterprise-class, object database that stores objects in their native format. Beyond the CFA features already discussed, the distinguishing ObjectStore features are its distributed resources, multithreaded server, and administrative tools. In an ObjectStore installation, the CFA server and CFA client are referred to as the ObjectStore server and ObjectStore client.

#### ⇒ **Distributed Resources**

In an ObjectStore installation

- An ObjectStore server can support many ObjectStore client applications.
- An ObjectStore client can simultaneously connect to multiple ObjectStore servers.

- An ObjectStore server can support several databases.
- An ObjectStore client can access multiple databases on many ObjectStore servers.
- ObjectStore servers and ObjectStore clients can reside on the same machine or on different machines connected by a high-speed local area network.

Distributed resources contribute to ObjectStore's scalability. ObjectStore has been deployed in applications that manage multi-terabyte databases.

Objects in one database can reference objects in another database, even if that database is managed by a different ObjectStore server. Distribution of objects across multiple databases and ObjectStore servers is transparent to your application. If an application follows a reference from one object to another, and the referenced object happens to reside in a different database, ObjectStore automatically opens the other database and connects the client to that database's ObjectStore server.

#### ⇒ **Multithreaded Server**

ObjectStore's server is a multithreaded process. Depending on the platform support, the server can improve throughput and responsiveness by taking advantage of support for native operating system threads, symmetric multiprocessor (SMP), and asynchronous I/O (AIO). Native operating system threads allow the server to concurrently handle multiple client requests. SMP allows the operating system to schedule multiple requests at the same time (on different CPUs). AIO allows a request to simultaneously perform I/O operations and CPU processing.

On some symmetric multiprocessor architectures, the native thread package is enhanced to support running threads from a single process on different processors. Support for this enables the ObjectStore server to leverage the SMP architecture and achieve even greater performance advantages.

#### ⇒ **Administrative Tools**

ObjectStore includes enterprise-class administrative tools for configuring *failover/high availability systems, replication, backing up and restoring data, and archive logging.*

### **Caching Applications That Access Relational Databases**

CFA has often been deployed in multiuser, caching applications that provide in-memory, object-oriented access to data stored in relational databases. Such applications translate data between relational and object formats. This caching ability lets you implement an application that takes advantage of CFA's performance features to service large numbers of requests for data in enterprise data stores.

After an application translates data from corporate data stores into object format, a CFA caching application puts the objects in storage caches. Storage caches are ObjectStore databases that are used as temporary storage for objects your application operates on. CFA caching applications can use as many storage caches, in addition to client page caches, as needed. Data in the storage caches is in the same form in which it is used. The application accesses this data without having to map between its format in the relational data sources and the format used by the application.

### **Small-Footprint Object Databases**

As processing capability grows and device size shrinks, an increasing number of applications that run on smaller devices can also apply the advantages of CFA. CFA has been embedded in network devices, handheld devices, and even copiers and sewing machines. ObjectStore PSE Pro was designed to leverage the aspects of CFA that are appropriate for these kinds of small-footprint deployments. PSE Pro's memory footprint is approximately 400 KB, which makes it ideal for these embedded scenarios.

ObjectStore PSE Pro is a single-process storage engine that manages objects in their native format and stores them in databases, which are local, operating system files. PSE Pro embeds the CFA server in the CFA client process.

In PSE Pro's embedded environment, VMM maps objects directly from disk to memory. CFA's single-level storage ensures that the format of an object in the database is the same as its format in memory. The data a PSE Pro application operates on is in memory. PSE Pro's storage management is transactional, however; it provides the full ACID characteristics of a conventional DBMS. This transactional behavior is an important advantage of using PSE Pro instead of a simple, "home grown," flat-file-based approach.

PSE Pro benefits from the performance, productivity, and transaction management advantages of CFA. As a single process, embedded, data management engine, PSE Pro, of course, cannot benefit from data distribution, locking, and caching. One application at a time can update a PSE Pro database. Multiple applications can simultaneously read from the same PSE Pro database.

The API for using PSE Pro is a subset of the API for using the ObjectStore object database. This makes it easy to change an application from PSE Pro to the ObjectStore object database as the needs of the application change.



## *Conclusion*

CFA is optimized to bring data management capabilities into the memory space of middle tier application servers. For C++ and Java applications that perform operations such as:

- Managing data for rich C++ or Java object models
- Caching enterprise data in the middle tier
- Building real time, high-performance event processing systems

ObjectStore's CFA offers:

- Application performance at in-memory speed
- Scalability for handling a deluge of data because VMM dynamically maps data into and out of virtual memory as needed to extend the operating system's virtual address space
- Ease of use because the interface is standard C++ or Java
- Elimination of bottlenecks associated with managing distributed data because you can use clustering to determine where to store each object

The CFA elements that make this happen are:

- Virtual memory mapping — VMM dynamically maps objects into memory as needed.
- Single level storage — Objects are stored on the server with the same representation as objects in memory.
- Distributed data processing — CFA servers and clients share data management and data processing responsibilities.
- Client caching — Clients keep recently used objects and locks.
- Transaction management — Clients share lock management with servers.
- Scalable and integrated deployment options — CFA can be the foundation for an enterprise-class object database, a cache application for relational databases, or a small-footprint object database that is a replacement for file-based storage.

## Glossary

This glossary defines CFA concepts as well as features supported by ObjectStore.

archive logging	Records transaction activity, which provides the information you need to recover modifications made after a backup.
backup and restore	Copies data to secondary storage, and restores data from secondary storage to primary storage.
CFA server	Manages physical data on disk, arbitrates among CFA client processes that are requesting objects, and ensures that all clients have consistent views of data.
CFA client	Provides the interface between your application and the CFA server. Your application uses a CFA client to get access to a logical view of your data and to apply business rules to that data. For example, in a real time data processing application, a CFA client can perform a query over a body of data that includes new data coming in as well as historical data.
client page cache	On each client host, CFA maintains a page cache for each client process. The client page cache holds pages that contain objects your application has recently used.
clustering	Lets you specify where to store an object. For example, if there is a group of objects that you frequently use together, you can store them near each other to increase locality. Conversely, you can store objects in different locations to ensure maximum concurrency.
CMTL	The C++ Middle-Tier Library provides a data caching technology that moves processing away from the back end and concentrates it in the middle tier, making data readily available to the business logic at in-memory speed while ensuring data consistency and transactional integrity.
collections	Objects such as sets, bags, or lists that serve to group together other objects.
compaction	Frees storage space so it can be used by other objects.
concurrency control	Provides simultaneous access to objects to ensure that an update to one object does not interfere with an update to another object.
deadlock	Occurs when two clients waiting for resources are each waiting for the resources being used by the other client.
deadlock detection	Detects and breaks deadlocks by aborting one of the transactions involved in the deadlock. This causes the victim's locks to be released so that other processes can proceed.
dump/load	Lets you dump data into an ASCII file and generates a loader executable that is capable of creating, given the ASCII file as input, an equivalent database or group of databases.
failover	Ensures that the failure of a CFA server, or the failure of the machine on which the server is running, does not prevent service to CFA clients. When failover is in effect, the failure of the protected server is immediately detected, and its service is picked up by another server running on its own machine, providing clients with continued access to objects.

heterogeneity	A CFA server can support a client of any type regardless of the client's data format, byte ordering, floating-point representation, or data alignment. A client with one processor architecture can generate data to be read by client applications residing on machines with processors that are different from the original client. For example, a Sun server can support Sun, Hewlett-Packard, IBM, and Windows clients simultaneously.
JMTL	The Java Middle-Tier Library provides transparent, high-performance storage for Java objects. JMTL caches objects accessed by client-initiated transactions, maintains the consistency and recoverability of the transactional caches, maintains each transaction's required isolation level, and schedules transactions to optimize throughput.
locality	Degree to which objects that are used together are stored near each other on the CFA server.
metaobject protocol	Library of classes that allows you to access ObjectStore schema information.
MVCC	Multiversion Concurrency Control allows objects that are being accessed by multiple read-only transactions to be simultaneously updated by one update transaction.
notification	Lets a CFA client notify other clients that an event has taken place. Typically, the event is a change in the objects to which other clients have access. Using the notification service, clients can send and receive notifications.
page fault	Operating system mechanism that notifies CFA that a client is trying to access an object. CFA installs its own page fault handler.
queries	Returns a collection that contains those elements of a given collection that satisfy a specified condition. Queries can be executed with an optimized search strategy, formulated by the query optimizer.
query indexes	The query optimizer maintains indexes into collections based on user-specified keys, that is, data members, or data members of data members, and so on. With these indexes, implemented as B-trees or hash tables, you can minimize the number of objects examined in response to a query. Formulation of optimization strategies is performed automatically by the system. Index maintenance can also be automatic.
recovery	Process of returning a database to a transactionally consistent state if any process aborts or any host crashes, or in the event of network failure.
replication	Continuously updated copy (or replica) of your objects.
relationship facility	Models binary relationships between pairs of objects and maintains referential integrity when required.
schema evolution	Modification of schema information associated with stored objects, and modification of any existing instances of the modified classes.
storage cache	ObjectStore database used for temporary storage during application run time.
transactional cache	Transactional caching is the underlying concept of the CFA middle-tier libraries. It refers to the automatic routing of client

requests to separate CMTL and JMTL caches, allowing in-memory access to cached objects. Transactional caches ensure data integrity by providing transactional consistency among the caches, even while servicing multiple, concurrent transactions against the same data.

transactions

Execution of a sequence of statements that operate on objects as a logical unit. That is, the operations performed by the statements are performed all together or not at all.

VMM

Virtual memory mapping is the element of CFA that dynamically maps objects into memory.



## *About Progress Real Time Division*

The Progress Real Time Division provides event stream processing, data management, data access and synchronization products to enable the real-time enterprise. Our products manage and analyze real-time event stream data for applications such as algorithmic trading and RFID; accelerate the performance of existing databases through sophisticated caching; manage and process complex data in the industry's leading object database; and support occasionally connected mobile users requiring real-time access to enterprise applications. The Progress Real Time Division is an operating unit of Progress Software Corporation (Nasdaq: PRGS), a global software industry leader. Headquartered in Bedford, Mass., they can be reached at [www.progress.com/realtime](http://www.progress.com/realtime) or +1-781-280-4000.

**PROGRESS**  
SOFTWARE

**Real Time Division**

[www.progress.com/realtime](http://www.progress.com/realtime)

#### Worldwide and North American Headquarters

Progress Real Time Division, 14 Oak Park, Bedford, MA 01730 USA Tel: +1 781 280 4000

#### UK and Northern Ireland

Progress Real Time Division, 210 Bath Road, Slough, Berkshire, SL1 3XE England Tel: +44 1753 216 300

#### Central Europe

Progress Real Time Division, Konrad-Adenauer-Str. 13, 50996 Köln, Germany Tel: +49 6171 981 127

#### France

Progress Real Time Division, 3 Place de Saverne, Les Renardières B, 92901 Paris la Défense Tel: +33 1 41 16 16 56

© 2006 Progress Software Corporation. All rights reserved. Progress, ObjectStore, Event Store and Cache-Forward are trademarks or registered trademarks of Progress Software Corporation, or any of its affiliates or subsidiaries, in the U.S. and other countries. Any other trademarks or service marks contained herein are the property of their respective owners. Specifications subject to change without notice. Visit [www.progress.com/realtime](http://www.progress.com/realtime) for more information.