# SanFrancisco performance: A case study in performance of large-scale Java applications

by R. Christ
S. L. Halter
K. Lynne
S. Meizer
S. J. Munroe
M. Pasch

***This is a study of the performance progress of the SanFrancisco™ project from the time the decision was made to base it on the Java™ language up to the time SanFrancisco applications were successfully deployed in the marketplace—from February 1997 until late 1999. We document the challenges, decisions, and technologies that were encountered during the three-year development period that saw performance improve by orders of magnitude. Key areas that allowed us to achieve this improvement were intelligent object caching, improved object access strategies, use of commands (function shipping), efficient mapping of objects to the underlying database, appropriate usage of Java, programmer education, and acquiring (or building) needed tools. We also discuss several areas where challenges remain and more progress is needed.***

The SanFrancisco* product establishes a new paradigm for building business applications. Targeted at independent software vendors (ISVs), SanFrancisco provides a distributed object infrastructure, common business objects, and business process components. Together these prebuilt components provide a platform-independent application structure designed for extension by ISVs to produce end-customer applications. The overall objective is to provide the ISVs with at least 40 percent of an overall business application in prebuilt content designed to be extended.

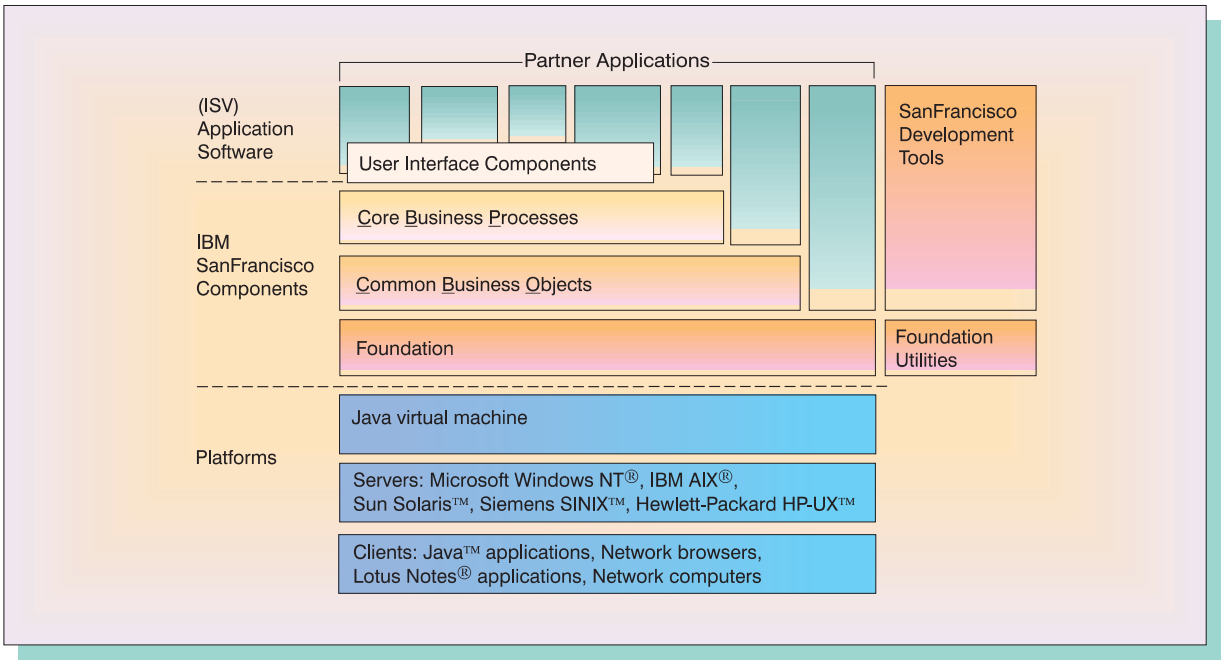
The SanFrancisco product is written almost entirely in Java** and is believed to be one of the largest Java development efforts at this time. This paper focuses on the performance concerns the SanFrancisco project faced and how we addressed them.

The SanFrancisco architecture consists of three layers of reusable code that can be extended by application developers (see Figure 1). The highest layer, the core business processes (CBPs), provides business objects and default business logic for various "vertical" application domains. These are sometimes also referred to as "towers." The second layer, the common business objects (CBOs), provides general-purpose business objects, interfaces for interoperability across core business processes, and reusable




0018-8670/00/$5.00 © 2000 IBM

**Figure 1 SanFrancisco framework architecture**




design pattern implementations. The lowest layer, the foundation, provides the infrastructure and services that are required to build industrial-strength applications using distributed, managed objects, across multiple platforms and relational databases. Application developers can use the prebuilt content at any of the three layers.

## Background and history

The SanFrancisco project came into existence approximately five years ago as a result of independent software vendors coming to IBM seeking a solution to problems they were encountering. An increasingly large percentage of their development budget was being spent on multiple platform support, technology infrastructure, and business logic that was not unique to their industries. This resulted in insufficient resources remaining to react to changing customer requirements and to implement new features that differentiate each vendor from its competitors. They were also facing the need to update their aging application bases to handle issues like Year 2000 preparation and European currency conversion. Most ISVs wanted to update their programming methodologies and skills from procedural to object-oriented development, for increased flexibility to adapt quickly to new requirements and for reduced maintenance costs. Many ISVs realized they could not afford the risk to proceed on their own, and needed to join forces with other companies to deliver competitive solutions. Thus the SanFrancisco project was conceived, and IBM worked with a key set of these ISVs to define a comprehensive solution to address their concerns.

When we chose the Java technology as our implementation environment in early 1996, it was not yet mature, and we knew that it would present significant technical challenges for the SanFrancisco project. So in November 1996 we established a dedicated SanFrancisco performance team to focus on creating meaningful benchmarks and goals, measuring, analyzing, prototyping, documenting, and driving performance improvements into the final product. Since our initial path-length benchmark measurement in late 1996, SanFrancisco performance has improved by a factor of 500.

Many benchmarks are available for small or numerically intensive Java client applications, and these show great results when just-in-time (JIT) compiler technologies are used. However, the SanFrancisco product workload of server-oriented business appli-

cations did not match the workload measured by these benchmarks. As a result the performance team wrote various Java server benchmarks that were inspired by the industry-standard TPC-C** benchmark.[1] Using our benchmarks, we have found only about a 40 percent gain from JIT technology, which is far short of the results obtained with other benchmarks.[2] This is due to the greater complexity of SanFrancisco frameworks over simple program loops on which most Java benchmarks are based.

This complexity includes database I/O, contention for shared resources (objects), and some native code. At this time, we continue to explore new JIT technology as well as the benefits of static compilation technology.

Performance tools are always critical in developing serious business applications, and Java-based performance tools have not been adequate up to this point. However, the existing tools did allow us to find numerous bottlenecks in SanFrancisco code, Java virtual machines, and Java class libraries and to track memory allocation problems. The performance team created several lock contention analysis tools to aid in discovery of improvements for higher concurrency and overall throughput.

While SanFrancisco currently supports good throughput with subsecond transaction rates and continues to make significant advances, it has not yet reached its aggressive goal of matching traditional procedural applications, which still have at least a 50 percent performance advantage.

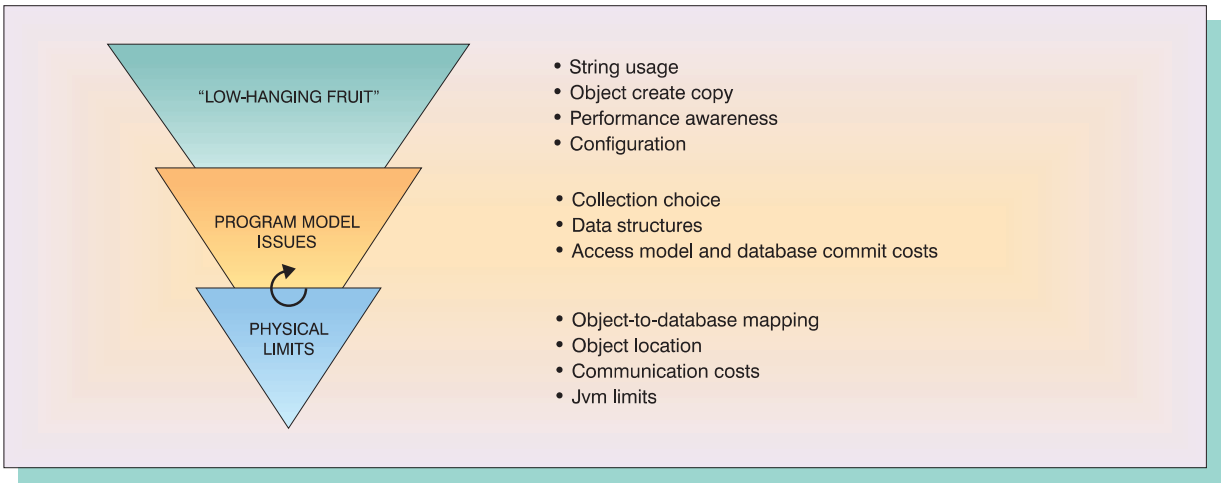
We continue to work with others in IBM and in the industry to ensure that the next generation of Java technology works well for this emerging class of new applications.

## Problems

Given the complexity of the SanFrancisco environment (see Figure 1), it should be no surprise that there are numerous opportunities for performance bottlenecks and problems. There are of course the time-honored pitfalls that can ensnare any application: inefficient or badly coded algorithms, excessive contention for shared resources, system configuration errors, communication bottlenecks, etc. Added to these "ordinary" problems are SanFrancisco-specific problems because of its architecture—cross-platform, distributed, object-oriented, component-based—and new Java technology. These problems stem from some combination of (1) being a framework that can be used (or abused) in many ways by the end application, (2) being written in Java, thus requiring a separate run-time environment that uses a specific set of run-time library routines, and (3) providing data persistence via a relational database (RDB) in an object-oriented environment. The performance problems the SanFrancisco team faced were of both the ordinary and "extraordinary" kind, and we categorize them as follows:

1. *Excessive code path.* Too much code was being generated and executed for the amount of underlying logic. Sometimes this was just due to bad choices of algorithms or poor coding practices, but more often it was due to some aspect of the Java language or run-time library that was not well understood. And of course because Java is often executed interpretively, these problems are a little worse. Additionally, the C++ skills many Java programmers already have can unknowingly lead to inefficient coding practices. In C++, for example, "strings" (null-terminated arrays of characters) are quite efficient, while the analogous "String" class in Java is much less so, and should be avoided in performance-sensitive situations.
2. *Inefficient framework usage.* SanFrancisco provides programmers with many choices in developing applications. For example, there are multiple ways to organize collections of objects (arrays, lists, maps, sets, and extents) and many ways to access and lock them. While at a high level they all accomplish basically the same thing, they have radically different performance characteristics, and the performance penalty for choosing incorrectly can be severe.
3. *Configuration problems.* Because of the added complexity of both the Jvm (Java virtual machine) and additional configurable structures in the SanFrancisco frameworks themselves, proper configuration of the resources on the host machine becomes more complex as well. Besides the obvious things—making sure the system has enough virtual memory (paging space) and the database has enough buffer space—there are the size of the Jvm heap, the SanFrancisco object caches, and the schema-mapping caches to consider as well. An imbalance in the allocation of resources among these can create performance bottlenecks that may be hard to spot without special tools and knowledge.

**Figure 2 Performance methodology**




4. *Jvm idiosyncrasies.* Most SanFrancisco code runs within some Jvm (the exception is a small amount of C++ code for database connectivity). But SanFrancisco runs on a number of different platforms, and each platform has its own Jvm. Although each of these Jvms is functionally equivalent, the Java specification does not guarantee performance characteristics, and some of these characteristics have profound ramifications. In particular, the garbage collection algorithms, the efficiency of the supplied JIT compiler and run-time libraries, and the availability of tuning and configuration parameters are all factors. The SanFrancisco project has had to deal with all of these in order to understand and address specific performance problems across the various platforms it supports.
5. *Object persistence.* The efficient mapping of Java objects to relational database structures has been a particularly daunting challenge within the SanFrancisco project. SanFrancisco uses native C++ code (via the Java Native Interface) to store and retrieve the contents of persistent SanFrancisco and Java objects within an RDB. This was done, rather than using Java Database Connectivity (JDBC**), to give us the ability to use two-phase commit techniques and control more precisely the database mapping. Doing this efficiently so that the query capabilities of the underlying RDB can be effectively utilized is absolutely required in order to ensure acceptable performance over large numbers of objects.

The next sections detail how the SanFrancisco team confronted and ultimately prevailed (at least partially) over most of these problems. Particular attention will be paid to those problems that were caused or made worse by the object-oriented Java language and the Java run-time environment.

## Methodology

The first step to solving the performance problems was to identify a methodology to use. Figure 2 represents the performance focus areas in the SanFrancisco project.

The top level represents simple changes that can be made while coding or doing low-level design. Most of these types of problems can be averted by simply being performance-aware, making them topics for developer education on the performance characteristics of Java technology.

The middle section of the diagram represents the SanFrancisco programming model's effect on performance. A designer needs to understand the capabilities of the programming model. For instance, the designer must make a proper choice of what collection to use, and what objects to use in general. The designer or implementer must also make good decisions on what access model to use when accessing the data.

Finally, the lowest level of the diagram represents the fundamental limitations of the underlying hardware, data store, and Jvm (efficiency of JIT compil-

ers, garbage collection, etc.). These are areas that we were not able to significantly change. The SanFrancisco performance team has been working with Jvm and garbage collection providers to improve the Jvm performance for applications. This effort is beginning to bear fruit with the improved garbage collection behavior of the IBM Developer Kit for OS/400* (Operating System/400*), Java Technology Edition, Version 1.1.7.

**One of the first steps in improving the performance of a program is to identify where the time is being spent.**

### Solutions

One of the first steps in improving the performance of a program is to identify where the time is being spent. A good set of tools can help this process enormously. When we first began the SanFrancisco project, there were really only two tools available.

The first, and crudest, method of timing a piece of code is to bracket the code with calls to System.currentTimeMillis( ). This has the advantage of not interfering with the bracketed code. However, this requires that the tester either know what specific code may be causing performance inefficiencies or go through a tedious process of gradually narrowing the scope of the bracketed code.

The second tool available to us was the profiling capability of the Jvm itself. To invoke this profiling, the Java program is called with java_g -prof. By default, this produces a file called java.prof. (The file can be named something other than the default by specifying java_g -prof: name.) The profile file is basically a view of the amount of time spent in all of the methods a program is calling. The raw file is fairly hard to work with, but there are various tools that make this much simpler. Unfortunately, new versions of the Jvm often cause these tools to fail.

While not specific to Java, both the Windows NT** and OS/400 systems provide various tools for measuring the performance of code running on them. On Windows NT, the performance monitor and task manager tools can be used to measure different aspects of the system, such as CPU and disk utilization. On OS/400, the performance explorer (PEX) tool can be used.

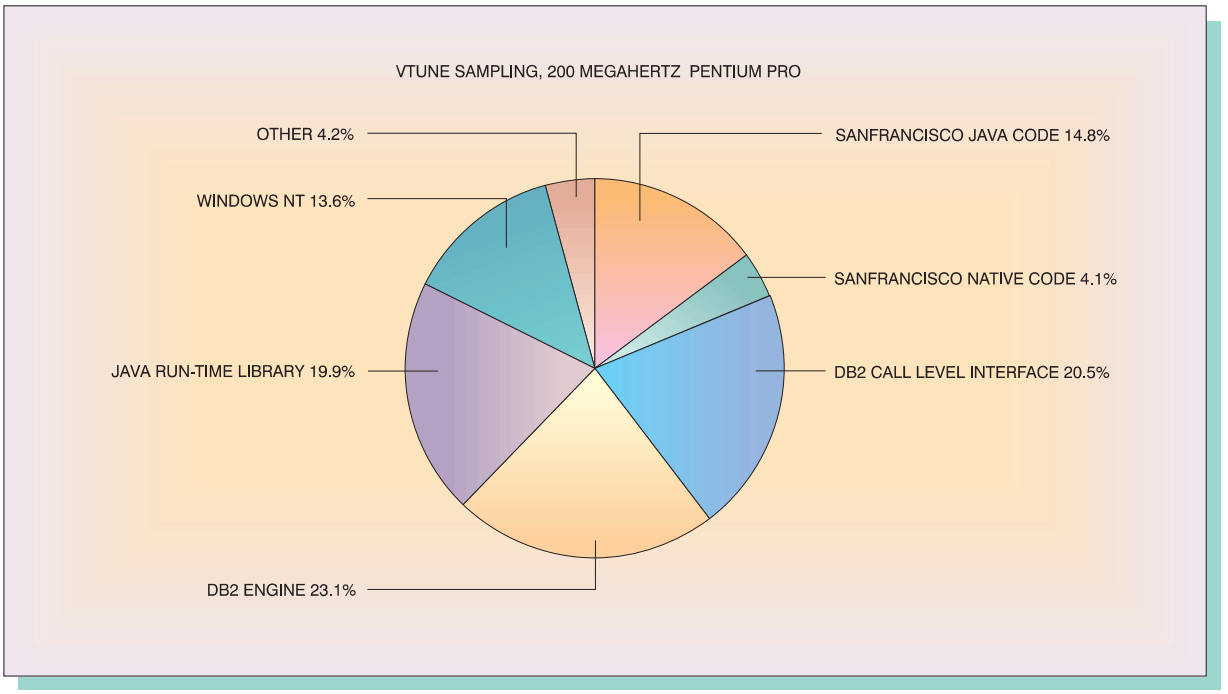
After several years of Java usage, third-party performance tools started to become available. Two tools that we found to be useful were JProbe** (http://www.klgroup.com) and OptimizeIt!** (http://www.optimizeit.com). These tools provide similar functionality, profiling the run-time activity of Java code. Both are fairly simple to use with nice graphical representations of time and memory usage.

Another useful performance tool is Intel's VTune** (http://developer.intel.com/vtune/index.htm), which can be used if the code is running on an Intel microprocessor-based computer. The advantage of this tool over the Java-specific profiling programs is that it shows everything that is using the CPU in the system, not just the Java code (see Figure 3). This gives the relative amount of time spent in Java code vs database code, for example. This can be very useful in showing where to start devoting performance resources. If only 5 percent of the time is spent in a particular piece of code, then no amount of optimization of that piece will give more than a 5 percent performance gain.

**SanFrancisco-specific tools.** In addition to figuring out where the raw execution time is being spent, it is important to identify where there may be bottlenecks in the code. Since SanFrancisco maintains its own object cache and object locking strategies, these were obvious places to look. To aid in this we developed a set of SanFrancisco-specific tools that would show us the caching statistics and an analysis of locking behavior. The lock tool can be used to determine if there are high-level objects that are being locked inappropriately and are thus producing bottlenecks. Once these objects are identified, it may be possible either to change the lock or to decompose the objects into smaller logical pieces that allow for more parallelism.

In SanFrancisco, object queries can be performed on collections of objects. When the object query is performed, the SanFrancisco infrastructure determines whether it is able to directly "push down" the query to the underlying data store or whether it has to perform the query itself by activating objects. We call the second form of query a "live object" query. The push-down form of the query is much faster and

**Figure 3 Total contribution of Java technology to SanFrancisco performance costs**




more desirable. To help determine what queries are being done and whether they are of the live object or query push-down form, we developed a "query trace" tool. This tool and several other SanFrancisco performance tools are available with the SanFrancisco product or from our Web site.[3]

**Programmer and ISV education.** As we started solving the performance problems, we realized that for us to be successful, the development team needed to be educated in the importance of performance. After our initial beta release, we began a series of Java and SanFrancisco performance education sessions. Since all of the developers on the SanFrancisco project were initially C++ programmers, one key part of our education was to identify the Java performance pitfalls that C++ programmers encounter. Among these pitfalls are the excessive use of strings, which in the Java language are immutable objects, and excessive use of exceptions, which are more expensive in Java code. In addition to the Java pitfalls, the SanFrancisco programming model has its own performance pitfalls. For each of the pitfalls, we identified one or more alternatives that exhibited better performance. We repeated this education on a frequent basis to remind the development team of the importance of performance and to present new discoveries. To supplement the education, we developed a set of performance tips and techniques that were published on the IBM intranet. These tips covered Java and SanFrancisco coding tips, as well as system, database, and SanFrancisco configuration tips.

As the project progressed, and the size and complexity of the product increased, the SanFrancisco performance team determined that it needed the assistance of key developers on each of the development teams. These key developers became "performance ambassadors," serving as liaisons between the performance and development teams. The ambassadors were given the responsibility of achieving performance goals. To make the ambassadors more effective, the performance team developed more intensive performance education for them.

Since the SanFrancisco product is intended to be extended by ISVs, we also concluded that the education materials developed internally were fundamentally needed for the ISVs and the SanFrancisco technical consultants. To achieve this goal, the ed-

ucation materials were converted into a course and presentations for our customers. Additionally, the performance tips and techniques were made available to the customers. Since new performance tips are constantly being discovered, we decided to make this information public via the Internet, updating it frequently. Also, the performance tips and external presentations were expanded to contain more detailed information.

Even with the performance information made available to the customers, we felt that each customer would have unique problems. To assist the customers in identifying their own performance problems, a SanFrancisco "porting and tuning" lab was established. This is located in Rochester, Minnesota, and consists of a large number of server and client systems running SanFrancisco and staffed by technical consultants, assisted by the SanFrancisco development team.

**Jvm infrastructure.** The Jvm infrastructure (Jvm, JIT compiler, and run-time environment) contributes to the success or failure of a Java application server. The run-time environment of an application server is very different from the client (Web content) environment where Java normally runs. Application servers run for many hours, whereas client applications live for only minutes. Application servers are shared by many different users, which implies more threads, more classes loaded, and larger object heaps than usually encountered in client applications.

Originally the Java language was controversial because it was interpreted. This is no longer a concern due to availability of JIT and more traditional compilers for Java. Current JIT compilers are very effective and can meet or exceed the run-time performance of compiled Java code. While we are expecting additional performance improvements from compiler technology, this is no longer a major concern. We should also keep in perspective the fact that the Java technology is only part of the total performance profile of a complete application server.

From Figure 3 we see that for a well-tuned workload, the direct contribution of Java technology is less than 20 percent. Most of the total CPU load is in the database (DBMS) or the operating system. Since 80 percent of the CPU load is in the DBMS (here DATABASE 2* [DB2*]) and operating system, the efficiency of how the application (and object infrastructure) uses the DBMS is more important than the language it is written in. Of course this assumes that developers use language features appropriately.

**Garbage collection issues.** The more important issue with Java technology is its fundamental object-oriented nature and garbage collection-based storage management. All objects are dynamically allocated from a heap, where garbage collection is automatically done either when the heap is exhausted or based on a timer. Since Java String instances are immutable, this results in radically different performance characteristics from strings in other languages. The same is true for arrays and structures, which are allocated dynamically.

Application servers differ from clients in heap usage. Servers cache persistent shared objects to improve throughput and response time. This means the Java heap contains significantly more long-lived objects than are found for client applications.

The large object caches (and large Java heaps) required for SanFrancisco application servers significantly increase the time required for the garbage collector (GC). With Java object heaps of 100–500 megabytes (MB), GC times of 5–20 seconds are common (twice that if heap compaction is required). Most current Jvms use a simple "mark-and-sweep" GC algorithm that suspends all user threads while it is running. This creates noticeable pauses for all clients attached to the server, affecting both throughput and response times.

The heuristics used to drive the garbage collector are also important. Heuristics are the rules for deciding:

- When to expand the heap
- When to trigger heap compaction
- When to force finalization
- When to clear weak references (and how many to clear)

A Jvm with inappropriate heuristics can cause serious performance problems. For example, the GC shipped with the Sun Java Development Kit (JDK**), v 1.1.7 has a heuristic that performs multiple GC passes through the object heap whenever the free heap is below 25 percent of the total size. For a Jvm running applets in a 4 MB heap, this may be appropriate. However, for SanFrancisco's use of the Java heap, these heuristics are not particularly effective. A multiuser application server running in a 256 MB

heap will be stopped cold for minutes. We call this "garbage collector hell."

Fortunately we are seeing improvements in the garbage collectors and heuristics. The IBM Developer Kit for Windows, Java Technology Edition, has significantly improved the performance of the garbage collector and heuristics. Jvms in the IBM Developer Kits for OS/400 and OS/390* (Operating System/390) implement a concurrent GC that can run in parallel with user threads. This requires the addition of a "write barrier" so that the GC is aware of all object reference updates. This causes a slight increase in run-time overhead, but the user threads run without pause. This "pauseless" behavior is important to application servers since it maintains consistently good response times at higher CPU loads.

**Threading and synchronization.** An important feature of Java is the built-in threading and synchronization support. Thread safety dictates that general utility classes be synchronized to ensure correct operation. For example, objects of StringBuffer, Hashtable, and Vector classes are synchronized for most methods, but we found that most usage did not require synchronization.

Specifically, objects of the StringBuffer class are created implicitly (by the Java compiler) for string concatenations. As these StringBuffer instances are temporary and local to the invocation, they are never shared and do not require synchronization. In this case the only recourse is for the programmer to limit String usage and use StringBuffer instances explicitly for concatenation and formatting.

As another example, the SanFrancisco infrastructure uses various hash tables internally, but we found that most of the hash tables are part of a larger structure that was already synchronized. So in this case using objects of the synchronized Hashtable class was unnecessary. For SanFrancisco we developed our own, unsynchronized hash tables specific to our needs.

The SanFrancisco program model provides a generalized locking mechanism integral with transactional support. This locking mechanism obviates the need for application programmers to do explicit synchronization in their implementation. In fact the program model documentation clearly discourages the use of synchronization for application code. This minimizes the use of synchronization in general and isolates synchronization and monitor usage to the SanFrancisco foundation layer and the Jvm itself.

However, not all synchronization problems can be resolved at the application level. The Jvm uses monitors internally to synchronize class lookup, heap allocations, etc. Also, the core Java libraries java.lang.*, java.io.*, and java.net.* are heavily synchronized and cannot be practically avoided. Fortunately, over the evolution of Sun's JDK v 1.1.x we have seen improvements in both the implementation of monitors and avoidance of contention within the Jvm. For example, IBM Jvms have optimized the monitors for the uncontended case.[4] The internal run-time structures of the Jvms have also improved to limit contention in high thread-usage cases: for example, improved class look-up algorithms and thread local subheaps.

Each thread requires memory for the thread itself and for both Java and C stacks. The defaults for Sun's JDK 1.1 are 100 kilobytes (KB) for the C stack and 400 KB for the Java stack. This is 0.5 MB per thread, which is allocated from the C heap of the Jvm instance. For 10 to 15 threads this is not an issue, but for a Java application server supporting hundreds of clients, this can add up to a significant memory requirement. Remote method invocation (RMI) requires several threads per attached client, so 200 to 1000 threads would be nominal for a large Java application server. This is a significant virtual memory requirement (100–500 MB). This is also a significant real memory requirement, because the stacks must be scanned by the garbage collector and, as mentioned earlier, it is critical to avoid paging in a "stop-the-world" garbage collector.

Both the JDK 1.1 and the Java 2 Software Development Kit, Standard Edition, v 1.2.2 support command line options -ss and -oss to change the default C and Java stack sizes. Unfortunately, these parameters apply to all threads in the Jvm instance, so the settings must support the maximum requirement. It would be useful for application servers to be able to set the stack sizes at thread create time. The only recourse at this time is to manage the total number of threads and attempt to run the Jvm with the minimum stack size that will support the application.

**SanFrancisco code path fixes.** When talking about performance, path length is an important concept. The path length of a task, or a business process, is the number of instructions that are executed in the CPU and the I/O subsystem in order to complete a business process or a set of transactions. Obviously, the goal is to keep the path length as short as possible, and it can be achieved by using programming techniques such as object caching or object reuse.

High-use paths need to be identified, analyzed, and optimized. Actual steps in this process are:

- Implementing a benchmark application that is representative of "real life" applications
- Identifying benchmark application "hot spots" and developing solutions for them
- Prototyping and measuring significant performance changes
- Performing code inspections with the help of a list of known performance problems

This process was applied to the SanFrancisco framework and the findings in both framework code and benchmarks are applicable to SanFrancisco-based applications as well. Some apply to distributed applications in general. The most significant of those findings are presented here.

*Communication.* Communication, i.e., data transfer between client and server, or between two servers, is very expensive. Therefore the number of communication requests as well as the actual amount of data transferred should be reduced as much as possible.

Basically, there are three choices in working with objects in a distributed environment:

- The remote object can be transferred to the client's address space so that method calls against it run locally.
- The remote object can remain at its home location; all method calls against it will be remote calls.
- The business process function can be shipped to the remote object's home location where it runs locally (see subsection on SanFrancisco commands).

Making the right choice is critical for performance, yet the best choice in a specific scenario depends on multiple factors, which are discussed later.

In SanFrancisco most objects represent shared, persistent data and operations. These will be referred to as business objects. It is important to know where a business object to be processed is located. In SanFrancisco this information is encapsulated by instances of the Handle class: every shared business object has a unique identity within the logical SanFrancisco network. A Handle instance encapsulates this unique identity and can be stored for later access to the business object.

How persistent business objects are accessed has a large impact on performance. In SanFrancisco this is controlled by instances of the AccessMode class, which are used to determine the access characteristics. One access characteristic is the access location. The two options for the location are (see Figure 4):

- Local—The business object is transferred from its owning server process to the requester's process. Subsequent method calls are local calls and are therefore fast. If the business object is updated, it will be transferred back to its server process when the transaction is committed.
- Home—The business object remains in its server process. A proxy object is created in the requesting process. Subsequent method calls are remote method calls and will go through the local proxy to the remote business object.

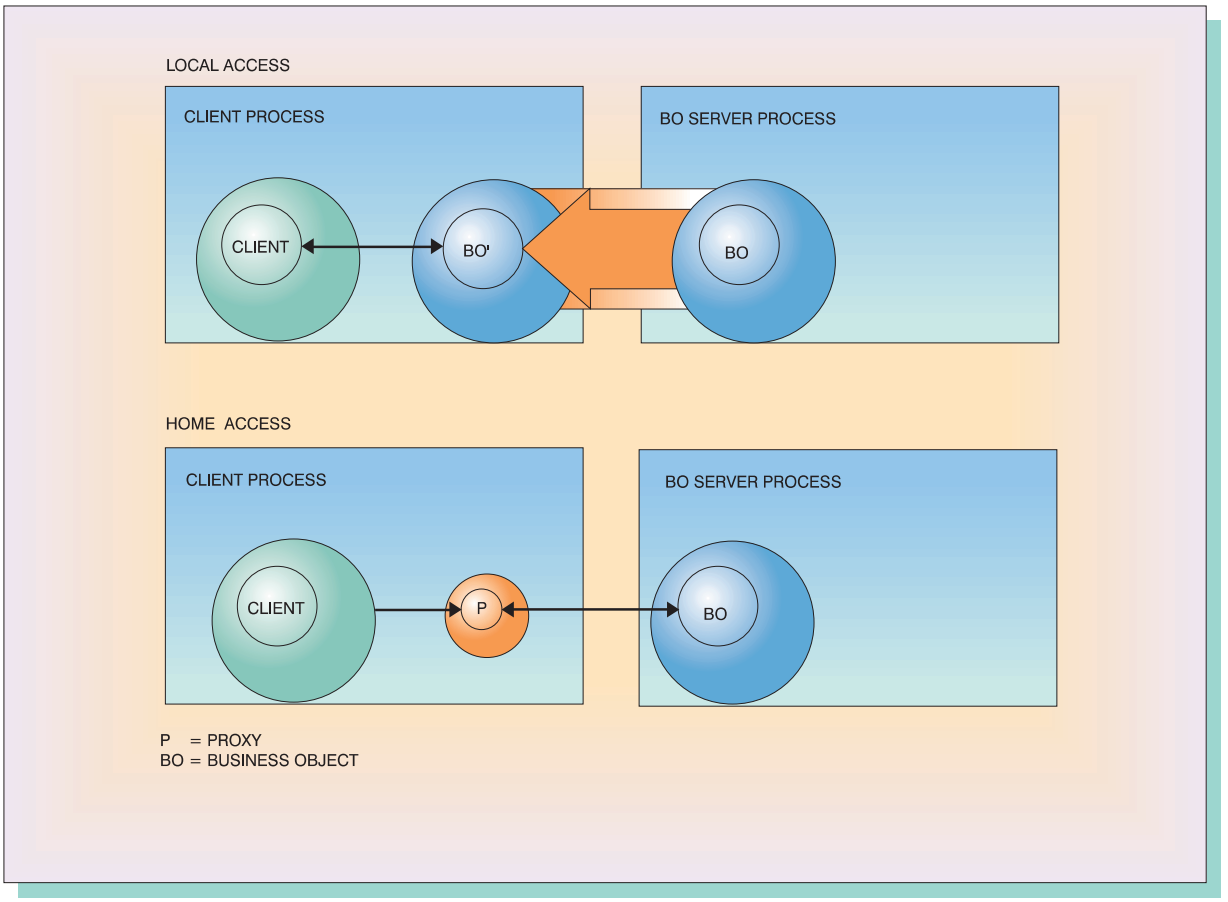
The choice between local and home is a trade-off between the overhead of transferring the business object back and forth between client and server (local), and the cost of having each method call flow back and forth (home). It depends on a number of factors:

- The size of the object
- The number of method calls
- The data flow associated with those method calls
- The communication link overhead
- The computing power of the server vs the computing power of the client

To make the proper choice, these different factors and how they interrelate must be well understood. It is not a trivial task, but the SanFrancisco performance team has found very few situations where local access is advantageous. The SanFrancisco default is to access remote objects at home, which is generally a good choice for an application.

As an alternative to remote method calls, SanFrancisco provides instances of the Command class. A command bundles a number of method calls, against one or more objects, that can be performed at a designated target location. The set of method calls implements a (business) task. The target location can be the local process or a remote server process, and it is specified as the handle of the business object that is the primary target of the command. When a command is being performed, a single (remote) method call triggers the execution of the task. Nec-

**Figure 4 Local vs home access**




essary input and resulting output data are passed as part of the state of the Command instance.

The following list summarizes the rules of thumb that the SanFrancisco performance team has established for choosing the right business object access:

- If there are very few requests (method calls), use home access.
- If there are many requests, and the object transfer cost is greater than the command transfer cost, use a command executing at the home location.
- If there are many requests and the object transfer cost is low, use local access.

*Caching.* Caching is especially important in a distributed environment like SanFrancisco, where each method call is potentially a remote method call or invokes remote methods internally. Frequently and repetitively used information should be cached in the local process space instead of being retrieved over the network or from a persistent data store over and over again as needed. Several types of caching exist:

- *Intramethod caching.* If the result of a method call will not change and will be used more than once, the result of the initial call should be cached in a local variable for later usage. In Java code, a local method call is at least three times more expensive than referencing a local variable. Caching method results is particularly important if the method will retrieve a business object.

- *Intermethod caching.* Another common coding problem, which is aggravated in a distributed environment, is the situation where two or more methods in a series of calls need to access the same data. In this case, the method that first accesses the com-

mon data should store it and pass it on to subsequent methods. This technique may require interface changes, i.e., additional parameters to pass the data.

- *Process/distributed process scoped caching.* Another opportunity for caching involves global data. These data can be cached within the scope of a process (in a static variable), or, in SanFrancisco, within the scope of a distributed process.

  A distributed process is like a normal process in that it is an anchor point for one or more threads that are actively working under it. The threads of a distributed process can be spread across one or more Jvms. The distributed process has a context, or shared information space, that each thread in the distributed process can access and modify. This context is an instance of the DistributedProcessContext (DPC) class. Changes made to the DPC object by one thread are available immediately to other threads in the distributed process.

*Object selection.* Often developers have a choice among different types of objects to accomplish a particular task. Selecting a type of the right "weight" is a critical issue for good performance. The most primitive choice meeting the requirements is generally the best choice. Choosing a type that is "heavier" than needed typically implies increased path length and storage requirements. If the appropriate type is not available, it should be created. Using an inappropriate type just because it is the only one available may cause performance problems that are difficult to correct.

*Object reuse.* The creation of a simple Java object is a relatively expensive operation in terms of system resources, and the creation of a SanFrancisco object even more so. Thus, creating unnecessary objects should be avoided, especially if the object's construction involves complex operations or the object is composed of many other objects. Reusing objects once they are no longer needed should always be considered. Object reuse not only saves object creation costs; it also saves garbage collection costs. The downside of reusing objects is that it may create contention points and may impede garbage collection if too many objects are reused.

Objects available for reuse can be collected in a reuse pool by type. They may also provide a reinitialize method that corresponds to the constructor for that type.

In some cases an object can be created as a copy of an existing one instead of being created "from scratch." The state of the new object may need to be updated, unless the new object is intended to be a clone of the original one. Copying will save some of the initialization cost, but the performance benefit may vary considerably depending on the complexity of creation logic and the amount of state that differs between the two instances.

*Exceptions.* Java exceptions and exception handling provide an elegant way to deal with exceptional conditions. With regard to performance, this support is not free. It costs to create an exception object, it costs to "throw" it and to "catch" it, and eventually it costs to reclaim its space through garbage collection. Therefore, exceptions should be reserved for very rare, truly abnormal conditions. In cases that are considered normal, return values should be used instead.

*Object persistence changes.* One of SanFrancisco's strengths is the ability to store persistent objects in an RDB. As with I/O in any product, efficiently storing these data to disk is critical to performance.

The process of transforming objects to rows in a database is known as *schema mapping*. The mapping information is provided by the application developer, giving the SanFrancisco schema mapping facility the knowledge of how to map complex object attributes to columns in a database. Without this information, the schema mapping facility will not always make optimal decisions regarding this mapping.

These decisions can even result in having to store the persistent object in a self-defining data stream. Of course, the more data written to the database, the poorer performance will be. Because of this, we have applied resources to ensure that objects can be optimally mapped to the database, with a primary focus on reducing the size of the data to be stored. Among the critical objects we have focused on are those of the DDecimal and DTime classes, both of which have specific counterparts in databases. Where persistent state is critical, we have moved away from complex Java classes, such as BigDecimal and GregorianCalendar, to primitive data types such as long floating point and integer.

Another critical part of SanFrancisco's persistence support is its object cache. Since I/O can cause performance problems, we wanted to avoid database access whenever possible. Because most RDBs do not "understand" objects, we could not rely on the da-

tabase caching capabilities. So SanFrancisco added an object cache that contains the most recently accessed objects. This object cache minimizes the number of database reads that a SanFrancisco application will perform.

One of the primary strengths of databases is powerful query functionality. For object-oriented programming, it is natural to have a query syntax that is based on objects and object method calls, which is unnatural for RDBs. To provide object-oriented developers with an object-oriented query function that performs and scales well, SanFrancisco provides a mechanism to perform object queries in an RDB. This function, *query pushdown*, allows the SanFrancisco applications to take advantage of the powerful database query capabilities. We located queries occurring in the SanFrancisco code, changing the query or configuring the schema mapping for the object being queried to ensure that the query occurred in the database for optimal performance.

*Scaling improvements.* Scalability is a complex topic covering all facets of running a SanFrancisco application. These facets include:

- The application itself
- The SanFrancisco towers and CBOs
- The SanFrancisco foundation layer
- The supporting software products, including the Jvm and RDB
- The supporting system platform, hardware, and operating system

All of these pieces interact in complex ways to effect scalability.

A scalable implementation can support a wide range of workloads. The workload is defined in terms of: number of concurrent users, throughput (e.g., business transactions per hour), and response time criteria (e.g., 90 percent of client requests complete in one second). Scalability is defined by the range, from the smallest configuration that will support the application, to the configuration that supports the highest throughput (or user count) without wasting resources.

This is all very much application-dependent. A simple application programmed directly to the SanFrancisco foundation layer will require fewer resources than a complex application making full use of features provided by one or more SanFrancisco towers. The scalability of an application tends to be inversely proportional to the complexity of the logic and the complexity of the data.

Data complexity contributes to scalability in multiple areas: the cost of schema mapping, the cost of object streaming, and the additional bandwidth required to read and write extra (meta) data. Techniques for managing code complexity and path length have been described. In addition, SanFrancisco provides extended schema mapping (ESM) tools for improving the schema mapping. Improved schema mapping can improve efficiency of data transformations and reduce the total data volume transferred to and from the RDB.

Another important scalability issue is the interaction between the path length and the application's lock strategy. To avoid deadlocks the application must develop and adhere to a totally linear lock order. A lock ordering strategy assigns each key resource (instance, class, or collection) in the application to a slot in the lock-order list. Each business transaction will follow this lock ordering for all the resources it uses. Observing a consistent lock ordering across the application avoids deadlocks, but may increase the duration of locks held for resources high on the list. This lock strategy affects the number, strength, and duration of locks. Excessive locking can severely limit throughput and scalability.

In SanFrancisco, locks are granted by the getEntity( ) method on the Factory class. SanFrancisco uses access modes to control lock strength and scope. Since AccessMode is a class, instances can be defined and used at any level of the application or framework. Access modes can also be passed as parameters to "get" methods so the returned object is locked at the appropriate strength.

Managing CPU and memory resources is an important topic for scalability. This applies for any application server, independent of language. However, the Java language can magnify some resource requirements. Dynamic memory allocation and garbage collection seems to increase the memory working set. Supporting persistent objects also inflates the record size and bandwidth required from the RDB and disk storage. SanFrancisco is perhaps an extreme example of this, since it assumes full polymorphic behavior from persistent objects, and this requires additional storage for the meta-data.

*Jvm scalability.* Complications include the interaction between the Jvm's garbage collector, available

real memory, and multiprocessor configurations. Garbage collector technology covers a wide range, from "stop-the-world" to concurrent.

In overcommitted memory situations, a "stop-the-world" garbage collector exhibits poor throughput and response time characteristics if the Jvm's object heap pages more than trivially. "Stop-the-world" garbage collectors also exhibit poor utilization of multiprocessor configurations. More sophisticated garbage collectors exhibit greater tolerance of paging and better utilization of multiprocessors.

For JDK v 1.1.x Jvms, it is critically important that the application server process (or processes) does not page. So an important tuning activity is adjusting the available tuning parameters so that paging is minimized. If the RDB server is running on the same processor as the Java server, it is important to constrain the RDB buffer allocation to avoid competing with the Java server for real memory pages. A good rule of thumb is:

- One third for application server heap space (SanFrancisco object cache)
- One third for RDB buffers
- One third for the operating system, file caches, runtime environment, etc.

Once the SanFrancisco server heap size is established, the container object cache sizes need to be adjusted. A good rule of thumb is to adjust the container cache threshold so that the Java heap is 60 to 70 percent utilized (30–40 percent free space) for steady-state operation. The heap space for each SanFrancisco entity (Entity is the class from which all persistent business objects inherit) is application-dependent and can be derived empirically by running the application server with Java parameter -verbosegc and observing the running application.

Depending on the Jvm and platform, there are practical limits to how large (the size of the heap and number of objects cached) a SanFrancisco server can be. For development using "stop-the-world" GCs, the practical limit seems to be 128–256 MB and 100 000–150 000 Entity instances cached. Beyond these empirically derived values, the GC overhead of scanning the large heap offsets any advantage of the larger entry cache. This effect is magnified on SMP (symmetrical multiprocessing) configurations for "stop-the-world" garbage collectors. Jvms with concurrent GC support larger Java heaps and entity caches. For example, on the AS/400* (Application System/400*), entity caches up to 400 000 have proven effective.

The total number of threads required to support a given number of attached clients can also become a limit. Each thread requires memory for thread control objects and stack space. How this is managed varies from Jvm to Jvm, but most implementations support command line options (-ss and -oss) to change the default storage allocation. However, the stack requirement is application-dependent and can only be determined empirically (the smallest stack size that does not throw the StackOverflowError exception!).

For large application servers it is better to manage the total thread count by multiplexing a large number of client connections over a smaller pool of threads. This is complicated by requirements enforced by most RDBs that transactions be bound to a specific thread. So once a client initiates a transaction, the server thread is bound to that client until the transaction completes (commits or is rolled back). Statistically only a subset of clients have an active transaction at any given time, so the server's thread pool needs to be large enough to cover the average number of "in-flight" transactions, plus a bit more. The total should be sufficient so that some percentage (90–95 percent) of the transactions can complete within a reasonable response time.

SanFrancisco servers support "thread throttling," which allows the administrator to control the size and behavior of the server's thread pool. In most cases setting the thread limit to twice the average number of "in-flight" transactions is adequate. This handles the steady-state load while controlling and smoothing out the peak loads that often occur after a GC event.

*Multiserver configurations.* If the application is partitionable, dividing the work across multiple SanFrancisco processes (separate processes, each with its own Java heap) can be an effective "work-around" solution for these problems. It is important that most object interactions are local (in the same server process). Objects with strong interactions must be placed on the same server, while unrelated objects and weakly interacting objects can be placed in separate servers. SanFrancisco provides the Container and AccessMode class mechanisms to support and manage partitioning. SanFrancisco's access mode mechanism strikes an effective balance between control and transparency.

Each SanFrancisco container represents a separate database instance and is the unit of distribution in SanFrancisco. Spreading an application's data across two or more containers is a prerequisite for multiple server partitioning. Once the container partitioning is established, assignment (or reassignment) of containers to server processes is easily configured. The LocationHandle class allows the application to ensure that new objects are created within the correct container. Since any object handle is also a location handle, the application can create new objects "near" (in the same container as) an existing object. This allows the application to correctly place the object within a container, without being concerned with the specific server and container configuration.

The location handle is also used to direct SanFrancisco commands to the appropriate server (that manages the container for the target object) for processing. Again, this allows clients to direct requests to the correct server process without knowing the specifics of the server/container configuration.

This optimizes the performance of each server while increasing the total system capacity. Of course, this complicates system tuning. While multiple server processes mitigate some of the effects of a "stop-the-world" garbage collector, it is still critical that the sum of all Java heaps (and the RDB data buffers) fit into the available real memory.

There are also practical limits to the number of servers an application or system configuration can use effectively. Each Jvm process must replicate its runtime data structures and the loaded classes required by the application. So two SanFrancisco processes with 128 MB heaps will require 16 to 32 MB more storage than a single process with a 256 MB heap.

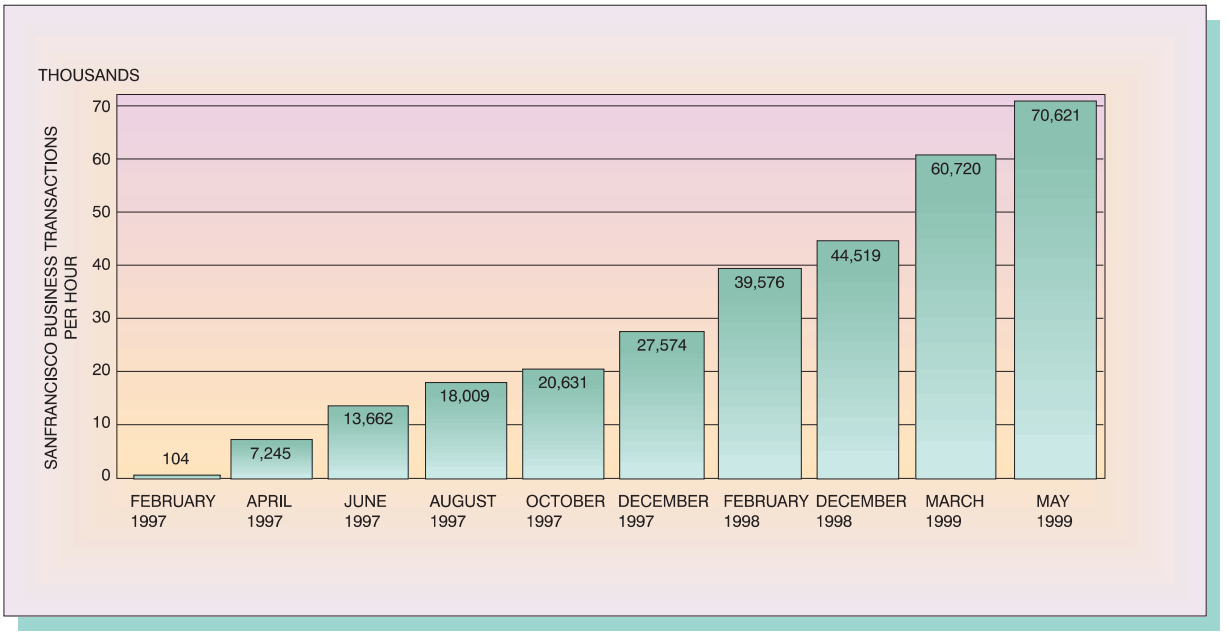
These are all empirically derived results from SanFrancisco's application benchmark (general business object benchmark [GBOB]) and any given application must use these values as guidelines for its own tuning. Much of this is simply working around the inherent limitation of the current development kits. While we have had some success with this, we still need to push for improvements in Jvm scalability. It is simply easier to manage (and tune) one server process than many, and easier to tune a single system (node) than a distributed system.

### Performance results

Performance has always been a key concern for the SanFrancisco team. From the earliest days when the transition to Java was just beginning, we have been tracking performance. Figure 5 shows one of the foundation benchmarks (a transaction-processing benchmark similar to TPC-C) and the performance improvement from the very first alpha release through SanFrancisco v 1.4.0, which is the second deployment release. The chart shows the number of business transactions per hour achieved by a benchmark (single client) written on the foundation layer of SanFrancisco running a 200 megahertz Pentium** processor with 1 gigabyte of main memory on a database of over 60 000 persistent objects. It shows the dramatic improvement from the earliest days, when a single transaction took over 30 seconds to complete, to today, where the same hardware can process nearly 20 transactions per second. This represents an improvement of over 500 times! Of course, the large factors of improvement early in this timeframe (from February 1997 through February 1998) were mostly due to rapid improvements in Java technology (for example, the introduction of JIT compilation) and the early SanFrancisco program model (for example, the use of commands for function shipping). However, the latter part of the chart shows performance almost doubling after both Java and SanFrancisco technology had already significantly matured.

The March 1999 (1.4.0) release of SanFrancisco is particularly noteworthy in that improving performance was the primary focus of the release, and much of the methodology mentioned earlier was included here. Besides the existing foundation benchmarks, additional benchmarks were developed for the towers code, specifically for the general ledger and order management towers. These benchmarks were tracked across the entire 1.4.0 release on a weekly basis. Figure 6 shows the improvement in the transaction rates for the key benchmarks of this release. These benchmarks all measured multiclient workloads on large (hundreds of thousands to millions of persistent object) databases, and we believe they are typical of the performance of deployed SanFrancisco applications. The foundation benchmark is based on a derivation of the TPC Benchmark** (though for legal and technical reasons it cannot be directly compared) that involves five types of transactions typical of an OLTP (on-line transaction processing) business workload. The benchmark is built using only classes from the SanFrancisco foundation layer. The general ledger benchmark is similar in structure to the foundation benchmark, but uses business objects from the general ledger tower. The order management benchmark models a typical sales-

**Figure 5** Foundation performance over time. Note that the chart is not in scale chronologically.




and-distribution (SD) application and mimics the SAP R/3** SD Benchmark (although not rigorously enough to invite comparisons). The specific numbers shown are less important than the relative improvement from release 1.3.0 to 1.4.0. Note that the greatest relative improvement (about 14 times) was in the (relatively new) order management tower and the least improvement (about 2 times) was in the (relatively mature) foundation with the (moderately mature) general ledger tower (with about 3 times improvement) in the middle. This matches our expectations that newer code (with the most "low hanging fruit") will show the most dramatic performance improvement with more modest—but still significant—improvements reserved for code that is at a later stage of maturity.

## Future directions

SanFrancisco has made tremendous strides in battling some of the performance problems inherent in object-oriented frameworks and has proven that commercial applications built from framework components written in the Java language can indeed achieve satisfactory performance. However the war is not yet won. There are still significant challenges to overcome and the solutions are not yet in hand. Some of the most significant obstacles and possible approaches to overcoming them are:

- *Simplifying the persistence of Java objects.* The SanFrancisco solution of platform- and database-specific C++ code to do this job is too expensive to maintain across the many databases and platforms SanFrancisco will eventually support. Either JDBC support must be improved (i.e., with two-phase commit transaction support) so that the native C++ interfaces are no longer necessary, or an object-oriented database engine (written in the Java language) that interfaces to the Jvm needs to be developed. In addition, more intelligent tools are needed for automatically mapping the object data to those databases; the current approach imposes too much of a burden on programmers and administrators. Obviously the trick is to do all this without sacrificing performance.
- *Automatic configuration.* The number of tuning parameters and configuration options gets much larger and more difficult to get right given the added complexity of SanFrancisco and the Jvm. It should be possible to develop a configurator that automatically provides near-optimal configuration parameters, either through explicit knowledge of the system or through learning and experience.

- *Improved Java technology and tooling.* Although Sun has made tremendous progress over the past couple of years in terms of JIT technology and Java implementation efficiencies, there is still a way to go. In particular, we are looking forward to having a nonblocking and generational garbage collector available for Jvms on all platforms. We would also like to see a standard set of instrumentation interfaces as part of the Java specification so that third-party tool vendors can develop a consistent set of powerful tools for analyzing complex (e.g., SanFrancisco-based) Java applications that will run on Jvms for all platforms. The JVMPI interface for Java 2 and later implementations is a good start in this direction.

Currently there are cross-industry efforts to address all the problems mentioned above. Some of the solutions are already being developed. Others will take longer. However, as the SanFrancisco project has proved, concerted effort applied to these problems will eventually bear much fruit; the future of Java technology and object-oriented frameworks depends on it.

## Acknowledgments

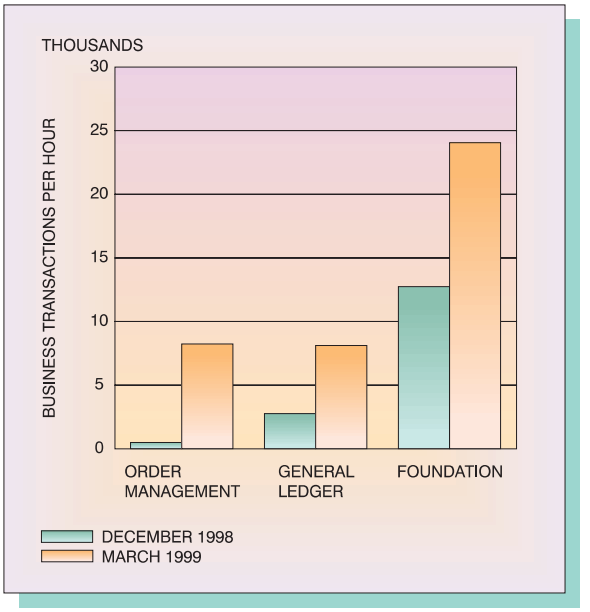
We thank Alan Stephens and Robert Berry for their helpful comments in reviewing this paper. Also, special thanks are due to the SanFrancisco development team, especially the "performance ambassadors": Christopher Abbey, Catherine Andresen, Jim Carey, Dan Dahl, Mary Dangler, Tim Graser, Dave Gross, Jay Johnson, Ken Lawrence, Michael Payne, Jon Peterson, Dianne Richards, Derek Studanski, and Andre Tost.



## Cited references and notes

1. Transaction Processing Performance Council's (TPC) Benchmark C Standard Specification, Revision 3.3 (April 8, 1997).
2. S. J. Baylor, M. Devarakonda, S. Fink, E. Gluzberg, M. Kalantar, P. Muttineni, E. Barsness, R. Arora, R. Dimpsey, and S. J. Munroe, "Java Server Benchmarks," *IBM Systems Journal* **39**, No. 1, 57–81 (2000, this issue).
3. Tools that work with or are specific to SanFrancisco are described at: http://www.software.ibm.com/ad/sanfrancisco/tools.html.
4. D. Bacon, R. Konuru, C. Murthy, and M. Serrano, "Thin Locks: Featherweight Synchronization for Java," *ACM Conference on Programming Language Design and Implementation*, Montreal, Canada (June 17–19, 1998).

**Figure 6** SanFrancisco performance 1.3.0 (December 1998) vs 1.4.0 (March 1999)




*Accepted for publication September 30, 1999.*

**Ralph Christ** *IBM Software Group, Rochester Laboratory, 3605 Highway 52 North, Rochester, Minnesota 55901 (electronic mail: christ@us.ibm.com).* Mr. Christ is a senior software engineer and has been the manager of the SanFrancisco Architecture, Plans, and Performance team from its inception in March of 1995. He has held numerous key positions working on the development of the System/38™ and AS/400 operating systems. For the past ten years he has been involved in a variety of object-oriented development projects, ranging from applications to operating systems. In early 1995, he was a key proponent and advocate within IBM for the move to Java technology. In May of 1999 he moved to the WebSphere team to help with project plans and strategy.

**Steven L. Halter** *IBM Software Group, Rochester Laboratory, 3605 Highway 52 North, Rochester, Minnesota 55901 (electronic mail: shalter@us.ibm.com).* Mr. Halter is a staff software engineer and was a member of the SanFrancisco performance group at the time this paper was written. He has worked in the area of object persistence and object infrastructures, an area in which he holds two United States patents. He received B.S. and M.S. degrees in computer science from Iowa State University.

**Kenton Lynne** *IBM Software Group, Rochester Laboratory, 3605 Highway 52 North, Rochester, Minnesota 55901 (electronic mail: klynne@us.ibm.com).* Mr. Lynne is an advisory programmer and was the team leader for the SanFrancisco performance group at the time this paper was written. He has also worked in various

system software areas for the IBM System/34, System/36™, and AS/400 products. He holds five United States patents and has a research interest in neural networks and machine learning algorithms. He received a B.S. degree in psychology and an M.S. degree in computer science from the University of Wisconsin, Madison.

**Stephanie Meizer** *IBM Software Group, Rochester Laboratory, 3605 Highway 52 North, Rochester, Minnesota 55901 (electronic mail: smeizer@us.ibm.com).* Ms. Meizer joined IBM Germany in 1994 as a software engineer and is currently on assignment to the SanFrancisco project in Rochester, where she is working in the performance group. Previously, she has been a developer on the VisualAge® for C++ Open Class Library and the SanFrancisco common business objects. She received an M.S. degree in computer science and business administration from the Berufsakademie, Mannheim, Germany, in 1994.

**Steven J. Munroe** *IBM Software Group, Rochester Laboratory, 3605 Highway 52 North, Rochester, Minnesota 55901 (electronic mail: sjmunroe@us.ibm.com).* Mr. Munroe is a senior software engineer and a member of the IBM SanFrancisco performance group. He has also worked on architecture and design in various system software areas for the IBM System/38™ and AS/400 products. He holds nine issued patents. He received a B.S. degree in computer science from Washington State University, Pullman.

**Mark Pasch** *IBM Software Group, Rochester Laboratory, 3605 Highway 52 North, Rochester, Minnesota 55901 (electronic mail: mpasch@us.ibm.com).* Mr. Pasch is an advisory software engineer with the SanFrancisco performance group. He has also worked in various system software areas for the AS/400 products, including a large object-oriented project. He received a B.S. degree in computer science from Purdue University. He has four patents filed.

Reprint Order No. G321-5714.