Selecting and Implementing an Embedded Database System

Although embedded systems share many characteristics with their desktop and server counterparts, the unique advantages, limitations, and requirements of the applications they run demand a careful selection process and tailored implementation.

With the increasing deployment of computers as embedded systems that provide new and interesting services, the computing landscape is much richer and more diverse than it was even a decade ago. Developers can choose from an enormous variety of hardware, operating systems, and tools for the embedded systems they build.

This range of platform and tool choices provides both a blessing and a curse. On one hand, developers can choose precisely the tools and services their applications require. On the other hand, finding the right products to use can be difficult.

Despite their differences, embedded systems share important characteristics with the desktop and server systems they supplant. Because embedded systems interact with users or with their environment, they need an input/output system. Many embedded systems perform multiple tasks and need some sort of operating system for scheduling and task management. Embedded systems also often require sophisticated database management services.

The key strategy in choosing database tools for embedded systems is to focus on the application’s requirements. Embedded database products vary widely from vendor to vendor. Some will do less than a particular application needs; some will do much more. By surveying the choices carefully, you can choose the tool that most closely matches your requirements.

After choosing the operating system, hardware platform, and database software for a new embedded system, you must design a system that runs reliably with little or no human intervention. Unlike desktop and server systems, embedded systems cannot ask for operator help when the application encounters a problem.

Finally, performance matters. Designing for performance up front, and evaluating it once you’ve built the application, is crucial. Fortunately, you can choose from a variety of techniques for evaluating and improving performance in database applications.

To arrive at the best embedded-database-system solution, you must select the product that best matches your specific needs, then integrate that solution with your application. Start this process by evaluating which services your embedded application will provide.

DATABASE SERVICES

Because embedded devices provide a huge variety of services, different embedded systems place different demands on the database systems they use.

At the high end, some embedded systems need the high concurrency and scalable transaction-processing services that established relational database vendors have delivered to desktop and systems developers for years. However, the established enterprise relational databases are seldom a good choice for embedded systems because of platform and packaging differences.

Enterprise relational database engines use well-understood techniques to process transactions. One example of these processing services is write-ahead logging, which keeps track of how a record looks both before and after it is changed so that the change can be backed out or reapplied as necessary. Another example, two-phase locking, keeps values locked until
all changes are complete so that other users cannot modify them in the meantime. Several embedded database products use these techniques to provide scalable, robust transaction-processing services in much smaller packages than an enterprise relational database system.

Some embedded systems do not need full-blown disaster recovery because they only require modest concurrency and data volumes. Others don’t have concurrent users and don’t need transactions, so they only require a high-performance data storage engine.

Database vendors, of course, want to sell their products to as many customers as possible, so they try to address as wide a range of requirements as they can. Many embedded databases are configurable, which allows developers to include or exclude services—such as write-ahead logging and two-phase locking—as a particular application requires.

**DATABASES FOR EMBEDDED SYSTEMS**

The hundreds of embedded operating systems available today run on an amazing variety of processor hardware, which makes choosing a good database engine for an embedded application harder than choosing one for a desktop or server system.

To date, embedded operating systems have made little progress in interface standardization. The EL/Ix standardization effort, sponsored by Red Hat, aims to put a Linux-like Posix 1003.1 interface atop embedded operating systems, but the standard lacks a reference implementation and OS vendor interest. Database developers have an enormous job porting their software to the platforms their potential customers use. As a result, many embedded operating systems simply do not run any commercial-grade database management tool.

The platform story will likely get better as time goes by. Today, a few operating systems dominate the embedded market, including Microsoft’s Windows CE, Wind River Systems’ VxWorks, and QNX’s Neutrino. Although embedded Linux has captured relatively little market share to date, I expect it to do well in the embedded market. Consolidation among operating-systems vendors and pressure to standardize interfaces will produce more and better tool choices for embedded-systems developers.

The market for embedded databases will grow with the market for embedded systems. In stark contrast to the situation in the desktop and server systems market, no market leader dominates the embedded database space. As the opportunity becomes apparent, new and established database companies will invest heavily to roll out new products and establish themselves in this market. In the long term, embedded-systems developers will have a variety of good database tools from which to choose.

Unfortunately, developers rarely have the luxury of operating in the long term. In the short term, embedded developers must evaluate available products carefully to find one that meets their requirements. Yet sorting through the marketing claims of the various companies can be difficult when, for example, different vendors define the terms “embedded systems” and “embedded database” to mean different things.

**Classifying database systems**

Tables 1, 2, and 3 categorize embedded database products from a number of vendors. I have arbitrar-

---

**Table 1. Relational client-server products.**

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Product</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centura Software</td>
<td>Velocis</td>
<td>Runs on desktop, server, and embedded operating systems. Supports both SQL and</td>
</tr>
<tr>
<td><a href="http://www.centurasoft.com">www.centurasoft.com</a></td>
<td></td>
<td>programmatic interfaces for data management.</td>
</tr>
<tr>
<td>Empress</td>
<td>Empress RDBMS</td>
<td>Runs on the major embedded operating systems. Supports both SQL and programmatic</td>
</tr>
<tr>
<td><a href="http://www.empress.com">www.empress.com</a></td>
<td></td>
<td>interfaces to manage data.</td>
</tr>
<tr>
<td>Microsoft</td>
<td>MSDE</td>
<td>Upward-compatible with Microsoft’s SQL server database engine. Supports only</td>
</tr>
<tr>
<td><a href="http://www.microsoft.com">www.microsoft.com</a></td>
<td></td>
<td>limited concurrency and relatively small data volumes. Runs only on Microsoft</td>
</tr>
<tr>
<td>Oracle</td>
<td>Oracle 8i</td>
<td>Runs on Linux, but not on embedded operating systems. A client-server system that</td>
</tr>
<tr>
<td><a href="http://www.oracle.com">www.oracle.com</a></td>
<td></td>
<td>supports the company’s standard interfaces.</td>
</tr>
<tr>
<td>Pervasive</td>
<td>Pervasive.SQL</td>
<td>Available for embedded operating systems; also runs on desktop and server CSs.</td>
</tr>
<tr>
<td><a href="http://www.pervasive.com">www.pervasive.com</a></td>
<td>2000</td>
<td>Supports a simple programmatic interface for operating on data; can be configured</td>
</tr>
<tr>
<td>Polyhedra</td>
<td>Polyhedra</td>
<td>A client-server database engine that supports SQL; runs on common desktop and</td>
</tr>
<tr>
<td><a href="http://www.polyhedra.com">www.polyhedra.com</a></td>
<td></td>
<td>server operating systems as well as some embedded operating systems.</td>
</tr>
<tr>
<td>Solid</td>
<td>Solid Embedded</td>
<td>Offers database management services on a variety of server and embedded operating</td>
</tr>
<tr>
<td><a href="http://www.solidtech.com">www.solidtech.com</a></td>
<td></td>
<td>systems. Engine provides data access via SQL with ODBC and JDBC interfaces</td>
</tr>
<tr>
<td>Sybase</td>
<td>SQL Anywhere</td>
<td>An SQL engine that may be configured to run only the queries a particular</td>
</tr>
<tr>
<td><a href="http://www.sybase.com">www.sybase.com</a></td>
<td></td>
<td>application requires. Heavy emphasis on synchronization with a Sybase enterprise</td>
</tr>
<tr>
<td>TimesTen</td>
<td>TimesTen</td>
<td>Provides standard relational data access via SQL, ODBC, and JDBC, but is designed</td>
</tr>
<tr>
<td><a href="http://www.timesten.com">www.timesten.com</a></td>
<td></td>
<td>for large memory systems. Represents memory allocation differently in memory than</td>
</tr>
</tbody>
</table>

---
ily divided the products into Table 1’s relational client-server products. Table 2’s object-oriented client-server products, and Table 3’s embedded libraries, which do not require a separate server process. Each category has advantages and drawbacks and is popular for different reasons. All but one of the products listed here, the gdbm library in Table 3, support transactions and different architectures of embedded operating systems.

Client-server relational system vendors. Some vendors, such as those shown in Table 1, offer client-server relational systems. Companies like Microsoft, Oracle, Sybase, and Informix sell high-end enterprise database systems, and their enterprise database engines, to embedded-systems developers. Client-server relational products are quite popular, because many programmers are already familiar with the SQL (Structured Query Language) and relational database design. The main drawbacks of client-server systems are the extra run-time cost of communication between the client and the server and the additional complexity of installing, running, and maintaining a separate server process in an embedded system.

Client-server object-oriented system vendors. Other vendors, such as those listed in Table 2, bring their object-oriented database expertise to the embedded systems market. Their products seem like good choices for embedded-systems use, but seldom get serious consideration. Why? Because vendors designed the popular object-oriented databases for Unix systems and gave them deeply ingrained assumptions about their memory-management system and interprocess-communication behaviors. These products port to the very different architectures of embedded operating systems only with difficulty. They may, however, prove useful on Linux-based systems and some Unix variants such as FreeBSD, which are used in some embedded devices.

Object-oriented databases are popular because of their close integration with the C++ and Java programming languages, and because they generally hide the complexity of database design from the application programmer. Their main drawbacks in the embedded market are the overhead that client-server communication imposes and poor vendor support for embedded operating systems.

Embedded library system vendors. Vendors listed in Table 3 sell systems explicitly designed for embedded use. All these systems link directly into the address space of the application that uses them. They provide a simple language-level API that does not require SQL to manipulate data. The key advantages that embedded libraries offer are:

- faster execution, since database operations need not communicate with a separate server, and
- increased reliability because fewer components run on the embedded system.

Library products suffer one significant drawback: They require developers to master nonstandard programming interfaces.

CHOOSING AN EMBEDDED DATABASE SYSTEM

You should consider several criteria when selecting a database engine for an embedded system.

Platform support

First, and often most critical, platform support determines many subsequent options. For example, if your embedded system will run under Linux or VxWorks on a Pentium chip set with a conventional magnetic disk storage system, you can choose from several database systems.

Developers who use less popular embedded operating systems, who need to run on more than one embedded operating system, or who use more exotic processor boards and storage will have fewer choices. Embedded operating system vendors typically maintain lists of partner companies whose products run on their operating systems. Consult that list for a good look at your first choices. Some database vendors distribute their product in source code form, so developers can port it to new platforms and new processors themselves. For some less common platforms, which lack vendor support, this option may be the only practical choice.

Resource load

You must also consider the resource load the database imposes on your embedded system. Does memory footprint make a difference to the application? Will the system support a client-server database system, or must the database be truly embedded in the application’s address space? As a rule, database sys-
Table 3. Embedded libraries.

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Product</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centura Software</td>
<td>RDM, db.linux</td>
<td>Distributed as source code, RDM compiles into a library that links into the application. It runs on embedded operating systems. The company offers its Linux port of RDM as a separate product called db.linux.</td>
</tr>
<tr>
<td><a href="http://www.centurasoft.com">www.centurasoft.com</a></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FairCom</td>
<td>c-tree Plus</td>
<td>Distributed in source code form; compiles into a library that links directly into the application address space. Officially supports the QNX and LynxOS embedded operating systems, but users can port the product's source code to other embedded operating systems.</td>
</tr>
<tr>
<td><a href="http://www.faircom.com">www.faircom.com</a></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Free Software Foundation</td>
<td>gdbm</td>
<td>Provides a simple programmatic API to data records; does not support concurrent read-and-write access or transactions. Distributed in source form at no charge under the Library General Public License. Does not have the Free Software Foundation's explicit support for embedded operating system ports. Nevertheless, many ports are available, and users can freely port the software to new operating systems.</td>
</tr>
<tr>
<td><a href="http://www.gnuro.org">www.gnuro.org</a></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Informix</td>
<td>C-ISAM</td>
<td>Embeddable data management library that offers many of the services embedded systems require. The library runs only on desktop and server operating systems, however.</td>
</tr>
<tr>
<td><a href="http://www.informix.com">www.informix.com</a></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sleepycat Software</td>
<td>Berkeley DB</td>
<td>Supports a simple programmatic interface for database management; can be configured to include or exclude components as the embedded application requires. Distributed in source form, runs on desktop and server operating systems and some embedded operating systems.</td>
</tr>
<tr>
<td><a href="http://www.sleepycat.com">www.sleepycat.com</a></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3. Embedded libraries.

- Performance evaluation
  Concurrency and scalability are major considerations. Will the application have multiple control threads working with the database at the same time? How big will the database get?
  Some systems were designed to support high concurrency and large databases; others were not. Products that scale well often compete well with low-end products that have few users and small amounts of data. The converse is not generally true, however.
  If scalability matters, you have fewer choices. You must test systems carefully to be sure they work as the vendor claims. Desktop- and server-system developers can rely on transaction-processing benchmarks to characterize database performance. Embedded systems have no such standard to quantify performance. Evaluating the actual application's performance is critical because it is the only embedded-system benchmark that matters. Most database vendors make their products available for evaluation at no charge, and you should take advantage of this opportunity to "test-drive" candidate products.

- Required services
  Once you have found a database system that runs well on the platform you plan to use, that fits in the available space, and that can handle your planned workload, you must next consider what services the embedded system actually requires.
  Will the database be used by a single control thread or by more than one? If the system requires no concurrency, the application will run faster without locking. Some database systems run without locks or permit you to disable locking.
  Does recovery from failures matter? Some embedded systems simply rebuild the database from scratch every time they start up. Others must be able to survive unexpected shutdowns and restarts. If recovery matters to you, choose a system that provides this service. If recovery doesn’t matter, you may want to choose a system that excludes the service or permits you to disable it, so that your application runs faster.

- Price
  After selecting products according to your technical requirements, you may still have more than one choice. If so, understanding product pricing can help you make your final decision.
  Several popular embedded database systems are distributed at no charge under certain conditions. Even some commercial vendors allow no-cost use because they expect to sell you services or other software products later.
  Vendors that charge for their products apply a dizzying variety of pricing policies. Common policies include fixed fees for each developer who works with the database software, for each application that embeds it, or for each platform on which it is deployed. Many vendors refuse to do fixed-price licensing, opting instead for per-copy or per-user royalty pricing, although such licensing is less common in the embedded systems market than in other markets.
  You should also understand at what point you must pay for the software. Some vendors charge for use during development, while others only charge when you deploy the embedded system.
  The best way to determine the actual cost of deploying a particular database is to speak to the vendor that distributes it. Pricing policies vary so widely that you can only compare them by applying each candidate product to your embedded system deployment plans and seeing how much it costs.

DESIGNING THE APPLICATION

Once you have selected an embedded database system, you must design your embedded system so that the database performs well. Realizing this goal demands that you consider reliability, speed, and predictability. Your users should get the right answers
quickly, and they should not be subjected to failures or unexpected responses.

Designing for speed
To ensure that your database responds quickly to searches and updates, configure it correctly from the start. Data layout and resource allocation are the main issues to consider. You must make at least three important decisions when designing your database.

Data representation. If the application and database use different representations for values, every fetch and every store operation will require translation. Most embedded database tools operate on a fixed set of data types, in a record format mandated by the database engine. For example, a typical database might require that a C structure or the attribute values in a Java instance be translated into records containing integers, dates, character strings, and other known types. A few database systems—generally, those that are libraries—let programs store data in program-native format, rather than translating it to a database format. In this case, the database requires no data translation.

Access patterns. Data should always be laid out with a view to the queries the application will execute. Should records be sorted on some key? Can the application choose keys so that related records (those the application is likely to need at the same time) reside physically close to each other in the database?

Most database engines support B+-tree storage, and some support other storage structures such as hash tables. As a rule of thumb, B+-trees perform better than hash tables, because B+-trees physically cluster data on a key and can exploit locality of reference with adjacent keys.

Further, you must consider the searches and updates the application will do and understand the physical layout the database system provides. Data can only be physically clustered on one key. If the application is likely to look up records on more than one key, consider defining secondary indices on the additional search keys. Such indices make the classic computer science trade-off: To save time during program execution, the designer duplicates some data, which consumes additional space.

Configuration. You can configure most embedded database systems in either more or less detail. Common configuration parameters include the amount of memory used for secondary caches, whether data should be written to disk or merely stored in memory, and the granularity the locking system uses to acquire locks on objects. Less commonly, embedded database systems may permit you to turn whole subsystems—such as locking and logging—on or off as the application demands. Single-user applications require no locking. Applications that can completely rebuild the database at startup do not need logging to recover changes after a crash.

The configuration control that the various database systems offer varies enormously. Thus, you must understand the chosen system’s behavior and consider its configuration carefully. In general, you should disable any unnecessary subsystems to save time and space.

Further, you must analyze the space that database records consume and size caches appropriately. Embedded applications have a wide variety of requirements, and database vendors often choose the wrong defaults.

Designing for predictability
Embedded systems may need to run when no human is present—or their users may not even realize they are working with computers. Thus, predictability is even more important in embedded systems than in end-user desktop applications.

Given these constraints, you must fanaticize check return values and other error indicators in your application so that it can recognize and recover from failure without external intervention. For example, if a write fails because of insufficient storage space, the application must clean up storage so that the write can succeed. If a network connection fails, the system must transparently redirect clients to another connection.

Designing for predictability is not easy, however, because database requests can fail for many reasons. The type of failure depends on the database system in use and how the application uses it. Examples include deadlocked transactions and illegal duplicates or other out-of-range errors in individual data elements.

Resource exhaustion is a potential cause of database operation failure. Mismanagement of system resources is a common error, particularly the failure to allocate and release memory properly. Embedded applications can run for a very long time, and even small memory leaks can eventually use up all available space. From the end user’s viewpoint, the application performs more and more sluggishly, then fails altogether. Tracking allocated resources and releasing them as soon as they are no longer needed prevents resource exhaustion in conventional databases.

Using an embedded database complicates resource management. If the database vendor’s library performs sophisticated resource allocation and management, the programmer must be sure the application code understands the library’s behavior. Many libraries include their own resource managers, such as file descriptor pools and memory managers, which differ from the services the operating system provides. Calling the operating system’s function to release a block of memory can be a serious bug if the database

To ensure that your database responds quickly to searches and updates, configure it correctly from the start.
Choosing the appropriate embedded database system is the key to satisfying the need for hands-off administration.

For end users, predictability means that the system continues to work without requiring manual administration. This means that the system has no backups, no recovery procedures, and no periodic reorganization. If the software uses a log to track changes, the application must ensure that the database periodically reclaims unneeded log entries. If the database must run recovery after a system crash, the application program must notice when recovery is required and take appropriate action that is transparent to the user.

Choosing the appropriate embedded database system is the key to satisfying the need for hands-off administration. For a given application, you should decide what services you require. Does the application need transactions so that changes will not be lost after a crash? If so, the database system will likely need to run recovery on restart after an abnormal shutdown. In that case, the recovery system must be callable from the application program without requiring human intervention by the end user.

**PERFORMANCE TUNING**

Virtually all database applications suffer from the same performance bottlenecks. The three most common causes for poor performance in database applications are two or more threads contending for frequently accessed—or hot—data, disk-to-memory transfers, and deadlocks.

**Contention for hot data**

Most database systems lock the values they touch during processing. Read locks are generally shared, but write locks are exclusive. If many threads try to touch the same database object at the same time, and some need exclusive access, some will be forced to wait while the others complete their processing.

Contention for scarce resources causes many performance problems. Whether your application is waiting for the disk arm, a locked object, or the right to append to the log, reducing contention is the best option for improving performance.

System metadata, which describes the content and structure of tables and records, is usually hot, so most database vendors treat it specially. They make locks on such data less often exclusive and release exclusive locks as early as possible to improve system throughput.

Shared structures that coordinate among multiple threads and processes can also be the focus of contention. Locks generally protect the pool of buffers, for example, but commercial systems are designed to hold these locks for very short periods. As a result, contention generally results from high demand for relatively few records stored in a table that the application defined.

If an application frequently waits on locks held by other transactions, the database probably contains hot data. If your database supports record-level locking, try using it to reduce contention. If not, making your database pages smaller offers a good surrogate technique. Smaller pages contain fewer records, so the page-level locks that the database acquires behave more like record-level locks.

Changing your transactions so that they touch the hot data last and holding hot locks for a shorter time also reduce contention.

**Disk-to-memory transfers**

Moving data between disk and memory can cause bottlenecks as well. Latency depends on a disk drive's electromechanical properties, and reducing latency generally requires effort and money. Database programmers generally try to read and write as little data as possible, then make their accesses sequential when they must do them.

When developing an embedded system, you can choose from a wide variety of hardware platforms. Some of these systems use conventional magnetic disk for storage, but others use more exotic technology, like flash RAM, to store data persistently. Flash RAM, with its ability to hold data across system shutdown and restart, makes possible embedded database systems that ignore disk accesses altogether. Such systems can optimize their page layout and data access to take advantage of large memory blocks with uniform access times, rather than making page-based accesses because of an architecture that a spinning-disk storage system imposes.

Yet, surprisingly, no established embedded-database-system vendor has created such a system. TimesTen, described in Table 1, runs primarily from memory by design, but offers only limited support for embedded operating systems. Even TimesTen forces write-throughs to disk for persistence.

Most embedded database systems assume the existence of a magnetic disk storage system as a backing store for information in memory. Some can manage data wholly in memory, with no backing store, provided that available memory is large enough to hold the entire database. A few embedded database systems can use flash RAM in place of magnetic disk when necessary. These systems generally treat flash RAM as if it were disk. Here, I focus on performance tuning for disk-to-memory transfers, but my advice applies equally to memory-to-flash-RAM transfers.

A system generally requires disk I/O when it encounters a problem with hardware configuration or the application's data access pattern. Database systems generally use RAM as a buffer cache for data
that normally resides on disk. In the steady state, database programmers want the cache to satisfy most page accesses without requiring a disk transfer. Having insufficient available memory in the buffer cache will force pages out during normal processing, and they must be reread when they are next required.

Systems should thus have a large enough buffer cache to hold all commonly accessed data at the same time. Cache misses and evictions show that the cache is too small to hold the application’s complete working set. Naturally, some I/O activity is inevitable, but performance tuning strives to reduce the number of disk reads and writes required.

If the cache hit percentage is low, you should consider increasing the cache size. What “low” means will depend on your application, but a reasonable goal is to find at least 50 percent of required pages in the cache.

You can also improve the cache hit percentage by making sure your queries and data have the same locality patterns. If your application looks up records by date, and requests for a particular date make requests for nearby dates more likely, you should order your records in a B+tree table. Doing so will put nearby dates close to one another in storage, so that satisfying queries will require fewer page reads. This technique can backfire, however. If locality of reference causes many control threads to compete for records near one another, you may end up with too much contention.

If you repeatedly scan an entire table looking for records of interest, you should consider creating an index on the table that lets the application find records more quickly. For example, if your customer table is keyed by ID number, but you often search by last name, create a secondary index that keys on last name. Indexes organize data for faster lookup on a particular key, and this fix can drastically reduce your system’s I/O.

**Deadlocks**

Most database systems use a two-phase locking technique to guarantee that different concurrent users do not interfere with each other’s changes. Each control thread acquires locks during a transaction. It holds them until the transaction completes, then releases them all at once. Figure 1 shows how two-phase locking can produce deadlocks when multiple control threads compete for locks on the same objects.

**Two sources.** If two control threads lock the same objects in a different order, each may be forced to wait on a lock held by the other. In this case, neither can progress and the system is deadlocked. Similarly, if two control threads share a lock on the same object, and both want to upgrade to an exclusive lock, neither will be able to complete until the other releases its shared lock. Again, the system is deadlocked.

All database systems susceptible to deadlocks have mechanisms for detecting them. Generally, they interrupt one thread in the deadlock and let others proceed. Any changes that the interrupted thread made are rolled back. If the application produces many deadlocks, several threads will begin working, only to have their changes rolled back before they complete. The work they do will be wasted and system throughput will suffer.
Several solutions. In embedded systems, turning off the locking subsystem may eliminate deadlocks altogether. If the database system supports it, and if the application permits it, this solution can speed up execution across the board. Single-user applications and multiuser read-only applications provide the best common opportunities for disabling locking.

If you cannot disable locking, shortening transactions is your simplest alternative. If the application has one big transaction that runs for a long time, consider breaking it into multiple smaller transactions. Each transaction will release its locks sooner, reducing the probability of interference from other transactions.

A less obvious but more powerful solution ensures that all your transactions acquire their locks in the same order. Deadlocks arise when one transaction locks object A and wants object B, while another locks object B and wants object A. If you write your code so that transactions touch objects and acquire their locks in the same order, such deadlocks become impossible.

Another powerful technique obtains the correct lock up front. Many embedded database systems let an application specify the lock it wants on a value before touching it. If it reads a value that it later plans to write, it may not be able to upgrade the read lock to a write lock if another reader arrives in the meantime. To avoid this situation, the application needs to get the write lock the first time it touches the object. Doing so locks out the other reader; although this technique introduces contention, it eliminates the deadlock. Generally, contention is less disruptive to system performance than a deadlock. To get a write lock on a read access, you should specify that its access will be read-modify-write, or RMW.

Finally, if all other techniques fail, you should shorten the deadlock detection interval, which determines how often the database system checks for deadlocks. The longer an undetected deadlock persists in an application, the more likely other transactions will get involved. Should this situation occur, you’ll most likely need to roll back more than one transaction to eliminate the deadlock, so keeping others out is a good strategy.

**OPTIMISTIC CONCURRENCY CONTROL**

Some database systems use a strategy called optimistic concurrency control in place of two-phase locking to avoid deadlocks. Because they do not acquire locks during operation, such systems are deadlock-free—although achieving this state introduces other performance issues.

Optimistic concurrency control makes copies of the data it updates, saving the changed copies until the transaction ends. When the transaction commits, the system checks to see if any conflicting updates have happened since the transaction started. If not, the changes are copied into the main database. Otherwise, it discards the changes. This policy is optimistic because the transaction assumes that its updates will work, then checks that assumption after all work has completed.

If two transactions update the same record simultaneously, their updates conflict and one must fail. Systems that use optimistic concurrency control discover these conflicts only at the end of the transaction, at which point they discard all work the transaction did. For long-running transactions that do a lot of work, this can be a serious performance problem. Under two-phase locking, one of the transactions waits until the other completes, thereby avoiding wasted work.

Systems that have little contention can use either two-phase locking or optimistic concurrency control, and will perform well. Two-phase locking seldom leads to deadlocks, and optimistic concurrency control rarely forces the discarding of work because of conflicts.

Applications that do have write contention may need to choose between two-phase locking and optimistic concurrency control. As a rule of thumb, if the writes genuinely conflict, two-phase locking is better. If two transactions must update the same value in the database, forcing them to run serially will produce the best throughput. Under false sharing, however, optimistic concurrency control is better. In this case, two transactions can update different values in a single record, so their updates do not conflict. Optimistic concurrency control permits both transactions to run to completion without waiting, while two-phase locking serializes them.

The design, construction, and deployment of high-quality embedded systems pose different engineering challenges from those you would encounter when building desktop and server systems. Meeting these challenges requires focusing on the requirements of the embedded application and choosing tools wisely.

Michael A. Olson is vice president of sales and marketing at Sleepycat Software, a vendor of open source embedded database software and services. His interests include data management and infrastructure for wireless and mobile computing. Olson received an MS in computer science from the University of California at Berkeley. Contact him at mao@sleepycat.com.