A Practical Guide to Identifying Slow Code During Development

A paper by Technology Strategy Research
Executive Summary

Application performance is one of the top concerns of enterprises today. Applications that perform poorly in production can cost an enterprise in terms of lost business opportunities, poor customer service, and unscheduled downtime. Often slow applications manifest other errors, such as timeouts, synchronization issues, and thrashing.

Most enterprises monitor application performance during production—when the application is live and serving customers and other users. However, by this time, any performance issue is already causing problems for the business. Fixing a performance problem at this point is likely to require downtime and an immediate shift of significant IT resources to analyze, diagnose, and address it.

Also, it’s simply more expensive to fix issues as an application progresses through its lifecycle. The earlier you can fix a performance issue, the less time and resources it costs to do so.

The need to find and fix performance issues early means that it’s up to developers to ensure that applications meet performance requirements. Often performance issues are not found until load testing or even production because some issues don’t manifest themselves until the application is under load. But developers can identify many potential problems well before that time, with the right tools to assess performance.

The only way developers can get a jump start on application performance is through comprehensive performance analysis. Most developer performance tools do statistical profiling, which samples the performance of various components during unit test or later integration or feature testing. While profilers can be useful for obtaining a high-level overview of code performance, they lack the granularity to pinpoint specific performance issues and tie those issues to coding practices.

Instead, developers should use full performance analysis that traces the amount of time it takes to execute every line of code, and every operation, as well as how many times that line is executed. Performance analysis provides the ability to fully understand how fast it takes to execute code, down to each individual line.

Potential Causes of Slow Code

Many seemingly innocent coding practices have the potential to cause code to execute slowly. In a number of cases, individual programming techniques can be innocuous, but in the aggregate can weigh heavily on application performance. Programmers should understand the performance implications of the code they write by looking at performance analysis during unit tests and as a part of code reviews.

Select Programming Constructs Wisely

The goal of programming for performance in managed code under the .NET Framework is to optimize computationally expensive operations. These include database calls, memory allocations (especially large ones), floating point operations, and certain Framework calls. Using .NET performance analysis enables programmers to determine which calls are more expensive than others.

Programmers have to perform most or all of these expensive operations as a matter of course. The trick is to execute them as few times as possible to optimize execution time. In some cases, it may be possible to find a more efficient way than the one actually used. This is the value of combining performance analysis with code reviews—armed with performance information code reviews can focus on refactoring to achieve better code execution times.

While much of the focus of the development team is on the time it takes to execute code, equal attention needs to be paid to the number of times a particular piece of code is executed. While a single call might be relatively inexpensive, placing that call in a loop could mean that an expensive operation is executed thousands of times. This might be the case for a database call or a Framework call that involves large memory allocations.
That, in fact, represents a significant difference between managed and native code. Memory allocations are inexpensive in native code, and releases are comparatively expensive. In managed code, the opposite is true—allocations are expensive, and releases are inexpensive.

The .NET Framework provides alternative means of performing many actions. When there are several choices available, developers should choose the calls that perform the actions they require, without a lot of additional computation. Developers should analyze the performance of alternative Framework calls to identify the best calls for their purposes.

Application Memory
Poor memory usage is also an area that can slow down application execution. Even in managed environments such as .NET, attention to memory is important, even though the system allocates and reclaims all memory. Improper memory management can lead to large numbers of long-lived objects, huge objects that serve no programmatic purpose, and object leaks. All of these contribute to memory bloat and ultimately slower execution. If not resolved, they can ultimately lead to application crashes and out of memory errors.

Even in a managed environment, many programmers prefer managing their own memory, and the .NET Framework provides a limited ability to do so. However, the garbage collector in the .NET Framework will almost always do a superior job of managing memory than the programmer.

Still, there are techniques that the programmer can use to make sure the memory footprint doesn't grow unnecessarily. One important goal for programmers at this point is to ensure that as many objects have as short a life as possible. This is done by making as many variables local as possible, and enabling them to go out of scope when their method is complete. Once out of scope, their memory can be quickly reclaimed by the .NET garbage collector.

Database Calls
Database calls tend to be computationally expensive because they frequently go out to disk and queries can take long periods of time to execute. If an application makes frequent database calls, it will slow significantly as the application pauses to write and retrieve data from disk.

The best programming practice is to create an in-memory database object, using ADO.NET, LINQ, or similar technique, in order to bring the data needed by a particular routine into application memory. However, even this approach isn't guaranteed to provide needed performance. In many cases, this is because developers don't construct their database objects appropriately. In such cases, additional data may have to be retrieved, or
the application retains data in the database object long after it should be returned to the database and the memory reclaimed by the system.

Databases themselves also offer caching, and it is incumbent for the development team to work with the DBA to ensure caching is optimized for the application. While this is likely to involve a measure of trial and error, database tuning at this point in the application lifecycle will ensure better performance during production.

Finding Slow Code

One way of locating and analyzing slow code is through the use of tools such as Micro Focus® DevPartner Studio performance analysis. The performance features of DevPartner Studio include code timings at the line or method level, as well as the number of times that code is executed during a unit test or other type of test. It also incorporates comprehensive memory analysis so that programmers can examine .NET memory usage dynamically while running unit, functional, or regression tests.

Because database access is very slow relative to the execution of application code, it is easy to identify. If you instrument at the line level, you can look at the actual code to see if there is an SQL call. An SQL call by itself isn’t necessarily a cause for concern because it could reasonably be performed on occasion to populate a database object or to return updated data back to the database.

Even if you simply look at method-level performance, it is fairly straightforward to identify poor database management in application code. Method-level results can also help to pinpoint other performance problems such as excessive work being done in loops. It’s possible you will have to do further investigation to identify the exact source of the slow code, but an examination of the source code in a slow method should provide significant clues as to where the issue resides.

Master Application Performance

It can be a challenge to code for optimum performance at the same time you are implementing application features. The focus is typically on adding new features, and performance is usually an afterthought.

To integrate performance analysis early in the development cycle, software developers should consider the following steps:

1. Start performance analysis in conjunction with unit tests early in the development process, and continue through functional and regression testing.
2. Use comprehensive performance analysis that includes both the time needed to execute all code as well as the number of times that code is executed.
3. Use performance analysis in conjunction with code reviews as a way of rewriting methods. Method-level performance analysis can help identify code that can benefit from refactoring.
4. Run a final comprehensive performance analysis prior to releasing the application from development.

Performance analysis during development won’t catch all performance problems. Some problems clearly won’t manifest themselves until the application is tested under load, or even during production. But dealing with problems later in the application lifecycle means that developers will be under pressure to find, diagnose, and fix those problems quickly and won’t have the best data or conditions to do so.

Once the application leaves development, any problems found in production use will be addressed in haphazard ways, often by overcompensating with servers, memory, or continuous database tuning. It’s better for the application, and less costly for the enterprise, to find slow code before it becomes a business problem.

Figure 3. Looking at method-level performance first lets developers determine where to concentrate their efforts.