Performance Engineering

DGP has been designed and built to deliver high performance executables. This is necessary because of the end of Moore's Law, which means that performance improvements can no longer be accomplished by using newer, faster hardware to solve the problem of poor software system performance. Going forward, performance improvements can only be achieved by improving the architecture and code of the software systems themselves.

High performance is very beneficial to software systems in a number of ways. First, high performance helps provide an excellent user experience. Second, it enables the use of a server-centric architecture that consolidates all of the logic and data of a system in the middle tier, which improves the security of a system and greatly simplifies maintenance, debugging and deployments. However, this architecture is only feasible if the services of the middle tier are very fast. Third, high performance executables reduces the amount of hardware and software needed to support a given number of concurrent users, reducing the cost of a system. Finally, the simplification of a system's architecture and code that is needed to deliver high performance executables reduces the number and cost of the staff needed to develop, test and support the system and also shortens the time required to produce deliverables.

The term "high performance" is defined as follows:

- Every user action in a UI should be displayed and the UI should be fully interactive in under one second.
- To meet that requirement (especially on mobile devices), server-side processing must be completed in under 100 MS.
- These are the minimum performance standards. Scalability can be increased, and total costs can be reduced even more with higher levels of performance and efficiency (which will eventually be subject to diminishing returns).

Performance Engineering References

An excellent book that has served as a guide for many of the performance improvements in DGP is Ben Watson's *Writing High Performance .NET Code* : https://www.writinghighperf.net/ .

Video explaining the importance of the 1 second limit: https://www.youtube.com/watch?v=ll4swGfTOSM

Performance Engineering Overview

Everything in software development is a series of compromises, including high performance. The objective is to determine which tools provide the best possible performance at the lowest cost and require the shortest amount of time to deliver. This means developer productivity is very important to minimize TCO and TTM, balanced with the need to maximize performance.

General Types of Programming Languages:

- Interpreted (Python, JavaScript)
- Managed (Java, .NET, Go)
- Low level (C/C++, Rust)

The objective when selecting programming languages is to choose whatever option delivers the best results from the perspective of those that will use those results (users, clients, customers, etc.), and will almost always require balancing multiple conflicting needs. For example, the need for high performance executables must be balanced with the cost of development (developer productivity plus developer availability/cost). Interpreted languages such as Python will generally have the highest developer productivity, but the worst performance. Low level languages like C/C++ provide the best performance, but also have the lowest developer productivity.

As one example, for IOT development the performance and efficiency of the executable is the primary constraint – more important than developer productivity and all other considerations. In this case, languages that are able to deliver the smallest, fastest and most efficient executables such as C or Rust are going to be the best choices.

In contrast, for web services and other long running processes, one of the most important factors is the ability to defragment/compact the system's memory periodically while it is running in order to maintain its high initial level of performance over time. Managed languages such as .NET, when used carefully to optimize performance, provide the best balance of good developer productivity, good executable performance, and the ability to compact memory as part of the process of garbage collection. That is why they are generally going to deliver the best results for web services instead of C/C++ or Rust.

Improving the Performance of Managed-code Web Services

For managed code, the overall objective is to minimize the workload placed on the GC so that it will have time to 1) keep up with the rate that objects are being allocated, and 2) execute the last step of garbage collection, which is compaction of the managed heap.

Steps to minimize the GC workload:

1. Simplification of the system architecture
   a) Remove any functionality that generates code (ORM's, etc.) from the design
   b) Remove all possible layers of abstraction (IOC containers, etc.) from the design
   c) In general, reduce the number of subsystems and tiers as much as possible
2. Simplification of the system code
   a) Reduce the object allocation rate of the system. This is accomplished by designing your system with the fewest classes and fewest methods possible in order to allow for the instantiation of fewer objects.
   b) Reduce object lifetime in order to collect objects in Gen0 or Gen1 memory segments.
   c) Reduce the depth of object reference trees. In general, try to minimize references between objects as much as possible.
   d) Avoid allocation of large objects. Objects on the LOH tend to be large strings or collections. Try to find ways to limit these objects to less than 85,000 bytes if at all possible.
   e) When objects cannot be collected in Gen0/Gen1 or must be larger than 85,000 bytes, reuse them rather than allocating new instances every time.
   f) Avoid computationally expensive functionality within a language or platform (reflection, late-binding in general, etc.).

Refer to Ben Watson's book for many additional recommendations.

When considering the performance of distributed software systems, Little's Law from queuing theory is very useful. The takeaway from Little's law is that the latency experienced by users is the single most important factor, and that is what should be minimized.

Rules for Measuring Performance Metrics

1. Only end-to-end measurements of the latency (time spent waiting) experienced by users of the software are useful. Only these types of end-to-end metrics can be used to compare very different software systems to each other. That means that the

performance of a system must be measured by the distributed remote applications provided to users, and testing the system's functionality should also work in the same way.

2. Only measurements collected while a system is under realistic workloads are accurate and meaningful. Collecting performance measurements will small numbers of concurrent users will produce unrealistically good metrics. Generating synthetic workloads at sufficient scale to simulate realistic workloads can be very difficult and expensive. The best way to accomplish this in practice is to collect end-to-end performance measurements during the actual use of a system in production, without affecting the system.

Note: variations in the performance measurements of a given system will sometimes occur due to the unpredictable combination of several features:

- Caching. To state the obvious, cache hits will improve performance, while cache misses will cause a noticeable decrease in performance. There are multiple levels of caching that are built into the server-side tiers of DGP Lattice:
    - IIS web server – by default, IIS application initialization can take a significant amount of time. From version 8.0 and later, IIS can be configured to pre-load a web service, greatly reducing the time needed to reply to a first request.
    - .NET framework – by default, the .NET framework JIT compiles and loads each method into memory the first time it is called. NGEN and .Net Native offer ahead-of-time (AOT) compilation to significantly improve application startup times.
    - DGP – the core web services cache reusable data in memory using the ASP.NET Application and Cache objects. A cache miss requires an RPC query to the database server.
    - SQL Server – SQL Server caches queries, query plans, pages of data, indexes, etc. in memory as much as possible.
- Network congestion will vary from time to time, and can have a big effect on the network latency metrics.
- Concurrent processes sharing host resources will vary from time to time, and can have a big effect on server processing time (the infamous "noisy neighbor" problem).

The Lattice UI has been built to store end-to-end performance metrics during its use of each environment of a system. This is the mechanism used to collect performance metrics from production systems under realistic workloads. Since the data is measured at the remote endpoints, and only a small set of users are collecting performance data (based on role membership), the collection and storage of those metrics do not add any significant overhead to the system being measured.

High Performance Summary

The importance of optimizing systems to deliver high performance executables cannot be overstated. It not only improves the customer/client experience when using the software, but also decreases the cost of literally all parts of the software system by reducing the amount of resources needed to support a given number of concurrent users. In practice, this means significantly less hardware, less software, less infrastructure and fewer people, which translates to very large cost savings for the business.

Performance Verification

1. *The status bar at the bottom of the DGP Lattice UI shows the end-to-end round trip time and the server-side processing time for the API methods called by the application during normal use in order to prove that the standards are being met in every environment of a DGP system. The calculation of the server performance metrics are designed and built into each web service, and are returned to the client app in every response message. The end-to-end performance metrics are calculated by each client application.*
2. *For members of the RemoteMonitor role, the Lattice application saves the performance metrics displayed in the status bar to the LatticeMetrics table of the SysMetrics database. This is also a form of monitoring the health of a system.*
3. *The API Tester shows the end-to-end round trip time and server processing time for every API method tested in each environment. The results of test runs can be saved to the TestResults table of the SysMetrics database for analysis. Those results can also be saved to a CSV file for import into Excel, etc.*
4. *The AutoWork Tester shows the duration of each automated process, and in many cases also shows the duration of the individual steps within those processes. The performance of the automated processes directly affects the number of concurrent threads needed to handle the automated workload of a system.*
5. *Caching authorization data in each account record results in authentication and authorization querying for a single record from a single table (APIUser) in the SysInfo database. This greatly improves the performance of the message pipeline class.*
6. *Caching the UserInfo object in each web service not only eliminates the query to the APIUser table, but also the need to decrypt the returned password value stored in the table. This results in a significant performance improvement for each API request*