IRONGRID

Continuous Performance:
The Next Advance in Software Development

An IronGrid Position Paper by Cody Menard

Executive Summary

Faced with ever-shrinking development cycles, many Java developers have implemented Continuous Integration, an elegant best practice which ensures that development teams conduct builds at least once a day, and ideally several times daily. Continuous Integration supports the logical notion that by checking functionality and testing for code conflicts frequently, programmers can fix bugs faster and easier than if they were to wait for weeks to test new code.

Performance-minded development teams, however, should consider expanding their Continuous Integration strategies to include ongoing performance tuning, or Continuous Performance. This concept borrows from the basic Continuous Integration theory of multiple builds: Continuous Performance recognizes that testing performance several times a day makes performance tuning vastly more efficient. This approach enables development teams to avoid the long and arduous performance- tuning sessions that today typically occur at 30-day coding milestones, and that can cost developers
time and money. Even better, Continuous Performance can result in software that performs better overall, because code is cleaner throughout the development process.

IronGrid, Inc., a small group of developers in Austin, Texas, provides low-cost Java tools that can be installed quickly and require no source code changes. IronGrid's products are underpinned by the company's dedication to the developer community, the benefits of Continuous Performance, and the technology used to create the P6Framework Open Source tool set.

Java is hot. Just seven years old, Java is rapidly becoming one of the leading development environments in the world. More than 1 million programmers and thousands of companies use it[1], and half of all IT managers expect to deploy J2EE applications this year.[2]

But Java's popularity hasn't necessarily made it easy for the growing population of hard-working Java code jockeys. Ever-shortening product cycles have kept the heat on programmers, who increasingly work in large teams to meet production milestones. And every day, those teams come face to face with an immutable law of software development: The more code you write, the more bugs you'll get – bugs that cost time and sap quality and performance from applications.

The Rise of Continuous Integration

In response, a growing number of teams are embracing Continuous Integration. This best practice advocates the habit of conducting code builds at least once – and if possible, several times – every day. Continuous Integration explicitly recognizes that development teams struggle with the odious task of ferreting out code conflicts and bugs each time the group contributes new code into a central repository. By maintaining shorter time frames between builds, teams find it easier to test code. They also can take advantage of natural breaks in which to identify and fix conflicts and bugs while the amount of fresh code is still manageable. Like so many best practices, Continuous Integration is brilliant in its simplicity, which has unquestionably helped with its adoption. And even busy Java programmers, many of whom resist procedural changes simply because they are initially disruptive, quickly detect the benefits of this developer-friendly approach to engineering software.

Continuous Integration initially sprang up in "Extreme Programming" environments, a byproduct of hurried Web application development whose acceptance ascended in lockstep with Java. Some still find the notion of continuous builds to be revolutionary, however, leading to an undeserved reputation as an edgy approach to programming. Nothing could be further from fact: Continuous Integration, in most environments, is simply a smarter approach to product development, resulting in better quality products.

One group that recognized this early on is the Open Source community, an informal legion of programmers dedicated to the idea that a group of talented developers can evolve software faster than any single company can. The Linux operating environment, originally dismissed by Microsoft and other industry leaders and now available on every major hardware platform, is the poster child of Open Source success. Instrumental in driving the growth of Continuous Integration, Open Source efforts have given rise to valuable Java frameworks like the enormously popular JUnit, which allows developers to write utilities that test key features of new code. SourceForge.net, the primary destination for Open Source postings and software, is home to more than 50,000 projects. Other sites, such as JUnit.org, are resources for specific tools and frameworks.

The Natural Next Step: Performance

Thanks to Open Source and a commitment to efficient programming, Java developers are making dramatic inroads with Continuous Integration. Yet in its current iteration, Continuous Integration has yet to specifically address the next natural area of focus for development teams: Performance.

Every time a development team executes a build, programmers pass up a golden opportunity to test and tune performance. It's almost startling that this is the case, particularly in the face of growing visibility into the dangers and costs associated with inefficiently managing performance issues.

The leading cause of performance failures is the traditional reactive approach to performance tuning that development teams exhibit during the coding process. It's understandable why project managers adopt a "fit it later" approach as they face cost containment goals and looming delivery deadlines. The conventional approach is to ignore performance until an issue presents itself in a way that cannot be ignored.[3] By then, however, the root cause of the performance problem often is extraordinarily difficult to pinpoint.

Most development cycles are segmented into phases, with each phase completed when the team reaches key programming milestones. Java application development cycles tend to be shorter than other coding projects, and three-month delivery schedules for J2EE applications are common. Three-month projects commonly are split into three phases, each a month long.

Even those who practice Continuous Integration often don't concern themselves with performance issues until they reach a project milestone. By that time, however, they are faced with tuning an entire month's worth of code. If they are lucky, they will observe minimal variances in seek times and process speeds, or the few problems they encounter will be easy to diagnose and correct.

For most developers, performance tuning feels neither lucky nor easy. The sense of accomplishment that programmers experience when they reach a development milestone can quickly vanish as they spend the next week tracking down and fixing code complications that cause the application to generate 500 SQL statements instead of one, or that make database calls take three times longer than targeted to complete. It isn't unusual for Java development teams to spend 20 percent of their time – up to a week every month – tediously fixing performance problems.

This isn't just an inconvenience. Poor performance costs the software industry millions of dollars annually in lost revenue, decreased productivity, increased development and hardware costs, and damaged customer relations.[4] Assuming a software developer's average salary to be a conservative $6,250 per month, or about $39 per hour, the time lost to ad-hoc performance tuning can run as much as $1,560 per developer, or $15,600 for a development team of 10.[5] Over the life of a three-month development project, the costs of ad-hoc performance tuning amount to $46,800.

"Nearly 80 to 85 percent of database performance problems arise from the application database's design or the application's own code. Good transaction throughput requires an application designed from the database up, with performance and scalability in mind."

DB2 Magazine

The damage is further compounded by an eventual reality: Left with shrinking development cycles, programming teams will deliver poorly performing software that can cause serious downstream problems for companies with customer-facing Web applications. According to CyberAtlas, some one-quarter of all online transactions are not completed.[6] For transaction-intensive Web sites, resulting losses can amount to as much as $275,000 an hour if only 20 percent of transactions are not completed, according to Forrester Research Inc.[7] Clearly, this is a problem worth fixing.

Introducing Continuous Performance

Jack Shirazi, in his book *Java Performance Tuning*, notes: "Performance tuning of Java applications (is) second only to the primary functionality of an application."[8] Most developers would agree. Then why is performance tuning typically addressed only after weeks of coding have passed?

It shouldn't be. In fact, performance optimization should be worked into Continuous Integration, making this best practice even better. We call this approach Continuous Performance, an important subset of the Continuous Integration concept.

Continuous Performance shares the same underlying "build and test often" philosophy and leverages the well-wrought best practices of Continuous Integration:

- Constant monitoring of software's performance against preset parameters;
- The ability to find the root cause of performance variances before too much time passes; and
- The ability to regularly fix performance problems before they inflate into unmanageable headaches requiring weeks of recoding and further testing.

If performed concurrently with build tests, Continuous Performance also will alleviate developers' abhorrence for extensive performance tuning sessions. But there is another, more financially rewarding, benefit: Continuous Performance eliminates the factor of compounded performance degradation.

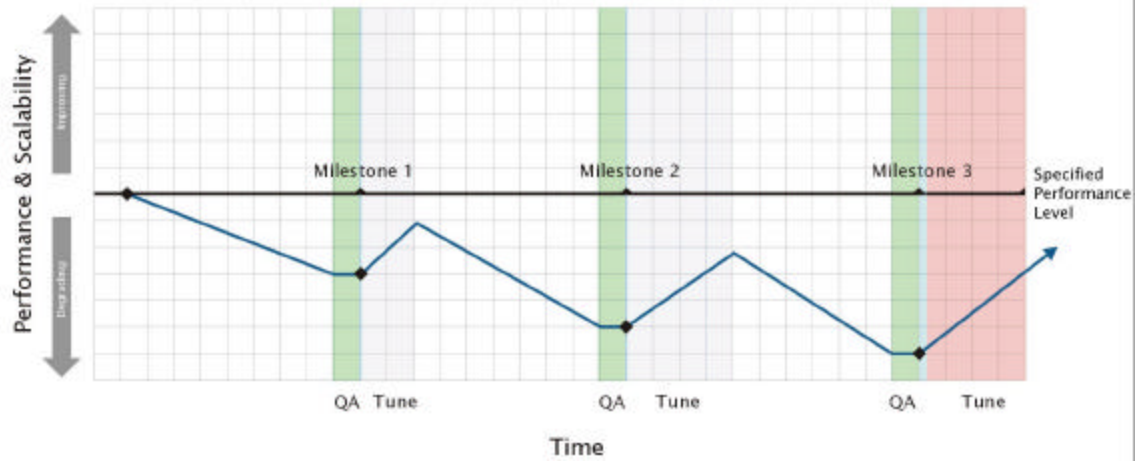
In team-based development environments, individual performance problems layer onto one another until variances reach startling levels. If Jeff's new code causes a 400 millisecond delay on a SQL call, it will sit on top of the 600 millisecond lag resulting from Diane's code that was added in the latest build. Between the two of them, they have contributed a 1-second variance that will continue to grow as new code is written, but not tuned.

Picture that kind of performance-killing handshake occurring with every build. After 30 days, when it's finally time to address performance variances, the labyrinth of interrelated bugs can amount to a black hole for developers. In many production environments, teams faced with unraveling complex performance snarls against tight deadlines inevitably make concessions, resulting in slackened performance parameters and an application that is slower than intended. This broken process repeats itself through every phase of the production cycle, with each set of concessions heaped onto the last until the final product varies significantly from its original performance parameters.

Consider, for instance, three teams working on various aspects of a transaction-based database application for a major online retailer. One team writes the interface, the next handles the logic, and the third focuses on the database. With each daily build, new code arrives from each team as they tackle different functions of the application. For instance, on Day One, the interface team may handle customer log-in, while the logic team focuses on browsing for specific products, and the database team zeros in on producing the item from the available online inventory. The Day One build most likely will contain a few performance latencies, but nothing earth shattering after just one day of writing code. True, they might fall outside the performance variances set when the project specs were originally determined, but at this point, the team is more interested in testing functionality than performance.

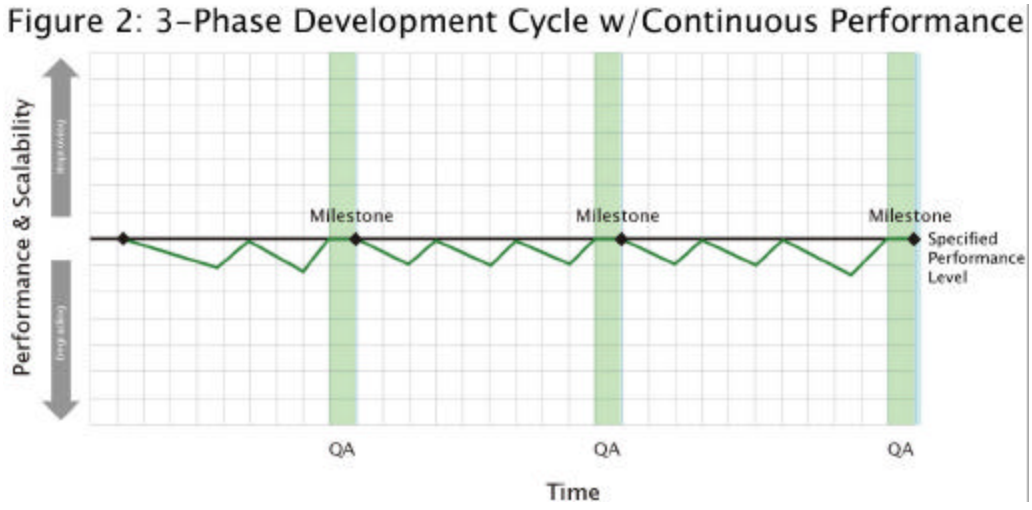
The danger lurking in that approach is that Day One's performance problems compound with Day Two's, and on and on, until a month passes by and the team finally takes a breather from writing code to test and tune performance. By that point, the small performance variances created with every day's build have become an enormous problem that takes a week to tune. Worse, the team endures this three times during the development life cycle. And each time, it becomes increasingly difficult to fix everything that is degrading performance. As a result, the final product falls well outside the performance parameters originally set for the application, or, the project team must spend more time and exceed cost budgets to tune the application closer to its intended performance specification. (See Figure 1.)

Figure 1: Typical 3-Phase Development Cycle



The Continuous Performance model, however, doesn't allow for this compounding effect because performance is tested and tuned with every build. This approach is particularly attractive to Java developers who already practice Continuous Integration, because the procedural steps required to add Continuous Performance to an ongoing development environment are relatively minor. It is merely a matter of accessing the right tools and ensuring that performance tests and tweaks are included in every post-build debugging. Yet Continuous Performance practices deliver benefits that are realized throughout the development process, vastly reducing the pain factor for development teams and resulting in better software on the back end. What's more, because they are spared week-long tuning nightmares, teams actually can deliver better code sooner. (See Figure 2.)

Figure 2: 3-Phase Development Cycle w/Continuous Performance



Understandably, developers seek out functionality that allows them to more easily build and test applications. Yet not enough tools are available to help developers tune their applications early in the development process. Worse, current solutions are bulky and expensive, and generally are designed less for development than for system testing or, at best, for operational functions such as simple performance monitoring.

Implementing Continuous Performance with IronGrid

For more than a year, the founders of IronGrid, Inc. have been looking for a solution to this problem. IronGrid's philosophy is to provide developers with specific problem-solving tools that are inexpensive, easy to install, and easy to use, that deliver immediate value, and can be seamlessly integrated in their development process. In this way, Continuous Integration strategies easily can be expanded to include Continuous Performance strategies. This concept underpins IronGrid's overall mission, which is to provide a series of low-cost Java tools that can be installed quickly and require no source code changes.

IronGrid's original founders created the P6Framework (www.p6spy.com) to validate these concepts. The P6Framework offered to the Open Source community a set of tools to test and improve the performance of database access in Java and J2EE applications. More than 10,000 developers have downloaded the P6Framework and related utilities. Many developers and even industry leaders like IBM and BEA Systems have contributed to this project and expanded it to support various J2EE platforms. The acceptance of the P6Framework by Java/J2EE developers convinced the founders of IronGrid to respond to the needs of the development community by creating a company dedicated to enabling Continuous Performance.

IronGrid recognizes that most development organizations buy extensive and costly software suites incorporating tools that developers neither need nor want. Instead, IronGrid focuses on making tools that solve specific problems, deliver immediate results, and can be seamlessly integrated into a development team's build strategy. IronGrid's flagship products increase the performance and scalability of Java-based applications.

IronEye SQL exposes all of the SQL being generated by an application and summarizes the information in a simple, easy to understand interface. Without changing a line of application code, a developer can instantly identify excessive or long-running SQL statement generated by code in development. Ant tasks identify code with database performance problems as part of the build process, enabling companies to implement Continuous Performance strategies throughout development. This ongoing visibility allows performance problems to be corrected early on in the development cycle, when performance tuning is dramatically less time-consuming and costly. IronEye SQL is available for download today at www.irongrid.com.

IronEye Cache represents the next level of performance enhancement tools from IronGrid. IronEye Cache further improves the performance and scalability of software through intelligent caching. This new tool allows developers to store frequently requested SQL statements in cache to minimize database calls and deliver commonly requested information quickly.

Available in Spring 2003, IronEye Cache provides a straightforward, "point and click" alternative to traditional caching systems, which can be complicated, invasive, and often times cause more problems than they solve. Using a console that displays all of the SQL

flowing between an application and a database, developers can select and cache any SQL statement simply with the click of a button and with no code changes. IronEye Cache also enables developers to tune performance and scalability on the fly by conducting "what if" caching scenarios. While not only giving developers a continuous view into SQL query performance, IronEye Cache also provides a fast, easy method for exploring caching strategies so teams can address performance issues immediately. This allows maximum flexibility for tuning code so that teams can seamlessly implement Continuous Performance strategies to deliver optimal code performance.

IronGrid's core product values underscore the company's admiration for the Open Source community and its approach to software evolution. The company is devoted to providing tools at low cost, with installation in as little as 15 minutes. IronGrid also encourages public bug tracking on its Web site, a tacit acknowledgement that IronGrid understands how crucial other Java developers are as sources of ideas for enhancements, fixes and new products. For an optional price, IronGrid also releases its source code to developers who have the time and resources to modify and improve the tools for their own use, and for the community at large.

Leveraging the terrific inroads paved by Continuous Integration, developers can make that best practice even better by addressing performance with every build. Indeed, it's not just a good idea: If Java is to evolve into a truly mature and robust development environment, advances like Continuous Performance will be vital. And once development teams see the benefits of tuning performance as they code, programmers will be happier, and software will perform better. And isn't that what a best practice is all about?

Cody Menard is Chief Technology Officer at IronGrid, Inc., in Austin, Texas. He has over two decades of experience in product design and engineering, with senior executive and programming positions at Covasoft, BMC Software and AT&T Bell Labs. He holds a bachelor of science degree in electrical engineering from Farleigh Dickinson University.

For more information about how IronGrid can help your development team achieve Continuous Performance with its growing line of Java Developer Tools, please visit us at:

www.irongrid.com

Or, contact us at:

IronGrid, Inc.
205 Brazos, Suite A
Austin, TX 78701

512.474.7400 phone
866.314.4766 toll free
512.474.7400 fax
info@irongrid.com

Resources

[1] Brookshear, Daniel. "Applying Murphy's Law to Java Development," April 19, 2002, InformIT. Retrieved from http://www.informit.com on Feb. 3, 2003.
[2] Bear Stearns Equity Research, "Enterprise Software Application Development Trends"; June, 2002.
[3] Williams, Lloyd G., and Smith, Connie U., "The Case for Software Performance Engineering." March 2002. Software Engineering Research and Performance Engineering Services.
[4] Williams, Lloyd G., and Smith, Connie U., "Five Steps to Solving Software Performance Problems." June 2002. Software Engineering Research and Performance Engineering Services.
[5] "Return on Investment Analysis of VA Assist Enteprise/J," Instantations, Inc.
[6] CyberAtlas, Dec. 22, 2000.
[7] "Nonstop E-Commerce," Forrester Research Inc. January 1999.
[8] "How to Make Java Applications Go Faster: O'Reilly Releases a New Edition of 'Java Performance Tuning,'" O'Reilly & Associates Press Release, January 30, 2003. Retrieved from http://press.oreilly.com/javapt2.html on Feb. 3, 2003.