SPK and Associates

www.spkaa.com
Ph: 888-310-4540

SPK and Associates
900 E Hamilton Ave, Ste.100
Campbell, CA 95008

Accelerating Your Software Build - A Customer Experience

Introduction

Your software compile turn-around time is one of the key metrics which determine your organization's productivity. Long builds equate to fewer opportunities (lower number of available cycles) to be able to add new features and focus on quality. The following describes one of our customer experiences in helping to accelerate their software build system.

Bad Things Sometimes Happen with Slow Builds

Engineers are very smart people. If they are on a tight schedule and the build cycle time is holding them back, they will find ways to work around it. However, sometimes the approaches taken have a negative impact on the overall process. Examples include:

- The Engineer may only build his/her piece and opt out of building the larger component. The risk incurred may include missing dependencies which ultimately could break the system (CM) build.

- The Engineer may hold his/her code until ready to build all changes at once. The risk incurred may include a lack of incremental change sets making debugging (or backing out) code more difficult.

- The Engineer may reduce creation and/or execution of unit tests. This may lead to reduced quality and the discovery of issues later in the overall process (which is more costly).

All these behaviors (and more) were occurring with this one particular customer ISV.

Setting the Stage

The customer's software product was being generated from a GNU makefile infrastructure. The code base was primarily C & C++. There were several build platforms in play but we focused primarily on the developer's core OS environment which was Redhat. The configuration management system was ClearCase. The build job was a single-threaded compile on a centralized resource. The overall build time was taking close to 6 hours. Engineers were provided with a "local build" for their team launching every night on this server. If all went well, the code was then promoted to formal CM integration build on a different server the next day.

The goal was to achieve a 30 minute or less build cycle. Doing so would allow the company to adopt iterative or agile development techniques.

Improvement - 1st pass

We are able reduce the local build turn-around time by roughly 50% by doing the following:

- We modified code hierarchy access process to leverage snapshot views from ClearCase instead of dynamic. By reducing the number of times the build needed to go back to the

"VOB" over the network, we gained some speed-up.

- And we moved the build to a multi-core machine with local disk and passed a "make –J2/3" flag to create some concurrency in our compile threads.

Next Steps – 2 Steps forward, One Step Back
We then hit a wall. Our "make –J" worked great for the 1st 2X/3X of speedup, but as we got beyond that (J4 +), our builds would break.

The source of the build problems was the lack of the identification of the compile's implicit dependencies (where the developer didn't fully specify them in the makefile).

To make matters worse, in some cases, we got an obvious broken build. But in other cases, we got an inaccurate, but apparently correct build which caused all sorts of headaches and very hard to debug.

The other problem we encountered with make –J4+ was with those buildmeisters who wished to continue leveraging the ClearCase dynamic view for source hierarchy access. The more threads (–J's) they used, the more file stats() were sent back to the VOB and View Server. That reduced the performance of both the VOB and View server.

It became very obvious that we were not going the get the build time down low enough using traditional methods (buying more hardware, using things like make –J, or trying to use build avoidance technologies like wink-in).

Improvement – 2nd pass
We were able to bring our builds down to 29 minutes (an additional 7X improvement), by leveraging Electric Cloud's ElectricAccelerator solution. Getting there required us to swap out the existing GNU make with ElectricAccelerator's emake. After some minor cleanup, we were able to migrate all of makefiles with no issues.

ElectricAccelerator achieves its impressive parallelization partly by monitoring the file access level, what files are read and what files are written, by every target in the build. From that it creates a dependency map. So, if something was run out of order in a later compile, it can build it quickly, while the build is still in flight, throw away that initial result for that target and rerun it so that the build is still guaranteed to be clean. That allowed us to push our compile to utilize 20 CPUs in parallel without breaking the build.

Another benefit is ElectricAccelerator's caching technology, which lowers the number of calls to the ClearCase VOB/View server.

Fine Tuning the Build
We were also able to utilize Electric Cloud's ElectricInsight companion product to ElectricAccelerator. The tool reads the output of Accelerator and gives you a visualization of the structure of the software build. The GUI shows you every job in the build and helps you identify dependencies, conflicts and serializations and other performance bottlenecks. This made it much easier to debug what commands were run, where they were run, and to visually see where the long poll opportunities for further parallelization existed.

SPK and Associates is a partner with Electric Cloud. Call us today to help speed up your build environment to improve your operations ROI.

Carlos Almeida
SPK and Associates
Architect, Software Engineering