Tom Was Right: Integration is Hard

James Larus
EPFL
Reps at 60
September 11, 2016
Google’s monolithic repository provides a common source of truth for tens of thousands of developers around the world.

BY RACHEL POTVIN AND JOSH LEVENBERG

EARLY GOOGLE EMPLOYEES decided to work with a shared codebase managed through a centralized source control system. This approach has served Google well for more than 16 years, and today the vast majority of Google’s software assets continues to be stored in a single, shared repository. Meanwhile, the number of Google software developers has steadily increased, and the size of the Google codebase has grown exponentially (see Figure 1). As a result, the technology used to host the codebase has also evolved significantly.

This article outlines the scale of that codebase and details Google’s custom-built monolithic source repository and the reasons the model was chosen. Google uses a homegrown, custom-built system to host one large codebase visible to, and used by, most of the software developers in the company. This centralized system is the foundation of many of Google’s developer workflows. Here, we provide background on the systems and workflows that make feasible managing and working productively with such a large repository. We’ll outline Google’s “trunk-based development” strategy and the support systems that enable a maintainable and healthy Google codebase healthy, including software for static analysis, code cleanup, and streamlined code review.

Google-Scale Google’s monolithic software repository, which is used by 95% of its software developers worldwide, meets the definition of an ultra-large-scale system, providing evidence the single-source repository model can be scaled successfully.

The Google codebase includes approximately one billion files and has a history of approximately 35 million commits spanning Google’s entire 18-year existence. The repository contains 86TB of data, including approximately

Google has shown the monolithic model of source code management can scale to an extremely large database of files and be used by tens of thousands of developers.

Benefits include unified versioning, extensive code sharing, simplified dependency management, atomic changes, large-scale refactoring, collaboration across teams, flexible code ownership, and code visibility.

Drawbacks include having to create and scale tools for development and execution, as well as potential for codebase complexity (such as unnecessary dependencies).

key insights
Google practices trunk-based development on top of the Piper source repository. The vast majority of Piper users work at the “head,” or most recent, version of a single copy of the code called “trunk” or “mainline.”

Trunk-based development is beneficial in part because it avoids the painful merges that often occur when it is time to reconcile long-lived branches.
I Couldn’t Help Thinking of This Paper

Programmers are often faced with the task of integrating several related, but slightly different variants of a system. One of the ways in which this situation arises is when a base version of a system is enhanced along different lines, either by users or maintainers, thereby creating several related versions with slightly different features.

The need to integrate several versions of a program into a common one arises frequently, but it is a tedious and time consuming task to integrate programs by hand. Anyone who has had to reconcile divergent lines of development will recognize the problem and identify with the need for automatic assistance.
First Step in a Larger Project

Program slicing, differencing, merging, chopping, “etc.”
A Bit of History

Tom and Susan started at UW in 1985 as professors

I joined UW in 1989 as a professor

Program slicing was Tom and Susan’s big research project

I left UW in 1997 to join MSR
Program Integration

Alice and Bob make concurrent changes to code
What Could Go Wrong?

Textual Conflict

```
x = x * 2;
```

```
x = x + x;
```

Compiler-Detectable Semantic Conflict

```
x = x << 1;
```

```
class foo {
  ...
  int unused;
  ...
  if (p == null) ...
  ...
  if (p != null) x = *p;
}
```

Compiler-Detected "unused"

Slicing-Detectable Semantic Conflict

```
p = malloc();
if (p == null) ...
...
if (p != null) x = *p;
```
Analogous to Concurrent Update Problem

- Write-Write Conflict
  - $x = 1;$
  - $y = x;$
  - $y = \&x; \ast y = 1;$

- Read-Write Conflict
  - $y = x;$

- Unanalyzable Conflict
  - $x = 1;$
  - $y = \&x; \ast y = 1;$
What To Do About This?

- Concurrency control
  - Use synchronization (lock or people) to serialize access
Microsoft, before Windows XP

SLM
(internal version of RCS)

- RCS
  - “Revision Control System” (Walter Tichy)
  - State of the art ~1980s
  - Single-file locking
    - No atomic commits
  - Branching unusable
Microsoft, after Windows XP

SourceDepot
Microsoft version of Perforce

Windows Repo (kernel)
Windows Repo (file system)
Windows Repo (utilities)
Integration window (3 days/team)
Modern (Git) Practice

“freshening integrations”  “essential integration”

atomic and conflict-free

Aside: closer to transactional memory (optimistic concurrency) than locking
How Do We Know Integration is Correct?

Yes, software testing
Program Slicing

• Not used in practice
• Does not come up in discussions in SE
Testing \(\sim\) Semantic Analysis?

- When you can’t perform static analysis (too slow, too imprecise), then test
- In practice, testing is still the primary way developers find bugs and ensure code quality
  - Not the only way

What does this say about SW?
What does it say about Tom’s research?
1980s -> 2010s: Enormous Progress

1980s
• lint and compiler warnings were primary development tools
• Interprocedural analysis was not well understood or widely used
• Pointer analysis not well understood
• PDGs just published
• SAT was the canonical NP-complete problem
• 8MB was a large memory and 300MB was a large disk

2010s
• Large number of open source and commercial tools
  • MS: SLAM, Dafny, Z3, Pex, Slayer, ...
  • Even some viable companies
• Program analysis well grounded and understood
  • Thanks Tom!
• Main memory in GB (heading to TB), disk in PB (heading to EB)
• Development tools still not well understood or widely used
Why Are Bugs Not Found By Tools?

• Legacy languages
• Legacy software
• Non-zero false positive / false negative rates
• Developers’ inability to understand / write specifications
• Diversity of bugs
• Fixing existing code is a fool’s errand

• Not because of immaturity of program analysis
Tom’s Role

• Tom’s contributions to program analysis are broad and fundamental
  • Interprocedural analysis
  • Pointer and shape analysis
  • Multi-threaded program analysis
  • Path problems
  • Model checking
  • …

• Without Tom’s research, the field of program analysis would be on a far shakier and less rigorous basis
Aside: Hope on the Horizon

- Verified software
- CompCert (Inria), seL4 (UNSW), IronFleet (MSR)
- Co-design and verification

The ability to change and rearrange code in discussion with the design team (to predict performance impact) was an important factor in the verification team’s productivity.
Conclusion

• Tom’s contributions to program analysis are broad and fundamental
• We aren’t there yet
  • But, we are a lot further along because Tom’s insights and contributions

• Tom’s many other virtues
  • Leader and role model in his field
  • Extraordinarily well-written papers
  • Excellent advisor with many talented and successful students and postdocs
  • Generous colleague
But

• Are you really 60?