Living Downstream
Without Drowning

LLVM Dev Meeting 2015
Paul Robinson & Mike Edwards
Sony Computer Entertainment
TERMINOLOGY

➢ Upstream = llvm.org project
➢ Downstream = project with your changes
➢ Local change = one of your changes that will (or could) go upstream
➢ Private change = one of your changes that you intend NOT to send upstream
  ➢ All private changes are local
  ➢ Not all local changes are private
Open-source Commit Data for 2014
➢ For LLVM+Clang specifically, ~50 commits/day
➢ Just a bit higher now...
➢ Plus: compiler-rt, compiler-tools-extra, libcxx, lld...
➢ Another ~20 commits/day
Sony’s historical local changes
➢ Some big lumps (X86-64 instruction subsets)
➢ Some smaller but still intrusive features (partial re-initialization; dllimport/dllexport for ELF)
➢ Toolchain stuff (driver)
➢ Default C++11 (trivial coding, >100 tests)
➢ Misc other stuff
THE TSUNAMI HITS

➢ Rebase from LLVM 2.9 to 3.0
➢ ~8 months of upstream changes
➢ Took 3 months of my time (+ help) to finalize
➢ 4-way merge review

Upstream old
Upstream new
Our old
Our new
LEARNING TO SWIM

- Dog-paddle
  - Pull from llvm.org every 2-3 months
  - Still took ~1 month each to finalize
- Moving with the current
  - Patch tactics
- Need a life-boat
  - Automation
Consider long-term project direction
➢ Do the big feature
➢ Do the re-design/refactoring
➢ Textual consistency of source basically irrelevant
   ➢ clang-format pretty much expected
➢ Although gratuitous churn considered unfriendly
MAINTENANCE MODE

Maintenance, sustaining, continuing engineering…
They all mean the same thing

➢ Fix the bug as safely as possible
➢ Minimize risk of introducing a new bug
➢ “Surgical” fix
➢ Smallest possible change, textually and functionally
➢ Very limited use of this in LLVM
LONG-TERM LOCAL CHANGES

DO

➢ Maintenance mode is your friend
➢ Minimize textual scope of changes
➢ Create a subclass for your special behavior
➢ Put local tests in local (new) files

➢ Use “local change made here” comments
➢ Diffs provide better info
➢ Bug reference helps archaeology
➢ Distinguish your changes from mistakes
Diff for a function we added to Path.h

```c
/// SCE: begin bug 2844.
+#ifdef LLVM_ON_WIN32
/// @brief Converts a string to UTF8 encoding and prints it to an output stream.
/// @param InputStr Input string to convert.
/// @param OutputStr Output stream.
/// @result True if the conversion succeeded.
bool convEncExternalToUTF8(const StringRef InputStr, std::string& OutputStr);
+#endif
/// SCE end.
```
CHANGE TAGS EXAMPLE 2

➢ From a recent conflict report

```cpp
+++<<<<<<< HEAD
+#include "llvm/ADT/Triple.h"
+#include "llvm/MC/MCDirectives.h"  // SCE: bug 10867
++++=======
+ #include "llvm/ADT/TargetTuple.h"
+>>>>>>> opensource
```
LONG-TERM LOCAL CHANGES
DON’T

➢ Never delete upstream code (use #if 0)
➢ clang-format is your enemy
➢ Textual consistency of the source is crucial
➢ Avoid the end of the file/namespace/class
PUSHING CHANGES UPSTREAM

➢ If you possibly can, do it upstream first!
➢ Or, you undo all the no-merge-pain tactics
  ➢ Do the refactor
  ➢ Do the reformat
  ➢ Put the change where it belongs
    ➢ Even at the end of the file/namespace/class...
  ➢ Integrate into existing tests where it makes sense
  ➢ Upstream review is a Good Thing™
• Automation is our lifeboat
• by implementing Continuous Integration and a phased building approach we’re working to get a handle on constant flow of commits.
• Automation helps us to be effective at merging commits and build/test
• Through Healthy Investments in infrastructure we can build and test extremely quickly!
5 Add'l Clicks: How Do We Navigate The Stream…

• Into The Boat - Begin the process of integrating upstream work w/internal patches - no automation here yet!
  • by working through the process manually we are able to pinpoint exactly what automation will work best for us when we are ready to implement
• Starting The Motor - when we notice volume of commits occurring upstream
  • Try to automate things we think are worth automating
  • Apply automation quickly but try to avoid unintended consequences
• Panic - Keep engineers involved with the process of dealing with merge conflicts - Merge Pain
• Underway - Employ Continuous Integration - bot which attempts merge and files tickets if merge not clean
• Headed Upstream - Get to a place where integration with upstream happens on a consistent regular basis, with minimal human intervention
5 Add’l Clicks:

- Start with commits to the llvm.org master branch
- After a passing build/test we import those commits to our opensource branch, which is a merged tree of llvm, clang, compiler-rt and lld
- Our staging branch is used to merge upstream commits with commits from our private branch which is represented by the bottom line labeled master
- End goal is to reduce the iteration time so we can eventually automate as much as possible.
• We are still doing manual bulk merging
• An incorrect merge would be really bad and cause much unnecessary work
• The Gorilla represents Paul R. as he is the one tasked with protecting our branches from the rest of us doing something wrong.
• Our opensource branch is the only branch currently managed 100% by a bot
  • This allows us to gather more data to ensure automation deployed is as close as possible to what we need
Earlier this year we were able to contribute new hardware to the community
• This bot builds our triple
• Acts as first line of defense against new commits breaking
  • llvm, clang and lld and of course our triple
• Runs on Ubuntu - on average typical build is ~ 3min and test is 16 sec.
• Windows bot TBD - hopefully by end of the year
Phase 1
- Release Build
- No Assertions
- Target X86 Only
- No Test
- Dirty Object Directory

Phase 2
- Release Build
- With Assertions
- Target X86 Only
- Tests
- Clean Object Directory

Phase 3
- Debug Build
- With Assertions
- Target X86 Only
- Tests
- Clean Object Directory

Phase 4
- Debug Build
- With Assertions
- All Targets
- With Tests
- Clean Object Directory

INTERNAL BUILD PHASES

LLVM, Clang, Clang Tools Extra, Compiler-rt & LLD

~27 sec.
~6 mins.
~8 mins.
~9.5 mins.

5 Add’l Clicks
- Four phased build approach on open source branches
- Building on Linux, Mac and Windows Hardware to maximize coverage
- Phase 1 - Fast - RO - Dirty - X86 Only ~27sec
- Phase 2 - Turn on Asserts and Tests, Clean - ~6min
- Phase 3 - DA & Clean - ~8min
- Phase 4 - DA & Clean & All Targets - ~9.5min
- Commit total travel across phases ~25min total
- Private branches use 3 phases because we omit Phase 4
8 Addl Clicks!

- Start with commit from upstream and we build & test it
- On completion a build asset is produced
  - if success an asset is a built compiler with its' supporting files
  - if failure an asset is mostly just a collection of log files
- A failing asset triggers a notification and the process just goes back to waiting for the next commit
- A successful asset is archived in a git repository
  - Why git - it was already available and we knew how to use it - a simple solution
  - Also, git provides the ability to easily search for an asset later on
- Once archived an asset is published via internal API
  - Makes asset available to other processes, internal teams and clients
  - Then trigger any remaining phases and go back to waiting

BUILD PIPELINE
• Engineers like to see green bots!
• Developed internally by one of our colleagues
• Each Column is a commit
• Each Row is a specific test run
• Layout is very dense, however Engineers are able to quickly find their commit and see how it is performing across all of the tests
• Each icon is a hyperlink to a detail page for that test run
- Our Merge Pain Tracker
- Developed internally as well
- Used to help keep track of conflicts we accumulate in between merges of our opensource and staging branches
- This assists us in surfacing the code which requires the most human attention and allows us to focus on getting that code upstream
- Over time we hope this tool will help us realize a healthy reduction in the amount of time it takes to resolve merge conflicts
• We have tools...we have process..
• Maybe we should just...
• Automate all the things!
• Sounds like a good idea, right?
• As it turns out, that's not exactly true
• Trick w/ automation is can’t allow automated processes to outpace the humans ability to keep up with it.
• Classic example is the “I Love Lucy” episode where Lucy & Ethel are working in the chocolate factory
• Conveyor goes crazy and chocolate ends up everywhere
• We wanted to avoid a similar scenario if we turned on 100% automated merging of llvm.org to staging and master without human intervention
• Paul would have pushed me out of the lifeboat
• Bots would turn red
• Engineers would be flooded with failure notifications
• and would eventually lose interest and bots would be forgotten about.
• Morale of the story, invest in infrastructure, automate as much as you can, but make sure your humans can keep up!
• A little bit about us, what we do and how we got here
• Now it’s your turn
• How do you deal with merge pain?
• How much automation have you deployed?
• How do you deal with private branches?
• Comments, question stories?
THANK YOU!

Paul Robinson
paul_robinson@playstation.sony.com

Mike Edwards
michael_edwards@playstation.sony.com