Tab 1

Equality, Debug Serialization, and Size Estimation in Protobufs, LITE_RUNTIME

Dan Murphy
modified: Apr 10, 2025
PUBLIC DOCUMENT

This is a very brief summary of my exploration for how to add non-hand-rolled functionality for the protobufs in the web_applications/PWA area of Chromium.

Background

protobufs, chromium uses proto2.

Protobufs

Protocol buffers are used in Chromium & other places to store serialized data.

Chromium enforces that only the "LITE_RUNTIME" version can be used in its binary, to reduce binary size.

- This means that the implementation has less functionality - no reflection
    - No generated equality method
    - No generated ToDebugString methods
    - No generated EstimateMemoryUsage methods
    - no reflection, etc.

PWA Code

The PWA code stores protobufs in a database, and we require equality checks as well as debug printing for our chrome://web-app-internals page. Currently we:

- Copy protobuf data to our own class (web_app.h)
- Custom-roll ToValue methods
- Custom-roll equality
    - When we do store a proto, we call SerializeAsString for the whole proto, and compare that.

Sync Code

The sync code has requirements too:

- ToValue creation for debug output & reports etc
- Size estimation

It does this via generating a visitor capability pattern for each proto. This allows the two functionalities to be implemented via visitors that get called for each field. This does not work for equality.

Use Cases

- PWA code
  - Needs ToValue
  - Needs equality
  - The hand-rolled code and duplication of fields is silly.
    - We could least try to re-use sync code for generation of these.
    - However, the sync pattern doesn't allow for equality generation
- Sync code
  - Generated code is ok, but binary size could maybe be reduced.
  - Needs ToValue
  - Needs size estimation
- Anecdotes from pkasting@:
  - Other teams have needed this for a feature, but because it wasn't there, just didn't do the feature.
  - There are likely other places that hack this / custom-roll code for this.

Problem: Inconsistent `--proto-path` when generating proto files

I attempted to move all of Chromium from LITE_RUNTIME to SPEED or CODE_SIZE, so that the full proto `Message` type can be used which has the required functionality.

The main problem I hit is one with how we compile protobufs and how they reference each other:

- non-LITE_RUNTIME implementations use `descriptor_table_*` symbol definitions. The `.pb.h` file declare these, and they are defined in the implementation file
- When referencing other proto files (proto files include other proto files), non-LITE_RUNTIME generated C++ references the other proto implementations using these symbols.

- The `descriptor_table_*` format depends on the proto file name and relative location to the `--proto-path` dir passed to `protoc`
    - Examples for `content/browser/background_fetch/background_fetch.proto`
        - 1) with `--proto-path` being the root directory, this would be **`descriptor_table_content_2fbrowser_2fbackground_5ffetch_2fbackground_5ffetch_2eproto`**
        - 2) with `--proto-path` being `content/browser/background_fetch/`, this would be **`descriptor_table_background_5ffetch_2eproto`**
- When the generated code references other protobuf files...
    - in chromium it seems to assume that 1) is true for all referenced proto files - it expects the symbol to be fully qualified with the path to the project root
    - however, almost always, 2) is the case, where the declared symbols in that other proto file don't have the path.
- Put another way, to have all of the cross-file imports work correctly, the `--proto-path` dir must be consistent for all proto generation. This is not the case today

Attempting to fix this in `third_party/protobuf/proto_library.gni` leads to a number of problems with cpp output location, etc. It became too hard and time consuming for a project I don't have that much time to work on to fix.

Possible fixes:

- Perhaps there is a way to know, for each dependency proto file, the symbol it defines (somehow protoc would have to be able to take in per-proto-file configuration? likely not)
- Perhaps there is a way to force all protoc calls to use the root as the path, and not mess anything else up?
    - likely complicated due to other random projects that use it... unclear.

The only way forward would be to fix all of the proto files without a --proto-path specified. This seems possible to do incrementally.

Proposal - Write a protoc plugin to generate files with functionality

See the existing proto-to-value converter here: https://source.chromium.org/chromium/chromium/src/+/main:components/safe_browsing/core/common/proto_to_value/

I will rename proto_to_value to proto_supplements (or proto_extras, WDYT?), have each part be options designed to be 'false' by default:

- omit_value_serialization (default false)
- generate_stream_operator (default false)
- omit_equality_operator (default false)

It will also be modified to support proto2.

If binary size becomes an issue, we can create 'minified reflection' for these protobuf messages, allowing the added functionality here to use that reflection instead of completely denormalizing the functionality. That can further reduce binary size.

Alternative
- Create a separate target/template for the equality conversions
    - proto_to_value (existing)
        - modified to support proto2 and also generate stream operators (optionally)
    - proto_comparison
        - generate comparison operators for protobuf messages.

alternative options

- generate_value_conversion (default true)
- generate_stream_operator (default false)
- generate_equality_operator (default true)

Alternative - Allow devs that need more functionality to use compile-time-generated 'extras' type, which creates the missing functionality

Prototype CL

Do a version of what sync did, but
- Make it generate the required functionality at compile-time instead of having dynamic visitors.
- Hopefully replace the sync stuff with this?

This should limit binary size impact:
- We get to delete hand-rolled stuff, which at worst is a non-impact
- We get to delete the dynamic nature of the sync generated things, which might remove symbols / binary size too

This also was pretty easy & contained.

Also easy to 'upgrade' to something else later - as it's all hidden behind a type.

Alternative - Incremental change to use protobuf full and CODE_SIZE by default, and SPEED when necessary

CODE_SIZE is better for binary size when message count is high, vs LITE_RUNTIME

- email

An investigation is done in the Proto Full Investigation tab, using this patch.

The conclusions, trying to do the most here to save binary size:

- Upfront binary size cost: ~940kb (850+90)
- per-proto-message savings for usage of extras:
  - .pb.o message: -0.5kb
  - base::Value conversion: -2kb
  - memory estimation: -0.1kb
  - equality: -0.77kb

What does this mean for each one individually?

- 1900 proto files need
- 420 serializes needed
- 1220 equalities needed
- 9400 memory estimations needed.

The current patch (size comparison) has 474 proto messages, 361 conversions, 260 memory estimations, and 5 equalities, and thus we are at 560kb left!

However:

- Compile size & android binary size grow quite a bit
  - Android Binary Size +125,245 bytes
  - Android Binary Size (arm64 high end) (TrichromeLibrary64.apk) +339,423 bytes
- Many static initializers are added in the code, which can make startup time increase. Unclear how acceptable this is. One initializer per protobuf message or file?
  - It's hard for me to find this again without recompiling the patch above all over - so unclear how avoidable this is / what it would take to remove these

Pros

- Easier to write functionality on top of protobufs, we can mostly remove this 'extras' project.

- Possibly reduction of desktop binary size if all protobuf message are moved to CODE_SIZE, and only a few stay as SPEED.

Cons

- Seems hard to avoid the Android build size increase unless a bunch of expensive/large protobuf messages are found that have hand-rolled serialization/equality situations.
- Static initializers seem bad?

Alternative - Use (internal) cc_proto_descriptor_library, as referenced here, as a way to generate a `Message` type from a `MessageLite`

This uses the message descriptor data, would require generating descriptors (which is currently supported by the gn build method for protobuf libraries, at least)

Design doc

This seems to require extra stuff

- Import this into third_party (not public at all yet, currently internal only?)
- Get it building
- This requires outputting the descriptors for proto files that we need this for, which is possible as that's an option for the GN proto_library build
- Unclear binary size cost.

Proto Full Investigation

Background

proto files

- web_applications proto files: 50 messages
- background fetch: 9 messages
- web_app_specifics: 2
- total: 61 messages

Experiments

Moving web_applications protos to SPEED

linux binary size analysis

increases:

- ~800kb: Moving to full (aka `Message` instead of `MessageLite`) increases third_party binary size by
- ~120kb: global ELF data increase (not sure which this should be attributed to)
- ~25kb: protobuf files in web_applications
- ~3kb: protobuf files for background_sync
- ~2kb: protobuf file web_app_specifics.proto in sync
- protobuf total: 30kb

kb increase per message?

- 30kb / 61 = **0.5kb**, over LITE_RUNTIME

Moving web_applications protos to CODE_SIZE

linux binary size analysis

increases

- (same) ~800kb: Moving to full (aka `Message` instead of `MessageLite`) increases third_party binary size by
- (same) ~120kb: global ELF data increase (not sure which this should be attributed to)

decreases

- ~32kb: protobuf files in web_applications
- ~8kb: protobuf files for background_sync
- ~1kb: protobuf file web_app_specifics.proto in sync

kb decrease per message?

- 0.6kb, over LITE_RUNTIME (1kb over SPEED)

Migrating the custom ToValue and equality code to reflection:

linux binary size analysis

- 2 or 3 tovalue methods
- equality for all of the os integration, and some sub-ones.

increases

- (more) ~850kb: Moving to full (aka `Message` instead of `MessageLite`) increases third_party binary size
    - includes, likely, the reflection class used to convert toValue (~100kb extra), but that is a one-time cost.
- (same) ~120kb: global ELF data increase (not sure which this should be attributed to)

decreases

- ~50kb: protobuf files in web_applications
    - cost of manual serialization
        - web_app_os_integration_state.to_value.cc -8.87 KiB
            - 1 message
        - web_app_os_integration_state.equal.cc-5.61 KiB
            - 7 messages
        - web_app.to_value.cc-2.32 KiB
            - 2 messages
        - web_app.equal.cc-0.86 KiB
            - 2 messages
        - **~18kb** total? matches, as 32 + 18 = 50
    - 32kb proto files savings
- ~7kb: protobuf files for background_sync
- ~1kb: protobuf file web_app_specifics.proto in sync
    - web_app_specifics.equal.cc-1.51 KiB
        - 2 messages
    - web_app_specifics.to_value.cc-0.86 KiB
        - 1 message

using reflection for tovalue:

- 4 messages, 12.3kb savings
- 3kb / message savings doing reflection instead of custom toValue

for equals

- 11 messages, 8.5kb savings
- 0.77kb / message savings doing reflection instead of custom equality.

Conclusions

- Upfront cost of protobuf full, and reflection stuff used for ToValue and Equality calculations:

Adding in sync, remvoing their visitors & using TotalSizeLong & reflection tovalue support

Sync protobufs:

- 260 messages

(before, only 60)

super size results

increases

- (more) ~852kb: Moving to full (aka `Message` instead of `MessageLite`) increases third_party binary size
  - includes, likely, the reflection class used to convert toValue (~100kb extra), but that is a one-time cost.
- (more) ~260kb: global ELF data increase (not sure which this should be attributed to)
  - This is +140kb from last time

decreases

- protobuf saving
  - 380kb savings total
  - to_value savings vs manual rolling
    - proto_value_conversions.cc-108.82 KiB
  - memory estimation savings
    - proto_memory_estimations.cc-22.44 KiB
  - 250kb savings protobuf files.
  - 130kb savings for handrolling -> reflection
- 250/260 = ~1kb savings per proto message

Conclusion for reflection for equals/size/tovalue

- tovalue
  - original 4 messages, 12.3kb savings
  - new 260 messages, 108kb savings
  - ~0.45kb savings per message
- memory estimation

        - 260 messages, 22kb savings
        - ~0.1kb savings per message
- equality
    - 11 messages, 8.5kb savings
    - 0.77kb / message savings doing reflection instead of custom equality.

proto message savings vs LITE_RUNTIME

- (250kb + 32kb) / + (260 + 60) = ~0.9kb per proto message savings.
- but - increase of ELF?
- 120kb for 61 messages, 261 for 320
    - mathing that means:
    - ~**90kb** is static cost, likely due to protobuf library
    - ~0.5kb per proto file?
- So - about ~0.5kb per proto file

Also updating safe_browsing

analysis

extra proto messages:

- 139 + 15 = 154

Increases

- (more) ~852kb: Moving to full (aka `Message` instead of `MessageLite`) increases third_party binary size
    - includes, likely, the reflection class used to convert toValue (~100kb extra), but that is a one-time cost.
- (more) ~315kb: global ELF data increase (not sure which this should be attributed to)
    - This is +140kb from last time

decreases

- protobuf saving
    - 200kb savings total
    - to_value savings vs manual rolling
        - csd.to_value.cc-35.02 KiB
            - 65 in csd
        - realtimeapi.to_value.cc-5.66 KiB
            - 9 messages
        - safebrowsingv5.to_value.cc-1.17 KiB
            - 5 messagse
        - connectors.to_value.cc-8.58 KiB
            - 14
        - total

            - 51kb
    - savings protobuf files.
        - 151kb

Conclusions

Conclusion for reflection for equals/size/tovalue

- tovalue
    - original 4 messages, 12.3kb savings
    - new 260 messages, 108kb savings
    - new 93, 51kb savings
    - total: 2kb / proto savings. (not including ELF)
        - 357/171
- memory estimation
    - 260 messages, 22kb savings
    - ~0.1kb savings per message
- equality
    - 11 messages, 8.5kb savings
    - 0.77kb / message savings doing reflection instead of custom equality.

proto message savings vs LITE_RUNTIME

- (250kb + 32kb + 151kb) / (260 + 60 + 154) = ~0.9kb per proto message savings.
- but - increase of ELF?
- 120kb for 61 messages, 261 for 320, and 315 for 474
    - gemini question
        - for the formula a + bx = y, with data points a + b*61 = 120, a + b*320 = 261, and a + b*474 = 315, approximately solve for a and b using a line of best fit
        - > Therefore, the approximate line of best fit for the given data points is y = 95.2 + 0.48x, where a ≈ 95.2 and b ≈ 0.48.
    - 95.2kb is the static cost
    - 0.48kb is the per-proto-message cost
- So - about -~0.5kb per proto file

Conclusion for cost calculations

- Upfront binary size cost: ~940kb (850+90)
- per-proto-message savings
    - .pb.o message: -0.5kb
    - base::Value conversion: -2kb
    - memory estimation: -0.1kb
    - equality: -0.77kb

What does this mean for each one individually?

- 1900 proto files need
- 420 serializes needed
- 1220 equalities needed
- 9400 memory estimations needed.

The current patch has 474 proto messages, 361 conversions, 260 memory estimations, and 5 equalities - 560kb left!

However:

- Compile size & android binary size grow quite a bit
    - Android Binary Size +125,245 bytes
    - Android Binary Size (arm64 high end) (TrichromeLibrary64.apk) +339,423 bytes
    - Compile size: Delta: +3.35 GiB (+3596283162)
        - not as big of a deal, this is just text size.