Tab Freezing + BFCache

Tracking IPCs

This Document is Public

Authors: chrisha@chromium.org
Last Update: August 2019
Status: Final | Draft

Status

Work finished, results of analysis available here (discussion here).

- Add IPC handler program counter to PendingTask and TaskAnnotator. (Landed)
- Integrate PendingTask::ipc_program_counter into logging messages. (Landed)
- Add IPC program counter decoration to all IPC handlers. (Landed)
- TaskAnnotator: Remove use of ThreadLocalOwnedPointer. (Landed)
- Integrate IPC decoration into tracing. (Landed)
- Add trace event when IPC-caused task is posted to a frozen task queue. (Landed)
- Add lifecycles tracing category to BENCHMARK_RENDERERS. (Landed)

One-page overview

BFCache (in progress) and tab freezing (shipping) are both efforts that revolve around freezing a tab for considerable periods of time. They both suffer from a similar problem: an accumulation of tasks caused by inbound IPCs. The reasons are many, but the symptoms are typically the same: an inbound IPC gets processed and indirectly results in a task being posted to a task queue that is associated with a frozen frame. The tasks continue to accumulate, resulting in a kind of memory leak. In the case of BFCache there are also privacy considerations related to this problem. Imagine a tab that has access to the geolocation API, but this tab has been navigated away from and inserted into the BFCache. While in the BFCache position updates continue to be delivered to the frame, accumulating in a task queue. If the user navigates back to the page that task queue will start processing events again, and it will have access to all of the position information of the device covering the period when the page was in the cache. This goes against user expectations, as from their point of view the site was unloaded during this time period.

We would like to identify the set of IPCs that lead to tasks being posted to frozen task queues. Once identified the long term plan is to endow the remote endpoints with knowledge of lifecycle states, and to avoid dispatching the IPCs in the first place.

Summary

We will be adding tracking of inbound IPCs by message type, and tagging any tasks posted during the context of handling the message. These tags will similarly be applied to any downstream tasks posted from the original labeled task. The tracking will involve a small amount of instrumentation in IPC message processing code, a TLS entry for tracking the active IPC message context, storage in PendingTask, plus a small amount of code for propagating message contexts in TaskAnnotator. Additional logic will be added to the renderer scheduler to detect when a task is being posted to a frozen task queue, and to log the IPC message that caused the task to be posted. This will be integrated into tracing, allowing this data to be collected in the wild from the canary channel. Similarly, it will be integrated into various pieces of logging to surface this additional context to developers where it is generally useful.

See a related discussion and design by eseckler@ here.

Platforms

All (some portions will not be realized on iOS)

Teams

chrome-catan@google.com, chrome-bfcache@google.com

Bug

950668

Code affected

base/logging.cc
base/pending_task.(cc|h)
base/task/common/task_annotator.(cc|h)
base/task/sequence_manager/task_queue_impl.cc
ipc/ipc_message_macros.h
mojo/public/tools/bindings/generators/cpp_templates/interface_definitions.tmpl
mojo/public/tools/bindings/generators/cpp_templates/module.cc.tmpl

---

Design

Storing The Data

We propose adding a single new pointer field to PendingTask. Additionally, we propose augmenting TaskAnnotator to allow setting and clearing a current IPC message context. The TaskAnnotator implementation will be responsible for decorating a task with the current context via TaskAnnotator::WillQueueTask. Similarly, decorated tasks will set the context (causing other posted tasks to be decorated) in TaskAnnotator::RunTask.

Instrumenting IPC Entry Points

The next step is to instrument the entry point. Both the legacy and Mojo IPC mechanisms have code generation choke points that are relatively easy to augment. Both of these mechanisms work via a switch statement which dispatches messages to the various registered message handlers. We propose augmenting the generated switch statements with code that wraps the message handling and sets/unsets a decoration using TaskAnnotator. To make this easier to manage we propose creating a ScopedIpcMessageTaskAnnotator helper that directly manages the TaskAnnotator TLS data.

There are a few alternative data representations that can be considered:

- ***Strings.** Specifically, a pointer to a static string that uniquely identifies the interface name and the message name, or to the file and line number. This does not require symbolization but it does require that the strings be statically linked into the binary. There is a general desire to avoid introducing new strings into the binary, although this might be the simplest approach for debug builds.*
- ***Instruction pointer.** This is easy to collect, and compact. Unfortunately, the information is opaque and requires symbolization in order to be actionable. Additionally, it is not directly aggregatable across Chrome versions, as the relative PC changes from one build to the next. The slow report, tracing and crash pipelines already have symbolization support[1], but it will require a non-trivial amount of work to convert PCs to message names.*
- ***String hashes.** It is possible to encode the context using a compile-time (constexpr) hash of the message name. This requires a custom symbolization step, but the raw data is trivially aggregatable across versions. It is also possible to provide a lightweight symbolization tool directly into the Chromium build for developers to use, and even to integrate the symbolization directly into debug build logging.*

[1] The pipeline currently supports function-name / source file symbolization, but not line number information. Adding line number support is expensive in terms of resources consumed, and would require substantial work. An additional symbolization step is still required to convert the filename/line-number data to a message name, involving access to the exact source code used to build the binary plus some custom parsing.

Impact on Binary Size

The new instrumentation results in a size increase on most platforms of ~52kB. This is identical to the size increase associated with esecker@'s initial related proposal. The following numbers are for official Chrome branded builds, where the instrumentation uses instruction pointers.

Platform	***Binary***	***Before (bytes)***	***After (bytes)***	***Delta***
Android ARM	*libchrome.so*	*49,456,032*	*49,509,280*	*53,248 (0.10%)*
Win32	*chrome.dll*	*54,123,520*	*54,168,064*	*44,544 (0.08%)*
Win32	*chrome_child.dll*	*80,398,336*	*80,451,584*	*53,248 (0.06%)*

A new trace event will be added to log the IPC message associated with a posted task as it is run, by extending code in TaskAnnotator::RunTask. The information will also be integrated with logging, so that fatal errors will also include IPC message context on the console.

It is expected that the proposed annotations will cover a majority of the code paths that lead to tasks being posted to a frozen task queue in a renderer. However, it is entirely possible that the IPC message being processed doesn't result in a PendingTask being posted, but rather it results in some other unit of work being posted (imagine a custom queue of unannotated callbacks, that is drained and processed periodically). The end result would be unannotated tasks being posted to a frozen task queue, for which no IPC message could directly be blamed. We propose to additionally introduce an UMA histogram which counts tasks posted to frozen task queues, counting those with and without annotations. If the relative number of undecorated callbacks is much higher than expected we could consider extending the decoration to base::Callback and base::Bind directly to increase coverage, rather than having it live in PendingTask. Note that this mechanism isn't perfect either, as pending work can be represented and queued using an unlimited number of mechanisms. We anticipate that the PendingTask decoration will cover the vast majority of common cases.

Getting the Instruction Pointer

We compared using base::GetProgramCounter() (calls a function which returns the return address via eax) and the GNU C++ extension that allows taking addresses of labels (ends up being "mov eax, CONSTANT"). In terms of binary size the two methods are effectively indistinguishable, with the label technique incurring relocation table entries. The label technique is marginally more efficient at runtime (avoids one call per IPC message handler), however given that it relies on a custom compiler extension it is less portable to use it.

On 64-bit a further alternative would be to use the rip-relative addressing mode of the "lea" instruction, but other than avoiding a call this incurs the same cost in code size.

Instrumenting Task Queues

It remains to detect when a task is posted to a frozen task queue. Ultimately, all tasks posted to the sequence manager end up in a TaskQueueImpl, which itself is enabled or disabled via various mechanisms. We proposed modifying TaskQueueImpl to emit a trace event if an IPC message decorated task is posted to a disabled task queue. This is known immediately after the call to TaskAnnotator::WillQueueTask.

We similarly wish to generate a list of IPCs that cause messages to be posted to a non-frame-associated task queue. This will aid in refining renderer scheduler logic to ensure that all frame-associated IPCs end up in the appropriate frame-associated task queue.

Crash Integration

We propose to integrate with logging.cc to ensure that both stack and task traces emitted to stdio include IPC message information. Furthermore, we propose integrating with logic in task_annotator.cc to ensure that the IPC entry point is included as part of the task backtrace, ensuring it is on the stack in the scope of a running task and included in crash reports.

Analysis

The proposed design will result in trace events being emitted as tasks are posted to task queues, including necessary queue metadata (frozen state, for example). A random sampling of representative traces is currently gathered by the slow reports back-end. These traces can be symbolized and analyzed in aggregate, and lists of IPC messages compiled.

Alternatives Considered

Other than the alternatives discussed in this document, one additional alternative was explored in order to reduce binary size impact. Rather than adding a ScopedIpcMessageDecorator within the context of each message handler dispatch, a single point of instrumentation was added to each dispatch function (identifying the message class) which also stored the enum ID of the message being handled (identifying the message itself). This results in about 50% less generated code, meaning the impact to Chrome is about 25 kB rather than 50 kB. However, in order to symbolize this information it would require a custom build to aggregate all message IDs, an upload to a centralized server for all official builds, plus a custom symbolization code path.

It is also conceivable to imagine generating the binary with a 4 GB virtual unmapped section where the IDs can be treated as offsets into that section, and IPC symbols generated for this binary as a

custom build step. This would allow the existing symbolization mechanism to work out of the box, but would still require custom build steps.

Metrics

Success metrics

TODO

Regression metrics

TODO

Experiments

TODO

Rollout plan

TODO

Core principle considerations

Speed

The proposed changes cause a small amount of additional state to be gathered and maintained on every task posting and every IPC message being processed. This amounts to adding an additional pointer to a data structure that is roughly 25 pointers in size. The expected overhead is very minimal, but profiling is required to ensure that performance is not impacted on what amounts to a critical code path. This will be validated via benchmarking, the performance waterfall, and finally monitoring of the feature in the wild.

Security

The proposed changes do not introduce any new attack surfaces, as they are all minor extension to existing code paths.

Privacy considerations

The suggested changes are causing a minor amount of additional information to be logged and reported in traces that are being collected from users in the wild. The state can typically be inferred via manual inspection by a developer, and messages with related strings already exist in traces.

Testing plan

The integration points of this code are mostly very thoroughly tested. However, additional testing will be necessary to ensure that the end-to-end pipeline works as expected, including surfacing the new trace events in the slow reports back-end.

Followup work

Once the logging infrastructure is in place to collect IPC message information there is work necessary to modify the remote endpoints, endowing call-sites with knowledge as to the lifecycle state of each frame, and to avoid sending the messages when the destination is frozen.