[public] FeatureParam Optimization

Author: Takashi Toyoshima
Last Modified: Aug 9, 2024

Context

FeatureParam<T>::Get() is often used to run Finch experiments, and more developers start using it as recent projects need more optimization parameters. On the other hand, it takes a visible time to call it, i.e. ~50ms in users' trace for total calls per benchmark. Developers who are careful about performance may design a static local cache so that the caller would not call it multiple times, but people don't in most cases, or such optimizations are unexpectedly removed by other developers while their refactoring changes.
So, if we can provide a common infrastructure to enforce having a cache, we can avoid such unexpected performance regressions. It's a memory vs performance trade-off, but in most cases, the parameter is stored as a class member, or so, and doesn't increase so much once we establish a standard approach.

Preliminary Evaluation

PoC CL shows 2-5ms performance gain on each benchmark and +0.01 score on speedometer3 on telemetry bot. The CL enforces the cache only for Blink side experiments using non-enum types. So, there is more room to be improved in total.

Design

In the final design, I modify existing FeatureParam<T> struct to be able to have an optional getter function, and prepare a macro that passes an external per-instance function that can have a static variable to cache the parsed value inside.

I applied this change to most of Blink features except for enum cases and test affecting cases, and results show 1-5ms speed up on each speedometer3 benchmark, and gets +0.02 point in the total score, but also APK size increased by 14.5 KiB for 135 FeatureParam instances. So, roughly said 100 B cost per parameter.

Header changes

```
#define BASE_DECLARE_FEATURE_PARAM(T, feature_object_name) \
  extern constinit const base::FeatureParam<T> feature_object_name

#define BASE_FEATURE_PARAM(T, feature_object_name, feature, name, \
                           default_value)                         \
  namespace field_trial_params_internal {                         \
  T GetFeatureParamWithCacheFor##feature_object_name(             \
      const base::FeatureParam<T>* feature_param) {               \
    static const T param = feature_param->GetWithoutCache();      \
    return param;                                                 \
  }                                                               \
  } /* field_trial_params_internal */                             \
  constinit const base::FeatureParam<T> feature_object_name(      \
      feature, name, default_value,                               \
      &field_trial_params_internal::                              \
          GetFeatureParamWithCacheFor##feature_object_name)

template <>
struct FeatureParam<T> {
  constexpr FeatureParam(
      const Feature* feature,
      const char* name,
      T default_value,
      T (*cache_getter)(const FeatureParam<T>*) = nullptr)
      : feature(feature),
        name(name),
        default_value(default_value),
        cache_getter(cache_getter) {}

  BASE_EXPORT T Get() const;
  BASE_EXPORT T GetWithoutCache() const;

  // RAW_PTR_EXCLUSION: #global-scope
  RAW_PTR_EXCLUSION const Feature* const feature;
  const char* const name;
  const T default_value;
  T (*const cache_getter)(const FeatureParam<T>*);
};
```

Implementation changes

```
int FeatureParam<T>::Get() const {
  if (LIKELY(cache_getter)) {
```

```
    return cache_getter(this);
  }
  return GetWithoutCache();
}

// Type-specific GetWithoutCache() implementation
// that was the original Get() implementation.
int FeatureParam<int>::GetWithoutCache() const {
  return GetFieldTrialParamByFeatureAsInt(*feature, name, default_value);
}
```

Rollout Plan

Step 1: Land the baseline

Land the code change that allows the `cache_getter` in the FeatureParam<T>. Then apply it to the blink features in a follow-up CL as I tried in the POC.
Launch Blink-only enforcements via Finch to see the real world performance impact. The cache in the FeatureParamWithCache<T> can be gated by another dedicated base::Feature for the comparison.

Step 2: Apply to more cases

1. Land the ScopedFeatureList fix that resets `cache_getter` in required cases to override the parameters in the test.
2. Support enum cases, preparing one more macros to pass one more argument for enum definition.

Step 3: Evaluate performance impact

Run A/B test behind a finch that controls the local cache, and remove the finch once the evaluation completes.

Discussion: should we enforce the cache use for all cases?

POC shows APK size is increased by 14.5 KiB. So, we need a discussion at binary-size@chromium.org. We may apply the cache selectively, and a simple strategy is

not apply for std::string cases that may use more memory, or they already have a local std::string instance respectively.

Technical Notes

Static local variables approach

This approach results in a single pair of a comparison and a conditional branch for the second and later runs. This looks ideal from the viewpoint of performance.

C++ code:

```
void foo() {
  static Foo* foo = new Foo();
  ...
}
```

Compiled binary example:

```
foo():
  push    rbp
  mov     rbp, rsp
  sub     rsp, 32
  cmp     byte ptr [guard variable for foo()::foo], 0
  jne     .LBB0_5
  movabs  rdi, offset guard variable for foo()::foo
  call    __cxa_guard_acquire
  cmp     eax, 0
  je      .LBB0_5
  ...
```

Atomic pointer with a value approach

This approach is still fast enough and achieves performance improvements. But it was difficult to remove static initializers even if we use constinit. My best attempt resulted in remaining a large code that registers dtors of FeatureParamWithCache<>s to atexit() in the static initializer list.

```
namespace field_trial_params_internal {

template <typename T>
```

```
struct LOGICALLY_CONST FeatureParamCache {
  explicit constexpr FeatureParamCache<T>(const FeatureParam<T>*
feature_param)
      : feature_param(feature_param) {}

  const T& Get() const {
    T* value_ptr = atomic_value_ptr.load(std::memory_order_acquire);
    if (value_ptr == nullptr) {
      std::unique_ptr<T> new_value =
std::make_unique<T>(feature_param->Get());
      T* expected = nullptr;
      if (std::atomic_compare_exchange_strong(&atomic_value_ptr, &expected,
                                              new_value.get())) {
        value = std::move(new_value);
        value_ptr = value.get();
      } else {
        value_ptr = expected;
      }
    }
    return *value_ptr;
  }

  void Reset() { value.reset(); }

  RAW_PTR_EXCLUSION const FeatureParam<T>* feature_param;
  mutable std::atomic<T*> atomic_value_ptr = nullptr;
  mutable std::unique_ptr<T> value;
};

}  // namespace field_trial_params_internal

template <typename T>
struct BASE_EXPORT LOGICALLY_CONST FeatureParamWithCache {
  constexpr FeatureParamWithCache<T>(const Feature* feature,
                                     const char* name,
                                     T default_value)
      : feature_param(feature, name, default_value),
        feature_param_cache(&feature_param) {}

  const T& Get() const { return feature_param_cache.Get(); }

  const struct FeatureParam<T> feature_param;
  const field_trial_params_internal::FeatureParamCache<T>
```

```
feature_param_cache;
};
```

Simple approach

This approach is not thread-safe that is required for base::FeatureParam.

```
template <typename T>
struct BASE_EXPORT FeatureParamWithCache<T> {
  constexpr FeatureParamWithCache<std::string>(const Feature* feature,
                                               const char* name,
                                               T default_value)
      : feature_param(feature, name, default_value) {}

  const T& Get() {
    // We will gate this via our own base::Feature to evaluate the impact
    if (!value.has_value()) {
      value = feature_param.Get();
    }
    return *value;
  }

  void Reset() { value.reset(); }

  const struct FeatureParam<T> feature_param;
  std::optional<T> value;
};

#define BASE_DECLARE_FEATURE_PARAM(T, feature_param) \
  extern base::FeatureParamWithCache<T> feature_param

#define BASE_FEATURE_PARAM(T, feature_param, feature, name, default_value) \
  base::FeatureParamWithCache<T> feature_param(feature, name, default_value)
```

References

- Perf Try Bots (recommendation is PGO build with 128 runs)