Shiny Objects: Object-Centric Characterization of Chromium

SAKET UPADHYAY, University of Virginia, USA
ASHISH VENKAT, University of Virginia, USA

Modern web browsers manage millions of dynamic objects across tabs, frames, DOM elements, and JavaScript contexts. However, fine-grained behaviors related to object allocation, lifetime, and memory usage in production browsers remain elusive. Chromium's modular and extensible design, use of specialized memory allocators, and sensitivity to instrumentation overhead further complicate precise object tracking. To this end, we develop a lightweight, thread-safe, and non-intrusive profiling framework. Using this infrastructure, we present an empirical characterization of Chromium's memory object behavior across twelve diverse, user-centric workloads. We examine object lifetime events, size diversity, spatial locality, type diversity, and memory activity, and reflect on their broader software and architectural implications. Our study offers a systems-oriented view into Chromium's architecture and memory behavior, and highlights structural challenges in efficient memory management in large-scale and diverse systems.

CCS Concepts: • **Software and its engineering** → **Object oriented development**; *Software performance*; *Software system structures*; **Compilers**; • **General and reference** → Empirical studies.

Additional Key Words and Phrases: object lifecycle profiling, compile-time instrumentation, memory characterization, browser performance, LLVM, microarchitectural analysis

ACM Reference Format:
Saket Upadhyay and Ashish Venkat. 2026. Shiny Objects: Object-Centric Characterization of Chromium. *Proc. ACM Meas. Anal. Comput. Syst.* 10, 1, Article 20 (March 2026), 27 pages. https://doi.org/10.1145/3788102

1 Introduction

The open-source *Chromium* web browser project [23] has been widely adopted due to its modularity, flexibility, extensibility, and performance-oriented nature. Its multi-process architecture integrates rendering engines, graphics libraries, JavaScript runtimes, state-of-the-art memory allocators and garbage collectors, and sandboxing technologies. It has been the bedrock of several modern browsers such as Google Chrome, Microsoft Edge, Amazon Silk, and DuckDuckGo, and forms the core of several desktop application frameworks such as Electron, used in popular applications such as Slack and Visual Studio Code Editor. It is also used in security-conscious virtualized environments such as Citrix Workspace [3] for secure web rendering. Owing to its wide deployment, even minor inefficiencies could have a major impact on the user's browsing experience, cloud resource utilization, and memory pressure.

Conventional performance characterization, profiling, and optimization strategies typically employ a code-centric perspective, following Amdahl's law [12–14, 25, 30], which involves identifying and analyzing hot code regions and functions of interest to identify computational bottlenecks and avenues for performance optimization. Data-centric strategies typically involve memory profiling to examine potential memory leaks, fragmentation, and corruption. However, these techniques are largely coarse-grained and scale poorly to large object-oriented codebases such as Chromium that

Authors' Contact Information: Saket Upadhyay, saket@virginia.edu, University of Virginia, Charlottesville, Virginia, USA; Ashish Venkat, venkat@virginia.edu, University of Virginia, Charlottesville, Virginia, USA.

ACM 2476-1249/2026/3-ART20
https://doi.org/10.1145/3788102

combine several complex performance-critical software components such as rendering engines, custom memory allocators, garbage collectors, and modules featuring event-driven computation that span several object lifetimes and contexts.

In contrast to code-centric characterization, this work shifts the lens to an **object-centric** perspective, focusing on how dynamic C++ objects are allocated, accessed, and destroyed across the browser's subsystems, rather than keeping focus on identifying hot and tight loops for performance optimization, thereby uncovering patterns and inefficiencies that remain hidden in aggregate code-centric views. In particular, we perform a large-scale, albeit low-cost instrumentation of Chromium with the goal of analyzing object lifetime behavior, access patterns, placement in hardware caches and memory pages, deallocation (e.g., graceful destruction vs. reclamation through garbage collection) and reallocation patterns, and interaction with microarchitectural structures such as the branch predictor and execution units, allowing us to draw insights and make recommendations to software developers and architects to improve the performance potential of Chromium.

This work introduces a **lightweight, thread-safe, and source-agnostic profiling framework** based on compile-time instrumentation. Our tool tracks object allocations, access frequency, and lifetimes without requiring modifications to the Chromium source code. By mapping object behavior across multiple browser subsystems, we provide fine-grained visibility into how different components contribute to memory pressure, latency, and performance anomalies.

While developers can modify source code with custom counters or debug logic, conducting a characterization effort at this scale demands significant time, effort, and resources. In contrast, this work offers a highly portable tool that performs all instrumentation automatically at the LLVM IR level at a low overhead, without the need for source modifications. Our approach applies to any C/C++ codebase compiled with LLVM that employs object-oriented allocation patterns. We select Chromium as the vehicle for this exploration due to its scale, architectural complexity spanning renderer, GPU, network, and storage subsystems, and its multi-process design with diverse memory management strategies (discussed in Section-2).

This methodology generalizes beyond browsers. Game engines such as Unreal Engine, database systems like ClickHouse[46] and RocksDB[18], and multimedia frameworks like FFmpeg[19] share similar characteristics: modular C++ architectures, custom allocators, and complex object lifecycles. Insights derived from Chromium can transfer directly to these domains. However, instrumentation of a codebase at Chromium's scale entails significant challenges.

First, low-overhead profiling without source modifications requires precise integration to avoid disrupting normal execution. Our framework uses compile-time instrumentation with inline assembly blocks inserted into target control paths where lightweight hooks invoke the RDTSC instruction for high-resolution timestamping at minimal cost. Second, Chromium employs custom memory allocators (Section 2.5), rendering standard library allocation hooks at the OS level inaccurate. We implement per-module allocation tracking by targeting specific atomic system calls that Chromium's allocators use to get accurate event logs and avoid interfering with its memory management. Third, multithreaded timing is complicated by core migrations, context switches, and unsynchronized clocks. We address this with per-thread timing blocks that use frequency-invariant and core-synchronized timestamp counters. Fourth, overhead is critical; components like networking and IPC are latency sensitive, and delays from instrumentation can cause timeouts. Excessive heap use or interference with reserved memory pools can crash Chromium. Our design limits logging-related allocations to 16 bytes per event to minimize heap usage. Traditional logging incurs high overhead from libc I/O and synchronization. Instead, we generate compile-time identity tags, use static buffers, and flush logs via minimal inline assembly blocks that invoke relatively low-overhead system calls, bypassing standard libraries entirely. Our profiling and instrumentation infrastructure is generic and can target any object-oriented application that can be compiled with the LLVM/Clang compiler – ideal targets include browsers, graphics libraries, media players, and web servers. In this work, we leverage Chromium as a demonstration vehicle.

There are several important implications of an object-centric characterization. First, precisely tracking object lifetime behavior, allocation, deallocation, and reallocation patterns could help better tailor memory management strategies. We find that the number, types, and sizes of objects that are ephemeral or persistent, and their destruction and reclamation patterns could vary across different execution flows, providing hints and heuristics to guide better memory management, garbage collection, and lifetime refactoring strategies. Second, it could help identify *zombie* objects (those that will no longer be accessed, and yet aren't reclaimed into the memory pool). This could not only help improve the efficiency of the garbage collector, but could also allow for targeted placement of deallocation hooks and automatic migration to safer and efficient C++ constructs such as smart pointers. Third, by analyzing the composition of memory pages by object lifetime categories (e.g., persistent, ephemeral but frequently reallocated, zombie, gracefully destroyed), valuable insights could be gained regarding potential fragmentation patterns and allocator inefficiencies. Fourth, identifying object access patterns and placement in hardware caches and memory pages could uncover memory hotspots and bottlenecks (e.g., false sharing) and enable the deployment of tailored optimizations such as cache conflict-averse placement and profile-guided prefetching.
In summary, we make the following major contributions:

- **A systems-oriented tutorial on Chromium's architecture and memory behavior.** We provide a concise, structured overview of Chromium's key components and their interaction with memory allocation subsystems, offering valuable context for researchers studying large-scale, multi-threaded applications.
- **A scalable, low-overhead instrumentation framework for object-centric characterization.** Our lightweight, thread-safe infrastructure enables detailed tracking of object allocations, deallocations, memory reuse, and access patterns across Chromium's complex runtime without requiring source code modifications.
- **A detailed characterization of object allocation, deallocation, and reallocation dynamics.** We quantify allocation intensity, highlight hotspots across browser subsystems, and uncover patterns of allocation churn and reuse not captured by conventional profilers.
- **An in-depth analysis of object lifetime distributions, spatial placement, and locality.** Our study reveals midlife and persistent object classes, memory pollution from zombie objects, and cache-level compaction behaviors that shape memory efficiency in Chromium.
- **Identification of object types and memory behaviors critical to performance.** We surface dominant object classes by frequency, longevity, and memory footprint, whose management disproportionately impacts memory retention and allocator efficiency. In addition, we also study the architectural impact of object layout and choice of data structures.
- **Recommendations for runtime and allocator-level optimizations.** Based on our findings, we outline directions for improving garbage collection heuristics, lifetime-aware memory placement, and mechanisms for mitigating memory fragmentation.

2 Chromium Browser Architecture

In this section, we provide requisite background on the Chromium architecture, describing its software organization, salient components and features, and their interactions, in addition to a brief overview of its memory management and garbage collection utilities.

One of the key defining features of Chromium is that it adopts a modular and a multi-threaded architecture, where each major subsystem is encapsulated within its own process or module for improved security and performance isolation and scalability. At startup, Chromium launches two primary processes: (a) the browser process, responsible for coordinating core browser functionality, and (b) the initial renderer process, which hosts the *Blink* rendering engine and the *V8* JavaScript engine.

As users open new tabs, Chromium dynamically spawns additional renderer processes. Furthermore, browser extensions, plugins, and standalone applications may also execute within dedicated processes, contributing to the overall process diversity.

2.1 Elements of a webpage

Figure-1 illustrates Chromium's modular architecture in relation to the tasks involved in loading a webpage. A webpage in Chromium is composed of several fundamental abstractions: *page*, *frame*, *Document Object Model (DOM) window*, and *document*. A *page* encapsulates the logical concept of a browser tab. While each renderer process may handle multiple pages, a page typically maps to a single tab from the user's perspective. A *frame* models the hierarchical structure of a web page, including both the main frame and any nested `iframe` elements. Each page may contain multiple frames arranged in a tree-like fashion. Each frame owns a single *DOM Window*, which corresponds to the global `window` object in JavaScript and provides the execution environment for scripts. The *Document* object, accessible via `window.document`, contains the parsed HTML content and serves as the root of the DOM tree for the frame's content. Further, to unify execution across different threading models, Chromium introduces the *execution context* abstraction. An execution context represents either a `Document` (for main-thread execution) or a `WorkerGlobalScope` (for worker-thread execution), enabling a consistent interface for script execution, resource access, and event handling regardless of the underlying context.

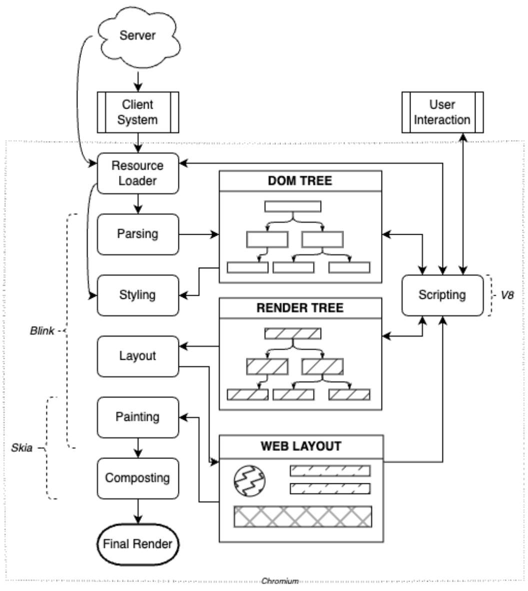


Fig. 1. Page loading overview in Chromium

2.2 Blink Rendering Engine

Blink, Chromium's rendering engine, is responsible for DOM construction, style and layout computation, JavaScript execution, and integration with the Chrome compositor. [2]

Rendering Pipeline. Parsing begins with HTML, generating the *DOM tree*, a structured representation of the page and the primary interface for JavaScript. The *V8 engine* accesses the DOM through *bindings*, thin wrappers that expose C++ structures to JavaScript. A document can include multiple DOM trees. Next, *CSS* rules are matched against DOM nodes to compute styles. The engine then constructs a parallel *Layout Tree*, whose nodes calculate positional geometry using element-specific layout algorithms. During the *Paint* phase, layout objects emit *display items*: commands like "draw rectangle" or "render text." Multiple display items may represent different visual layers of the same element. Display items are converted into pixel data in the *Rasterization* phase using the *Skia* [15] graphics library. Rasterization produces a bitmap by issuing hardware-abstracted draw calls, including Bézier paths and vector shapes. Finally, raster commands are bundled into a *GPU command buffer*, transmitted over *IPC*, and executed by the GPU to render pixels onscreen.

Threads and Memory. Blink runs in sandboxed renderer processes, each with a dedicated main thread and optional worker/internal threads. The *main thread* handles JavaScript, DOM, style, and layout, making Blink mostly single-threaded. Chromium's site isolation model enforces separate renderer processes for distinct sites. For example, an isolate specific origin policy could be specified such that mail.google.com and photos.google.com appear to share a site, but flights.google.com does not. Memory is managed using a hybrid model that comprises of: (a) *Oilpan*, a garbage collector-based heap allocator, used for most object lifetimes, and (b) *PartitionAlloc*, a manual allocator, used for performance-critical objects or objects with simpler lifetimes. We discuss this in greater detail in Section 2.5.

Task Scheduling. The *Blink Scheduler* manages task queues on the main thread to optimize system responsiveness [16]. Figure 2 illustrates the scheduler architecture. Tasks are assigned semantic labels (e.g., input, rendering) for context-aware prioritization. APIs abstract threading complexities, allowing developers to schedule tasks based on behavior rather than platform-specific details.

In Blink, tasks are defined as *base::OnceClosure* objects, posted via *TaskRunner::PostTask* or *TaskRunner:: PostDelayedTask*. Blink avoids general closure methods, using *WTF::Bind* for same-thread tasks and *CrossThreadBind* for cross-thread tasks. Once posted, tasks run atomically until completion. Task runners link tasks to execution contexts. *FrameScheduler::GetTaskRunner* ties tasks to specific frames, enabling the scheduler to prioritize or freeze tasks based on frame state. Specialized task runners manage tasks like garbage collection and compositing, each tied to specific threads. Tasks are categorized by *blink::TaskType*, which determines scheduling priority. Input tasks receive the highest priority, followed by compositor tasks when user gestures are detected. Other tasks are deferred or throttled to maintain responsiveness and conserve power. The scheduler also pauses tasks during synchronous dialogs or debugger breakpoints. Tasks that do not require the main thread are offloaded using *worker_pool::PostTask*, utilizing a thread pool. Tasks needing sequential execution should use *worker_pool::CreateTaskRunner* to ensure ordering, avoiding the overhead of dedicated threads. Efficient task scheduling directly influences memory behavior, such as garbage collector pauses, object allocation timing, and interactions with rendering subsystems.

2.3 V8 JavaScript Engine

V8 is Google's high-performance JavaScript and WebAssembly engine implemented in C++. Its execution pipeline includes both interpretation and just-in-time (JIT) compilation for optimized performance. The compilation process begins with a custom parser that constructs an abstract syntax tree (AST). This AST is then converted into V8 bytecode by Ignition, V8's interpreter. Subsequently, TurboFan, the optimizing compiler, compiles the bytecode into machine code for execution. V8's compilation infrastructure, previously referred to as Crankshaft, consists of four core components. First, the*base compiler* generates machine code quickly with minimal optimization, enabling fast startup. Second, the *runtime profiler* identifies frequently executed hot code paths during program execution. Third, the *optimizing compiler (TurboFan)* recompiles hot code with optimizations, with type feedback gathered during execution guiding the optimizations. Finally, *de-optimization* supports speculative optimization by allowing fallback to baseline code

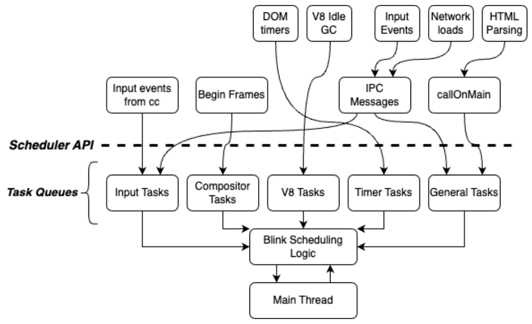


Fig. 2. Overview of Blink task scheduler

when assumptions made during compilation are invalidated at runtime. V8 dynamically optimizes and re-optimizes code based on runtime behavior, enabling adaptive performance tuning that balances execution speed with resource usage.

2.4 Component Interaction

Chromium follows a multi-process architecture for isolation and fault tolerance, separating browser and renderer processes. The browser process handles UI, system integration, and process control; each renderer process manages web content. Each renderer maintains a global `RenderProcess` object to store per-process state and facilitates communication with the browser. On the browser side, a matching `RenderProcessHost` manages the renderer and serves as its IPC endpoint. Communication occurs over Mojo, Chromium's IPC framework, which offers cross-platform communication, an Interface Description Language (IDL), and code-generated bindings. Within a renderer, each web document maps to a `RenderFrame`, paired with a `RenderFrameHost` in the browser. `RenderFrame` encapsulates document state and interacts with Blink; `RenderFrameHost` controls its lifecycle and coordination. Frames are identified via routing IDs, unique within each renderer; globally unique frame IDs combine the routing ID and `RenderProcessHost`. Visual content (e.g., images, input regions) is managed by `RenderWidget` objects, paired with `RenderWidgetHost` in the browser. These hosts coordinate rendering and input delivery. Finally, Mojo (or legacy IPC) channels are monitored for failures. If a renderer crashes (e.g., via handle state), Chromium notifies affected tabs, replaces the view with a placeholder, and spawns a new renderer upon reload.

2.5 Memory Management

Chromium employs a hybrid memory management approach combining a custom allocator, *PartitionAlloc*, with tracing garbage collectors for higher-level subsystems (Blink and V8).

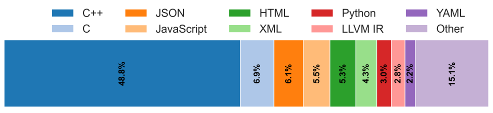


Fig. 3. Distribution of code across different programming languages within the Chromium project.

PartitionAlloc: Chromium's Primary Allocator. PartitionAlloc is Chromium's default memory allocator, replacing standard allocation functions via a unified shim that intercepts `malloc()`, `free()`, `new`, and `delete`. It is optimized to reduce fragmentation, isolate performance across subsystems, and enable memory safety features such as pointer tagging and address pool partitioning. Note that, at the lowest level, all allocators ultimately use `malloc`/`new` and related primitives, as the Chromium codebase is predominantly C/C++ (Figure 3).

Memory is reserved in 2 MiB-aligned *super pages*, subdivided into fixed-size *partition pages*, which are further divided into one or more *slot spans* made up of fixed-size *slots*. Allocation requests are bucketed by size, and each bucket tracks active, empty, and decommitted slot spans. The fast path allocates via freelist pointers embedded in slots; new pages are provisioned on demand, deferring physical memory commitment until required.

Renderer processes leverage dedicated partitions to improve cache locality and security. The *LayoutObject* partition is optimized for rendering data structures, *Buffer* stores script-exposed containers like `Vector` and `String`, *ArrayBuffer* is dedicated to backing JS `ArrayBuffer` memory, and *FastMalloc* is a general-purpose fallback allocator.

Garbage Collection. Chromium employs two major garbage collection (GC) techniques. Blink uses *Oilpan* (C++), while JavaScript memory is managed by V8's generational GC. Oilpan uses a mark-and-sweep collector with optional compaction. Heap pointers are wrapped in GC-aware smart types (`Member`, `Persistent`, etc.). It supports two GC styles: (a) *conservative* that scans native stacks to find potential pointers, and (b)*precise* that runs at known safe points (e.g., event loop boundaries). It also supports three execution modes: (a) *atomic* that forces all execution to be stopped while garbage collection is in progress, (b) *incremental* that interleaves execution with the mutator via write barriers, and (c) *concurrent* that runs marking on background threads. During *marking*, reachable objects are traced from root sets; in *sweeping*, destructors run and memory is reclaimed. Pre-finalizers allow inter-object cleanup before destructor execution, which occurs on the allocating thread to preserve C++ semantics. It also supports weak references, although they may be elided if both referent and holder are collected in the same cycle.

V8, on the other hand, uses a generational, incremental mark-and-sweep collector optimized for ephemeral objects. The heap is split into (a) *young generation* that is collected frequently via minor GCs, and (b)*old generation* that is collected less often via major GCs and contains long-lived objects. Objects typically start in "Eden", a space where all newly allocated objects begin their life in the young generation, and may be promoted after surviving cycles. Both spaces support incremental and concurrent collection to minimize pause times. Relocation reduces fragmentation but incurs pointer update overhead. As the old generation grows, promotion cost and GC latency may increase, affecting responsiveness.

3 Object-Centric Instrumentation

In this section, we discuss the design of our instrumentation methodology, outlining the mechanisms used to capture object life-cycle events with minimal runtime overhead and high semantic fidelity.

3.1 Instrumentation Pipeline

Our pipeline consists of three phases: instrumentation, execution, and analysis. In the first phase, we compile Chromium with lightweight hooks to trace *allocation*, *constructor*, *destructor*, and *deallocation* events using our LLVM [35]. We simultaneously extract static metadata such as type layouts and symbol tables for annotating runtime events. In the second phase, we run the instrumented binary on representative workloads. This produces detailed logs capturing object life cycle events, types, and source locations. We also collect hardware-level data using Intel VTune, including retired loads/stores, data addresses, and cache hit/miss statistics. The third phase analyzes these logs. We correlate dynamic events with static metadata to infer object lifetimes, resolve type boundaries, and eliminate redundant records. The resulting trace captures allocation/deallocation patterns, object placement at page and cache levels, and fine-grained access behavior.

Instrumentation generates four raw data streams per component: allocations, deallocations, constructor calls, and destructor calls. We identify objects by matching constructor invocations to preceding allocation events, i.e., constructor address falls within the allocated region and occurs after the allocation. We resolve object types using a compile-time `type_map` that links runtime allocations to known types. For each object, we retrieve its type and use the corresponding size to infer boundaries. An object is considered valid if its boundary lies within the allocated region. Resolved objects are added to the final analysis set.

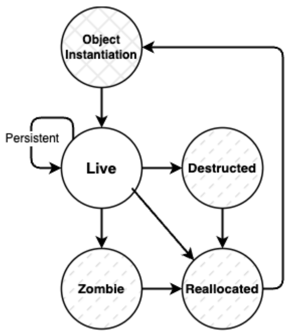


Fig. 4. Classification of object states.

3.2 Object Lifetime Taxonomy

We next present a taxonomy we use in our object lifetime analysis. Recall that an object's lifetime begins with the process of memory allocation, followed by instantiation that occurs explicitly through a constructor call. Memory is typically allocated from a free pool of memory. Objects once created enter the *live* state, which implies that they might be referenced in

Algorithm 1 Estimate Object Lifetimes

```
Require: Sorted Objects, Destructors
 1: alive_addrs ← {}
 2: for all $O \in$ Objects do
 3:     $O$.death ← $\infty$; $O$.alive ← $\infty$
 4:     $O$.reuse ← $\infty$; $O$.cod ← P
 5:     $addr$ ← $O$.addr; $t_b$ ← $O$.ctor_ts
 6:     if $addr \in$ alive_addrs then
 7:         $O$.alive ← $t_b$ − alive_addrs[$addr$]
 8:         $O$.death ← $O$.reuse ← $t_b$
 9:         $O$.cod ← R
10:     else
11:         alive_addrs[$addr$] ← $t_b$
12:     end if
13:     for all $D \in$ Destructors do
14:         if $D$.addr = $addr \wedge D$.cycle > $t_b \wedge$
                          $D$.ts < $O$.reuse then
15:             $t_d$ ← max($D$.cycle)
16:             $O$.death ← $t_d$
17:             $O$.alive ← $t_d - t_b$
18:             $O$.cod ← D
19:         end if
20:     end for
21: end for
```

the future. We refer to the objects that are actively being referenced in any given execution interval as *in-use* objects. The lifetime of an object ends upon destruction. This typically happens through an explicit destructor call (graceful destruction) or via garbage collection. However, several objects may *persist* or remain live throughout execution as their contexts and access patterns span several different modules. If a live object has not been used for a long time, but has not been explicitly destroyed, it could have entered a *zombie* state, which means that the object might no longer be referenced. Objects may stay in the zombie state for a long time until they get picked up by the garbage collector. In many instances, they might never be marked for garbage collection, artificially inflating the memory consumed. Memory reclaimed via garbage collection might be reallocated at a later point of time, followed by a constructor call, signifying the birth of a new object. It is important to distinguish between a live object that was instantiated after a first-time allocation vs. a reallocation to assess the effectiveness of the underlying memory allocation and management scheme in limiting memory fragmentation. Figure 4 shows the object lifetime state transitions described above.

Algorithm 1 estimates object lifetimes by tracking address reuse events and explicit destructor calls, inferring the time interval between object allocation and its eventual disposal. The algorithm operates on a sorted list of objects and a list of destructor events. For each object, the algorithm initializes its lifetime parameters, including the death time, duration of liveness, and reuse cycle. If the object's address has been reused, the algorithm updates the object's lifespan based on the difference between its current and previous allocation times. If a destructor call for the object is found, the death time is updated to the destructor's timestamp, and the liveness duration is recalculated. The object's status is marked as either reused ('R') or destroyed ('D') based on the presence of a

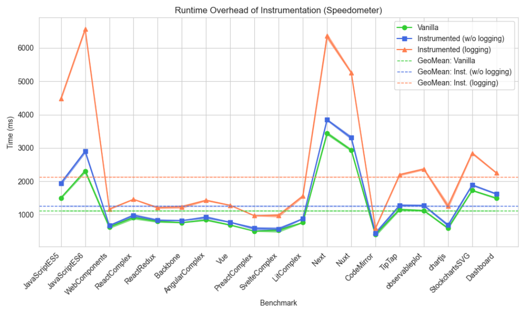


Fig. 5. Overhead of object tracking instrumentation, measured using the WebKit Speedometer suite.

reuse event or destructor call. In the absence of a corresponding reuse event or destructor call, an object is marked as persistent ('P'). Post-memory access mapping using VTune, if a persistent object exhibits no load or store activity, it is further classified as a zombie ('Z').

3.3 Instrumentation Overhead

Our instrumentation framework introduces minimal performance overhead. The inserted hooks, state checkers, and supporting logic incur a only 1.1× runtime overhead relative to the baseline. Enabling object data logging increases this overhead to an average of 1.9×. Figure 5 presents the runtime performance impact, as measured using the WebKit Speedometer benchmark suite [17], comparing a baseline Chromium build ('Vanilla') with two instrumented variants: one with logging disabled ('Instrumented (w/o logging)') and another with logging enabled ('Instrumented (logging)'). In terms of binary size, the average size of shared object files increases from 526,274.24 bytes in the baseline to 578,230.46 bytes in the instrumented build, an increase of approximately 9.8%. This moderate code size overhead stems from the inclusion of logging metadata and support routines, and remains acceptable within Chromium's modular build architecture. Compile-time overhead is similarly negligible. Using 8-thread parallel builds with `gn` and `ninja` on a system with an Intel i9-13900KS and 64 GB of memory, instrumentation extends total build time from ≈12,262 to ≈12,534 seconds (+2.2%).
To assess the feasibility of tracking dynamic object allocations in Chromium, we used Microsoft's Sysinternals VMMap [38] during the Speedometer 3.1 benchmark [17]. Chrome committed over 200 MB to the heap (50,006 pages), compared to 1.8 MB on the stack and 16 MB in mapped I/O regions. This heap-heavy profile (>90%) reflects typical browser behavior driven by DOM structures, JavaScript objects, and the V8 engine. These results, consistent with Chromium's architecture, confirm that our method captures the dominant memory usage patterns under realistic workloads.

3.4 Existing Memory Profiling Approach

Table 1 compares our approach against three representative tools native to the Chromium ecosystem.

MemoryInfra [4] provides timeline-based profiling through chrome://tracing, offering subsystem level attribution via the `MemoryDumpProvider` interface. Components self-report their allocations at configurable intervals, light dumps every 250ms and heavy dumps every 2 seconds. While this approach integrates seamlessly with Chromium's tracing infrastructure and imposes low overhead, it requires manual implementation of dump providers for each subsystem and cannot track individual object lifecycles or detect transient allocation patterns between snapshots.

Heap Profiler [7] captures per-allocation backtraces, enabling developers to attribute memory consumption to specific call sites. This granularity comes at significant cost: storing complete stack traces for every allocation increases memory overhead substantially, and type resolution requires debug symbol availability. The profiler tracks allocations but not deallocations, precluding lifetime analysis.

AddressSanitizer (ASan)[1, 48] and **LeakSanitizer (LSan)**[9] instrument memory accesses to detect errors such as use-after-free, buffer overflows, and memory leaks. These tools excel at identifying correctness bugs but do not characterize normal memory behavior, they report only error events and leaked allocations at program termination, providing no visibility into object lifecycles or access patterns during execution.

Our approach enables continuous per-object tracking with lifetime state transitions, automatic module attribution without manual instrumentation, and the ability to detect inefficiencies like zombie objects. The LLVM instrumentation pass requires no source modifications, enabling immediate extension to new subsystems as Chromium evolves.

Table 1. Comparison of memory profiling methods for object tracking in Chromium.

Dimension	**This Work**	**MemoryInfra**	**Heap Profiler**	**ASan/LSan**
Granularity				
Tracking Unit	Per-object lifecycle	Per-subsystem aggregates	Per-allocation backtraces	Per-allocation errors
Event Types	alloc, ctor, dtor, dealloc	Periodic snapshots	Allocation only	Error events only
Type Resolution	Compile-time `type_map`	Self-reported by provider	Debug builds only	None
Temporal Data	Continuous (RDTSC)	Intervals (250ms/2s)	Cumulative	None
Overhead				
Runtime	1.1×–1.9×	0.05–0.10×	High (variable)	~2×
Memory	16 bytes/event	Minimal	High (backtraces)	2–3×
Binary Size	~0.098×	Negligible	Negligible	~2×
Developer Effort				
Source Modifications	None	Required per subsystem	None	None
New Component Support	Automatic	Manual `MemoryDumpProvider`	Automatic	Automatic
Build Integration	LLVM pass	Native	Flag-based	Flag-based
Relationship Mapping				
Object-to-Module	Automatic via IR	Manual attribution	Indirect (backtrace)	None
Lifetime States	Live/Zombie/Destroyed/ Reallocated/Persistent	None	None	Leaked only
Reuse Tracking	Address reuse chains	None	None	None
Access Correlation	VTune integration	None	None	None

Table 2. Summary of Browser Interaction Experiments

Experiment	**Type**	**Affected Modules**	**Description**
`OPENHOMEPAGE`	Static Navigation	`content`, `net`, `blink`	Load static homepage; baseline for memory use.
`CLICKLINK`	Navigation	`content`, `blink`, `net`	Navigate via hyperlink; observe DOM teardown.
`FILLFORM`	Form Interaction	`blink`, `v8`	Enter text into forms; track transient JS objects.
`MULTISTEPFORM`	Form Workflow	`blink`, `v8`, `content`	Submit multi-page form with validation/state.
`AJAXLOAD`	Dynamic Content	`v8`, `net`, `blink`	Trigger AJAX; monitor JS heap behavior.
`NAVBACKFWD`	State Transition	`content,blink`, `nav.`	Use history API; test caching and DOM restore.
`REFRESHPAGE`	Page Reload	`content`, `blink`, `v8`	Full reload; analyze memory cleanup/reuse.
`EXECJAVASCRIPT`	Scripting	`v8`, `devtools`, `blink`	Inject and run JS; stress JS engine.
`COOKIEMANAGEMENT`	Persistent State	`net`, `content`, `storage`	Read/write/delete cookies; test storage APIs.
`DRAGDROP`	UI Interaction	`blink`, `ui`, `content`	Perform drag-and-drop; test UI/DOM updates.
`HANDLEALERT`	Modal Handling	`content`, `blink`, `ui`	Interact with alerts; monitor UI thread.
`TAKESS`	Rendering/Output	`blink`, `viz`, `gpu`, `content`	Capture page state; check rendering pipeline.
`WEBGL2D`	2D Graphics	`gpu`, `blink`, `viz`, `v8`	Animate circles; simulate 2D graphics load.
`MEMSTRESS10-60`	Memory Stress	`v8`, `blink`, `GC`	Allocate JS objects; simulate memory pressure.

4 Experimental Methodology

To systematically observe and analyze Chromium's[1] memory and object behavior under realistic interaction patterns, we develop a suite of automated browser experiments using `pytest` and the Selenium WebDriver. These experiments are representative of a range of user browser interactions, covering both typical (e.g., page navigation, form interactions) and edge-case behaviors (memory

[1]Commit hash `c61aa6fa0196356fda60dca670ed358c26f9fd3b`

stress tests), while capturing fine-grained object and memory traces. Table 2 summarizes the scenarios we implement, including interaction types, targeted browser subsystems, and intended stress patterns. All experiments are conducted on an Intel(R) Core(TM) i5-12900KS processor (3.6 GHz), running Ubuntu 24.04. For microarchitectural and memory access analysis, we develop a wrapper around Intel VTune Profiler [8] (version 2025.1.0, build 629847) that logs data via Intel Sampling Drivers. To ensure consistency, we disable Address Space Layout Randomization (ASLR), Intel's Turbo Boost (frequency scaling), and hyper-threading, and set all CPU governors to performance. Profiling results were exported with Data Address as the pivot, allowing us to map memory accesses back to individual objects.

5 Evaluation

In this section, we present our observations and findings, highlighting key insights and affirmations regarding Chromium's memory management behavior and design choices.

5.1 Object Lifetime Characterization

Object Events. We begin by characterizing the number of key life cycle events - allocations, deallocations, constructor invocations, and destructor invocations (Figure 6). Constructor calls occur most frequently, outpacing allocations, which highlights a common pattern of in-place object memory reuse without fresh memory allocation. This reuse is indicative of memory pooling

strategies designed to reduce allocator overhead. In contrast, we observe fewer deallocations than allocations, reflecting deferred reclamation or remapping strategies that avoid conventional free paths. Most notably, destructor invocations are significantly lower than constructor calls. This discrepancy arises from the widespread use of bulk memory reclamation mechanisms (garbage collection), which eliminate the need for explicit destruction. These trends collectively reveal that an object life cycle is heavily optimized for performance and tailored to the demands of low-latency rendering and high-throughput DOM manipulation, aimed for *minimizing* allocation, deallocation, and destruction overhead.

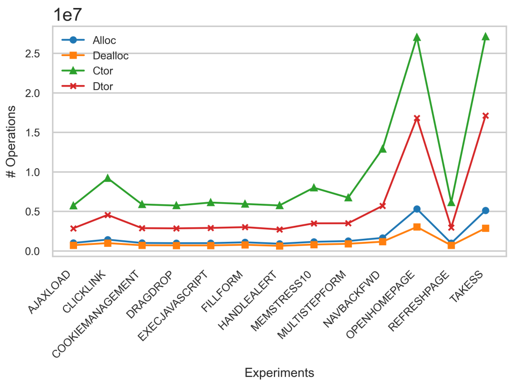


Fig. 6. Object allocation (Alloc), deallocation (Dealloc), constructor (Ctor), and destructor (Dtor) events across different experiments shown in Table 2.

Insight 1

Constructors consistently outpace allocations, while explicit deallocations are infrequent, suggesting extensive in-place object reuse and the prevalence of deferred reclamation.

Recommendation 1

Developers should continue to focus on taking advantage of the reuse and deferred reclamation model to avoid frequent allocation/deallocation overhead, particularly in performance-critical code regions.

To quantify the impact of memory pooling and deferred reclamation, we model two scenarios:

(1) No memory-pooling: every object construction directly invokes the allocator.
(2) No deferred reclamation: requires graceful destruction of an object upon its death.

We conservatively estimate cycle estimates based on PartitionAlloc [11], TCMalloc [26], and Mallacc [32] as follows: average allocation latency of 150 cycles, deallocation latency of 100 cycles on the fast path, a typical constructor cost of 300 cycles, and destructor cost of 200 cycles. These values represent optimistic yet loosely bounded estimates suitable for comparative analysis. In practice, slow-path allocations that involve OS system calls may incur ≈500-5000 cycles, and constructor/destructor costs vary substantially with object size, initialization complexity, and design decisions. Our projections show that *no memory pooling* increases average allocator invocations from 7,903 to 2.09M operations, equivalent to 519.5M additional cycles, slowing down execution by 240X. On the other hand, *no deferred reclamation* forces an additional 1.39M destructor calls on top of 695,446 baseline calls, imposing a 3.33× overhead. Together, these results indicate that pooling and deferred reclamation jointly avoid approximately 797.6 million wasted cycles in our execution trace, underscoring their substantial role in amortizing allocation and reclamation costs.

Object Lifetime Distribution. Figure 7 shows the distribution of object lifetimes, averaged across all experiments shown in Table 2. Clearly, the distribution exhibits a classic exponential decay, with the vast majority of objects being ephemeral (i.e., having short life spans), a pattern consistent with transient allocation behavior commonly observed in large-scale applications. This profile is strongly aligned with the performance assumptions of generational garbage collection (GC) and validates the architectural choices in Chromium's memory management subsystem (see Section 2.5).

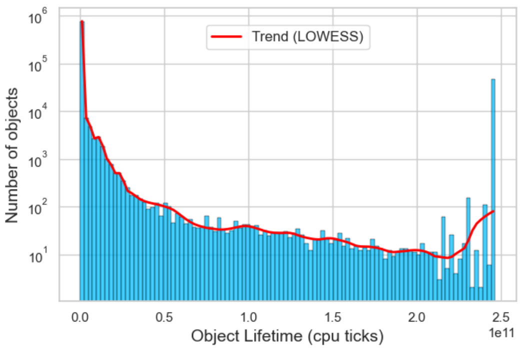


Fig. 7. Distribution of object lifetimes plotted on a log-scaled y-axis. The red curve overlays a LOWESS-smoothed trend line to highlight the underlying pattern and deviations across the lifetime spectrum.

Following the initial decay, the lifetime distribution enters a relatively flat phase, indicating a cohort of objects with mid-range lifetimes. Most of these objects are GPU-related objects such as GrDirectContextPriv, GrShaderVar, and GrYUVATextureProxies that persist across draw passes or video decode phases, encapsulating shader state and texture resources. Similarly, SkShader instances live across paint operations and may be cached within rendering pipelines. Platform abstractions like ScopedGObject is similar to a scoped_refptr for GObject types, with lifetimes scoped to UI components or platform events. Utility classes such as ClearCollectionScope and SkAutoMutexExclusive support scoped object management and synchronization. These objects are typically associated with components that implement caching, session state, or similar persistent-but-not-global semantics.

Beyond this flat phase, the tail of the lifetime distribution becomes more erratic, exhibiting greater variance and a pronounced spike,indicating the presence of highly persistent or effectively zombie objects. Upon closer inspection, the dominant type in this region is PrefService (~39%), defined in components/prefs/pref_service.h, which provides centralized access to user preference data and policy-controlled settings. This is followed by GrGLGpu and GrGLAttribArrayState, which together account for ~30% of objects in this region. Both originate from Skia's OpenGL backend (skia/src/gpu/ganesh/gl) and manage critical aspects of GPU state: GrGLGpu oversees rendering operations and framebuffer configuration, while GrGLAttribArrayState tracks vertex attribute bindings. The next most frequent type is GURL (~20%), from Chromium's core URL handling library, which encapsulates parsed URLs and normalization logic. Finally, ContentSettingsPattern appears frequently in this tail, serving as a key structure for mapping content permission rules (e.g., cookies, JavaScript) to URL patterns.

Overall, the majority of persistent objects fall into three domains: Inter Process Communication (mojo, ipc), URL parsing (e.g., `GURL`, `URLPattern`), and graphics state management (e.g., Skia and `cc/paint`). A breakdown of object mortality at the tail shows that 96.10% are persistent/zombies, while 3.74% are explicitly destroyed, and only 0.14% are reclaimed due to address reuse.

Insight 2

Chromium employs a mix of different memory management policies based on the locality profiles and performance requirements of the subsystem - Blink/V8 leverage Mark & Sweep and Generational Garbage Collection, Skia uses arena-based allocation, and browser services employ scoped reference pointers alongside long-lived persistent global objects.

Recommendation 2

For future subsystems, developers should leverage garbage collection for dynamic language bindings and object graphs, arenas for rendering, and scoped computation for services.

5.2 Object Size Heterogeneity

We categorize the memory footprint of objects based on allocation size, highlighting contrasting behaviors of small and large object allocations.

Small Objects (1, 4, and 8 Bytes). The smallest observed objects are 1-byte, followed by 4-byte and 8-byte allocations. These are typically low-level constructs used for management and synchronization. Examples include AsanUnpoisonScope, SkSpinlock, SkSafeMath, SkOnce, AllLABsAreEmpty, and DirtyBit. SkOnce (1-byte type), defined in SkOnce.h (Skia), supports thread-safe one-time initialization. While useful, widespread use of SkOnce may lead to unnecessary allocation churn if not carefully managed. Auditing such usage may reveal optimization opportunities. Most 8-byte allocations are smart pointers, primarily used for lifetime tracking and shared ownership. Although expected in large codebases, the sheer volume of such pointers implies non-trivial overhead from pointer-based indirection. These small objects, while lightweight individually, contribute disproportionately to allocator overhead and heap fragmentation. Their frequency compounds memory pollution when they are long-lived or never deallocated.

Large Objects (3,000-65,568 bytes). The largest observed allocations, up to 65,568 bytes, originate from SkSTArenaAlloc, Skia's arena allocator. Designed for high-throughput, low-fragmentation allocation, SkSTArenaAlloc allocates memory in exponentially increasing block sizes following a Fibonacci sequence. SkSTArenaAlloc begins with a user-provided block or a default size and allocates new blocks as needed. Block sizes grow as $F_n = F_{n-1} + F_{n-2}$ for minimizing block churn and deallocations. This

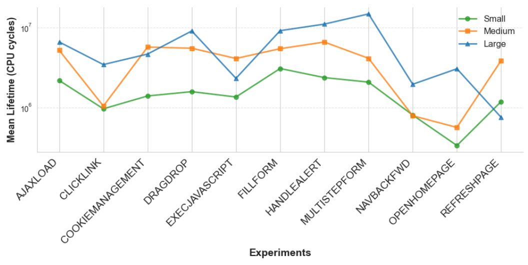


Fig. 8. Geomean Object Lifetimes

strategy benefits performance but may lead to internal fragmentation and memory waste, particularly when arena lifetime outlasts that of constituent objects. As a result, persistent or zombie objects in these blocks cause significant memory retention. While smaller than SkSTArenaAlloc, instances of SelectedKeywordView, which is responsible for "tab-to-search" UI functionality in the browser's location bar, appear frequently during a session. This class interprets user input like "search google <query>" to trigger contextual actions. Frequent instantiation under interactive workloads can compound memory pressure. We next examine the correlation between object size and lifetime. As shown in Figure-8, across all operations, lifetimes of a small objects are dramatically shorter than that of large objects. This aligns with the expectation that small objects are likely transient or temporary data, while large objects are often related to persistent resources (like graphic buffers, DOM Trees, Render Trees, etc.).

We further examine the state of the object in its life cycle according to the taxonomy described in Section 3.2. We find that most small objects are gracefully deallocated via destructors or efficiently reclaimed and reused by the garbage collector. Only a small fraction of these remain suspended in an *zombie* state. In contrast, while large objects are also typically terminated through destructors, their memory is not immediately reclaimed. This aligns with the principles of generational garbage collection, where older objects are collected less frequently. Notably, even the youngest large objects in Chromium tend to exhibit longer lifetimes than their smaller counterparts.

Insight 3

Most objects are small and ephemeral in their lifespan, dominating the allocation events, but contributing minimally to total memory usage. However, a significant outlier of large arena-allocated persistent objects exists, related to graphics buffers and rendering.

Recommendation 3

Developers should audit and minimize the allocation of small objects as they tend to have a high allocator cost per byte, i.e., their overhead is mostly hidden in allocator metadata and fragmentation, not raw memory usage.

Page Pollution. The primary challenge posed by *persistent* and *zombie* objects is not their sheer count, but their spatial distribution across memory pages. These objects fragment memory, preventing full page reclamation and undermining the efficiency of memory reuse mechanisms. Figure-9b illustrates that pages with a high number of zombies tend to contain small objects (averaging 1-10 bytes) while large zombies (>1 KB) are more likely to appear isolated or in low density.

High-density pages filled with numerous small, inactive objects exacerbate memory fragmentation and complicate the process of reclaiming memory. Although individual objects may be relatively small, their combined footprint can prevent page reuse or efficient compaction. Moreover, these objects may persistently occupy cache lines or TLB entries, contributing to broader system inefficiencies. Consequently, these pages are more challenging to evict or reuse without precise object-level tracking. In contrast, low-density pages containing large Zombie objects serve as clear targets for coarse-grained reclamation. Since these pages primarily host a few bulky, dead allocations, they can be efficiently identified and released through page-level mechanisms, provided that allocation tracking metadata supports this functionality. This observation motivates the design of tiered cleanup strategies: aggressively compact or clear high-density pages of small Zombies, while directly deallocating low-density pages with large ones.

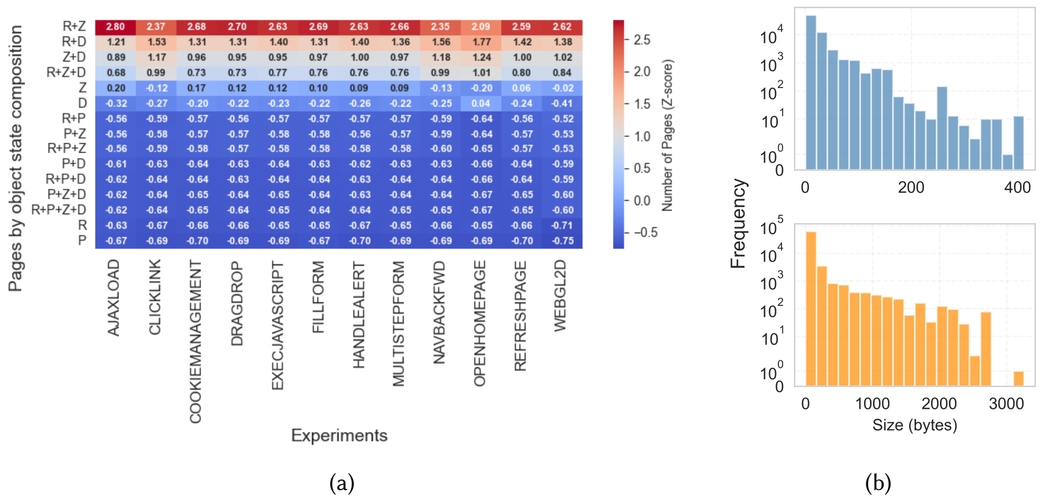


Fig. 9. (a) Page composition by object states (R: Reallocated, Z: Zombie, D: Destroyed, P: Persistent), normalized column-wise. (b) Zombie count and mean size per page.

Further analysis reveals a structural impediment to effective memory reclamation. Figure-9a shows that zombie objects frequently coexist with objects that were instantiated on reclaimed memory (i.e., pages with R+Z status in the figure). This co-residency hinders the ability of garbage collectors or memory managers to reclaim entire pages, even when parts of them are reallocated and assigned to newly created objects. While majority of objects can be reclaimed and reallocated efficiently, zombie objects prevent the wholesale deallocation of these pages, resulting in a phenomenon we term *page pollution* by zombies.

This effect is particularly pronounced for small zombie objects, which, despite their insignificant individual footprint, scatter across memory and become interleaved with live allocations. Consequently, memory managers are compelled to adopt a partial reuse model, where page reuse is constrained by the liveness of the smallest objects. This fragmentation severely restricts optimizations such as huge-page coalescing (where the operating system combines smaller memory pages into larger *huge pages* to improve performance by reducing page table overhead), OS-level page release, and slab reclamation. It is worth noting that this distribution holds across various workloads, reinforcing the conclusion that Zombie behavior is a structural consequence of Chromium's memory semantics, rather than a mere workload-specific anomaly. This suggests that runtime systems could benefit from identifying polluted pages as a unit of optimization, whether for eviction, migration, or triggering garbage collection, without the need for comprehensive program analysis. Effective mitigation may require isolating short-lived or inactive allocations, potentially through better segregated allocation pools or lifetime-aware object placement strategies, to minimize zombie-liveness interference and enhance page-level cleanup potential.

Insight 4

Small zombie objects can also contribute to spatial fragmentation, preventing entire memory pages from being reclaimed even if most of their contents are dead, and interfering with system-level optimizations such as huge page coalescing.

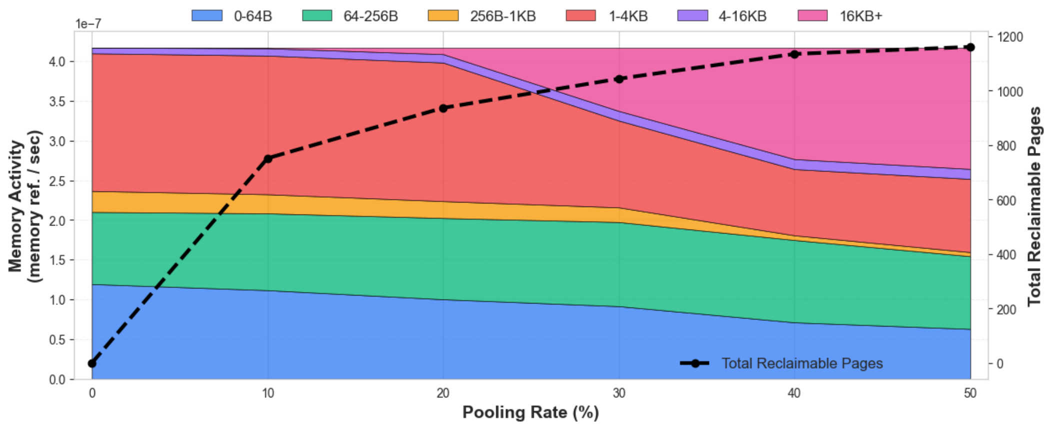


Fig. 10. Affect of size and lifetime-aware object pooling on Memory access rates and total reclaimable pages per size bucket.

Recommendation 4

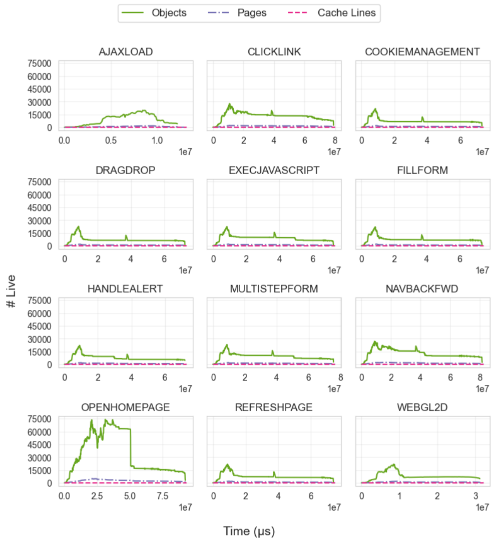
Use lifetime-aware placement to ensure small objects that have the potential to become zombies are contained within a small subset of pages and employ lifetime-aware arena partitioning to prevent long-lived zombie objects from pinning entire blocks in memory.

To evaluate the impact of size and lifetime-aware object pooling, we model a configuration in which objects belonging to the same size class and exhibiting similar lifetimes are pooled together to enable batched reclamation. We project both allocation overhead and memory cost under progressive size promotion, where smaller objects are aggregated into larger buckets. As shown in Figure 10, object pooling and migration markedly alter memory activity (measured in memory references per second) across size categories. As smaller objects (1-4 KB) are pooled and promoted into larger buckets, we observe a pronounced shift in memory activity toward the 16 KB + size range. Although the relative change for very small objects (0-64 B) appears modest, their contribution to reclaimable pages is substantial, as reflected in the right-hand panel of the figure, highlighting the disproportionate effect of small-object pooling on overall memory reclamation efficiency.

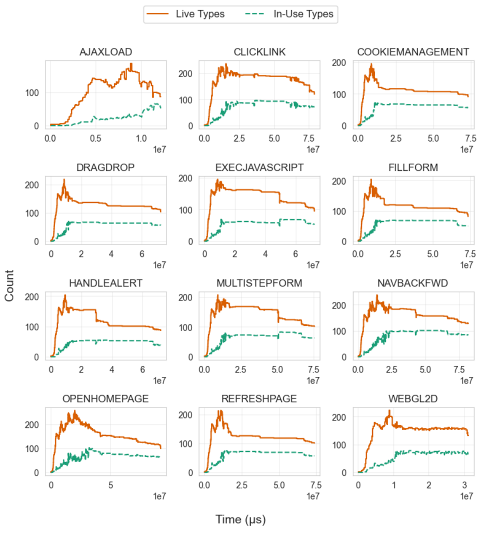
5.3 Object Activity Characterization

Live vs In-Use. Figures 11a, 11b, and 12a present the number of live and in-use objects, and their types, along with the corresponding number of pages and L3 cache lines containing the objects, as observed across each experiment. We make several observations.

First, even though tens of millions of allocations are made throughout execution, the number of live objects at any given point in time is a small fraction of those allocations, mostly capped at tens of thousands of objects across the different experiments. The number of in-use objects is even smaller, hovering over a few hundreds to thousands of objects at any given point. This suggests that most code regions involve computations featuring a limited set of objects and the liveness of most objects do not extend beyond short time windows. Second, even though a sizeable number of objects remain live at any given point in time, they typically map to only a small set of pages



(a)



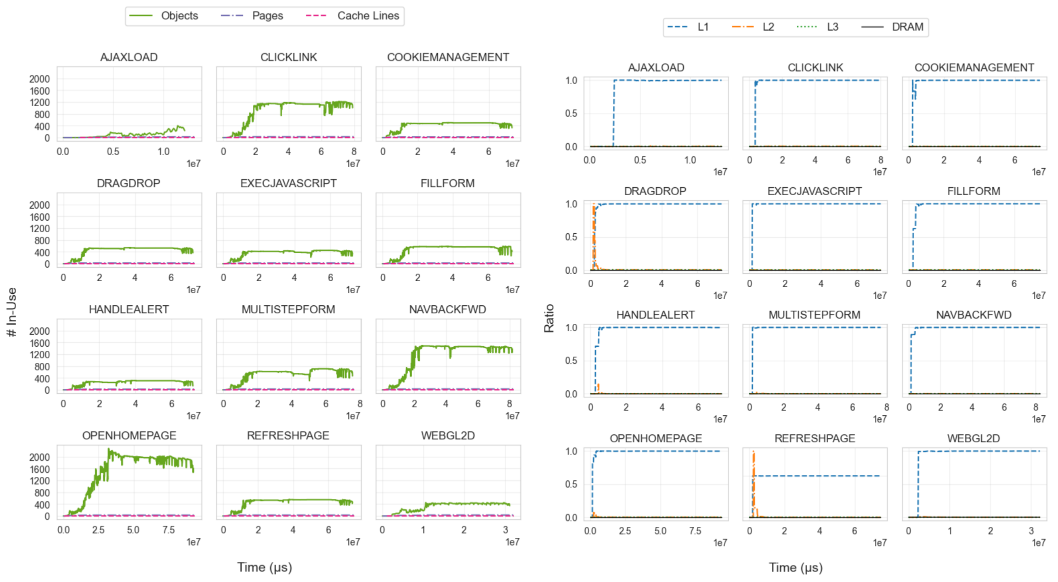
(b)

Fig. 11. (a) Live Objects, Pages, and LLC Lines. (b) Live vs In-Use Data Types.

and cache lines, suggesting significant compaction enabled by the memory allocators coupled with high locality.

In fact, resident page counts vary by at most ±10% (0.5-1.5K pages) and the working set size typically does not exceed the capacity of the L1 cache. This is confirmed in Figure 12b that shows that most of the accesses are L1 hits across all of the experiments. In some cases, particularly during the initialization phase, we observe that objects contend for the same L1 cache set, resulting in conflict misses, although this behavior is not observed once steady state is reached. Third, the number of distinct C++ data types of objects that are live or in-use are an order of magnitude smaller than the actual number of objects that are live or in-use, suggesting that multiple objects that belong to the same data type are typically used at once. This also aligns with the expectation that developers typically tend to work with a small set of objects and data types at any given point in time, despite the sheer number of classes and complex data types defined across all the modules and subsystems in Chromium. This behavior can be observed in Figure 11b.

Fourth, all runs exhibit a rapid increase in the number of live objects within the first 10-20 μs, following the initial rise, live-object counts stabilize and persist for 40-80μs. This expected behavior is cause by the initial setup procedure where lots of objects are created to support different modules of Chromium as discussed in Section-2. A secondary spike in live object counts is observed in most experiments between 35-38μs. This corresponds to Chromium's delayed asynchronous initialization strategy, where non-critical background tasks (e.g., metrics collection, service startup) are deferred to avoid blocking the main UI thread during early execution. AJAXLOAD and WEBGL2D do not exhibit this behavior, as the experiment duration is insufficient to reach the deferred initialization phase. In `OPENHOMEPAGE`, the dominant scale of the initial allocations visually obscures the spike.



(a) (b)

Fig. 12. (a) In-Use Objects, Pages and LLC Lines. (b) Ratio of L1, L2, L3 Hit and DRAM bound Loads. Total accesses: L1 hit + L1 miss + FB hit.

Insight 5

Despite millions of allocations, the active working set of objects is very small and typically fits within the L1 cache. These objects belong to a small set of distinct high-level data types, indicating that developers refrain from working with too many objects and data types at once.

Recommendation 5

Developers should continue to carefully optimize the active working set to be cache-friendly.

At $\approx 5.07 \times 10^7$ μs, OPENHOMEPAGE exhibits a sharp drop of ~34K live objects. These objects originate due to asynchronous iterable bindings implemented by the Blink engine for JavaScript. The objects serve as V8-backed wrappers and iterators bridging C++ and JavaScript asynchronous control flows. Their coordinated destruction via C++ destructors indicates the completion of a major asynchronous rendering phase. This drop aligns with the browser having fully loaded, suggesting that associated tasks, e.g., stream readers or worker-based iterators, are shutting down.

Insight 6

Most Chromium tasks are characterized by a rapid increase in the number of live objects, followed by a period of stabilization, and a secondary spike to delayed asynchronous initialization of objects from background non-critical threads.

Recommendation 6

Developers should manage deferred asynchronous initialization carefully so as to avoid sharp spikes in memory usage during performance/user experience-critical phases.

Recommendation 7

System designers should employ phase-aware memory management policies by using tight compaction and short-lifetime arenas during startup, transitioning to steady-state garbage collection tuning after stabilization, and triggering asynchronous cleanup post-deferred initialization phase.

5.4 Object-Centric Characterization of CPU microarchitectural Behavior

Software performance depends not only on the algorithmic complexity, but on how predictably it exercises the underlying μarchitectural optimizations. In Figure 13, we examine two key μarchitectural metrics that capture these behaviors: branch mispredictions per kilo instructions (MPKI) and heavy operations per kilo instructions (HPKI). MPKI quantifies branching entropy, he extent to which the processor's branch predictor fails to anticipate control-flow direction, causing pipeline flushes and front-end stalls. HPKI, on the other hand, reflects backend inefficiency, including cache-miss penalties and latency from complex arithmetic or memory-dependence chains. The particular Intel performance counter measuring *heavy operations* accounts for complex x86 instructions that break down into more than one micro-operation, which are typically common for pointer arithmetic instructions that employ register-memory addressing mode.

Traditional code-centric analyses study MPKI and HPKI at the level of individual algorithms or functions, seeking to restructure code or scheduling logic for improved predictor friendliness or data reuse. In contrast, we examine these effects from an object-centric perspective, and in particular, inspect how MPKI and HPKI vary when specific object types are accessed during Chromium's execution. This approach decouples performance from static code structure and instead relates it to dynamic object behavior, e.g., how data encapsulation and event-Our analysis indicates that this perspective exposes architectural hotspots.

First, we observe that accesses within the GPU layer (e.g., GrResourceAllocator) occur during phases of higher relative branching entropy (and thus higher branch MPKI) due to heavy use of pointer indirection and complex control flow in state-dependent checks (e.g., those required for resource eviction and synchronization). GrResourceAllocator, in particular, incorporates dynamically sized containers such as hash maps and live interval-tracking structures that not only entail complex control flow, but also irregular memory access patterns with poor locality. This memory-bounded nature of operations involving objects of the type GrResourceAllocator is also reflected in it high HPKI. Second, UI-related object types such as BrowserFrame are typically vulnerable to poor control flow predictability. This is because of their event-driven design that relies heavily on deeply nested virtual dispatches (e.g., browser_view_->browser()->profile()->GetProfileType()) that are often part of conditionals. Thus, accesses to BrowserFrame typically occurs during phases of severely elevated branch MPKI, which is a direct result of the branch history table being polluted from heterogeneous and unpredictable event sequences. Furthermore, nesting of operations (in this case, virtual dispatches that typically entail a chain of complex call instructions that decompose into two or more micro-operations) automatically implies the presence of long dependency chains and hence low instruction-level parallelism, resulting in low backend efficiency as well. Third, objects pertaining to the UI layer (e.g., ToolbarActionView, ContentSettingImageView, Tab) also

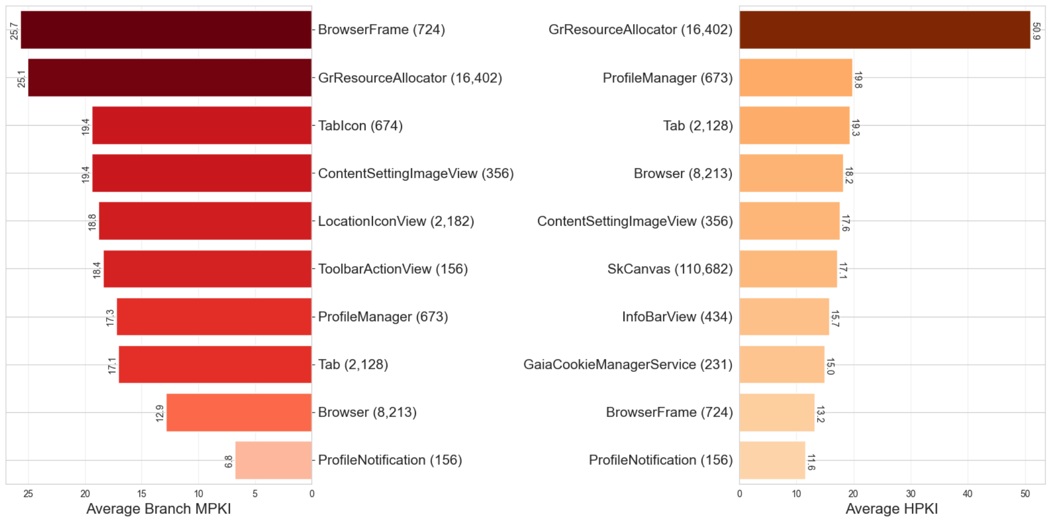


Fig. 13. Comparison of top object types exhibiting high branch misprediction rates (MPKI, left) versus heavy operation overhead (HPKI, right) across 12 experimental workloads.

suffer from conditional branch explosion from combinatorial state spaces, i.e., numerous feature flags, permissions, and transient view states resulting in deeply nested conditional hierarchies.

Notably, these inefficiencies arise from the very abstractions that make Chromium's software architecture expressive and modular. Event-driven designs, virtual dispatch hierarchies, and dynamically typed container patterns (e.g., hash maps and ref-counted handles) all optimize for software flexibility, composability, and algorithmic efficiency, yet they implicitly trade off μarchitectural predictability. Event-driven control introduces temporal irregularity that disrupts the branch predictor's history tables; virtual dispatch incurs indirect branches that the branch predictor cannot correlate across dynamic call targets; and hash-based containers, while reducing algorithmic time complexity, produce pointer-chasing access patterns with high cache-miss penalties and weak spatial locality. In essence, these abstractions optimize the big-O profile but degrade pipeline stability, cache efficiency, and branch predictability.

Reworking these abstractions to achieve the best of both worlds requires a shift from purely algorithmic optimization toward μarchitecturally informed abstraction design. For example, employing flat hash maps (Google's Swiss Tables) will allow us to retain O(1) access while improving cache contiguity. Similarly, replacing polymorphic dispatch with table-driven routing to convert indirect branches into direct indexed lookups, where possible, and batching or amortizing asynchronous event handling to align with predictable temporal windows, could allow the software to maintain its architectural abstraction boundaries while still exploiting the underlying μarchitectural optimizations.

Insight 7

Software abstractions that prioritize flexibility and algorithmic efficiency such as event-driven designs, virtual dispatch, and hash-based containers, could potentially introduce branching entropy and backend inefficiencies that fundamentally limit μarchitectural predictability and throughput.

Recommendation 8

Reconcile abstraction and performance by redesigning data structures and control paths for predictable access and locality using flat, contiguous containers, table-driven dispatch, and batched event handling to align software modularity with μarchitectural efficiency.

5.5 Optimization Workflow

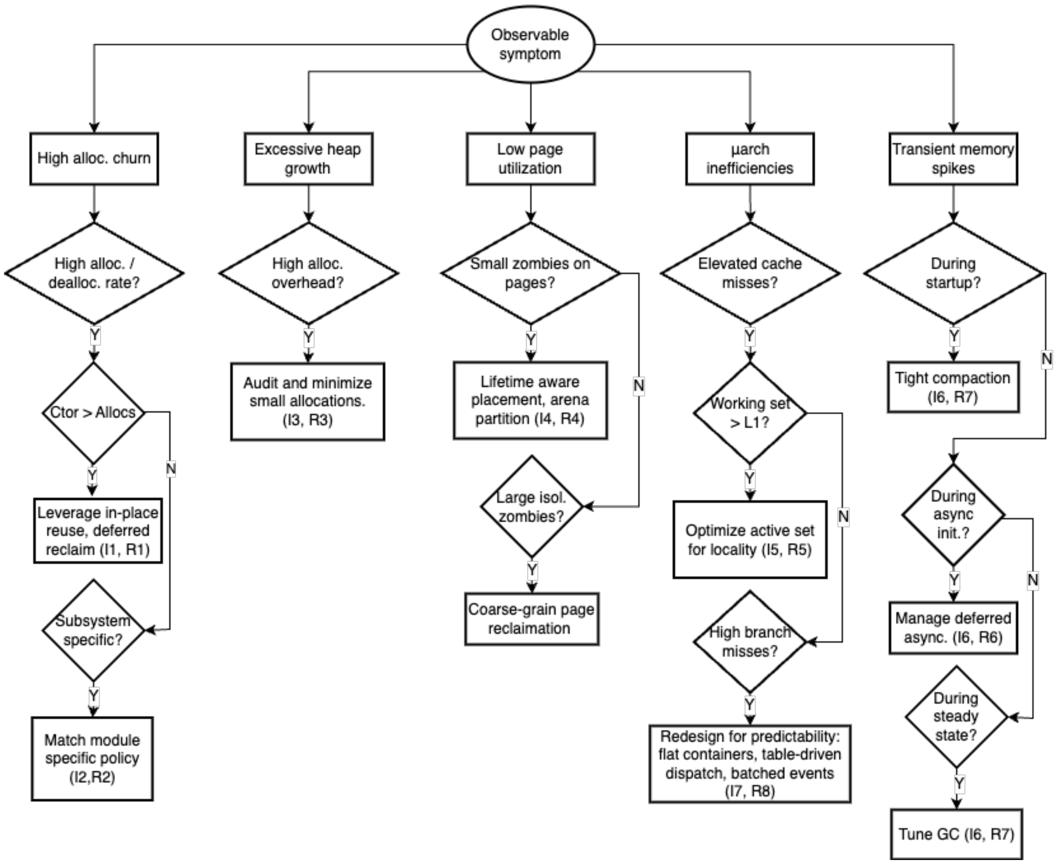

Observable symptom
High alloc. churn
Excessive heap growth
Low page utilization
µarch inefficiencies
Transient memory spikes
High alloc. / dealloc. rate?
High alloc. overhead?
Small zombies on pages?
Elevated cache misses?
During startup?
Y
N
Ctor > Allocs
Audit and minimize small allocations. (I3, R3)
Lifetime aware placement, arena partition (I4, R4)
Working set > L1?
Tight compaction (I6, R7)
Leverage in-place reuse, deferred reclaim (I1, R1)
Large isol. zombies?
Optimize active set for locality (I5, R5)
During async init.?
Subsystem specific?
Coarse-grain page reclamation
High branch misses?
Manage deferred async. (I6, R6)
Match module specific policy (I2,R2)
Redesign for predictability: flat containers, table-driven dispatch, batched events (I7, R8)
During steady state?
Tune GC (I6, R7)

Fig. 14. Symptom-driven optimization workflow synthesizing insights from object lifetime, size heterogeneity, spatial placement, and μarchitectural characterization. The diagnostic tree enables systematic identification of memory inefficiencies and maps them to actionable optimization opportunities.

Based on our insights and recommendations, we propose an optimization workflow for object-centric code optimizations in large-scale, object-oriented software systems. This workflow adopts a symptom-driven diagnostic approach, mapping observable metrics to actionable interventions. We identify the following five primary symptom categories, each leading to a decision tree depicted in Figure 14-

(1) **High allocation churn,** measured as combined allocation and deallocation events per second, indicates excessive allocator activity traceable to insufficient object reuse. The diagnostic process begins by examining constructor-to-allocation ratios: when constructors consistently exceed allocations, the subsystem can leverage in-place reuse or deferred reclamation, and

further optimization should focus on extending these patterns to additional object types. When allocations dominate, optimization should align memory management strategies with subsystem semantics by adopting module-specific policies such as arena allocation or memory pooling (Insights 1, 2).

(2) **Excessive heap growth,** observable as committed bytes increasing over time, often stems from small-object dominance rather than large allocations. In these cases, overhead accumulates in allocator metadata and internal fragmentation rather than application data, resulting in elevated per-byte costs that are not immediately apparent from aggregate memory statistics. Optimization should focus on auditing high-churn small allocations and consolidating objects with similar size classes and lifetimes into dedicated pools (Insight 3).

(3) **Low page utilization,** quantified as the ratio of live bytes to committed bytes, reflects inefficient page-level memory use caused by zombie objects that prevent wholesale reclamation. The spatial distribution of zombies determines the appropriate intervention: small zombies scattered across many pages require lifetime-aware placement and arena partitioning to concentrate them into reclaimable regions, whereas large isolated zombies occupying low-density pages are amenable to coarse-grained page reclamation through direct OS hints (Insight 4).

(4) μ**architectural inefficiency,** measured via hardware performance counters as misses per kilo-instruction (MPKI) or heavy operations per kilo-instruction (HPKI), arises from either working set overflow or abstraction-induced unpredictability. The diagnostic process first determines whether the active working set exceeds L1 cache capacity; if so, optimization should target improved locality through cache-friendly data layout and access pattern restructuring. If elevated miss rates persist despite a reasonably sized working set, the cause is typically branch entropy introduced by event-driven designs, virtual dispatch hierarchies, or pointer-chasing data structures. These inefficiencies may respond to abstraction redesign using flat containers, table-driven dispatch, and batched event handling (Insights 5, 7).

(5) **Transient memory spikes,** characterized by the ratio of peak to steady-state memory consumption, indicate phase-correlated memory pressure occurring during startup, asynchronous initialization, or steady-state transitions. Each phase benefits from distinct interventions: startup spikes respond to tight compaction and short-lifetime arenas; initialization-phase spikes require careful scheduling of deferred background tasks to avoid overlapping with user-facing activity; and steady-state fluctuations indicate opportunities for garbage collection tuning to align reclamation with natural idle periods (Insight 6).

Quantifying Optimization Potential.
Our analysis identifies three optimization axes with quantifiable impact. First, existing pooling and deferred reclamation strategies already save approximately 797.6M cycles per session by reducing allocator invocations from 2.09M to 7,903 operations and eliminating 1.39M unnecessary destructor calls. Disabling these mechanisms would result in significant slowdowns, with 240x and 3.33x reductions, respectively (see Section 5.1, ‘Object Events’). Second, lifetime-aware object placement offers additional memory efficiency gains. Currently, zombie objects co-reside with live allocations on the majority of partially occupied pages, preventing wholesale reclamation even when the effective utilization is low. By segregating short-lived small objects (under 64 bytes) into dedicated arenas, we can enable page-granularity reclamation. This reduces committed memory and improves TLB efficiency by reducing the number of resident pages and increasing the eligibility for huge-page promotion (see Section 5.2, ‘Page Pollution’). Third, μarchitectural optimizations targeting high-MPKI object types, particularly GPU resource managers and event-driven UI components, present opportunities for measurable improvements in IPC. Replacing

virtual dispatch chains with table-driven routing eliminates indirect branch mispredictions, while substituting pointer-chasing hash maps with flat containers improves cache line utilization and reduces the number of heavy operations (see Section 5.4). Collectively, these optimizations address distinct bottlenecks: memory pooling reduces allocation costs, lifetime-aware placement minimizes reclamation costs, and abstraction redesign restores μarchitectural efficiency lost due to increasing software modularity.

6 Related Work

In this section, we discuss related work spanning code instrumentation, memory profiling, and browser characterization. Table 3 summarizes key characteristics across representative techniques, situating our lightweight, lifetime-aware approach within the broader landscape.

Table 3. Summary of related work on code instrumentation, memory profiling, and browser characterization.

Method	**Category**	**Technique**	**Granularity**	**Limitation**
Pin [36]	Dynamic Binary	Register re-allocation, shadow mem.	Instruction	High runtime overhead
Valgrind [42]	Dynamic Binary	Runtime framework	Byte	Prohibitive for production
DynamoRIO [22]	Dynamic Binary	Kernel-level coverage	Instruction	Complex deployment
DTrace [21]	Process Probes	Zero-overhead probes	Function	Limited active granularity
ASan [48]	Compile-time	Shadow memory transforms	Byte	Error detection only
MSan [49]	Compile-time	Bit-precise shadow prop.	Bit	High overhead
MemoryInfra [4]	Runtime Profiling	MemoryDumpProvider-callbacks	Subsystem	No object detail; manual inst.
PROMPT [51]	Profiling Framework	Extensible multi-type	Configurable	Generic; not browser-specific
Resurrector [50]	Object Lifetime	Alloc/dealloc tracking	Object	Not browser-scale
SWAT [24]	Leak Detection	Adaptive statistical	Statistical	May miss infrequent leaks
CRAMM [52]	GC Profiling	Per-process ref. tracking	Reference	JS heap only
DMon [34]	Selective Profiling	Resource-bounded	Cache-level	Selective, not comprehensive
DevTools [10]	JS Heap Profiling	V8 heap snapshots	JS Object	No native allocations
WebCore [53]	Hardware Opt.	SRU + Browser Engine Cache	Architecture	Hardware modification req.
Kanev et al. [31]	Datacenter Profiling	Warehouse-scale analysis	System	Server-focused
Musleh et al. [37]	Mobile JS	Phase-specific profiling	Phase	JS-centric
Ogasawara [43]	Server JS	Runtime library profiling	Function	Server-side only
Hwang et al. [27]	Mobile I/O	Storage pattern analysis	I/O	I/O-focused
Radhakrishnan [45]	Browser-HW	Multi-browser profiling	System	No optimization framework

6.1 Code Instrumentation.

Code instrumentation is a common technique for logging, tracing, profiling, and optimization [28, 44, 47]. Instrumentation can occur during compilation (source-level) or execution (binary-level) [20, 42, 54]. Dynamic binary instrumentation frameworks like Pin [36] and Valgrind [42] enable runtime analysis without source code access, achieving efficient instrumentation through techniques like register re-allocation and shadow memory support. For kernel-level instrumentation, DynamoRIO extensions [22] provide comprehensive coverage of system-level code that browsers interact with during rendering and JavaScript execution. Production system instrumentation through sampling based approaches [5] and zero overhead frameworks like DTrace [21] help profile complex systems with minimal performance impact. Compile-time instrumentation offers advantages when source code is available. AddressSanitizer [48], widely used in Chromium development, achieves only 73% average slowdown for comprehensive memory error detection through efficient compile-time transformation. MemorySanitizer [49] demonstrates similar benefits with 2.5x execution overhead using bit-precise shadow memory propagation. A frequently used alternative involves instrumenting standard libraries instead of binaries, which enhances logging stability and is effective for micro benchmarks or small programs. However, it struggles in large projects due to possibility of self-triggering and results in execution pollution [39]. This method is unsuitable for Chromium, which overrides standard library calls via custom wrappers, including memory management (see Section 6.2). Hence, we implement compile-time instrumentation directly in the target binary.

6.2 Memory Profiling.

Chrome provides a timeline-based profiling tool called *MemoryInfra* [4], accessible via `chrome://tracing`. It offers coarse-grained visibility into memory usage across browser components but lacks object-level detail within individual modules. To log data, developers must register their components by implementing a `MemoryDumpProvider` and integrating it with the memory dump infrastructure. Modern profiling frameworks like PROMPT [51] provide extensible support for multiple profiling types including memory-dependence, value-pattern, object-lifetime, and points-to analysis with dramatically reduced implementation effort. Object lifetime profiling [50] enables optimization of real-world programs by tracking allocation and deallocation patterns. Shadow memory techniques [41] enable efficient metadata tracking by maintaining a shadow copy of application memory, forming the foundation for modern memory profilers and checkers. Memory leak detection in production environments requires low-overhead approaches. SWAT [24] uses adaptive statistical profiling to achieve less than 5% overhead, making continuous leak detection practical for live systems. For garbage-collected environments like JavaScript engines, CRAMM [52] demonstrates techniques for gathering per-process reference information at approximately 1% overhead. Selective profiling approaches like DMon [34] achieve only 1.36% average overhead through resource-bounded methodology that focuses on specific cache levels or memory based on identified bottlenecks. While tools like Chrome's DevTools Memory tab [10] provide insights into JavaScript heap usage, they often lack detailed information about native memory allocations and object-level granularity. This limitation hampers the ability to diagnose memory bloat and leaks effectively.

6.3 Hardware Optimizations for Browsers.

Browser workloads exhibit unique microarchitectural characteristics that differ fundamentally from traditional server applications. Microarchitectural profiling reveals that computation, rather than networking, is the primary bottleneck in modern browsers [30], with significant stress on instruction caches and memory hierarchies. Zhu et al. [53] proposed WebCore, a processor architecture optimized for mobile web workloads, incorporating a Style Resolution Unit (SRU) and Browser Engine Cache. Combined, these yield 22.2% performance gain, 18.6% energy savings, and up to 10× acceleration in specific tasks. Peters et al. [40] profiled browser activity on heterogeneous multi-processing (HMP) systems, breaking down CPU time and energy per thread and showing that DVFS, thread allocation, and power gating can significantly reduce power use. Hwang et al. [27] analyzed five mobile browsers, finding browsing to be write-intensive with 11.7× write amplification. Over 50% of I/Os relate to metadata/journaling, and 70% of Chrome's storage volume resides in SQLite. 57% of writes are synchronous, making up 68% of total write volume. Kanev et al. [31] profiled warehouse-scale computers running web-facing workloads, identifying a "datacenter tax" comprising nearly 30% of cycles from instruction cache misses and memory hierarchy stress. Web search workload analysis [29] reveals how web applications stress memory hierarchies differently than traditional server workloads, with implications for cache-conscious optimization. TailBench [33] provides comprehensive methodology for characterizing latency-critical web applications including cache behavior and memory hierarchy stress patterns. Musleh et al. [37] analyzed

JavaScript workloads on mobile, emphasizing memory pressure and phase-specific variability in execution due to inline caching and type predictability. They found that targeted optimizations for predictable phases outperform generalized techniques. Analysis of server-side JavaScript [43] reveals that 47.5% of CPU time is spent in C++ runtime libraries, highlighting cache and memory hierarchy challenges in engine implementations. GreenWeb [6] demonstrates how JavaScript patterns affect microarchitectural behavior including cache performance and energy consumption. Radhakrishnan [45] found major browsers (IE, Firefox, Chrome) *underutilize* system hardware across configurations, indicating inefficiencies in browser-hardware integration.

7 Conclusions

We present an empirical characterization of object allocation, lifetime, and memory behavior in Chromium using a lightweight, non-intrusive profiling framework. Our analysis across diverse workloads shows that while Chromium performs tens of millions of allocations, the number of live and in-use objects at any point remains small and exhibits strong spatial locality. These objects are densely packed into a limited number of pages and dominated by a small number of C++ types, reflecting compact and type-local runtime behavior.

We also observe that persistent and zombie objects, though limited in number, contribute disproportionately to memory fragmentation by hindering full-page reclamation. Small long-lived objects are especially prone to polluting memory, while large objects managed via arenas risk internal fragmentation when arena lifetimes exceed object lifetimes. These findings highlight structural challenges in memory reuse and point to the potential value of intent-aware memory management strategies in large-scale, event-driven systems.

Acknowledgments

The authors would like to thank the anonymous reviewers and the Shepherd for their insightful suggestions and comments. This research was supported by the NSF grant CCF-2238548 and a DARPA I2O cooperative agreement FA8750-24-2-0002.

References

[1] AddressSanitizer (ASan) - Chromium Docs.
[2] Blink (Rendering Engine).
[3] Browser Content Redirection | Citrix Workspace app for Windows.
[4] Chromium Docs - MemoryInfra.
[5] Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers.
[6] GreenWeb: Language Extensions for Energy-Efficient Mobile Web Computing (PLDI 2016 - Research Papers) - PLDI 2016.
[7] Heap Profiling with MemoryInfra.
[8] Intel® VTune™ Profiler.
[9] LeakSanitizer - Chromium Docs.
[10] Memory panel overview | Chrome DevTools.
[11] PartitionAlloc Design.
[12] A Performance Comparative on Most Popular Internet Web Browsers - ScienceDirect.
[13] Performance evaluation of web browsers in iOS platform | IEEE Conference Publication | IEEE Xplore.
[14] Scalable Dynamic Analysis of Browsers for Privacy and Performance | ACM SIGMETRICS Performance Evaluation Review.
[15] skia - Google.
[16] Task scheduling in Blink.
[17] WebKit/Speedometer: An open source repository for the Speedometer benchmark.
[18] facebook/rocksdb, Jan. 2026. original-date: 2012-11-30T06:16:18Z.
[19] FFmpeg/FFmpeg, Jan. 2026. original-date: 2011-04-14T14:12:38Z.
[20] Bernat, A. R., and Miller, B. P. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools* (New York, NY, USA, Sept. 2011), PASTE '11, Association for Computing Machinery, pp. 9–16.
[21] Cantrill, B. M., Shapiro, M. W., and Leventhal, A. H. Dynamic instrumentation of production systems. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (USA, June 2004), ATEC '04, USENIX Association, p. 2.
[22] Feiner, P., Brown, A. D., and Goel, A. Comprehensive kernel instrumentation via dynamic binary translation. *SIGARCH Comput. Archit. News 40*, 1 (Mar. 2012), 135–146.
[23] google. Chromium.
[24] Hauswirth, M., and Chilimbi, T. M. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, Oct. 2004), ASPLOS XI, Association for Computing Machinery, pp. 156–164.
[25] Hill, M. D., and Marty, M. R. Amdahl's Law in the Multicore Era. *Computer 41*, 7 (July 2008), 33–38.
[26] Hunter, A. H., Kennelly, C., Turner, P., Gove, D., Moseley, T., and Ranganathan, P. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. pp. 257–273.
[27] Hwang, T., Kim, M., Lee, S., and Won, Y. On the I/O characteristics of the mobile web browsers. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing* (New York, NY, USA, Apr. 2018), SAC '18, Association for Computing Machinery, pp. 964–966.
[28] Iwainsky, C., and Bischof, C. Calltree-Controlled Instrumentation for Low-Overhead Survey Measurements. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (May 2016), pp. 1668–1677.
[29] Janapa Reddi, V., Lee, B. C., Chilimbi, T., and Vaid, K. Web search using mobile cores: quantifying and mitigating the price of efficiency. *SIGARCH Comput. Archit. News 38*, 3 (June 2010), 314–325.
[30] Javad Nejati, Nejati, J., Aruna Balasubramanian, and Balasubramanian, A. An In-depth Study of Mobile Browser Performance. *The Web Conference* (Apr. 2016), 1305–1315. MAG ID: 2335825318 S2ID: d7c93547894236ee468a7cbe44d83cecc4368bd3.
[31] Kanev, S., Darago, J. P., Hazelwood, K., Ranganathan, P., Moseley, T., Wei, G.-Y., and Brooks, D. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (New York, NY, USA, June 2015), ISCA '15, Association for Computing Machinery, pp. 158–169.
[32] Kanev, S., Xi, S. L., Wei, G.-Y., and Brooks, D. Mallacc: Accelerating Memory Allocation. *SIGPLAN Not. 52*, 4 (Apr. 2017), 33–45.
[33] Kasture, H., and Sanchez, D. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)* (Sept. 2016), pp. 1–10.
[34] Khan, T. A., Neal, I., Pokam, G., Mozafari, B., and Kasikci, B. DMon: Efficient Detection and Correction of Data Locality Problems Using Selective Profiling. pp. 163–181.

[35] Lattner, C., and Adve, V. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* (Mar. 2004), pp. 75–86.

[36] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not. 40*, 6 (June 2005), 190–200.

[37] Malek Musleh, Musleh, M., Vijay S. Pai, and Pai, V. S. Architectural Characterization of Client-side JavaScript Workloads & Analysis of Software Optimizations. MAG ID: 797982555 S2ID: 68dcfb8e13a560e633ea09208e2075f62e69dfb3.

[38] markruss. VMMap - Sysinternals.

[39] Michael Factor, Factor, M., Assaf Schuster, Schuster, A., Konstantin Shagin, Konstantin Shagin, and Shagin, K. Instrumentation of standard libraries in object-oriented languages: the twin class hierarchy approach. *Conference on Object-Oriented Programming Systems, Languages, and Applications 39*, 10 (Oct. 2004), 288–300. MAG ID: 1994729403 S2ID: 65cd6de2de27ed1a105763da6772a5c2612bd55a.

[40] Nadja Peters, Peters, N., Sang-Young Park, Park, S., Samarjit Chakraborty, Chakraborty, S., Benedikt Meurer, Meurer, B., Hannes Payer, Payer, H., Daniel Clifford, and Clifford, D. K. Web browser workload characterization for power management on HMP platforms. *International Conference on Hardware/Software Codesign and System Synthesis* (Oct. 2016), 26. MAG ID: 2512591860 S2ID: ca1f9ce0e0f6cb15590b142fc307a569594c05f2.

[41] Nethercote, N., and Seward, J. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd international conference on Virtual execution environments* (New York, NY, USA, June 2007), VEE '07, Association for Computing Machinery, pp. 65–74.

[42] Nethercote, N., and Seward, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not. 42*, 6 (June 2007), 89–100.

[43] Ogasawara, T. Workload characterization of server-side JavaScript. In *2014 IEEE International Symposium on Workload Characterization (IISWC)* (Oct. 2014), pp. 13–21.

[44] Parragi, Z., and Porkolab, Z. Instrumentation of C++ Programs Using Automatic Source Code Transformations. *Studia Universitatis Babeș-Bolyai Informatica 63*, 2 (June 2018), 53–65. Number: 2.

[45] Radhakrishnan, J. Hardware dependency and performance of JavaScript engines used in popular browsers. In *2015 International Conference on Control Communication & Computing India (ICCC)* (Nov. 2015), pp. 681–684.

[46] Schulze, R., Schreiber, T., Yatsishin, I., Dahimene, R., and Milovidov, A. ClickHouse - Lightning Fast Analytics for Everyone, 2024. Issue: 12 Publication Title: Proceedings of the VLDB Endowment Volume: 17 original-date: 2016-06-02T08:28:18Z.

[47] Sebastian Kreutzer, Christian Iwainsky, Jan-Patrick Lehr, and Christian Bischof. Compiler-Assisted Instrumentation Selection for Large-Scale C++ Codes. *Springer International Publishing eBooks* (Jan. 2022), 5–19. MAG ID: 4313433611 S2ID: 246cfa5e772e86fdc9f19edd5663a6cb10ac115e.

[48] Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. AddressSanitizer: A Fast Address Sanity Checker. pp. 309–318.

[49] Stepanov, E., and Serebryany, K. MemorySanitizer: Fast detector of uninitialized memory use in C++. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (Feb. 2015), pp. 46–55.

[50] Xu, G. Resurrector: a tunable object lifetime profiling technique for optimizing real-world programs. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications* (New York, NY, USA, Oct. 2013), OOPSLA '13, Association for Computing Machinery, pp. 111–130.

[51] Xu, Z., Chon, Y., Su, Y., Tan, Z., Apostolakis, S., Campanoni, S., and August, D. I. PROMPT: A Fast and Extensible Memory Profiling Framework. *Artifact for Paper "PROMPT: A Fast and Extensible Memory Profiling Framework" 8*, OOPSLA1 (Apr. 2024), 110:449–110:473.

[52] Yang, T., Berger, E. D., Kaplan, S. F., and Moss, J. E. B. {CRAMM}: Virtual Memory Support for {Garbage-Collected} Applications.

[53] Yuhao Zhu, Zhu, Y., Vijay Janapa Reddi, and Reddi, V. J. WebCore: architectural support for mobileweb browsing. *International Symposium on Computer Architecture 42*, 3 (June 2014), 541–552. MAG ID: 2161176479 S2ID: 20d0b7473429464fc2f9bfd59d513d63c844551c.

[54] Zhang, M., Qiao, R., Hasabnis, N., and Sekar, R. A platform for secure static binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* (New York, NY, USA, Mar. 2014), VEE '14, Association for Computing Machinery, pp. 129–140.

Received October 2025; revised December 2025; accepted January 2026