Allocation Folding Based on Dominance

Daniel Clifford Hannes Payer Michael Starzinger Ben L. Titzer

Google

{danno,hpayer,mstarzinger,titzer}@google.com

Abstract

Memory management system performance is of increasing importance in today's managed languages. Two lingering sources of overhead are the direct costs of memory allocations and write barriers. This paper introduces allocation folding, an optimization technique where the virtual machine automatically folds multiple memory allocation operations in optimized code together into a single, larger allocation group. An allocation group comprises multiple objects and requires just a single bounds check in a bump-pointer style allocation, rather than a check for each individual object. More importantly, all objects allocated in a single allocation group are guaranteed to be contiguous after allocation and thus exist in the same generation, which makes it possible to statically remove write barriers for reference stores involving objects in the same allocation group. Unlike object inlining, object fusing, and object colocation, allocation folding requires no special connectivity or ownership relation between the objects in an allocation group. We present our analysis algorithm to determine when it is safe to fold allocations together and discuss our implementation in V8, an open-source, production JavaScript virtual machine. We present performance results for the Octane and Kraken benchmark suites and show that allocation folding is a strong performance improvement, even in the presence of some heap fragmentation. Additionally, we use four hand-selected benchmarks JPEGEncoder, NBody, Soft3D, and Textwriter where allocation folding has a large impact.

Categories and Subject Descriptors D3.4 [Programming Languages]: Processors compilers, memory management (garbage collection), optimization

General Terms Algorithms, Languages, Experimentation, Performance, Measurement

Keywords Dynamic Optimization, Garbage Collection, Memory Managment, Write barriers, JavaScript

1. Introduction

Applications that rely on automatic memory management are now everywhere, from traditional consumer desktop applications to large scale data analysis, high-performance web servers, financial trading platforms, to ever-more demanding websites, and even billions of mobile phones and embedded devices. Reducing the costs of automatic memory management is of principal importance in best utilizing computing resources across the entire spectrum.

Automatic memory management systems that rely on garbage collection introduce some overhead in the application's main execution path. While some garbage collection work can be made incremental, parallel, or even concurrent, the actual cost of executing allocation operations and write barriers still remains. This is even more apparent in collectors that target low pause time and require heavier write barriers.

This paper targets two of the most direct costs of garbage collection overhead on the application: the cost of allocation bounds checks and write barriers executed inline in application code. Our optimization technique, allocation folding, automatically groups multiple object allocations from multiple allocation sites in an optimized function into a single, larger allocation group. The allocation of an allocation group requires just a single bounds check in a bump-pointer style allocator, rather than one check per object. Even more importantly, our flow-sensitive compiler analysis that eliminates write barriers is vastly improved by allocation folding since a larger region of the optimized code can be proven not to require write barriers.

Allocation folding relies on just one dynamic invariant:

Invariant 1. Between two allocations A_1 and A_2, if no other operation that can move the object allocated at A_1 occurs, then space for the object allocated at A_2 could have been allocated at A_1 and then initialized at A_2, without ever having been observable to the garbage collector.

Our optimization exploits this invariant to group multiple allocations in an optimized function into a single, larger allocation. Individual objects can then be carved out of this larger region, without the garbage collector ever observing an intermediate state.

Allocation folding can be considered an optimization local to an optimized function. Unlike object inlining [5], object fusing [21], or object colocation [11], the objects that are put into an allocation group need not have any specific ownership or connectivity relationship. In fact, once the objects in a group are allocated and initialized, the garbage collector may reclaim, move, or promote them independently of each other. No static analysis is required, and the flow-sensitive analysis is local to an optimized function. Our technique ensures that allocation folding requires no special support from the garbage collector or the deoptimizer and does not interfere with other compiler optimizations. We implemented allocation folding in V8 [8], a high-performance open source virtual machine for JavaScript. Our implementation of allocation folding is part of the production V8 code base and is enabled by default since Chrome M30.

The rest of this paper is structured as follows. Section 2 describes the parts of the V8 JavaScript engine relevant to allocation folding, which includes the flow-sensitive analysis required for allocation folding and relevant details about the garbage collector and write barriers. Section 3 describes the allocation folding algorithm

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISMM '14, June 12, 2014, Edinburgh, UK.
Copyright © 2014 ACM 978-1-4503-2921-7/14/06. . . $15.00.
http://dx.doi.org/10.1145/2602988.2602994

and shows how allocation folding vastly widens the scope of write barrier elimination. Section 4 presents experimental results for allocation folding across a range of benchmarks which include the Octane [7] and Kraken [13] suites. Section 5 discusses related work followed by a conclusion in Section 6.

2. The V8 Engine

V8 [8] is an industrial-strength compile-only JavaScript virtual machine consisting of a quick, one-pass compiler that generates machine code that simulates an expression stack and a more aggressive optimizing compiler based on a static single assignment (SSA) intermediate representation (IR) called Crankshaft, which is triggered when a function becomes hot. V8 uses runtime type profiling and hidden classes [9] to create efficient representations for JavaScript objects. Crankshaft relies on type feedback gathered at runtime to perform aggressive speculative optimizations that target efficient property access, inlining of hot methods, and reducing arithmetic to primitives. Dynamic checks inserted into optimized code detect when speculation no longer holds, invalidating the optimization code. Deoptimization then transfers execution back to unoptimized code[1]. Such speculation is necessary to optimize for common cases that appear in JavaScript programs but that can nevertheless be violated by JavaScript's extremely liberal allowance for mutation. For example, unlike most statically-typed object-oriented languages, JavaScript allows adding and removing properties from objects by name, installing getters and setters (even for previously existing properties), and mutation of an object's prototype chain at essentially any point during execution. After adapting to JavaScript's vagaries, Crankshaft performs a suite of common classical compiler optimizations, including constant folding, strength reduction, dead code elimination, loop invariant code motion, type check elimination, load/store elimination, range analysis, bounds check removal and hoisting, and global value numbering. It uses a linear-scan SSA-based register allocator similar to that described by Wimmer [20].

V8 implements a generational garbage collector and employs write barriers to record references from the old generation to the young generation. Write barriers are partially generated inline in compiled code by both compilers. They consist of efficient inline flag checks and more expensive shared code that may record the field which is being written. For V8's garbage collector the write barriers also maintain the incremental marking invariant and record references to objects that will be relocated. Crankshaft can statically elide write barriers in some cases, e.g. if the object value being written is guaranteed to be immortal and will not be relocated, or if the object field being written resides in an object known to be in the young generation. The analysis for such elimination is given in Section 2.3.1.

2.1 Crankshaft IR

Crankshaft uses an SSA sparse-dataflow intermediate representation which is built directly from the JavaScript abstract syntax tree (AST). All important optimizations are performed on this IR. Instructions define values rather than virtual registers, which allows an instruction use to refer directly to the instruction definition, making move instructions unnecessary and improving pattern matching. Instructions are organized into basic blocks which are themselves organized into a control flow graph with branches and gotos, and `PHI` instructions merge values at control flow join points. SSA form guarantees that every instruction I_n is defined exactly once.

[1] V8 might be considered the most direct descendant of the Smalltalk → Self → HotSpot lineage of virtual machines that pioneered these techniques.

Instruction	Dep	Chg
I_n = PARAMETER[K]		
I_n = CONSTANT[K]		
I_n = ARITH(I, I)		
I_n = LOAD[field](object)	Ψ	
I_n = STORE[field](object, value)		Ψ
I_n = ALLOC[space](size)	Λ	Λ
I_n = INNER[offset, size](alloc)		
I_n = CALL(I...)	*	*
I_n = PHI(I...)		

Table 1: Simplified Crankshaft IR Instructions.

Every definition must dominate its uses, except for the inputs to `PHI` instructions.

Table 1 shows a simplified set of Crankshaft instructions that will be used throughout this paper. Statically known parts of an instruction, such as the field involved in a `LOAD` or `STORE`, or the value of a constant, are enclosed in square brackets []. The inputs to an instruction are given in parentheses () and must be references to dominating instructions. The table also lists the effects changed and depended on for each instruction. Effects will be discussed in Section 2.2.2. We elide the discussion of the more than 100 real Crankshaft instructions which are not relevant to this paper.

2.2 Global Value Numbering

The analysis required to detect opportunities for allocation folding is implemented as part of the existing flow-sensitive global value numbering (GVN) algorithm in Crankshaft. Global value numbering eliminates redundant computations when it is possible to do so without affecting the semantics of the overall program. Extending GVN to handle impure operations gives the necessary flow-sensitivity for identifying candidates for allocation folding.

2.2.1 GVN for Pure Operations

GVN traditionally targets pure computations in the program such as arithmetic on primitives, math functions, and accesses to immutable data. Because such operations always compute the same result and neither produce nor are affected by side-effects, it is safe to hoist such computations out of loops or reuse the result from a previous occurrence of the same operation on the same inputs.

For each basic block in the method, the value numbering algorithm visits the instructions in control flow order, putting pure instructions into a value numbering table. In our simplified Crankshaft instruction set depicted in Table 1, we consider all arithmetic instructions ARITH(I_i, I_j) to be pure instructions[2]. Two instructions are value-equivalent if they are the same operation (e.g. both `ADD` or both `SUB`) and the inputs are identical SSA values. If a value-equivalent instruction already exists in the table, then the second instruction is redundant. The second instruction is removed, and all of its uses are updated to reference the first instruction.

Crankshaft uses the dominator tree of the control flow graph to extend local value numbering to the entire control flow graph. The dominator tree captures the standard dominance relation for basic blocks: a basic block D dominates basic block B if and only if D appears on every path from the function entry to B. It is

[2] In JavaScript, all operations are untyped. Arithmetic on objects could result in calls to application-defined methods that have arbitrary side-effects. In V8, a complex system of type profiling with inline caches, some static type inference during compilation, and some speculative checks in optimized code guard operations that have been assumed to apply only to primitives.

straightforward to extend the dominator relation on basic blocks to instructions, since instructions are ordered inside of basic blocks.

GVN applies local value numbering to each basic block in dominator tree order, starting at the function entry. Instead of starting with an empty value numbering table at the beginning of each block, the value numbering table from a dominating block D is copied and used as the starting table when processing each of its immediately dominated children B. By the definition of dominance, a block D dominating block B appears on every control flow path from the start to B. Therefore any instruction I_2 in B which is equivalent to I_1 in D is redundant and can be safely replaced by I_1. Since Crankshaft's SSA form guarantees that every definition must dominate its usages, the algorithm is guaranteed to find all fully redundant computations[3].

2.2.2 GVN for Impure Operations

Crankshaft extends the GVN algorithm to handle some instructions that can be affected by side-effects, but are nevertheless redundant if no such side-effects can happen between redundant occurrences of the same instruction. Extending GVN to impure instructions by explicitly tracking side-effects is the key analysis needed for allocation folding.

We illustrate the tracking of side-effects during GVN with a simple form of redundant load elimination. A load $\mathtt{L_2} = \mathtt{LOAD[field](O_i)}$ can be replaced with a previous load of the same field $\mathtt{L_1} = \mathtt{LOAD[field](O_i)}$ if $\mathtt{L_1}$ dominates $\mathtt{L_2}$ and no intervening operation could have modified the field of the object on any path between $\mathtt{L_1}$ and $\mathtt{L_2}$.

For load elimination, we consider `LOAD` and `STORE` instructions and an abstraction of the state in the heap. For the sake of illustration, in this section we will model all the state in the heap with a single effect Ψ, but for finer granularity, one could model multiple non-overlapping heap abstractions with individual side-effects $\Psi_\mathtt{f}$, e.g. one for each field $\mathtt{f}$[4]. Stores change Ψ and loads depend on Ψ. `CALL` instructions are conservatively considered to change all possible side-effects, so we consider them to also change Ψ.

While previously only pure instructions were allowed to be added to the value numbering table, now we also allow instructions that depend on side-effects to be added to the table, and each entry in the value numbering table also records the effects on which the instruction depends. When processing a load $\mathtt{L_1} = \mathtt{LOAD[field](O_i)}$, it is inserted into the table and marked as depending on effect Ψ. A later load $\mathtt{L_2} = \mathtt{LOAD[field](O_i)}$ might be encountered. Such a load is redundant if the value numbering table contains $\mathtt{L_1}$. When an instruction that changes a side-effect is encountered, any entry in the value numbering table that depends on that effect is invalidated. Thus any store $\mathtt{S_1} = \mathtt{STORE[field](O_i, V_j)}$ causes all instructions in the table that depend on Ψ to be removed, so that subsequent loads cannot reuse values from before the store.

We would like to use the idea above to perform global value numbering for instructions that can be affected by side-effects across the entire control flow graph. Unfortunately, it is not enough just to rely on the effects we encounter as we walk down the dominator tree, as we did in the previous algorithm. The dominator tree only guarantees that a dominator block appears on every path from the start to its dominated block, but other blocks can appear between the dominator and the dominated block. To correctly account for side-effects, we must process the effects on all paths from a dominator block to its children blocks.

[3] By induction on the structure of instructions.

[4] The actual load elimination algorithm in Crankshaft models several non-overlapping heap memory abstractions and also performs a limited alias analysis.

To perform this analysis efficiently, we first perform a linear pass over the control flow graph, computing an unordered set of effects that are produced by each block. Loops require extra care. Assuming a reducible flow graph, each loop has a unique header block which is the only block from which the loop can be entered. A loop header block is marked specially and contains the union of effects for all blocks in the loop. When traversing the dominator tree, if the child node is a loop header, then all instructions in the value numbering table that depend on the loop effects are first invalidated.

Armed with the pre-computed effect summaries for each block, the GVN algorithm can process the effects on all paths between a dominator and its children by first starting at the child block and walking the control flow edges backward, invalidating entries in the value numbering table that depend on the summary effects from each block, until the dominator block is reached. Such a walk is worst-case $O(E)$, since the dominator block may be the start block and the child block may be the end block, leading to an overall worst-case of $O(E * N)$, where E is the number of edges and N is the number of blocks. In practice, most dominator-child relationships have zero non-dominating paths, so this step is usually a no-op. Our implementation also employs several tricks to avoid the worst-case complexity, such as memoizing some path traversals and terminating early when the value numbering table no longer contains impure instructions, but the details are not relevant to the scope of this paper.

2.2.3 Side-Effect Dominators

Each effect [illegible] induces a global flow-sensitive relation on instructions that depend on [illegible] and instructions that change [illegible] We call this relation [illegible]dominance.

Definition 1. For a given effect [illegible] instruction D [illegible]dominates instruction I if and only if D occurs on every path from the function entry to I, and no path from D to I contains another instruction $D' \neq D$ that changes [illegible]

Given this new definition, it is easy to restate load elimination.

Predicate 1. A load $\mathtt{L_2} = \mathtt{LOAD[field_j](O_i)}$ can be replaced with $\mathtt{L_1} = \mathtt{LOAD[field_j](O_i)}$ if $\mathtt{L_1}$ Ψ-dominates $\mathtt{L_2}$.

We can also define an [illegible]dominator.

Definition 2. For a given effect [illegible] instruction D is the [illegible]dominator of instruction I if and only if D [illegible]dominates I and D changes [illegible]

It follows immediately from the definition of [illegible]dominance that an instruction can have at most one [illegible]dominator.

GVN for impure values computes both [illegible]dominance and the unique [illegible]dominator during its traversal of the instructions. It provides the [illegible]dominator as an API to the rest of the compiler. Crankshaft uses it for both allocation folding and for write barrier elimination, both of which are detailed in the following sections.

2.3 Write Barriers

V8 employs a generational garbage collector, using a semi-space strategy for frequent minor collections of the young generation, and a mark-and-sweep collector with incremental marking for major collections of the old generation. Write barriers emitted inline in compiled code track inter-generational pointers and maintain the marking invariant between incremental phases. Every store into an object on the garbage collected heap may require a write barrier, unless the compiler can prove the barrier to be redundant. This section details the tasks a write barrier must perform and some of the implementation details to understand the runtime overhead introduced by write barriers, and then explores conditions under which it is permissible to statically eliminate write barriers (Section 2.3.1).

Write barriers in V8 perform three main tasks to ensure correct behavior of the garbage collector while mutators are accessing objects on the garbage collected heap.

- Track Inter-generational Pointers: References stored into the old generation pointing to an object in the young generation are recorded in a store buffer. The store buffer becomes part of the root-set for minor collections, allowing the garbage collector to perform a minor collection without considering the entire heap. Every mutation of an object in the old generation potentially introduces an old-to-young reference.
- Maintain Marking Invariant: During the marking phase of a major collection, a standard marking scheme gives each object one of three colors: white for objects not yet seen by the garbage collector, gray for objects seen but not yet scanned by the garbage collector, and black for objects fully scanned by the garbage collector. The marking invariant is that black objects cannot reference white objects. To reduce the pause time of major collections, V8 interleaves the marking phase with mutator execution and performs stepwise incremental marking until the transitive closure of all reachable objects has been found. The write barrier must maintain the marking invariant for objects in the old generation, since every mutation of an object in the old generation could potentially introduce a black-to-white reference. Newly allocated objects are guaranteed to be white and hence cannot break the marking invariant.
- Pointers into Evacuation Candidates: To reduce fragmentation of certain regions of the heap, the garbage collector might mark fragmented pages as evacuation candidates before the marking phase starts. Objects on these pages will be relocated onto other, less fragmented pages, freeing the evacuated pages. The marking phase records all references pointing into these evacuation candidates in a buffer so that references can be updated once the target object has been relocated. As before, objects in the young generation are fully scanned during a major collection and their references don't need to be recorded explicitly. Every mutation of an object in the old generation potentially introduces a reference pointing to an evacuation candidate.

```
 1 store:
 2   mov    [$obj+field], $val
 3 barrier:
 4   and    $val, 0xfff00000
 5   test_b [$val+PAGE_FLAGS], VALUES_INTERESTING
 6   jz     skip
 7   mov    $val, 0xfff00000
 8   and    $val, $obj
 9   test_b [$val+PAGE_FLAGS], FIELDS_INTERESTING
10   jz     skip
11   call   RecordWriteStub($obj, field)
12 skip:
13   ...
```

Listing 1: Inlined write barrier assembly on IA32

The above three tasks require an efficient yet compact implementation of the write barrier code. This is achieved by splitting the write barrier into two parts: one that is emitted inline with the compiled code, and out-of-line code stubs. The assembly code in Listing 1 shows the instructions being emitted inline for an IA32 processor. After performing the store to the field (Line 2), the write barrier first checks whether the referenced object `$val` is situated on a page where values are considered interesting (Lines 4 to 6). It then checks whether the receiver object `$obj` is situated on a page whose fields are considered interesting (Lines 7 to 10). These checks perform bit mask tests of the page flags for the pages[5] on which the respective objects are situated. The code stubs recording the store are only called in case both checks succeed (Line 11). The write barrier can be removed if the compiler can statically determine that at least one of the checks will always fail.

During execution the garbage collector may change the page flags `VALUES_INTERESTING` and `FIELDS_INTERESTING` which are continuously checked by write barriers.

2.3.1 Write Barrier Elimination

Under some conditions it is possible to statically remove write barriers. Stores whose receiver object is guaranteed to be newly allocated in the young generation never need to be recorded. Such stores cannot introduce old-to-young references, they cannot break the marking invariant as newly allocated objects are white, and finally their fields will be updated automatically in case they point into evacuation candidates.

Using the GVN algorithm which handles side-effecting instructions, we introduce a new effect Λ, which tracks the last instruction that could trigger a garbage collection. We say that allocations, meaning instructions of the form $I_1 = \mathrm{ALLOC}[s](K_1)$, both change and depend on Λ. We consider all `CALL` instructions to have uncontrollable effects, so they implicitly also change Λ, as with Ψ.

With Λ, it is easy for Crankshaft to analyze store instructions and remove write barriers to objects guaranteed to be newly allocated in the young generation:

Predicate 2. $S_1 = \mathrm{STORE}[\mathrm{field}](O_1, V_1)$ does not require a write barrier if O_1 has the form $O_1 = \mathrm{ALLOC}[\mathrm{young}](I_1)$ and O_1 Λ-dominates S_1.

This approach to write barrier elimination is limited in that it can only remove write barriers for the most recently allocated young space object. As we will see in the next section, allocation folding enlarges the scope for write barrier elimination.

3. Allocation Folding

Allocation folding groups multiple allocations together into a single chunk of memory when it is safe to do so without being observable to the garbage collector. In terms of Crankshaft IR instructions, this means replacing some `ALLOC` instructions with `INNER` instructions. `ALLOC` allocates a contiguous chunk of memory of a given size, performing a garbage collection if necessary. `INNER` computes the effective address of a sub-region within a previously allocated chunk of memory and has no side-effects. According to Invariant 1, we can fold two allocations together if there is no intervening operation that can move the first allocated object. We can use that dynamic invariant to formulate the allocation folding opportunities on Crankshaft IR:

Predicate 3. Allocations $A_1 = \mathrm{ALLOC}[s](K_1)$ and $A_2 = \mathrm{ALLOC}[s](K_2)$ are candidates for allocation folding if A_1 is the Λ-dominator of A_2.

When candidates are identified, allocation folding is a simple local transformation of the code. If allocation $A_1 = \mathrm{ALLOC}[s](K_1)$ is the Λ-dominator of allocation $A_2 = \mathrm{ALLOC}[s](K_2)$, then a single instruction $A_{new} = \mathrm{ALLOC}[s](K_1 + K_2)$ can be inserted immediately before A_1, and A_1 can be replaced with $A_1' = \mathrm{INNER}[\#\,0, K_1](A_{new})$ and A_2 can be replaced with $A_2' = \mathrm{INNER}[K_1, K_2](A_{new})$.

Figure 1 presents an example control flow graph before allocation folding has been performed. The dominator tree is shown in light gray and is marked with the effects for each block. Blocks

[5] All pages in the collected heap are aligned at megabyte boundaries, hence computing the page header from an arbitrary object reference is a single bitmask.