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

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 $\mathtt{I_n}$ is defined exactly once. Every definition must dominate its uses, except for the inputs to PHI instructions.

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

Table 1: Simplified Crankshaft IR 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 $\mathtt{ARITH(I_i, I_j)}$ 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

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

[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 $\mathtt{L_2} = \mathtt{LOAD}[\mathtt{field}](\mathtt{O_i})$ can be replaced with a previous load of the same field $\mathtt{L_1} = \mathtt{LOAD}[\mathtt{field}](\mathtt{O_i})$ 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 $\mathtt{L_1} = \mathtt{LOAD}[\mathtt{field}](\mathtt{O_i})$, it is inserted into the table and marked as depending on effect Ψ. A later load $\mathtt{L_2} = \mathtt{LOAD}[\mathtt{field}](\mathtt{O_i})$ 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 $\mathtt{S_1} = \mathtt{STORE}[\mathtt{field}](\mathtt{O_i}, \mathtt{V_j})$ 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 ϵ induces a global flow-sensitive relation on instructions that depend on ϵ and instructions that change ϵ. We call this relation *ϵ-dominance*.

Definition 1. *For a given effect ϵ, instruction D ϵ-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 ϵ.*

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

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

We can also define an *ϵ-dominator*.

Definition 2. *For a given effect ϵ, instruction D is the ϵ-dominator of instruction I if and only if D ϵ-dominates I and D changes ϵ.*

It follows immediately from the definition of ϵ-dominance that an instruction can have at most one ϵ-dominator.

GVN for impure values computes both ϵ-dominance and the unique ϵ-dominator during its traversal of the instructions. It provides the ϵ-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 $\mathtt{I}_1 = \mathtt{ALLOC}[\mathtt{s}](\mathtt{K}_1)$, 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. $\mathtt{S}_1 = \mathtt{STORE}[\mathtt{field}](\mathtt{O}_1, \mathtt{V}_1)$ *does not require a write barrier if* $\mathtt{O}_1$ *has the form* $\mathtt{O}_1 = \mathtt{ALLOC}[\mathtt{young}](\mathtt{I}_1)$ *and* $\mathtt{O}_1$ *Λ-dominates* $\mathtt{S}_1$.

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

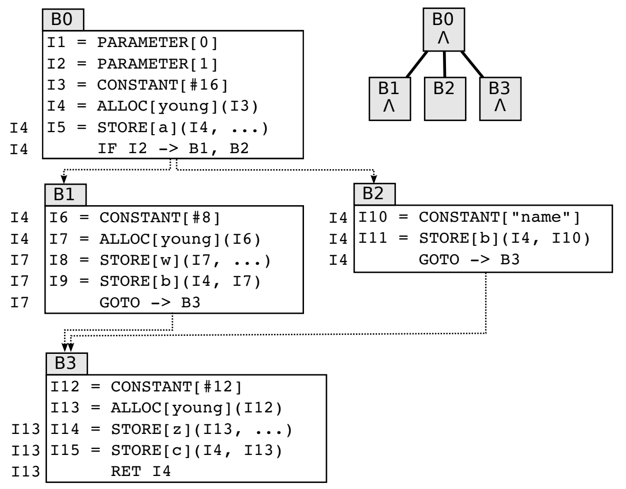
Allocation folding groups multiple allocations together into a single chunk of memory when it is safe to do so without being observable to the garbage collector. In terms of Crankshaft IR instructions, this means replacing some `ALLOC` instructions with `INNER` instructions. `ALLOC` allocates a contiguous 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* $\mathtt{A}_1 = \mathtt{ALLOC}[\mathtt{s}](\mathtt{K}_1)$ *and* $\mathtt{A}_2 = \mathtt{ALLOC}[\mathtt{s}](\mathtt{K}_2)$ *are candidates for allocation folding if* $\mathtt{A}_1$ *is the Λ-dominator of* $\mathtt{A}_2$.

When candidates are identified, allocation folding is a simple local transformation of the code. If allocation $\mathtt{A}_1 = \mathtt{ALLOC}[\mathtt{s}](\mathtt{K}_1)$ is the Λ-dominator of allocation $\mathtt{A}_2 = \mathtt{ALLOC}[\mathtt{s}](\mathtt{K}_2)$, then a single instruction $\mathtt{A}_{\mathtt{new}} = \mathtt{ALLOC}[\mathtt{s}](\mathtt{K}_1 + \mathtt{K}_2)$ can be inserted immediately before $\mathtt{A}_1$, and $\mathtt{A}_1$ can be replaced with $\mathtt{A}_1' = \mathtt{INNER}[\#0, \mathtt{K}_1](\mathtt{A}_{\mathtt{new}})$ and $\mathtt{A}_2$ can be replaced with $\mathtt{A}_2' = \mathtt{INNER}[\mathtt{K}_1, \mathtt{K}_2](\mathtt{A}_{\mathtt{new}})$.

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.

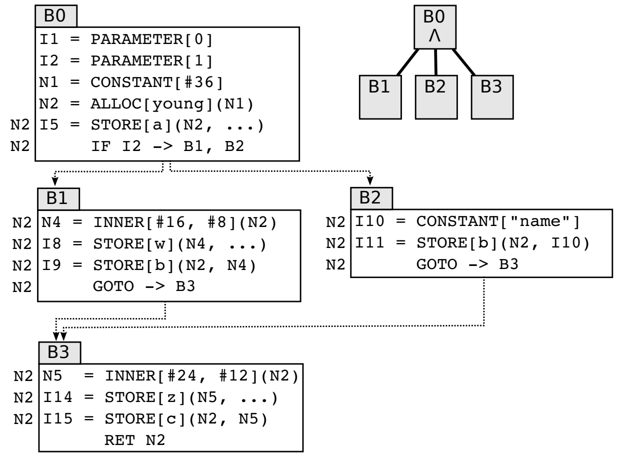

```
B0
I1 = PARAMETER[0]
I2 = PARAMETER[1]
I3 = CONSTANT[#16]
I4 = ALLOC[young](I3)
I4  I5 = STORE[a](I4, ...)
I4       IF I2 -> B1, B2

B1
I4  I6 = CONSTANT[#8]
I4  I7 = ALLOC[young](I6)
I7  I8 = STORE[w](I7, ...)
I7  I9 = STORE[b](I4, I7)
I7       GOTO -> B3

B2
I4  I10 = CONSTANT["name"]
I4  I11 = STORE[b](I4, I10)
I4        GOTO -> B3

B3
I12 = CONSTANT[#12]
I13 = ALLOC[young](I12)
I13  I14 = STORE[z](I13, ...)
I13  I15 = STORE[c](I4, I13)
I13        RET I4
```

Figure 1: Example CFG before allocation folding.


```
B0
I1 = PARAMETER[0]
I2 = PARAMETER[1]
N1 = CONSTANT[#36]
N2 = ALLOC[young](N1)
N2  I5 = STORE[a](N2, ...)
N2       IF I2 -> B1, B2

B1
N2  N4 = INNER[#16, #8](N2)
N2  I8 = STORE[w](N4, ...)
N2  I9 = STORE[b](N2, N4)
N2       GOTO -> B3

B2
N2  I10 = CONSTANT["name"]
N2  I11 = STORE[b](N2, I10)
N2        GOTO -> B3

B3
N2  N5  = INNER[#24, #12](N2)
N2  I14 = STORE[z](N5, ...)
N2  I15 = STORE[c](N2, N5)
          RET N2
```

Figure 2: Example CFG after allocation folding.

B0, B1, and B3 each contain an allocation instruction, therefore each is marked as changing Λ. The Λ-dominator is shown to the left of each instruction, outside the basic block. Note that some instructions, such as I12 and I13 do not have a Λ-dominator. In Figure 1, we can see that some, but not all, write barriers can be eliminated through local analysis. Write barriers associated with stores I8, I11, and I14 can be eliminated, since we can see that their Λ-dominator is the receiver of the store, and that receiver is an allocation in the young generation. However, write barriers associated with stores I9 and I15 cannot be eliminated because their Λ-dominator does not match the receiver object of the store.

Figure 2 shows the control flow graph from Figure 1 after allocation folding has been performed. Some instructions have been removed, and new instructions $\mathrm{N_n}$ have been inserted. The allocations in blocks B0, B1, and B3 have been folded into one larger allocation[6] in B0 and are replaced by INNER instructions that carve out individual objects from the allocation group. We can see that removing these allocations removes the Λ from these blocks because INNER instructions do not change Λ. By replacing ALLOC instructions with INNER instructions the number of program points at which garbage collection can happen is reduced. This then increases the opportunities for local write barrier elimination. The opportunities are evident in the changes to the Λ-dominators for each instruction. After allocation folding, the single, larger allocation Λ-dominates all the stores. All stores in the example are now into objects allocated from the same allocation group, which is allocated in the young generation. Since we know that stores into objects in the young generation cannot introduce old-to-young references, all write barriers in this example can be removed.

In this example we can see how allocation folding can give rise to memory fragmentation. If at runtime the code follows the path B0 → B2 → B3, then the space reserved for the inner allocation at N4 will have been allocated but not be used because we do not overlap the space reserved for the folded allocations. A straightforward approach to avoiding this source of memory fragmentation is to only fold allocations in the same basic block. We compare allocation folding with and without the basic block restriction and study the overhead of fragmentation by measuring the amount of each allocation group that is actually used, or the *allocation group utilization*, in Section 4.

Memory fragmentation gives rise to uninitialized memory regions between objects in the heap. This requires the garbage collector to be capable of handling a non-iterable heap. As a consequence a mark-and-sweep garbage collector must store the mark bits outside objects.

3.1 Allocation Folding in Crankshaft

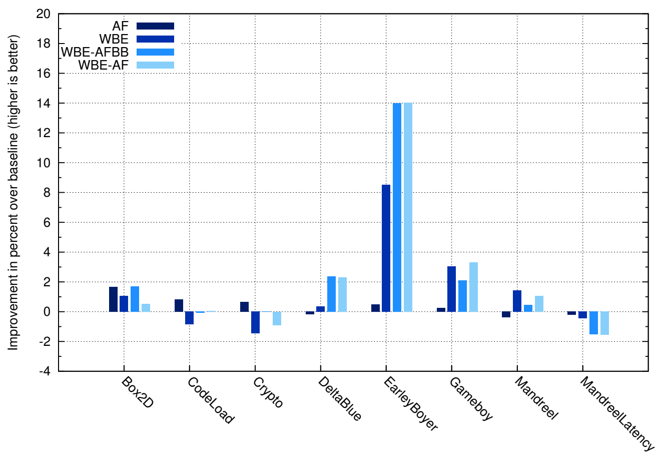
We present the pseudo-code of the allocation folding algorithm in Crankshaft in Listing 2. We perform allocation folding as part of GVN, after performing aggressive inlining, so that the maximum number of folding opportunities are available.

```
 1 HAllocate::HandleSideEffectDominator(dominator):
 2   if !dominator->IsAllocate():
 3     return;
 4   if AllocationFoldingBasicBlockMode() &&
 5       this->BlockID() != dominator->BlockID():
 6     return;
 7   dominator_size = dominator->Size();
 8   size = this->Size();
 9   if !dominator_size->IsConstant() ||
10      !size->IsConstant():
11     return;
12   new_size = dominator_size + size;
13   if this->DoubleAligned():
14     if !dominator->DoubleAligned():
15       dominator->SetDoubleAligned(true);
16     if IsDoubleAligned(dominator_size):
17       dominator_size += DoubleSize()/2;
18       new_size += DoubleSize()/2;
19   if new_size > MaxAllocationFoldingSize():
20     return;
21   new_size_instruction =
22     HConstant::CreateAndInsertBefore(
23       new_size, dominator);
24   dominator->UpdateSize(new_size_instruction);
25   inner_allocated_object_instruction =
26     HInnerAllocatedObject::New(
27       dominator, dominator_size);
28   this->DeleteAndReplaceWith(
29     inner_allocated_object_instruction);
```

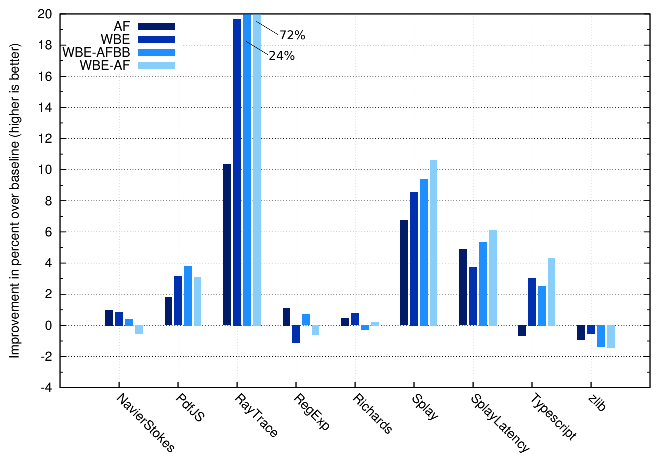
Listing 2: Allocation folding algorithm

A given allocation instruction can only be folded into its Λ-dominator if that Λ-dominator is itself an allocation instruction (Line 2). If the basic block restriction is enabled (Line 3), then only allocations in the same basic block will be folded (Line 4).

[6] Note that allocation I13 has no Λ-dominator until allocation I7 has been folded into I4. In general, allocation folding can be applied again whenever it introduces a new Λ-dominator for an allocation that previously did not have one due to merges in the control flow.



(a) Part 1



(b) Part 2

Figure 3: Improvement in percent of all configurations over the baseline on the Octane suite running on X64.

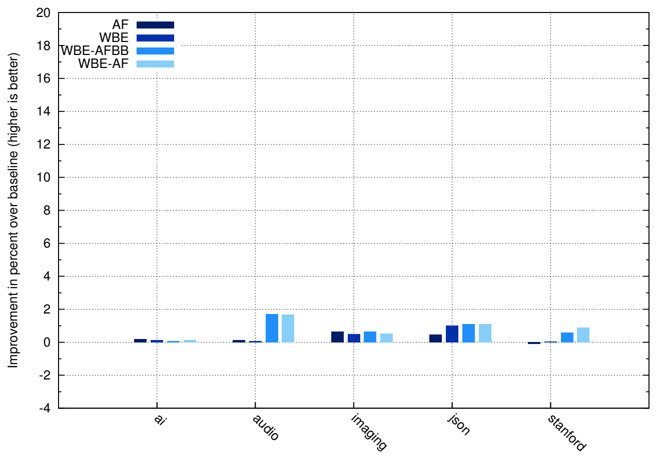


Figure 4: Improvement in percent of all configurations over the baseline on the Kraken suite running on X64.

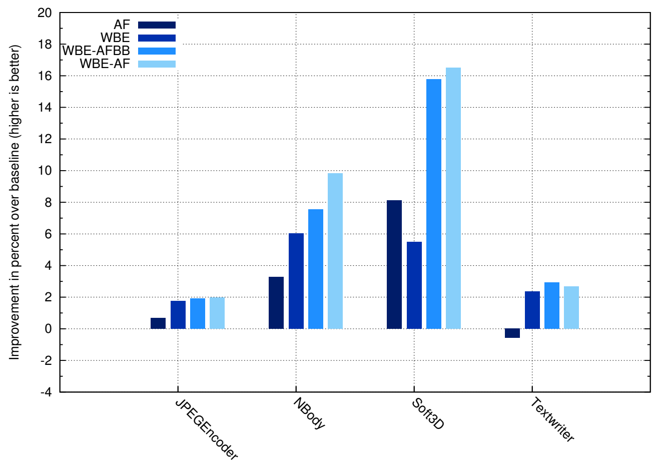


Figure 5: Improvement in percent of all configurations over the baseline on hand-selected benchmarks running on X64.

The allocation size must be a constant[7] (Line 9 and Line 10). The size of the the new dominator allocation instruction is the sum of the sizes of the given allocation instruction and its Λ-dominator (Line 12). If the given allocation instruction requires double alignment (Line 10) the Λ-dominator must be aligned as well and the extra space accounted for if necessary (Line 17 and Line 18). If the new allocation would be larger than a maximum size (a constant determined based on the size of the young generation) then the algorithm will not do the folding (Line 19). If all criteria are satisfied, the algorithm increases the size of the Λ-dominator allocation instruction (Line 24) and creates a new inner-allocate (`INNER`) instruction which refers to the end of the previous allocation group (Line 25). All uses of the previous instruction are replaced with uses of the new inner-allocate instruction (Line 28).

[7] Folding non-constant size allocations is possible in principle, but the gritty details means a lot of graph rewriting, since the computed sizes also need to be hoisted.

4. Experiments

We ran V8 revision r18926 on an X64 server machine with an Intel Core i5-2400 quad-core 3.10GHz CPU and 80GB of main memory running Linux. We performed the same experiments on IA32 and ARM, but found that the performance results were in such close agreement with X64 that we gained no new insights. We therefore chose to omit redundant data for space reasons.

For our experiments we used the complete Octane 2.0 [7] and Kraken 1.1 [13] suites, two standard JavaScript benchmarks which are designed to test specific virtual machine subsystems. In both cases we run each benchmark 20 times, each run in a separate virtual machine in order to isolate their effects from each other and report the average of these runs. We ran many other benchmarks where we measured no observable impact from allocation folding, but found no benchmarks where allocation folding was a measurable detriment to performance. However, we did find that for four other benchmarks, allocation folding had significant improvement: (1) a JPEGEncoder [16] written in JavaScript encoding an image, (2) NBody [10] solving the classical N-body problem, (3) a

JavaScript software 3D renderer Soft3D [12], and (4) the JavaScript benchmark Textwriter [1] originally designed to test string operation speed.

We use five configurations of V8 for our experiments: (1) *baseline* generates optimized code without write barrier elimination or allocation folding, (2) *allocation folding (AF)* is the baseline configuration with allocation folding only, (3) *write barrier elimination (WBE)* is the baseline configuration with write barrier elimination only, (4) *write barrier elimination and allocation folding on basic blocks (WBE-AFBB)*, performs write barrier elimination and allocation folding only on basic blocks, and (5) *write barrier elimination and allocation folding (WBE-AF)* is the previous configuration without the basic block restriction.

The WBE-AF configuration is the one used in production code and on average yields the biggest performance improvement. The other configurations are used to investigate the independent impact of the optimizations on the baseline performance without taking the positive interplay of allocation folding and dominator-based write barrier elimination into account.

4.1 Throughput

Figures 3-5 show relative throughout improvement for each of the benchmarks on X64. Allocation folding has the most impact on `RayTrace`, and here we measured a trend that is common to several benchmarks. In `RayTrace`, we measured an improvement with AF of more than 10% from saving bump-pointer allocation costs, with WBE of more than 20% from doing only dominator-based write barrier elimination, with WBE-AFBB of 23% from allocation folding at the basic block level, and with WBE-AF of more than 70% from allocation folding without the basic block restriction. WBE-AF improves `EarleyBoyer` by about 14%, `Splay` by over 10%, and `TypeScript` by about 4%. `DeltaBlue`, `PdfJS`, and `Gameboy` also improve by about 2-3%. The throughput improvement is less than 1% for most of the Kraken benchmarks. Other benchmarks had significant improvements with WBE-AF, such as `NBody` (10%) and `Soft3D` (16%).

With many benchmarks we see the same trend where the improvement from allocation folding alone is measurable, even significant, but the largest gains are from eliminating the cost of write barriers, as seen in `EarlyBoyer`, `RayTrace`, `Gameboy`, `NBody`, and `Soft3D`. Also notable is `DeltaBlue`, which only benefits from the combined effects of allocation folding and write barrier elimination, and sees almost no benefit from either independently. We also see that in several cases allocation folding on basic blocks gives results as good as the complete dominator-based algorithm.

Tables 2-4 show the proportion of folded and non-folded allocation sites in optimized code in our benchmarks.

4.2 Write Barrier Frequency

Tables 5, 6 and 7 show the static number of write barrier sites compiled into the optimized code as well as the dynamic number of write barriers executed.

There is a strong correlation between throughput improvement and fewer executed write barriers due to folded allocations. For example, in `RayTrace`, WBE eliminates about 38% of write barriers for a 20% speedup, and WBE-AF eliminates about 96% of write barriers resulting in a throughput improvement of 72%. `EarleyBoyer` executes even fewer write barriers in comparison to the baseline, with 98% eliminated and throughput improvement by 14% using WBE-AF. In `Soft3D` allocation folding reduced the number of executed write barriers by 86% for a speedup of 16% using WBE-AF. In `NBody` allocation folding removed the most write barriers, about 99% in WBE-AF. The results are consistent across the remaining benchmarks, with those that have the most write barriers eliminated experiencing the largest gains in throughput.

	AFBB	AF
Benchmark	Folded in %	Folded in %
Box2D	21	35
CodeLoad	28	28
Crypto	33	42
DeltaBlue	37	47
EarleyBoyer	26	42
Gameboy	3	3
Mandreel	38	38
MandreelLatency	38	38
NavierStokes	18	18
PdfJS	29	31
RayTrace	11	65
RegExp	27	27
Richards	26	65
Splay	22	40
SplayLatency	22	40
Typescript	6	12
zlib	82	82

Table 2: Static proportion of folded allocation instructions in Octane.

	AFBB	AF
Benchmark	Folded in %	Folded in %
ai	25	25
audio	37	38
imaging	0	0
json	0	0
stanford	23	24

Table 3: Static proportion of folded allocation instructions in Kraken.

	AFBB	AF
Benchmark	Folded in %	Non-folded in %
JPEGEncoder	18	55
NBody	76	88
Soft3D	56	58
Textwriter	8	17

Table 4: Static proportion of folded allocation instructions in hand-selected benchmarks.

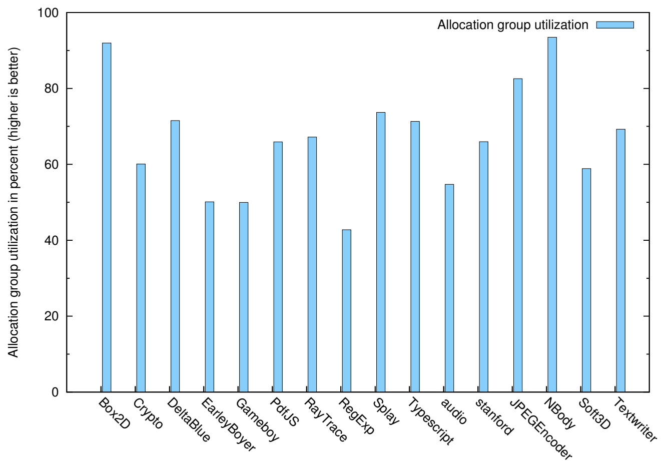


Figure 6: Allocation group utilization on X64.

	Static					Dynamic				
Benchmark	Baseline	AF	WBE	WBE-AFBB	WBE-AF	Baseline	AF	WBE	WBE-AFBB	WBE-AF
Box2D	3,166	3,103	2,446	2,449	2,188	41,083,162	41,083,425	25,453,150	25,432,225	20,811,285
Codeload	46,956	46,929	46,780	46,814	46,793	300,295	300,447	186,482	186,487	186,491
Crypto	720	717	353	257	237	1,690,104	1,692,164	494,311	329,287	312,298
DeltaBlue	518	516	307	209	187	472,408,298	472,408,568	329,751,772	251,689,699	197,782,399
EarleyBoyer	774	777	215	196	196	1,374,515,484	1,374,515,775	28,122,342	28,029,928	28,039,744
Gameboy	4,019	3,984	3,634	3,692	3,685	8,714,289	8,773,082	8,321,467	8,176,195	8,145,726
Mandreel	107	107	39	36	36	18,668	18,668	1,744	1,744	1,744
MandreelLatency	127	127	41	36	36	18,668	18,668	1,744	1,744	1,744
NavierStokes	218	216	111	109	109	27,887	27,887	26,039	26,039	26,039
PdfJS	4,983	5,186	4,165	4,173	4,091	215,806,576	216,156,555	38,048,742	39,857,551	38,115,351
RayTrace	998	998	621	614	238	1,188,216,950	1,188,216,565	741,557,641	741,554,567	54,467,024
RegExp	423	423	330	323	323	73,182,468	73,182,384	62,449,266	62,446,183	62,446,138
Richards	429	429	206	189	185	593,493,156	593,462,983	589,364,534	588,980,038	588,831,690
Splay	1,658	1,649	1,533	1,497	1,574	477,647,218	477,647,182	464,248,939	438,256,053	438,258,244
SplayLatency	1,641	1,678	1,519	1,503	1,515	477,646,746	477,646,841	464,241,636	438,255,521	438,256,651
Typescript	19,326	19,770	16,542	15,467	17,395	36,257,098	36,241,445	30,485,767	30,412,861	30,454,115
zlib	309	309	185	176	176	17,078	17,078	14,812	14,767	14,767

Table 5: Static and dynamic number of write barriers in optimized code in the Octane suite.

	Static					Dynamic				
Benchmark	Baseline	AF	WBE	WBE-AFBB	WBE-AF	Baseline	AF	WBE	WBE-AFBB	WBE-AF
ai	80	80	26	26	26	145,579	145,579	126,795	126,341	126,296
audio	261	261	78	70	63	125,518	126,034	55,694	33,364	30,990
imaging	40	40	2	2	2	1,967	1,967	51	51	51
json	40	40	2	2	2	1,910	1,910	51	51	51
stanford	773	766	489	455	441	1,780,963	1,766,298	847,108	730,193	717,309

Table 6: Static and dynamic number of write barriers in optimized code in the Kraken suite.

	Static					Dynamic				
Benchmark	Baseline	AF	WBE	WBE-AFBB	WBE-AF	Baseline	AF	WBE	WBE-AFBB	WBE-AF
JPEGEncoder	178	178	96	94	92	5,979,914	6,001,312	5,775,179	5,786,586	5,766,983
NBody	528	527	314	33	33	41,969,426	43,652,671	27,032,795	10,399	10,300
Soft3D	490	490	276	206	206	298,331,060	344,663,928	155,663,705	43,997,228	42,900,118
Textwriter	665	670	351	333	332	111,465,619	114,906,407	93,684,228	90,337,194	90,710,264

Table 7: Static and dynamic number of write barriers in optimized code in the hand-selected benchmarks.

4.3 Allocation Group Utilization

Allocation instructions folded into a given dominator from different branch successors may result in unused memory, which can be considered fragmentation. Figure 6 shows the percentage of memory allocated for the allocation group that is actually used as live objects by the program for the AF and WBE-AF configurations. Benchmarks with 100% allocation group utilization are elided for conciseness. Here we only consider memory dynamically allocated in allocation groups by optimized code, and do not count the memory allocated in normal allocations *outside* of allocation groups or in unoptimized code. Therefore this should not be considered a measurement of total heap fragmentation. Our measurements show that most of the benchmarks utilize between 50% and 80% of the memory allocated in allocation groups, with the exception of `RegExp` using only 42%, and `Box2D` and `NBody` using more than 90%. Lower memory utilization in the young generation results in more frequent young generation collections. We investigate this effect in the next section.

4.4 Garbage Collection Overhead

Intuitively, more frequent collections that result from higher memory fragmentation should lead to higher garbage collection overhead, but is this effect real, and is it more significant than the benefits from allocation folding? We studied this question by recording a number of garbage collection statistics, including the number of minor garbage collections, number of major garbage collections, and garbage collection time in milliseconds of the baseline, WBE, WBE-AFBB, AF, and WBE-AF configurations. We report the raw numbers in Table 8, Table 9, and Table 10. These numbers show that in most cases, there is almost no increase in garbage collection overhead, even though many benchmarks see a small increase in the number of minor collections. This is because the cost of scavenging is proportional to the size of live objects, so a small amount of fragmentation, which is by definition not live, has little cost other than cache effects and appears not to be measurable. However, in some cases we see the total garbage collection time increase, for example by 48 ms in `Soft3D` and by about 41 ms in `PdfJS`, with the former due to more minor collections and the latter due to more major collections. Even with the added garbage collection overhead of 56 additional minor collections, allocation folding is still an overall throughput improvement in `Soft3D`. The throughput of `PdfJS` slightly degrades in the AF and WBE-AF configurations, due to three additional major collections.

5. Related Work

Are write barriers really that expensive? This question was studied extensively by Blackburn and Hosking in 2004 [3], and a followup study in 2012 [22]. Their reported experimental results indicate average write barrier overheads in the range of 1-6% for the Java programs they study, for most of the write barrier types. At first glance,

	Baseline, WBE, WBE-AFBB (no fragmentation)			AF, WBE-AF (fragmentation)		
Benchmark	#Minor GCs	#Major GCs	GC time in ms	#Minor GCs	#Major GCs	GC time in ms
Box2D	111	5	99.37	111	5	100.97
CodeLoad	14	5	137.45	14	5	136.22
Cyrpto	62	0	2.25	69	0	1.5
DeltaBlue	585	0	44.57	587	0	45.11
EarleyBoyer	856	0	770.58	856	0	763.82
Gameboy	37	18	112.99	37	19	116.51
Mandreel	106	6	12.84	106	6	12.91
MandreelLatency	106	6	12.84	106	6	12.91
NavierStokes	16	0	2.18	16	0	2.26
PdfJS	555	26	772.54	555	29	813.28
RayTrace	2599	0	18.5	2610	0	22.11
RegExp	959	0	20.49	956	0	18.91
Richards	63	0	0.56	63	0	0.56
Splay	311	194	416.69	313	194	419.1
SplayLatency	311	194	416.69	313	194	419.1
Typescript	51	6	444.77	51	6	449.11
zlib	1	1	2.41	1	1	2.35

Table 8: Number of minor collections, major collections, and total garbage collection time in ms with and without allocation folding in the Octane suite on X64.

	Baseline, WBE, WBE-AFBB (no fragmentation)			AF, WBE-AF (fragmentation)		
Benchmark	Minor GCs	Major GCs	GC time in ms	Minor GCs	Major GCs	GC time in ms
ai	4	1	6.55	4	1	6.49
audio	42	6	18.37	43	6	18.31
imaging	2	4	6.45	2	4	6.33
json	21	2	4.16	21	2	4.26
stanford	45	4	16.96	45	4	16.97

Table 9: Number of minor collections, major collections, and total garbage collection time in ms with and without allocation folding in the Kraken suite on X64.

	Baseline, WBE, WBE-AFBB (no fragmentation)			AF, WBE-AF (fragmentation)		
Benchmark	Minor GCs	Major GCs	GC time in ms	Minor GCs	Major GCs	GC time in ms
JPEGEncoder	18	1	17.89	18	1	17.51
NBody	597	0	0.31	620	0	0.31
Soft3D	437	0	57.44	493	0	105.67
TextWriter	2213	0	6.28	2230	0	10.13

Table 10: Number of minor collections, major collections, and total garbage collection time in ms with and without allocation folding in the hand-selected benchmarks on X64.

the large speedups for some benchmarks yielded by our optimization technique would seem to contradict their estimate of write barrier overheads. However, a close reading of their data tables show several important outliers, and we believe these outliers are exactly the cases where our optimization technique works best. First, V8's garbage collector is incremental, requiring a heavier write barrier than any of those studied in these two papers. V8's write barrier is closest to the "zone" barrier reported in their study, which, though no attention was called to it in their discussion, shows between 10-50% performance overhead for several DaCapo benchmarks. This larger write barrier overhead is in closer agreement to the optimization potential exploited in this paper using allocation folding. Second, we believe that some of the applications in Octane are much more allocation intensive than those in DaCapo, if only by virtue of JavaScript's numerical model leading to excessive amounts of boxing double numbers in V8, which is extreme in the case in RayTrace. Third, garbage collection designs with heavier write barrier costs are becoming more important as language implementations pursue reducing latency versus maximum throughput. We showed allocation folding to be of particular benefit to V8, which has an expensive write barrier to support incremental marking.

Previous optimizations related to allocation folding fall into two categories: static analysis during compilation to reduce barriers and techniques to combine object allocations.

Barth [2] discusses minimizing the expense of reference-counted garbage collection through static analysis during compilation and is suggestive of later write barrier elimination based on static analysis [23]. Although eliding unnecessary reference-count decrement on freshly allocated objects is specifically mentioned, implementation details and empirical results are not presented as the author considered the technique impractical for the time.

Nandivada and Detlefs [14] present a static analysis pass to minimize write barriers at compile time for a snapshot-at-the-beginning style of garbage collector and document the empirical improvement of generated code using their techniques. Their approach bears similarities to ours as it exploits the property of freshly allocated objects always being colored white to remove write barriers. However, it is unable to leverage this property for multiple objects allocated in close proximity and the algorithm's ability to remove write barriers can actually diminish with objects allocated in clusters. Rogers [17] studies the problem of read barriers in a concurrent collector and uses techniques similar to partial redundancy elimination to hoist or sink potentially redundant parts of barriers. Pizlo et al.

[15] generate multiple copies of the code, with different versions of read/write barriers specialized to different phases of collection, but they do not describe the complete removal of barriers. Vechev and Bacon [18] study conditions under which write barriers may be redundant for concurrent collectors and study program traces. Their work may prove to be complimentary in that allocation folding could present even more covering conditions than previously known.

Automatic object inlining is well studied [4] [6] [5], however it relies on parent-child relationships between objects to make decisions to combine allocations. Object colocation [11] allocates related objects together in the same space, but requires explicit support from the garbage collector and is intended to reduce the cost of collection rather than to improve the efficiency of compiled code. Object and array fusing [21] uses colocation to improve the efficiency of accessing one object through the field of another in compiled code, but also requires explicit support from the garbage collector. Object combining [19] is closest to allocation folding in that it has fewer restrictions, but works best with patterns where an indirection can be eliminated, and the opportunity for eliminating write barriers was not recognized at the time.

6. Conclusion

In this paper we introduced allocation folding, a compiler optimization where multiple memory allocation operations in optimized code are folded together into a single, larger allocation group. Folding allocations together reduces the per-object allocation overhead and widens the scope for write barrier removal. Unlike previous work on object inlining, fusion, and colocation, allocation folding requires no particular connectivity or ownership relationship among objects, only a control-flow relation within a single optimized function. We presented a flow-sensitive analysis based on GVN with side-effects that computes the necessary dominance information to determine allocation folding candidates.

We implemented allocation folding in V8, a high-performance open-source JavaScript virtual machine and evaluated its effectiveness across a variety of standard benchmarks. Our results demonstrated that allocation folding can make a large improvement in throughput for allocation and write-barrier intensive programs. We measured the benefits of reducing bump-pointer operations and write barriers both independently and together. We found that memory fragmentation arising from allocation folding has negligible cost in most cases.

7. References

[1] C. Authors. Textwriter. http://www.chrome.org.

[2] J. M. Barth. Shifting garbage collection overhead to compile time. *Communications of the ACM*, 20(7):513–518, July 1977.

[3] S. M. Blackburn and A. L. Hosking. Barriers: friend or foe? In *Proceedings of the International Symposium on Memory Management*, ISMM '04, pages 143–151, New York, NY, USA, 2004. ACM.

[4] J. Dolby. Automatic inline allocation of objects. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '97, pages 7–17, New York, NY, USA, 1997. ACM.

[5] J. Dolby and A. Chien. An automatic object inlining optimization and its evaluation. *SIGPLAN Notices*, 35(5):345–357, May 2000.

[6] J. Dolby and A. A. Chien. An evaluation of automatic object inline allocation techniques. In *Proceeding of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '98, pages 1–20. ACM Press, 1998.

[7] Google Inc. Octane. https://developers.google.com/octane, 2013.

[8] Google Inc. V8. https://code.google.com/p/v8, 2013.

[9] Google Inc. V8 design. https://code.google.com/p/v8/design, 2013.

[10] I. Gouy. Nbody. http://shootout.alioth.debian.org.

[11] S. Z. Guyer and K. S. McKinley. Finding your cronies: static analysis for dynamic object colocation. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 237–250, New York, NY, USA, 2004. ACM.

[12] D. McNamee. Soft3d, 2008.

[13] Mozilla. Kraken. https://krakenbenchmark.mozilla.org, 2013.

[14] V. K. Nandivada and D. Detlefs. Compile-time concurrent marking write barrier removal. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pages 37–48, Washington, DC, USA, 2005. IEEE Computer Society.

[15] F. Pizlo, E. Petrank, and B. Steensgaard. Path specialization: reducing phased execution overheads. In *Proceedings of the International Symposium on Memory Management*, ISMM '08, pages 81–90, New York, NY, USA, 2008. ACM.

[16] A. Ritter. JPEGEncoder. https://github.com/owencm/javascript-jpeg-encoder, 2009.

[17] I. Rogers. Reducing and eliding read barriers for concurrent garbage collectors. In *Proceedings of the Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS '11, pages 5:1–5:5, New York, NY, USA, 2011. ACM.

[18] M. T. Vechev and D. F. Bacon. Write barrier elision for concurrent garbage collectors. In *Proceedings of the International Symposium on Memory Management*, ISMM '04, pages 13–24, New York, NY, USA, 2004. ACM.

[19] R. Veldema, J. H. Ceriel, F. H. Rutger, and E. Henri. Object combining: A new aggressive optimization for object intensive programs. In *Proceedings of the Conference on Java Grande*, JGI '02, pages 165–174, New York, NY, USA, 2002. ACM.

[20] C. Wimmer and M. Franz. Linear scan register allocation on SSA form. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '10, pages 170–179, New York, NY, USA, 2010. ACM.

[21] C. Wimmer and H. Mössenböck. Automatic feedback-directed object fusing. *ACM Transactions on Architecture and Code Optimization*, 7(2):7:1–7:35, Oct. 2010.

[22] X. Yang, S. M. Blackburn, D. Frampton, and A. L. Hosking. Barriers reconsidered, friendlier still! In *Proceedings of the International Symposium on Memory Management*, ISMM '12, pages 37–48, New York, NY, USA, 2012. ACM.

[23] K. Zee and M. Rinard. Write barrier removal by static analysis. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 191–210, New York, NY, USA, 2002. ACM.