Memento Mori: Dynamic Allocation-Site-Based Optimizations

Daniel Clifford Hannes Payer Michael Stanton Ben L. Titzer

Google, Munich, Germany

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

Abstract

Languages that lack static typing are ubiquitous in the world of mobile and web applications. The rapid rise of larger applications like interactive web GUIs, games, and cryptography presents a new range of implementation challenges for modern virtual machines to close the performance gap between typed and untyped languages. While all languages can benefit from efficient automatic memory management, languages like JavaScript present extra thrill with innocent-looking but difficult features like dynamically-sized arrays, deletable properties, and prototypes. Optimizing such languages requires complex dynamic techniques with more radical object layout strategies such as dynamically evolving representations for arrays. This paper presents a general approach for gathering temporal allocation site feedback that tackles both the general problem of object lifetime estimation and improves optimization of these problematic language features. We introduce a new implementation technique where *allocation mementos* processed by the garbage collector and runtime system efficiently tie objects back to allocation sites in the program and dynamically estimate object lifetime, representation, and size to inform three optimizations: *pretenuring*, *pretransitioning*, and *presizing*. Unlike previous work on pretenuring, our system utilizes allocation mementos to achieve fully dynamic allocation-site-based pretenuring in a production system. We implement all of our techniques in V8, a high performance virtual machine for JavaScript, and demonstrate solid performance improvements across a range of benchmarks.

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

Permission to make digital or hard copies of part or all 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 third-party components of this work must be honored. For all other uses, contact the Owner/Author.

Copyright is held by the owner/author(s).

ISMM'15, June 14, 2015, Portland, OR, USA
ACM. 978-1-4503-3589-8/15/06
http://dx.doi.org/10.1145/2754169.2754181

General Terms Algorithms, Languages, Experimentation, Performance, Measurement

Keywords Dynamic Optimization, Garbage Collection, Memory Managment, JavaScript

1. Introduction

Web application developers are deploying ever larger applications with ever greater demand for computational power. Sites across the web such as Gmail, Facebook, and Amazon include megabytes of JavaScript to run everything from complex interactive UIs to comment systems, shopping carts and ads tracking. But larger web applications aren't the only trend. JavaScript has emerged on the server side as well [33], allowing web developers to share frontend and backend code by running a type of headless JavaScript VM on the server. Such server code often deals with consulting data storage and assembling responses to requests in data buffers, string- and data-crunching tasks closer to traditional languages. However the biggest trend is that whole games are being developed and deployed in JavaScript [39], with engines that perform heavy numeric computation.

Since JavaScript lacks static typing, high performance virtual machines must resort to dynamic techniques like inline caching, type feedback, and dynamic compilation. V8 pioneered several new techniques for optimizing JavaScript, most importantly, hidden classes [14]. Our work extends V8's existing architecture with a new class of dynamic feedback that utilizes per-object instrumentation.

While previous work [17, 23] has primarily focused on allocation sampling to gather limited online object lifetime feedback, our work introduces a new object instrumentation technique called *allocation mementos* that generalizes online per-object temporal feedback in a more flexible way. Mementos are small objects placed next to objects in the young generation that track runtime information without requiring space in the host object. Mementos are created directly at the allocation site of the object and live only a short time before a garbage collection, avoiding a large space overhead on the program. Perhaps even more importantly, mementos can be selectively enabled on a per-function or even per-allocation-site basis, allowing a broader range of instrumentation choices than sampling based on allocated bytes.

Mementos serve as a general instrumentation technique within V8 which informs several optimizations. In this paper we show how the low-overhead temporal feedback provided by mementos is crucial in enabling two key optimizations, *pretransitioning* and *pretenuring*. We implemented these optimizations in a production JavaScript virtual machine and evaluated their effectiveness on a wide range of standard JavaScript benchmarks. We further demonstrate the flexibility and generality of mementos by showing a prototype of array *presizing* which took one person less than a day of work.

Pretenuring. A generational garbage collection architecture provides high throughput and low latency when most objects die young [34], but suffers when too many objects outlive the young generation, incurring large copying and scanning overhead. Pretenuring [8] is an optimization where some objects are allocated directly in the old generation with the goal of reducing this overhead. Effective pretenuring entails predicting the future; pretenure too many short-lived objects and the old generation will fill up too quickly, requiring more frequent major collections; pretenure too few and long-lived objects will go through multiple young generation collections before being promoted. This paper presents our technique for pretenuring as one of a suite of optimizations based on temporal allocation site feedback using mementos.

Pretransitioning. JavaScript semantics for arrays present tricky implementation issues. Yet despite many possible corner cases, most code is well behaved enough that arrays fall into a small number of categories that a virtual machine can implement efficiently. For such categories, V8 has different array representations with different space / time trade-offs, transitioning arrays on demand as they are mutated by the program. Such transitions are relatively expensive. We introduce *pretransitioning* which uses dynamic feedback about array evolution to choose the best initial representation on a per-allocation-site basis, avoiding future transitions.

Presizing. JavaScript arrays can be resized dynamically and offer methods that allow them to be used like queues or stacks. Dense representations can be inefficient if they reallocate and copy elements too often. We demonstrate a prototype of *presizing* which avoids internal elements copies by tracking the dynamic behavior of arrays and estimating an optimum size for particular allocation sites based on past behavior.

Summary of Contributions.

- a new instrumentation technique, *allocation mementos*, which efficiently gather temporal feedback about objects
- a fully dynamic allocation-site-based pretenuring system with online lifetime estimation
- the first feedback-based pretransitioning of array representations that avoids expensive transition operations
- the first prototype of feedback-based presizing of arrays
- an industrial strength implementation of our optimizations in a production virtual machine, shipping since Chrome 26.0.1410.0.
- a thorough evaluation of our optimizations on several suites of benchmarks

2. Background

V8 is an industrial-strength virtual machine for JavaScript that can be embedded into a C/C++ application through a set of programmable APIs. It is the JavaScript virtual machine used by both the Chrome and Opera web browsers and the server-side node.js framework [21]. V8 pioneered an implementation technique for optimizing JavaScript called *hidden classes* [14] that assigns every object a dynamically-evolving shape that describes its properties and prototype object. We follow V8's implementation and use the term *map* to describe the hidden class or shape of an object within this paper. A detailed description of maps is beyond the scope of the paper; other works offer more detail, e.g. [1].

2.1 V8 Adaptive Optimization

V8 uses a two-tier compile-only strategy where JavaScript functions are compiled to native machine code by a fast AST-walking compiler upon the first execution. This paper will refer to this first tier compiler as the baseline compiler and code it generates as baseline code. Inline caches (ICs) [10, 36] implement the basic JavaScript operations like binary addition, comparison, object creation, object property accesses, etc., and dynamically adapt to the types of input values that they encounter, recording type information that is later used during optimization. CallICs are inline caches that record the targets of function calls in the code. Such an IC starts in the state `UNINITIALIZED` and on the first call records the target function, transitioning to state `MONOMORPHIC`. As long as subsequent calls target the same function, then the IC remains `MONOMORPHIC`. If a new call target is seen, then the IC transitions to the state `MEGAMORPHIC`.

An optimizing compiler called Crankshaft recompiles hot functions for better peak performance. Type feedback guided by ICs is used to reduce complex JavaScript operations to primitive arithmetic and reduce property accesses to efficient, single-indirection loads, both for named properties of objects and array elements, usually guarded by a check of the object's map. Monomorphic CallICs can be queried to determine the unique caller, which may subsequently be inlined. Many general optimizations in Crankshaft are standard, such as constant folding, strength reduction, global value numbering, loop invariant code motion, dead code elimination, etc. If type or bounds checks fail or if the code encounters new types of objects at runtime or numbers outside of an assumed integer range, then the optimized code is *deoptimized* and control is transfered back to baseline code.

2.2 JavaScript Objects and Arrays

JavaScript has a prototype-based object model that does not have arrays in the traditional sense. A simplified but workable view is that objects simply have properties, property names are just strings, and a read of a missing property of an object results in a recursive read of that property on the prototype object of the object. Properties can be added and removed from objects at essentially any time and can even have getter and setter methods installed.

Instead of special types of array objects, any object can take on array-like qualities like a numeric length and properties that are named like integers. Shortcut notation like array literals `[e1, e2, ...]` and calls to the `Array` constructor create regular objects that have prepopulated length and elements. Accessing out of the "bounds" of an array behaves like accessing any other missing property as search continues on the prototype object. Arrays can store any type of value in the general case, but most do not, especially in numeric code. Achieving competitive performance on such code requires a highly optimized representation for arrays that contain only doubles or integers.

This is made more difficult because JavaScript is untyped, so it is not obvious what types of data an array will store when it is created, since even array literals are mutable. Worse, objects created with the `Array` constructor inherit, by virtue of their prototype, a host of methods like `push()` and `pop()` that allow the array to be used as a stack, adding or removing elements from the end of the array and adjusting its length. Arrays are used in different ways, even in different parts of the same program. Some arrays might be used like lists, yet others might store floating point numbers for an intense numerical computation. A contiguous memory representation would be horribly inefficient if internal copies were required to grow the storage on each operation; it is more efficient to allocate additional capacity to avoid too many copies.

Another peculiarity is the hole. Indexed properties can be *deleted* from the middle of an "array" in JavaScript. When an indexed property is deleted, further reads from that index do not produce `0` or `undefined` or throw an exception, but instead search continues on the prototype of the object just like any other property the object does not have. As we'll see in the next section, V8 uses a special sentinel value called the "hole" to mark elements that do not exist for more complex handling. We use the symbol • to represent the hole value in this paper.

2.3 V8 Array Elements

V8 internally stores the integer-indexed properties of an object one indirection away from the main object in a separate contiguous memory chunk called the *elements backing store*, or simply *elements*. The elements are represented in one of six ways, depending on the values and whether there are holes. We refer to the representation of the elements as the *ElementsKind* and list the different kinds in Table 1.

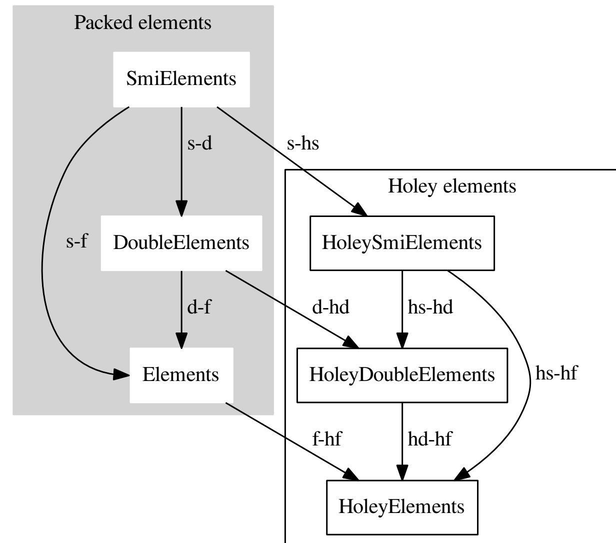

Figure 1. Allowed ElementsKind transitions in V8.

V8 distinguishes arrays that might have holes from those that definitely do not have holes by giving them different ElementsKinds and dynamically transitioning the ElementsKind when a hole is created. The acceptable transitions are given in Figure 1. In elements whose ElementsKind is either `HoleySmiElements` or `HoleyElements`, • is represented as a pointer to a special singleton object that identifies itself as the hole, whereas in those elements with ElementsKind `HoleyDoubleElements`, • is represented as a special 64-bit double `NaN` value that is impossible to produce otherwise through normal arithmetic. A read of an array element that might contain holes needs a dynamic check to see if the element is •, and if so, a prototype lookup will be necessary to implement the semantics dictated by JavaScript. A single typecheck in optimized code can often guard an entire loop, making it free from hole checks.

Array literals usually offer some clue to a good initial ElementsKind since they contain initial values and a fixed length. Static information is used when possible to start the array at the most appropriate ElementsKind.

```
(a1) var a = [1,2,3.5];
(a2) var b = [1,"hello",,];
```

In (a1) `a` will start with kind `DoubleElements`, since all elements are statically determined to be numbers. In (a2) however, `b` will contain both a string, an integer, and the value `undefined` (since there is an element missing), so it is given kind `HoleyElements`.

Normally, all arrays start with a default ElementsKind of `SmiElements` and V8 will transition the ElementsKind as necessary when the array is mutated by the program.

ElementsKind	Description	Contains	Size
`SmiElements`	small integers	tagged ints	$(2 + length) * wordsize$
`HoleySmiElements`	small integers with holes	tagged ints or •	$(2 + length) * wordsize$
`DoubleElements`	64-bit doubles	doubles	$2 * wordsize + 8 * length$
`HoleyDoubleElements`	64-bit doubles with holes	doubles or •	$2 * wordsize + 8 * length$
`Elements`	all value types	tagged values	$(2 + length) * wordsize$
`HoleyElements`	all value types with holes	tagged values or •	$(2 + length) * wordsize$

Table 1. ElementsKind in V8

```
(b1) var a = [1,2,3];
(b2) a[0] = 1.5;
```

In (b1), `a` begins with kind `SmiElements`, but the assignment in (b2) will dynamically transition the array to kind `DoubleElements`, which requires allocating a new backing store and copying elements. On 32-bit systems, a transition from `SmiElements` to `DoubleElements` requires four steps: 1) allocate a new chunk of memory of size $2 * wordsize + 8 * length$, 2) copy elements into the new chunk, converting each tagged integer to a 64-bit double, 3) set the object's elements pointer to the new elements, and 4) change the object's map to reflect the ElementsKind change.

```
(c1) var a = [1,2,3];
(c2) delete a[1];
```

In (c2) we see the introduction of a hole into an array. If the program deletes an array element or writes the value `undefined`, V8 instead writes • in the elements and set the ElementsKind to one of the `HOLEY` variants. The array from (c1) will start with ElementsKind `SmiElements` and the assignment (c2) will change the ElementsKind to `HoleySmiElements`. This transition is less expensive, since the array does not need to be reallocated nor the existing elements copied; however, it still requires a map change of the object to reflect the change of the ElementsKind.

A transition from a `SmiElements` to `Elements` is similarly inexpensive, requiring only a map change, because tagged ints are a subset of tagged values. The most expensive transitions are from `DoubleElements` to `Elements` and from `HoleyDoubleElements` to `HoleyElements` since in both cases every 64-bit double value must be boxed into a wrapper object on the heap.

2.4 V8 Garbage Collector

V8 uses a generational garbage collector with a semi-space scavenger strategy for frequent collections of the young generation and a mark-and-sweep collector with incremental marking, concurrent sweeping, and compaction for major collections of the old generation. A store-buffer write barrier tracks old-to-young generation reference stores, and the recorded store buffer entries become part of the root set for young generation collections. The write barrier must also maintain the incremental marking invariant and record references to objects that will be relocated during compaction.

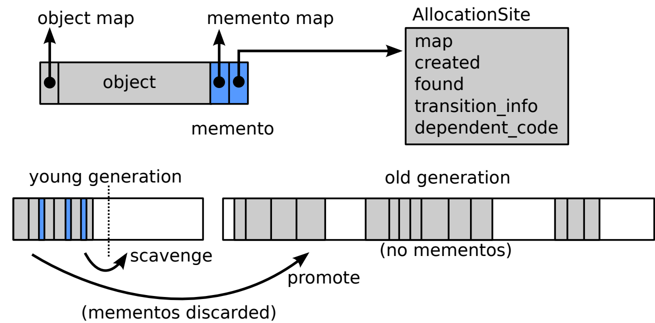

Figure 2. Allocation mementos in the V8 heap.

Newly created objects are allocated using bump-pointer allocation, which by default happens in the active semi-space in the young generation. If the end of the active semi-space is reached a young generation garbage collection is triggered. The scavenger iterates over the transitive closure of live objects in the young generation and copies them from the current semi-space to the other semi-space. Therefore the latency of a young generation collection depends on the size of live objects in the young generation.

Objects that were already copied once by the scavenger are promoted by copying them again into the old generation. This becomes inefficient if too many objects survive for a long time, especially for applications with a large memory footprint, since the old generation is populated by objects that have already been copied twice. For such objects, it is clearly more beneficial to allocate them directly in the old generation if possible.

3. Allocation Sites and Allocation Mementos

The purpose of an *allocation memento*, or simply memento, is to store metadata about an object in the heap and act as a programmable hook linking an object with a temporary payload of data. This is accomplished by placing the memento directly after an object in memory. The memento is *unrooted*: neither the object nor any outside pointers point to the memento directly. The payload that a memento carries is processed by the runtime system during array transitioning and array growth and by the garbage collector during a young generation collection. To avoid a large space overhead, mementos are only created in the young generation and never survive a garbage collection. Instead, when

the object is copied by the collector, the memento is simply discarded, so no space overhead is required in the old generation. Thus mementos are lossy; they are present for a short window of time early in an object's evolution until the object survives the scavenge. An illustration of mementos can be see in Figure 2.

We use the memento's payload to store a reference to the *allocation site* of the object, which corresponds to the textual location in the source code where the object was created. In our implementation, a memento occupies two words of memory where the first word is a pointer to the special *memento map*, which distinguishes the memory as a memento, and the second is a pointer to the AllocationSite. Using a special map word makes checking for mementos extremely efficient. It is sufficient to check a single word of memory directly after the object to determine if the object carries a memento.[1]

Why not use an extra word in the object or information in the metadata of the object to store allocation site information? What about a hash map? First, adding an extra word to the object imposes a space penalty on *every* object, even those that are no longer or never were interesting. Second, the map of an object is too coarse-grained, being type-specific rather than allocation-site specific.[2] Third, a hash map is too slow to consult on every transition and young generation collection. Fourth, placing mementos next to the object they instrument is potentially more cache efficient. The insight of allocation mementos is that the extra space needed is *object-specific* and *temporal*.

With mementos, we can simply "turn off" the learning mechanism on a site-by-site basis as soon as the system has learned enough to make an optimization decision. When a hot function is recompiled, Crankshaft harvests the information from AllocationSite instances in the baseline code to make optimization decisions. Since we expect most programs to spend most of their execution time in optimized code, that code should be as fast as possible and free from both the space and time overhead of creating mementos. Therefore, in our implementation, optimized functions never create mementos, though the architecture has no inherent limitation that prevents this.

3.1 Pretenuring

The goal of pretenuring is to avoid the overhead of copying and scanning objects that are often promoted to the old generation. Our approach utilizes temporal allocation site feedback to make pretenuring decisions for optimized code.

Dynamic allocation-site-based pretenuring tracks statistics in the AllocationSite data structure in Figure 2 to decide which allocation sites are good candidates for pretenuring. Only two integer-sized fields in each AllocationSite are needed. The `created` field stores the number of mementos created at the site since the last collection, and `found` stores the number of mementos found attached to live objects during the last collection.

[1] Spot the danger here? This design requires that uninitialized or leftover memory following each object be cleared, or at least set to anything *except* the memento map word, to prevent the system from interpreting garbage memory as a memento when the heap is not iterable.

[2] Other systems have experimented with inserting a level of indirection between an object and its type [28].

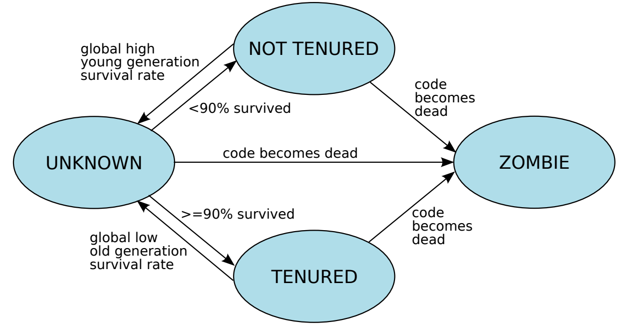

Figure 3. Lifetime states of an allocation site.

Baseline code is instrumented to create mementos. For each allocation site, the baseline compiler emits code that 1) allocates the object, 2) creates the memento directly following the object in memory, 3) initializes the memento to point back to the allocation site, and 4) increments the `created` counter in the AllocationSite.

The memento in this case serves both as a marker for the garbage collector to recognize that an object has been instrumented and as a pointer back to the allocation site for which to update the statistics. Only objects that have been instrumented with a memento are interesting. During a collection of the young generation, the garbage collector simply inspects the word following each live object to find mementos. If a memento is found, it follows its pointer to the AllocationSite and increments the found counter. At the end of the collection, these counts are used to calculate the survivor rate and decide whether the site is a good candidate for pretenuring.

An allocation site can be in four different lifetime states as depicted in Figure 3. An allocation site begins in the `UNKNOWN` state, and the number of created mementos at the site during one collection cycle must reach a threshold T before it can transition to a different state. We chose $T = 100$ in our implementation because it gave good results in practice. Another tuning parameter S controls the survivor rate at which pretenuring is initiated. After the threshold T has been reached within one garbage collection cycle, if more than $S\%$ of the objects allocated at the site survive that garbage collection cycle, then the AllocationSite is transitioned to the `TENURED` state; otherwise it is transitioned to the `NOT_TENURED` state. We chose $S = 90$ in our implementation to be conservative[3], since overestimating the sites that should be pretenured proves to be far more costly in

[3] 90% is also close to heuristics in other systems [17, 23]

practice than underestimating, which leads to more collections of the old generation.

During recompilation of a hot function, the optimizing compiler uses the state of AllocationSite instances for that function to generate inline memory allocation code. If the state of the AllocationSite has transitioned to `TENURED`, then the compiler will generate an inline allocation directly into the old generation, otherwise defaulting to an inline allocation into the young generation. In order to make the fastest possible optimized code, in neither case does the optimizing compiler emit code to create mementos. The space and time overhead of mementos is limited to the learning phase which happens in baseline code.

3.1.1 Zombies and Recovery

The `ZOMBIE` state is an implementation artifact of V8 that is a result of code objects being garbage collectable. Since V8 is a compile-only virtual machine and JavaScript has the ability to dynamically generate new source code with `eval`, V8 puts all code objects on the heap and collects and compacts them like all other objects. Code objects can die in one cycle, but the AllocationSite objects that they point to may still be pointed to by mementos. The `ZOMBIE` state allows these AllocationSite objects to survive for an extra garbage collection cycle to preserve heap integrity. This harder case only occurs in very long-lived applications when code objects start to die in large numbers.

Most benchmarks have very stable lifetime characteristics, but what if the lifetime characteristics of an allocation site change after optimization decisions have already been made? Productionizing pretenuring for more than a suite of benchmarks requires a bit more work.

Our feedback mechanism shuts itself off for performance reasons after optimized code has been generated. To avoid getting stuck with severely misguided pretenuring decisions, we use a global recovery mechanism which triggers when unstable conditions persist for two or more garbage collections:

1. When too many objects survive young generation garbage collection but no mementos are found, the recovery mechanism assumes that some non-tenured allocation sites would benefit from pretenuring. Since it lacks information on which site is misbehaving, it deoptimizes all optimized code that contains non-tenured allocation sites and new lifetime feedback is gathered.
2. When too many objects die in the old generation, the recovery mechanism assumes that too many sites are pretenured. In this case it does the reverse and deoptimizes all optimized code that contains tenured allocation sites and new lifetime feedback is gathered.

In both cases the `dependent_code` field of the AllocationSite objects is used to find the optimized code objects in the heap. Deoptimizing all optimized code that could have wrong tenuring decisions is relatively drastic, but protects the system from getting stuck in a slow state. The baseline code will warm up again and eventually be recompiled to new optimized code with new pretenuring and type feedback decisions. This does not occur in benchmarks but can happen with long-running applications.

3.1.2 Predecessors

Our system for dynamic allocation-site-based pretenuring replaces two previous approaches in V8 to reducing the problem of long-living objects. The first attempt was a *global high-promotion mode* which was triggered when the overall survivor rate from young generation collections consistently exceeded 90%. When this mode was activated, all objects surviving their first young generation collection would be immediately promoted to the old generation, reducing the number of copies from two to one. This gives a nice performance boost for certain phases of a program but still requires tenured objects to be copied at least once. The second attempt was a *global pretenuring mode*, triggered under the same condition of a consistent 90% survival from the young generation. At activation of global pretenuring, the system would deoptimize all optimized code and recompile new optimized code with tenured allocation sites.[4] All subsequent allocations from optimized code would go directly to the old generation. Both systems performed excellent on benchmarks and on certain phases of real applications, but were less performant on more general workloads. Worse, they required much more careful tuning since activation and deactivation drastically changed the behavior of the system.

Dynamic allocation-site-based pretenuring is superior in both peak performance (between 10% and 100% faster than the previous systems on Splay and SplayLatency), but also far more robust, since it is capable of approximating the lifetimes of individual allocation sites rather than just a phase of the program. The fewer drastic behavior changes associated with dynamic pretenuring also results in much smoother performance for applications in the wild.

3.2 Pretransitioning

The goal of pretransitioning is to avoid expensive array transitions by attempting to predict the optimal array elements representation for each allocation site. The small program in (d1) - (d9) illustrates the potential for this optimization.

```
(d1) function foo(a) { a[0] = 3.5; }
(d2) function bar(a) { a[0] = 'test'; }
(d3) for(var i = 0; i < 100; i++) {
(d4)   var a = [1,2,3];
(d5)   var b = [4,5,6];
(d6)   foo(a);
(d7)   foo(b);
(d8)   bar(b);
(d9) }
```

[4] One could also consider code patching for this purpose, but for technical reasons this was not feasible for fast-path allocations in V8.

Each time through the loop, a new array is allocated at (d4) and at (d5). Both arrays have an initial ElementsKind `SmiElements`. When the array `a` is passed to `foo()` in (d6), the assignment at (d1) assigns a double element, requiring a transition of the array to `DoubleElements`. The same happens at (d7) when `foo()` is passed `b`. The third call in (d8) to `bar()` assigns a string element to `b`, which requires a transition to `Elements`. All three transitions are relatively expensive and happen for every iteration in this example, resulting in a total of 300 array transitions.

Pretransitioning creates one AllocationSite for each of the places in the program where the optimal representation is yet to be learned (d4) and (d5). The baseline code emitted for those sites will allocate the array and also create a memento directly after the array in memory which points back to the AllocationSite object. We use the aptly-named `transition_info` field in the AllocationSite to store information about array transitions that have happened for objects allocated at the site. When transitions are required on an array, the runtime system looks for a memento following the array and if so uses the memento to find the AllocationSite object, updating the `transition_info` field.

With mementos, the system can connect the transitions that happen at (d1) and (d2) back to AllocationSite instances for (d4) and (d5). In this case the system will learn immediately that the ideal representation for arrays at (d4) is `DoubleElements` and those at (d5) is `Elements`. Subsequent arrays allocated at these sites will not require transitions, and the total number of array transitions for all iterations will be just 3.

Some array creations are not immediately obvious from the source code. For example, the `Array` built-in is a first class function that can be passed around the program like other functions. Calls to this function create new arrays. Because JavaScript has mutable bindings, V8 must use an IC to be sure of which function is called at runtime, including built-in functions which can have their names rebound by the program. The CallIC provides a convenient place to recognize such calls to the `Array` constructor, even if they are indirect calls.

The example in (e1-e7) illustrates the problem with indirect calls. Without pretransitioning, all arrays in this example are created with a default ElementsKind of `SmiElements`, and (e6) forces each to transition to `DoubleElements`, resulting in 100 total transitions.

```
(e1) function foo(f) { return f(); }
(e2) function custom_alloc() { return []; }
(e3) for(var i = 0; i < 100; i++) {
(e4)   var a = foo(Array);
(e5)   if (i == 50) Array = custom_alloc;
(e6)   a[0] = 3.5;
(e7) }
```

How does pretransitioning handle this case? In (e1) an indirect function call contains a CallIC. In the first few iterations of the loop (e3-e7), `foo` is called with the built-in `Array` function, and the CallIC state becomes `MONOMORPHIC`. V8 recognizes monomorphic calls to the special built-in `Array` with the CallIC and then creates an AllocationSite for the site (e1). During subsequent calls at (e1), the CallIC checks that `f` is the built-in `Array` function, and if so allocates the array with a memento that points to the AllocationSite for (e1).

After (e5) is executed, the binding for `Array` changes, causing the call at (e4) in the next iteration to pass `custom_alloc` to `foo()`. The CallIC in (e1) will now transition to `MEGAMORPHIC` and will lose its AllocationSite due to storage limitations. Calls in (e1) now go to `custom_alloc`, but arrays allocated inside will have an AllocationSite from (e2). Now the transition at site (e6) will update the AllocationSite for (e2). Overall, pretransitioning reduces the number of transitions in this example from 100 down to 2.

3.3 Presizing Prototype

To demonstrate the generality of mementos as an instrumentation technique, we developed a prototype of array presizing in one afternoon with just 150 lines of code [5].

The goal of presizing is to avoid expensive element reallocation operations without wasting memory by reserving the right amount of space when an array is allocated. Prior to our presizing work, V8 set a default elements capacity of 4 for all new arrays that do not have a specified size. A static policy wastes memory for smaller arrays and requires costly resizing for larger arrays.

Our approach is to use the memento and allocation site mechanism to track growth of arrays from calls to the `Array.push()` built-in JavaScript function. The prototype reuses the same mementos that are already created for arrays by baseline code for the pretransitioning optimization. The implementation of `Array.push()` simply looks for a memento when growing the elements backing store of an array and updates allocation site statistics with the new size. The statistics are ultimately used to choose a more appropriate capacity when generating optimized code.

Encouragingly, our prototype was able to learn the optimum allocation size for the important sites in the very sensitive DeltaBlue benchmark without prior knowledge, and preliminary measurements indicated a slight decrease in the maximum overall heap size in Octane, with no other performance loss.

4. Experiments

Our experiments were performed with V8 revision r24424 (October 2014) on an IA32 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 X64 and ARM but found that the performance

[5] [Reviewer's note: reference to open source patch deleted for blind review]

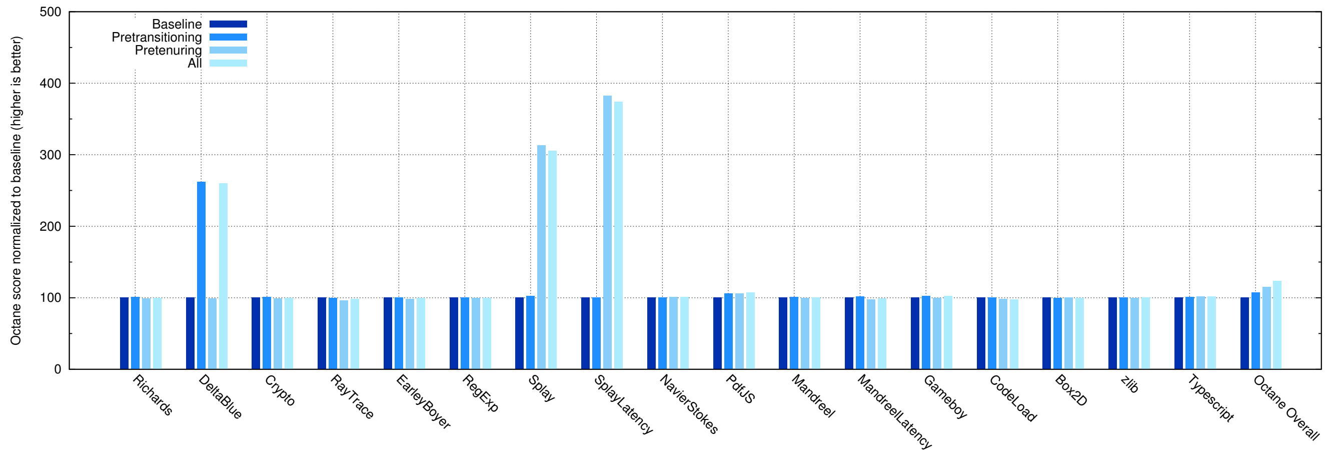

Figure 4. Octane score of baseline, transitioning, pretenuring, and all optimizations turned on normalized to baseline.

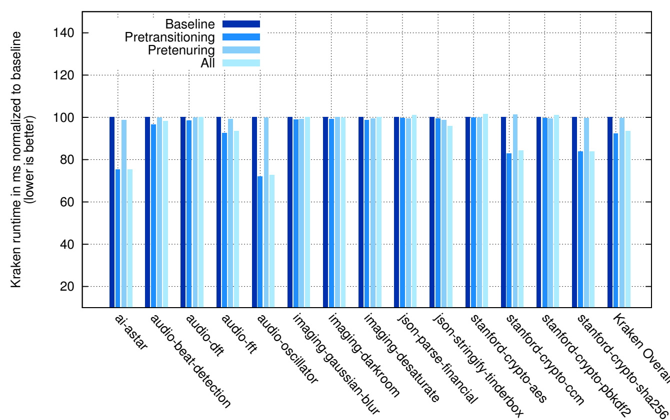

Figure 5. Kraken suite runtime of baseline, transitioning, pretenuring, and all optimizations turned on normalized to baseline.

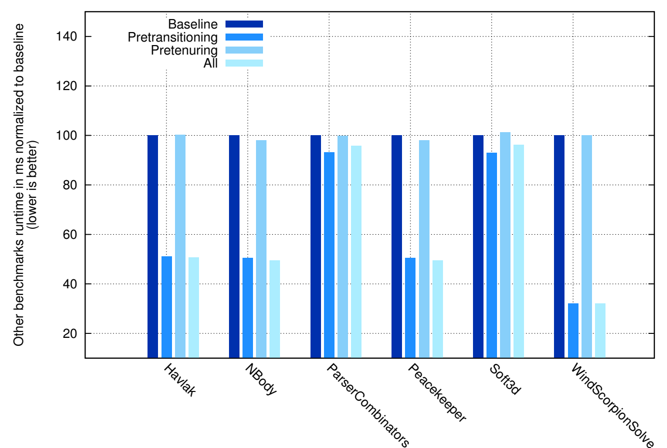

Figure 6. Other benchmarks runtime of baseline, transitioning, pretenuring, and all optimizations turned on normalized to baseline.

results were similar to IA32. We gained no new insights from these platforms and thus chose to omit redundant data for space reasons. V8's dynamic growing strategies for the young and old generation were used and no static limits were set.

For our experiments we used the complete Octane 2.0 [15] and Kraken 1.1 [27] suites, two standard JavaScript benchmarks which are designed to test specific virtual machine subsystems. Each benchmark is run 20 times with a fixed amount of iterations, each run in a separate virtual machine instance. There are no warmup iterations, to be sure to include any overhead that instrumentation may have added to baseline code. The average of the 20 runs is reported. In addition to these standard suites we analyzed many publicly available benchmarks but selected only a few for presentation. Of all the benchmarks we tested, none showed performance degradation. We show results for (1) Havlak [20], a loop recognition algorithm, (2) NBody [16], which solves the classical N-body physics problem, (3) ParserCombinators [24] a parser benchmark that uses a simple arithmetic expression grammar built from parser combinators, (4) the array combined version of Peacekeeper [11], (5) Soft3d [26], a JavaScript software 3d renderer, and (6) WindScorpionSolve [22], which solves linear equations.

We use four configurations of V8 for our experiments. The *Baseline* configuration without allocation-site-based optimizations, *Pretransitioning* with the pretransitioning optimization only, *Pretenuring* with the pretenuring optimization only, and *All* with both optimizations. For maximum repeatability all configurations run V8 in predictable mode which disables nondeterministic features like concurrent recompilation, concurrent sweeping, and concurrent on stack replacement. Note that we observe similar performance im-

	Baseline				Pretransitioning				Pretenuring				All			
	Scavenge		Mark-Sweep		Scavenge		Mark-Sweep		Scavenge		Mark-Sweep		Scavenge		Mark-Sweep	
Benchmark	#	ms	#	ms	#	ms	#	ms	#	ms	#	ms	#	ms	#	ms
Richards	4	4.8	0	0	4	6.4	0	0	4	3.2	0	0	4	5.4	0	0
DeltaBlue	331	35.5	0	0	328	35.4	0	0	331	38.5	0	0	328	38.4	0	0
Crypto	4	4.3	0	0	4	2.9	0	0	4	3.9	0	0	4	5.4	0	0
RayTrace	665	60.7	0	0	665	53.6	0	0	665	69.7	0	0	665	69.1	0	0
EarleyBoyer	779	887.2	0	0	779	939.4	0	0	779	1012.5	0	0	779	1059.2	0	0
RegExp	330	39.5	0	0	330	37.8	0	0	330	38.2	0	0	330	40.2	0	0
Splay	614	8196.9	23	4974.1	614	7920.7	23	4587.9	17	139.1	75	1670.3	17	115	75	1739.5
NavierStokes	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
PdfJS	610	978.8	4	248.3	610	928.4	4	236.7	528	546.3	8	489.1	527	516.3	8	428.9
Mandreel	17	2.8	2	37.7	17	4	2	38.2	17	3.3	2	37.9	17	3.2	2	38.7
Gameboy	33	19.6	1	11	33	15.6	1	11	33	16.3	1	11	33	19	1	11
CodeLoad	13	69.4	1	43.1	13	64.4	1	42.6	13	77.6	1	45.3	13	69.6	1	43.3
Box2d	101	129.6	1	16.8	101	109.2	1	17.4	101	114.1	1	17.2	101	101.4	1	16.4
zlib	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Typescript	41	359.4	2	146.8	41	388.4	2	170	41	410.2	2	157.9	41	401.5	2	179.3
Octane total	**3542**	**10788.5**	**34**	**5477.8**	**3539**	**10506.2**	**34**	**5103.8**	**2863**	**2472.9**	**30**	**2428.7**	**2859**	**2443.7**	**30**	**2457.1**
audio-beat-detection	25	2	0	0	18	1.5	0	0	24	1.8	0	0	17	1.6	0	0
audio-dft	15	0.8	1	16.1	15	0.8	1	16.5	15	1	1	16.2	15	1	1	16
audio-fft	22	1.4	0	0	15	1.4	0	0	22	1.6	0	0	15	1.7	0	0
audio-oscillator	29	1.9	0	0	21	2.1	0	0	29	2.1	0	0	21	2.1	0	0
imaging-gaussian-blur	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
imaging-darkroom	2	1.1	0	1.7	2	1.1	0	1.8	2	1.2	0	1.7	2	1.2	0	1.6
imaging-desaturate	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
json-parse-financial	9	1.5	0	0	9	1.4	0	0	9	1.5	0	0	9	1.7	0	0
json-stringify-tinderbox	7	2.2	0	0	7	3.2	0	0	7	3	0	0	7	2.9	0	0
stanford-crypto-aes	24	5.8	0	0	24	6	0	0	25	6.1	0	0	25	6.1	0	0
stanford-crypto-ccm	16	2.5	2	22	16	2.6	2	22	16	2.6	2	22	16	2.6	2	22
stanford-crypto-pbkdf2	32	1.7	0	0	32	1.4	0	0	32	1.4	0	0	32	1.5	0	0
stanford-crypto-sha256	13	1.4	0	0	13	1.1	0	0	13	1.7	0	0	13	1.3	0	0
Kraken total	**197**	**24.7**	**3**	**39.8**	**175**	**25**	**3**	**40.3**	**197**	**26.6**	**3**	**39.9**	**175**	**26.1**	**3**	**39.6**
Havlak	5	9.7	0	0	5	8.7	0	0	5	9.8	0	0	5	9.2	0	0
NBody	2	2.4	0	0	1	1.9	0	0	2	2.2	0	0	1	1.9	0	0
ParserCombinators	23	4.3	0	0	23	4.3	0	0	23	4.7	0	0	23	4.4	0	0
Peacekeeper	4	2.7	0	0	3	2.8	0	0	4	2.9	0	0	3	2.6	0	0
Soft3d	319	62.8	0	0	319	70.1	0	0	318	578	31	435.3	319	577.9	31	433.3
WindScorpionSolve	9	3.6	0	0	6	2.8	0	0	9	3.6	0	0	6	2.8	0	0

Table 2. Number of scavenges, scavenging time, number of mark-sweeps, and mark-sweep time in the benchmarks.

provements without predictable mode, but variance is much higher.

Results reported in Figures 4-6 focus on performance. The y-axis show the performance improvement of a given benchmark normalized to the baseline, with higher bars being better for Octane and lower bars being better for Kraken and the selected benchmarks. We report detailed garbage collector statistics in Table 2 and counts of transitions in Table 3. Table 4 reports the number of created and found mementos and the total allocated memory for each benchmark.

The obvious standout is the Splay benchmark, which allocates many long-living objects. Here, pretenuring improves the score of Splay by 3x and the score of SplayLatency by 3.6x. This is highly correlated with the reduction in garbage collection work visible in Table 2. The number of performed scavenges drops from 614 to 17 and the overall scavenging time reduced by 80x. Note that also the maximum scavenging pause time reduced significantly since fewer objects survive young generation collections. In Splay five allocation sites are pretenured with a semi-space survival rate of about 99%. PdfJs improves by about 5% due to pretenuring with 49 pretenured allocation sites. Scavenging time was reduced by about 380ms but mark-sweep time increased by 180ms. Pretenuring does not activate on the other benchmarks. However pretenuring does cause baseline code to create mementos, and we see a slowdown of about 3% in Soft3d. These high memento counts arise from getting stuck in baseline code because this benchmark has code that is hard for Crankshaft to optimize.

Pretransitioning has a large impact on the DeltaBlue benchmark in Octane, with a 2.6x improvement, which correlates strongly with a large reduction in the number of transitions as seen in Table 3, from 250,205 to merely 8. It also improves PdfJS, reducing the number of transitions from 226,136 to just 1555. Other benchmarks improved by pre-transitioning also show strong correlations with reduced transition counts, including several Kraken benchmarks and all of the third group of selected benchmarks. Overall pretransitioning improves Kraken by 8%. The improvements are more dramatic in the third group, with about 2x on Havlak, NBody, and PeaceKeaper and 3x on WindScorpionSolver. Anomalies are CodeLoad which pathologically creates a large amount of new source code using JavaScript `eval`, thwarting the learning mechanism of pretransitioning, and EarleyBoyer which has a solid reduction in transitions but almost no speedup; our statistics showed these transitions were of the less expensive variety that require only a map change. It is also interesting that pretransitioning alone

Benchmark	# elements transitions Baseline	# elements transitions with pretransitioning
Richards	25	6
DeltaBlue	250205	8
Crypto	5446	12
RayTrace	4005	8
EarleyBoyer	39140	2009
PdfJS	226136	1555
Gameboy	93	2
CodeLoad	1007	1208
Box2d	34104	11
zlib	24	28
Typescript	74416	6058
ai-astar	4852	7
audio-beat-detection	48	21
audio-dft	44	17
audio-fft	44	17
stanford-crypto-aes	4426	4250
stanford-crypto-ccm	8330	279
stanford-crypto-sha256	2005	12
Havlak	111808	21
NBody	10000	2
ParserCombinators	245750	5
Peacekeeper	9998	1
Soft3d	4777	13
WindScorpionSolve	520459	3

Table 3. Number of element transitions without and with pretransitioning for affected benchmarks.

reduces garbage collection overhead in some cases, e.g. the number of scavenges is reduced slightly in DeltaBlue, audio-dft, audio-oscillator, Havlak, NBody, and WindScorpion-Solve. This is because pre-transitioning avoids the types of transitions that require reallocating and copying the array's internal elements.

We can also see from the experimental results that these two optimizations are almost entirely orthogonal. Where we see an improvement with one optimization, that improvement is also retained with the other optimization also turned on. We can even see several cases where the speedup or slowdowns between the two optimizations are almost perfectly additive: Splay, SplayLatency, PdfJS, MandreelLatency, audio-fft, audio-oscillator, NBody. Coupled with the correlations with counter data, this gives us confidence that our performance measurements are sound.

We also wish to study the memory overhead introduced by allocation mementos. Table 4 reports the total memory allocated by each benchmark in the baseline configuration as well as the number of created and found allocation mementos for pretransitioning and pretenuring. Based on this data, we can see that the number of mementos is quite low, generally in the thousands, with a few outliers. This reflects the our design of restricting instrumentation to baseline code and disabling mementos when transitioning to optimized code. A few outliers are indicative of other performance problems that are beyond the scope of this paper. In particular, PdfJS, Box2d and stanford-crypto-aes seem to get stuck in baseline code due to limitations in Crankshaft and thus continue generating allocation mementos. Relative to the total amount of memory allocated by the benchmarks, the space overhead of mementos is negligible. Pretransitioning allocates only 696KB of mementos and pretenuring only 9MB of mementos, versus 23GB total allocated memory for the entire Octane suite. We did not count the number of mementos in the *All* configuration because mementos are actually shared by both optimizations.

	Baseline	Pretransitioning	Pretenuring	
Benchmark	allocated MB	# created	# created	# found
Richards	32.60	179	14010	13849
DeltaBlue	2563.55	2	11043	39
Crypto	35.16	302	11625	3
RayTrace	5154.96	3	3404	568
EarleyBoyer	5567.6	1030	6659	863
RegExp	2508.28	936	7685	4103
Splay	2546.23	1	7368	4134
NavierStokes	8.39	19	4	2
PdfJS	4265.86	51355	535062	35113
Mandreel	162.06	1	1997	2
Gameboy	226.83	22536	89388	34104
CodeLoad	109.84	1603	29121	10751
Box2d	734.02	64	466076	421
zlib	13.54	14	4178	50
Typescript	249.91	11170	25684	5894
ai-astar	4.9	774	14499	2139
audio-beat-detection	40.95	1033	83	10
audio-dft	41.66	19	33	15
audio-fft	39.48	27	21	7
audio-oscillator	51.33	7	3560	8
imaging-gaussian-blur	9.53	5	13	13
imaging-darkroom	11.33	0	5	5
imaging-desaturate	9.77	0	5	5
json-parse-financial	9.19	0	5	5
json-stringify-tinderbox	11.65	0	805	805
stanford-crypto-aes	43.91	1165	109269	8363
stanford-crypto-ccm	29.88	7355	23024	1048
stanford-crypto-pbkdf2	31.51	2530	3152	61
stanford-crypto-sha256	13.61	1243	8755	57
Havlak	36.64	49702	350545	53335
NBody	16.31	1	148	58
ParserCombinators	182.477	11	4000	1419
Peacekeeper	33.53	377	152	62
Soft3d	2441.95	14708	77739	8276
WindScorpionSolve	74.05	3	4157	60

Table 4. Number of created and found mementos in the benchmarks.

5. Related Work

Pretenuring was first studied in an offline setting by [8]. A heap profile is obtained by instrumenting the program to prepend an allocation site identifier to each object which is inspected for both live and dead objects at garbage collection time. Statistics from profiling runs are used to choose allocation sites to pretenure. This approach was extended by [4, 5] in the context of Java. Using execution profiles as an oracle, their system classifies allocation sites and adds an immortal space for objects that live longer than half of the program execution time. Both application-specific and combined pretenuring advice for libraries improved performance. In [25] the pretenuring classification is based on a program analysis which identifies patterns of lifetime behavior and compares them against a database of previous knowledge of so-called micro-patterns [12]. [32] studied more advanced classification schemes for pretenuring, considering metrics beyond allocation sites, such as types. Static techniques, offline techniques, and dynamic techniques based on training

data have the advantage of low runtime overhead but require prior knowledge of application behavior and cannot react to dynamic feedback. In contrast, our approach requires no offline training but also has low runtime overhead.

The most closely related work on dynamic object sampling techniques clusters around pretenuring. The first dynamic pretenuring system was described by [17] using instrumentation that samples allocation sites when the allocation buffer overflows. Similarly [23] takes samples every 2^n bytes of allocated memory, placing a magic word before sampled objects that is used to identify allocation sites during GC. The magic word is problematic since it is small and must encode the allocation site ID; the collector could be tricked if the word before an object is actually part of the end of the previous object. Mementos are more general, since the payload of a memento is configurable and it contains a real object header. Sampling based on allocation counters is generally more expensive than emitting mementos at allocation sites for two reasons. First, the out-of-line slow path usually involves spilling all the registers and at least one function call, whereas emitting a memento requires just a couple of inline machine instructions. Second, sampling windows must be much smaller than the young generation size in order to instrument an appreciable fraction of objects, leading to many slowpath allocations. Allocation-counter-based techniques also oversample large objects, cannot distinguish between allocations made from optimized code versus unoptimized code, require an as-yet-unexplored external control of the sampling threshold to tune overhead, and are only applicable to lifetime estimation, not the feedback needed for pretransitioning and presizing. We carefully considered many of these alternatives before developing mementos, and our approach of emitting mementos at allocation sites provides reliable feedback for our new optimizations while avoiding the above disadvantages.

Huang [19] dynamically tracks the lifetime of objects at the granularity of types and makes class-based pretenuring decisions, but types proved to be a weak indicator of lifetime. In [28], dynamic profiling is used to identify allocation sites that allocate objects that should be immediately promoted from the young generation upon the first garbage collection. This is a more local approach similar in spirit to our previous high-promotion mode (see Section 3.1.2) since it must copy each object at least once. This extra copy distinguishes it from ours and other systems discussed here that allocate tenured objects directly in the old generation. They also experimented with various schemes for mapping objects back to allocation sites, including hashcodes and an extra indirection from the object to the class. Mementos have the benefit of constant-time access from an allocated object.

The problem of reducing generational overhead can also be addressed by reconfiguring a stock generational system at runtime. For example, instances of this idea are dynamically adjusting promotion thresholds [30, 35], variable-sized young generations [2], and multiple generations with different garbage collection strategies. A combination of these strategies is typically used in production virtual machines.

Storage Strategies [6] as described by Bolz are analagous to V8's ElementsKind concept. They were developed later but independently, before any published material was available. The authors discuss V8's pretransitioning concept which was still in development.

Much work related to ideal data structures is carried out at the application level and is focused on instrumented detection of the ideal structure. In [9] a survey of different approaches to choosing ideal data representation motivates the development of a structured approach called *Just-In-Time data structures*. In [31] instrumentation in the virtual machine and libraries tracks information about data structure usage with the ultimate goal of selecting the best data structure alternative, but good decisions required gathering multiple levels of calling context which proved too expensive for an online system. Instead, dynamic information was summarized as advice to the programmer to make source code changes. Other work [13] [38] has addressed specific representation issues in collections. Some JVMs represent arrays with discontiguous array-lets [3, 7, 29] which save memory with zero-compression, copy-on-write, and lazy allocation and improve garbage collection pause times by limiting the maximum object size. Unlike JavaScript arrays, Java arrays don't change length or representation. In work published to date, using array-lets is a global VM policy rather than based on dynamic feedback.

Other work on tracking dynamic properties of objects runs a spectrum from extremely precise [18] to less precise but faster [37], and graduations between, with some offering programming client analyses. Our work has focused on performance with low-overhead instrumentation to drive the optimizations described in this paper. We believe allocation mementos could be useful as an implementation technique to more powerful analysis techniques.

6. Conclusion

This paper offers a new technique for virtual machines to collect temporal allocation site feedback. Allocation mementos efficiently tie objects to a small payload without a large space or time cost on the program. We made use of mementos and allocation sites to implement two important optimizations: *pretenuring* and *pretransitioning* and prototype a third: *presizing*.

We showed how pretransitioning can use information from allocation sites to avoid expensive array transitions and how pretenuring can reduce garbage collection overhead with better site-specific pretenuring decisions. The rapid proof-of-concept we built to presize arrays offers promise that mementos could be useful for a broader range of dynamic feedback in the future. Since creating mementos is a choice at the allocation time of an object, the overhead

can be tuned, a capability that we exploit to keep optimized code fast and free from the cost of creating mementos. We limit space overhead by only creating mementos in baseline code for objects in the young generation, even though the architecture could support mementos anywhere.

We measured and validated our results on several suites of benchmarks and showed solid performance improvements. We delivered our optimizations in the V8 production virtual machine which demanded important controls like a pretenuring recovery mechanism and comprehensiveness over a wide range of dynamic behavior.

References

[1] W. Ahn, J. Choi, T. Shull, M. J. Garzarán, and J. Torrellas. Improving javascript performance by deconstructing the type system. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, 2014.

[2] A. W. Appel. Simple generational garbage collection and fast allocation. *Softw. Pract. Exper.*, 19(2), Feb. 1989.

[3] D. F. Bacon, P. Cheng, and V. T. Rajan. Controlling fragmentation and space consumption in the metronome, a real-time garbage collector for java. *SIGPLAN Not.*, 38(7):81–92, June 2003. ISSN 0362-1340. . URL http://doi.acm.org/10.1145/780731.780744.

[4] S. M. Blackburn, S. Singhai, M. Hertz, K. S. McKinely, and J. E. B. Moss. Pretenuring for java. *SIGPLAN Not.*, 36(11), Oct. 2001.

[5] S. M. Blackburn, M. Hertz, K. S. McKinley, J. E. B. Moss, and T. Yang. Profile-based pretenuring. *ACM Trans. Program. Lang. Syst.*, 29(1), Jan. 2007.

[6] C. F. Bolz, L. Diekmann, and L. Tratt. Storage strategies for collections in dynamically typed languages. *SIGPLAN Not.*, 48(10), Oct. 2013.

[7] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, B. Mathiske, and M. Wolczko. Heap compression for memory-constrained java environments. *SIGPLAN Not.*, 38 (11):282–301, Oct. 2003. ISSN 0362-1340. . URL http://doi.acm.org/10.1145/949343.949330.

[8] P. Cheng, R. Harper, and P. Lee. Generational stack collection and profile-driven pretenuring. *SIGPLAN Not.*, 33(5), May 1998.

[9] M. De Wael, S. Marr, and W. De Meuter. Data interface + algorithms = efficient programs: Separating logic from representation to improve performance. In *Proceedings of the 9th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems PLE*, ICOOOLPS '14, 2014.

[10] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, 1984.

[11] Futuremark. PeaceKeeper, 2014.

[12] J. Y. Gil and I. Maman. Micro patterns in java code. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, 2005.

[13] J. Y. Gil and Y. Shimron. Smaller footprint for java collections. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, 2011.

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

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

[16] I. Gouy. NBody, 2011.

[17] T. L. Harris. Dynamic adaptive pre-tenuring. In *Proceedings of the 2Nd International Symposium on Memory Management*, ISMM '00, 2000.

[18] M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stefanović. Generating object lifetime traces with merlin. *ACM Trans. Program. Lang. Syst.*, 28(3), May 2006.

[19] W. Huang, W. Srisa-an, and J. M. Chang. Dynamic pretenuring schemes for generational garbage collection. In *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '04, 2004.

[20] R. Hundt. Havlak, 2014. URL https://code.google.com/p/multi-language-bench/source/browse/trunk/src/havlak.

[21] Joyent Inc. Node.js, 2014. URL http://nodejs.org/.

[22] JS-X.com. WindScorpionSolve, 2005. URL http://www.js-x.com/page/javascripts__example.html?view=1068.

[23] M. Jump, S. M. Blackburn, and K. S. McKinley. Dynamic object sampling for pretenuring. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, 2004.

[24] R. Macnak. ParserCombinators, 2013.

[25] S. Marion, R. Jones, and C. Ryder. Decrypting the java gene pool. In *Proceedings of the 6th International Symposium on Memory Management*, ISMM '07, 2007.

[26] D. McNamee. Soft3d, 2008. URL www.deanmcnamee.com.

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

[28] R. Odaira, K. Ogata, K. Kawachiya, T. Onodera, and T. Nakatani. Efficient runtime tracking of allocation sites in java. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '10, 2010.

[29] J. B. Sartor, S. M. Blackburn, D. Frampton, M. Hirzel, and K. S. McKinley. Z-rays: Divide arrays and conquer speed and flexibility. *SIGPLAN Not.*, 45(6):471–482, June 2010. ISSN 0362-1340. . URL http://doi.acm.org/10.1145/1809028.1806649.

[30] M. L. Seidl and B. G. Zorn. Segregating heap objects by reference behavior and lifetime. *SIGPLAN Not.*, 33(11), Oct. 1998.

[31] O. Shacham, M. Vechev, and E. Yahav. Chameleon: Adaptive selection of collections. In *Proceedings of the 2009 ACM*

SIGPLAN Conference on Programming Language Design and Implementation, PLDI '09, 2009.

[32] J. Singer, G. Brown, M. Luján, and I. Watson. Towards intelligent analysis techniques for object pretenuring. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*, PPPJ '07, 2007.

[33] S. Tilkov and S. Vinoski. Node.js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83, Nov. 2010. ISSN 1089-7801.

[34] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, 1984.

[35] D. Ungar and F. Jackson. An adaptive tenuring policy for generation scavengers. *ACM Trans. Program. Lang. Syst.*, 14 (1), Jan. 1992.

[36] D. Ungar and R. B. Smith. Self: The power of simplicity. *SIGPLAN Not.*, 22(12), Dec. 1987.

[37] G. Xu. Resurrector: A tunable object lifetime profiling technique for optimizing real-world programs. *SIGPLAN Not.*, 48 (10), Oct. 2013.

[38] G. Xu and A. Rountev. Detecting inefficiently-used containers to avoid bloat. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, 2010.

[39] A. Zakai. Emscripten: An llvm-to-javascript compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, pages 301–312, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0942-4.