Runtime and Compiler Support for HAMTs

Sona Torosyan
University of Utah
USA
s.torosyan@utah.edu

Jon Zeppieri
independent researcher
USA
zeppieri@gmail.com

Matthew Flatt
University of Utah
USA
mflatt@cs.utah.edu

Abstract

Many functional languages—including Racket, Clojure, and Scala—provide a persistent-map datatype with an implementation based on Hash Array Mapped Tries (HAMTs). HAMTs enable efficient functional lookup, insertion, and deletion operations with a small memory footprint, especially when taking advantage of implementation techniques that have been developed since the original HAMT implementation. Racket's latest HAMT implementation is based on an intermediate data structure, a *stencil vector*, that supports an especially compact representation of HAMTs with help from the compiler and memory manager. That is, stencil vectors provide an abstraction to improve HAMT performance without burdening the compiler with all of the complexity and design choices of a HAMT implementation. Benchmark measurements show that HAMTs in Racket have performance comparable to other state-of-the-art implementations, while stencil-vector HAMTs are more compact and run as fast as alternative representations in Racket. Although we only report on Racket, our experience suggests that a stencil-vector datatype in other dynamic-language implementations might improve HAMT performance in those implementations.

CCS Concepts: • **Software and its engineering** → **Compilers**; **Runtime environments**; **Data types and structures**.

Keywords: Hash array mapped trie, Racket

ACM Reference Format:
Sona Torosyan, Jon Zeppieri, and Matthew Flatt. 2021. Runtime and Compiler Support for HAMTs. In *Proceedings of the 17th ACM SIGPLAN International Symposium on Dynamic Languages (DLS '21), October 19, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3486602.3486931

1 Introduction

Key–value maps are pervasive in program implementations. In an imperative style, programs traditionally use *hash tables* as a general and efficient implementation of maps. To support a functional style, languages and libraries can offer a *persistent* form of maps, where adding or removing a key–value pair returns a new map value, instead of mutating the given map. To make that addition or deletion efficient, the new map value will share internal structure with the given map; the details of that sharing are private to the implementation, but it invariably involves a tree structure, as opposed to the mutable array that is used in a traditional hash table. That way, a functional update operation requires reallocating only nodes along the spine from the root of the internal tree to the changed node.

The tree within a persistent-map implementation might be a balanced binary search tree using the integers that are produced by a hash function. An alternative is to use a *trie*, where searching is based on inspecting a prefix of a number's representation instead of comparing whether one number is less than another. When hashing converts every key to a integer whose representation is bounded, a trie can have a bounded depth without needing to be balanced.

The Hash Array Mapped Trie (HAMT) data structure was first described and implemented in C++ by Bagwell [1]. It represents trie nodes compactly by associating a bitmap with each node, and only positions with bits set in the bitmap are represented in the node's array of child references. That is, the size of each node corresponds to the number of its non-empty children instead of its maximum number of children, which makes the representation much more compact. Subsequent refinements to Bagwell's design include reordering and distinguishing child references based on whether they represent leaves or subtrees, and those details also can be represented in the bitmap.

When a data structure is important enough to the performance of a programming language, it may receive special treatment by the compiler and runtime system. For example, implementations of Lisp typically have a special representation for cons cells, allowing them to take up just two words of memory (perhaps encoding *cons*ness through bits in the pointer), whereas other kinds of two-word values need an additional tag word. Operations on important datatypes are often hand-coded in a low-level layer or inlined by the compiler. HAMTs could similarly benefit from direct compiler and runtime support, but they are more complex than the kinds of datatypes that normally receive special treatment.

Racket's HAMT-based persistent map is important enough for its performance to be worth compiler support. For example, the macro expander makes heavy use of persistent maps to implement sets of scopes and binding environments. To gain the benefits of compiler support without embedding too many HAMT details into the compiler and runtime implementation, we have introduced an intermediate data structure into the Racket backend compiler (which is Chez Scheme): a *stencil vector*. A stencil vector is a one-dimensional array plus a mask bitmap, where the vector is conceptually as long as the bitmap is wide, but a vector element is represented in memory only if the corresponding bit is set in the mask. The stencil-vector datatype enables the compiler and runtime system to offer two benefits: (1) a stencil vector has a HAMT-style bitmap that determines the vector's size, so the representation does not have to include a redundant size field for safe-mode bounds checks and for garbage collection, and (2) a bitmap-based update operation can allocate and fill a new stencil vector between garbage-collection points, which avoids the need to initialize the vector's memory with collector-friendly values [15] or to coordinate updates with the collector through a write barrier.

In this paper, we describe Racket's HAMT implementation and its use of stencil vectors, and we provide benchmarks to demonstrate how this implementation performs better than other implementations that we have previously deployed in Racket. We also show performance comparisons to state-of-the-art implementations on the JVM—which is a kind of apples-to-oranges comparison, but it provides evidence that Racket's implementations are comparable to the state of the art. Thus, while stencil vectors are currently specific to Racket, the fact that they can improve a competitive implementation suggests that compiler and runtime support might usefully translate to other dynamic-language implementations.

2 Prior Work

The *trie* data structure was first implemented by Briandais [2] and named by Fredkin [6]. Bagwell [1] used the idea of tries with the bits of a hash code to serve as the strings, and he combined it with partitioning based on hash codes and the linear hash principles of Litwin et al. [8] to solve problems of collision management and storage growth. Bagwell's implementation of tries has a smaller memory footprint than previous implementations, and by using hash codes as keys, it guarantees an upper bound of $O(log_{32}(n))$ for lookup, insertion, and deletion.

Bagwell's mutable HAMT implementation was adapted by Hickey [7] to implement a functional and immutable HAMT data structure for Clojure. Scala through version 2.12 uses a HAMT implementation, but where the implementations of Bagwell and Hickey store key–value pairs for a node in the node itself, Scala's HAMT stores a key–value pair in a leaf node, and it also records a computed hash code alongside the key.

Steindorfer and Vinju [12] improve HAMT's implementation with their Compressed Hash-Array Mapped Prefix-tree (CHAMP) implementation. CHAMP improves locality and ensures that the tree remains in a canonical and compact representation after deletion. Scala 2.13 and later use CHAMP. Compared to the HAMTs of Scala 2.12 and Clojure, CHAMP achieves a smaller memory footprint through its layout for internal trie nodes, which saves space at the cost of more bitwise arithmetic: a mapped position in a node is represented by an array of slots that may refer to a key–value pair or subtree, instead of having separate key–value and subtree slots where only one or the other is used. At the same time, the slots are reordered so that subtree references are grouped together, which improves locality during iterations.

For applications that involve one-to-many mappings, a map data structure can be used with values that are collections, but the AXIOM data structure [13] provides more direct support for one-to-many mappings. It builds on the ideas of CHAMP, but it further refines the use of mask bits to distinguish node types, which lets it more compactly represent one-to-many relations and speed iteration operations over a combination of keys and values. The approach to distinguishing one-to-many mappings could also be used to distinguish one-to-zero mappings, enabling the same data structure to compactly represent both maps and sets.

Variations on the HAMT idea, like CHAMP and AXIOM, share many common implementation details. Some of that commonality is the target of stencil vectors as presented here. As even more direct support, Steindorfer and Vinju [14] describe a domain-specific language for generating trie-based implementations of collections.

3 HAMTs

Figure 1 illustrates the basics of how a HAMT representation works compared to a traditional array representation of hash tables. The picture shows keys without values, so it demonstrates a hash set rather than a hash map, but that's enough to cover the essentials.

Part (a) of the figure shows the hash codes for example key objects A, B, C, and D.

Part (b) depicts an array-based table, where each hash code is converted to an index by taking its modulus with respect to the table size. Also, each index is used directly as an array index, which is why the slots of the array are consecutively numbered in the upper left of the slots.

Part (c) shows a HAMT representation where only A and B have been added to the table. To determine an index within

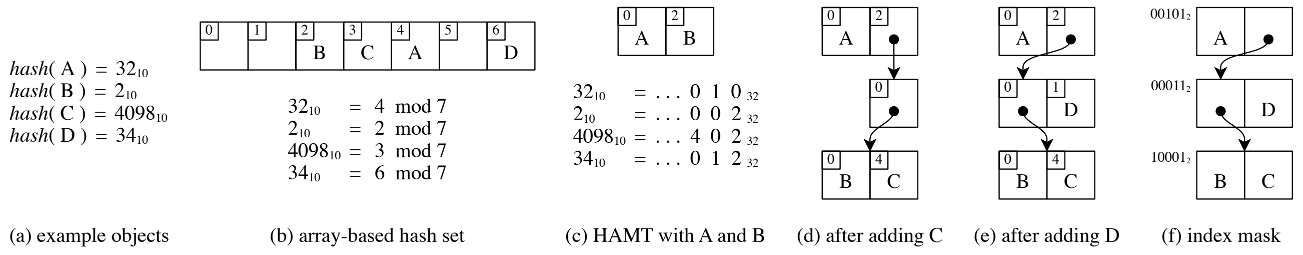

Figure 1. Insertion of A, B, C, and D objects into a HAMT, based on the example and picture by Steindorfer and Vinju [12].

the node, each hash code is considered as a sequence of base-N digits, where N is the HAMT's *branching factor*. For this example, N is 32. The hash codes of A and B differ in the first (least-significant) base-32 digit, so they are mapped to different indices in the topmost HAMT node, and no further nodes are needed. Within the node, the upper left of each slot shows the index for that slot. Although the indices are in order, they do not match the array indices, because only non-empty slots are represented in the node's array.

Part (d) shows a HAMT where C has been added. The hash codes of B and C have the same first two base-32 digits and differ in the third, so the table is now three nodes deep.

Part (e) shows the table after adding D, which has the same first base-32 digit as B and C, but a different second digit. If another key were added that has the same first two base-32 digits as D, then that element and D would be moved to another node, and the HAMT would start to look more tree-like.

Part (f) shows the same HAMT representation as (e), but where the indices on slots are encoded by a mask. For example, the first node's slots have indices of 0 and 2 because the set bits in 00101_2 are at positions 0 and 2 (counting from the least-significant end). A mask and an index can be combined with bitwise shift and popcount operations to convert an index to an array position.

Left implicit in figure 1 is some way of distinguishing node slots that contain keys versus subtrees. In a language like Java or Racket, `instanceof` or a structure predicate can be used to make that distinction. The CHAMP design [12] uses another possibility: use two masks, where one mask has a bit set for each slot that refers to a subtree, and another mask has a bit set for each slot that refers to a key (and the two bitmaps are mutually exclusive). Since branching factors N in the range of 8 to 32 tend to perform well, there can easily be room for two N-bit masks within one machine word. The masks could be interleaved or concatenated, but concatenation fits well with the idea of grouping all subtree slots before key slots, which tends to promote locality.

Separating subtree references from keys offers a further advantage for storing values alongside keys, since a subtree reference will occupy a single machine word, while a key and value pair will require two words. Racket HAMTs further separate keys from values using a third mask, which has a bit for each key that has a value. (Every bit set in the value mask must also be set in the key mask.) A key without a value is implicitly mapped to "true," which is useful for compactly representing sets without a separate data structure. This choice limits the branching factor N further, since it can be only 1/4 of the word size to preserve a power-of-2 N, but branching factors of 8 and 16 work well on 32-bit and 64-bit architectures, respectively.[1]

While there are tradeoffs in the details, many good choices for HAMT nodes create a correspondence between bits in a mask (or concatenation of masks) and the number of slots in a node. Whether slots are used to hold keys, values, subtrees, or metadata such as the total size of a tree, the general strategy is a mask combined with a compressed array. That general strategy can be supported to good effect in the runtime system through a new datatype, which brings us to stencil vectors.

4 Stencil Vectors

A *stencil vector* is a one-dimensional array of fixed size N, where each of its N elements is either present or absent. It is furthermore a compressed array, since only the elements that are present are represented in memory. Concretely, a stencil vector consists of (a) a mask bitmap of N bits and (b) an array of present elements. The "stencil" part of a stencil vector is that its bitmap indicates which slots of the array need to be represented. If bit i is the jth bit that is set in the bitmap, then the array's ith element is present and stored in the jth position of the array. If bit i is unset, then the ith element is not present and has no representation in the stencil-vector's array.

As long as a small N suffices, as it does for representing HAMT nodes, then a stencil vector's mask and array can be stored consecutively in memory, and accessing an element of a stencil vector requires only a few instructions. When an

[1]AXIOM [13] suggests an alternative, which is to use two bits total to represent four possible states: subtree, key with value, key without value, or empty. In Racket, however, slightly less than a word size is available for representing masks efficiently, so using two bits instead of three does not enable a larger branching factor.

element *i* is present (i.e., the *i*th bit is set in the stencil vector's mask), the element's array index is computed by counting the number of bits that are set below the *i*th bit, which on many machines is a bitwise "and" operation followed by a "popcount" operation.

Another key to stencil-vector performance is a functional-update operator to produce a new stencil vector. A functional-update primitive can be faster than allocating a plain array and filling it, because the runtime system sees filling an array as a side-affecting sequence that requires extra steps to cooperate with the generational garbage collector.[2] Limiting a stencil vector's size to a small enough N means that the allocation and filling step can reasonably be inlined and atomic with respect to garbage collection.

The stencil-vector API for Chez Scheme includes `stencil-vector` for creating a stencil vector, `stencil-vector-mask` for accessing its mask, `stencil-vector-ref` for accessing an element by array index, and `stencil-vector-update` for functional update of an existing stencil vector. In the examples below, we use numbers prefixed with `#b`, which is Scheme syntax for a base-2 number.

`(stencil-vector` *`mask element ...`*`)` allocates a new stencil vector whose mask and content are *`mask`* and the given *`element`* values. The *`mask`* must be a fixnum in the range of supported masks, which is 0 through 2^{58}-1 on a 64-bit platform and 0 through 2^{26}-1 on a 32-bit platform. The number of provided *`element`*s must match the number of bits set in *`mask`*. Note that the compiler can statically verify this correspondence in an immediate call to `stencil-vector` where the *`mask`* argument is a literal number. Examples:

```
> (stencil-vector #b10010 'a 'b)
#<stencil-vector #b10010: 'a 'b>
> (stencil-vector #b111 'a 'b 'c)
#<stencil-vector #b111: 'a 'b 'c>
```

`(stencil-vector-mask` *`st-vec`*`)` returns the mask for the stencil vector *`st-vec`*. The mask is always a fixnum.

```
> (stencil-vector-mask (stencil-vector #b10010 'a 'b))
18
```

`(stencil-vector-ref` *`st-vec j`*`)` returns the *`j`*th item in the array for *`st-vec`*, where *`j`* must be an index that is less than the number of bits set in *`st-vec`*'s mask. Except for a bounds check, which is omitted in unsafe mode, accessing an element from a stencil vector is the same at the machine level as accessing an element from a plain Scheme vector.

```
> (stencil-vector-ref (stencil-vector #b10010 'a 'b) 1)
'b
> (stencil-vector-ref (stencil-vector #b111 'a 'b 'c) 1)
'b
```

[2]Although our implementation of stencil vectors currently supports imperative updates with the associated administrative work to cooperate with the garbage collector, we ignore those operations for our purposes here.

`(stencil-vector-update` *`st-vec remove-mask add-mask element ...`*`)` performs a functional update of *`st-vec`*. This is the one operation that relies on a corespondence between bits set in *`st-vec`*'s mask and the positions of elements, because elements corresponding to *`remove-mask`* are removed, and then the *`element`*s are inserted at positions based on *`add-mask`*. The *`remove-mask`* argument must have only bits that are currently set in *`st-vec`*'s mask, while *`add-mask`* argument must have only bits that are unset after subtracting *`remove-mask`* from *`st-vec`*'s mask. If *`remove-mask`* and *`add-mask`* have bits in common, those are places where an element in the stencil vector is being replaced. The number of *`element`* arguments must match the number of bits set in *`add-mask`*.

```
> (define st-vec (stencil-vector #b101 'a 'b))
> (stencil-vector-update st-vec #b0 #b10 'c)
#<stencil-vector #b111: 'a 'c 'b>
> (stencil-vector-update st-vec #b0 #b1000 'c)
#<stencil-vector #b1101: 'a 'b 'c>
> st-vec ; unchanged by updates
#<stencil-vector #b101: 'a 'b>
> (stencil-vector-update st-vec #b1 #b1 'c)
#<stencil-vector #b101: 'c 'b>
> (stencil-vector-update st-vec #b100 #b100 'c)
#<stencil-vector #b101: 'a 'c>
> (stencil-vector-update st-vec #b100 #b0)
#<stencil-vector #b1: 'a>
```

A HAMT implementation needs additional functions that correlate mask bits to positions, but those functions do not need to be primitive. For example, an operation to access the element corresponding to a bit value can be implemented using `fxpopcount` and other bitwise operations:[3]

```
(define (stencil-vector-bitwise-ref st-vec bit-val)
  (define i (fxpopcount (fxand (stencil-vector-mask st-vec)
                               (fx- bit-val 1))))
  (stencil-vector-ref st-vec i))

> (stencil-vector-bitwise-ref (stencil-vector 5 'a 'b) 4)
'b
```

To get the last ounce of performance for Racket's HAMT implementation, we compile it with Chez Scheme's unsafe mode. In that mode, `stencil-vector-bitwise-ref` as above becomes as fast as any primitive version would be.

The implementation of stencil vectors for Chez Scheme requires about 150 lines of library code, about 90 lines of code in the compiler pass for inlining primitives, about 15 lines of support in the garbage collector's domain-specific language, and about a dozen lines of glue in the C-implemented kernel for boot-file loading and the C API.

5 From Stencil Vectors to HAMTs

A full implementation of HAMTs using stencil vectors is too long to fit in this paper, but we sketch some key functions in

[3]The `fxpopcount` operation returns the number of bits that are set in (the two's complement representation of) a fixnum.

the implementation to illustrate how stencil-vector HAMTs work.

Assuming a branching factor of 16 (which works on a 64-bit platform), we reserve 16 stencil-vector slots for each of the subtrees, keys, and values. A subtree slot will be mapped only if the key slot for the same index is not mapped, and vice versa. A value slot will be mapped only if the key slot for the same index is mapped.

```
(define BRANCHING-FACTOR 16)

(define SUBTREE-BITPOS 0)
(define KEY-BITPOS (+ SUBTREE-BITPOS BRANCHING-FACTOR))
(define VAL-BITPOS (+ KEY-BITPOS  BRANCHING-FACTOR))
```

When we want to add a key and value to a node where the index is not already mapped to a key or subtree, then a simple form of the addition is[4]

```
(define (node-add-key node key val idx)
  (stencil-vector-update
   node
   0
   (ior (<< 1 (+ idx KEY-BITPOS))
        (<< 1 (+ idx VAL-BITPOS)))
   key
   val))
```

Note that we know to provide the *key* and *val* arguments to `stencil-vector-update` in that order, because keys are always stored before values.

With 48 slots in a stencil vector reserved for subtrees, keys, and values, our implementation on 64-bit Chez Scheme would have 10 unused slots per node. Those used slots take up no additional memory outside the mask bits, of course, but they could be used to store additional metadata. In Racket's implementation of HAMTs, we allocate one extra slot in every stencil-vector node to record the total number of items in the subtree that is represented by the node; that way, getting the size of a HAMT is a constant-time operation without needing an extra wrapper object around the topmost node. The same metadata word also encodes whether the node is part of a HAMT that uses `eq?`, `eqv?`, or `equal?` hashing (again, without needing an extra wrapper object). Avoiding a wrapper object while making nodes larger by one word is a good trade-off when small HAMTs are common, as they are in Racket.

If we adjust the implementation to use a slot in a stencil-vector node to encode a count, then it turns out to be convenient to use the first slot for the count, so we shift the other slots and bit positions:

```
(define COUNT-BITPOS 0)
(define SUBTREE-BITPOS 1)
(define KEY-BITPOS (+ SUBTREE-BITPOS BRANCHING-FACTOR))
(define VAL-BITPOS (+ KEY-BITPOS  BRANCHING-FACTOR))
```

[4]We use `<<` for a bitwise left-shift, `>>` for a bitwise right shift, `ior` for a bitwise "inclusive or," and so on.

Now, `node-add-key` needs to update the count on a node by removing the old count and adding a new one, in addition to adding the key and value. The count slot is always first, so the updated count is supplied to `stencil-vector-update` before *key* and *val*. Furthermore, since the count is always in the first slot of a node, we can access the old count using `stencil-vector-ref` and array index 0:

```
(define (node-add-key node key val idx)
  (stencil-vector-update
   node
   (<< 1 COUNT-BITPOS)
   (ior (<< 1 COUNT-BITPOS)
        (<< 1 (+ idx KEY-BITPOS))
        (<< 1 (+ idx VAL-BITPOS)))
   (+ 1 (stencil-vector-ref node 0))
   key
   val))
```

Suppose that a key is added to a node where another key already exists at the same index. In that case, the two keys will be combined (with their values) in a new *subtree*. The node must be updated to remove the old key and value while adding the subtree (which must contain two keys):

```
(define (node-remove-key-add-subtree node subtree idx)
  (stencil-vector-update
   node
   (ior (<< 1 COUNT-BITPOS)
        (<< 1 (+ idx KEY-BITPOS))
        (<< 1 (+ idx VAL-BITPOS)))
   (ior (<< 1 COUNT-BITPOS)
        (<< 1 (+ idx SUBTREE-BITPOS)))
   (+ 1 (stencil-vector-ref node 0))
   subtree))
```

We have so far assumed that a value is always present in a node when its key is present. To more compactly represent sets, Racket HAMTs refrain from storing a value when it is `#t` (i.e., the "true" literal). When a key–value mapping is updated in a HAMT to replace an old value with a new one, the update operation will need to check whether the value bit is set and adjust it as appropriate. In some cases, the value bit and its associated slot are removed, in some cases they are added, and in some cases the bit stays the same while the slot may be updated:

```
(define (node-replace-val node idx val)
  (define bit (<< 1 (+ idx VAL-BITPOS)))
  (cond
    [(andtest? (stencil-vector-mask node) bit)
     ; old value is not #t
     (if (eq? val #t)
         (stencil-vector-update node bit 0)
         (stencil-vector-update node bit bit val))]
    [else
     ; old value is #t
     (if (eq? val #t)
         node
         (stencil-vector-update node 0 bit val))]))
```

As this function illustrates, there is some branching cost to saving space by omitting `#t` values, and similar branches are needed in other functions, like `node-add-key`. Happily, the leaves of the functions are efficient stencil-vector update operations. The time–space trade-off may vary for other contexts, and stencil vectors accommodate different implementation choices at that level.

6 Stencil Vectors as Scheme Objects

To understand the performance effect of stencil vectors and alternatives for HAMTs in Racket, it's helpful to understand the internal representation of values generally. Racket uses Chez Scheme as its core compiler and runtime system [5], and we have added stencil vectors at that level, so the representation details are Chez Scheme's [3]. At the end of this section, we reflect on how the details might be different in another dynamic-language implementation.

In Chez Scheme, every value is represented by a word-sized reference that may encode a memory address, but some references are for immediate values (such as small integers or booleans). The lower *tag bits* of a reference provide an initial layer of type information, including whether the reference encodes a memory address versus an immediate value. For some references that encode memory addresses (again, determined by the reference's tag bits), a second layer of type information is in a word at the start of the object's memory.

Figure 2 illustrates how a 64-bit reference xx...xxxxyyy$_2$ is decoded as a value, where the lower three bits yyy are used as tag bits. If the lower three bits are 000, then the value is an immediate integer, that is, a *fixnum*. If the lower three bits are 001, then the value represents a `cons` cell, and the bits after the tag bits determine the cell's location in memory;[5] no further tagging is needed at that memory address. We omit some other cases from the figure, but the most general case is when the tag bits are 111, in which case the reference refers to memory that starts with a word that has its own tag bits using a secondary encoding. All allocated objects are at least 8-byte-aligned,[6] so we preserve access to the full address space for pointers using the bits that would otherwise always be 000.

In the secondary encoding reached through 111, the bits 000 indicate a Racket vector, while the bits 011110 indicate our new stencil-vector datatype. After secondary tag bits, the rest of the first word of an object can be used for a type-specific purpose. In the case of a vector, the remaining bits indicate the vector's size. In the case of a stencil vector, the remaining bits hold a mask, and the size of the stencil

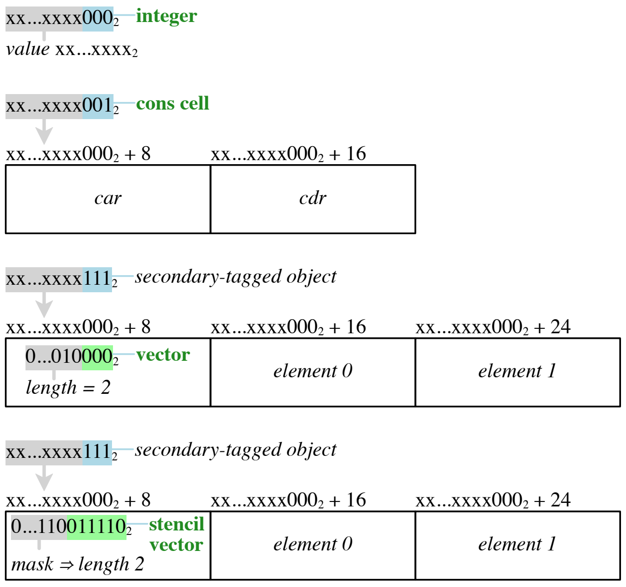

Figure 2. Examples of 64-bit value references

vector—as needed by the garbage collector or for bounds checking—is the number of bits set in the mask.

Given that the secondary tag 011110 uses six bits, a stencil vector's mask is limited to 26 bits on a 32-bit platform and 58 bits on a 64-bit platform. The stencil-vector implementation does not care how those bits are used for HAMT nodes, but our HAMT implementation will distribute most of the remaining bits equally among three groups: subtree slots, key slots, and value slots. As a result, our HAMT implementation is limited to a branching factor of 8 on a 32-bit platform and 16 on a 64-bit platform; happily, those branching factors perform well, and they leave at least one bit/slot available (outside the three groups) for additional HAMT-node metadata. It's also significant that the number of bits in a mask is no more than the number of bits available in a fixnum, which means all mask operations can be performed with fast fixnum arithmetic.

Many dynamic-language implementations use a two-level tagging regime similar to Chez Scheme's. We expect that our strategy for stencil vectors in Chez Scheme would carry over to most such implementations. In particular, our encoding does not change the primary layer of tagging, where bit combinations are an especially limited resource; some implementations use only one bit to distinguish immediate integers from other kinds of values, and some implementations use "NaN boxing" to distinguish immediate floating-point values from other kinds of values—but stencil vectors fall into the "other" category. Our approach affects only the second encoding layer, as stored in an object's header, which is typically more flexible. The details of a second-level encoding affect how many bits can be made available for a stencil-vector

[5]If you view the reference to a `cons` cells as a pointer, then it points to just before the actual memory of the cons cell, which has the sometimes-convenient property that all fields are at a positive offset from that pointer.
[6]The assumption of 8-byte alignment applies to both the 32-bit and 64-bit implementations. The allocator for 64-bit Chez Scheme aligns to 16 bytes, but an object reference is sometimes allowed to reach into the middle of another object to address a double-precision floating-point number.

mask, and if 26 or 58 bits were not enough to implement HAMT nodes, we might have shuffled other encodings in Chez Scheme so that fewer than 6 bits could distinguish a stencil vector from other values; in a different language implementation, a shuffling like that might be necessary.

7 Benchmarks

Figure 3 shows a comparison of the **stencil-vector HAMT** implementation in Racket, the **CHAMP** implementation in Java, and the **PersistentHashMap** from Clojure. The benchmarks, which we describe in more detail in section 7.3.1, are the ones from Steindorfer and Vinju [12]. Since the Java and Clojure implementations have a branching factor of 32 and run on a different virtual machine than the Racket implementation, and since those virtual machines use different implementation techniques (JIT versus ahead-of-time compilation), the value of this performance comparison is limited. Still, the comparison suggests that Racket HAMTs perform in the same neighborhood as state-of-the-art and widely used HAMT implementations, so our detailed measurements for HAMTs and stencil vectors within Racket address a meaningful point on the performance spectrum.

7.1 Persistent Maps in Racket

Our Racket-specific measurements compare three different implementations of persistent maps that reflect historical implementations over Racket's evolution:

- The **Patricia trie** implementation (lines of code: 383) is based on the one first implemented by Morrison [9] and then used by Okasaki and Gill [10] to represent a finite map with integer keys. The implementation of a Patricia trie is much simpler than HAMT implementations, and it performs especially well for small maps. A Patricia trie tends to be less compact than other alternatives; although it has less metadata, it has a branching factor of only two.
- The **vector HAMT** implementation (lines of code: 785) uses a plain Scheme vector for keys and subtrees within a node, plus a separate vector for values. A wrapper object for each node combines those vectors with separate mask bitmaps for keys and subtrees, so it is not necessary to distinguish subtree values from other kinds of values in the key/subtree vector—which means, in turn, that subtrees and HAMT values (which could be keys in other HAMTs) can be represented in the same way. Unlike in CHAMP, keys and subtrees are interleaved in the key/subtree vector. The branching factor for nodes is 16.
- The **stencil-vector HAMT** implementation (lines of code: 1075) uses stencil vectors directly. We take advantage of the fact that stencil vectors are used only for HAMTs in Racket, so a stencil-vector value can be used directly as a HAMT or subtree representation and remain distinct from all other kinds of run-time values. Each node in the stencil-vector implementation uses CHAMP-style layout that groups subtree and key entries separately, and values are kept in a third contiguous portion of the vector. The branching factor for nodes is 16 on a 64-bit platform.

These implementations of persistent maps change many design parameters at once. They're comparable in the sense that each one starts with a premise (implementing Patricia tries, using plain vectors, or using stencil vectors), and other design parameters have been tuned to make that premise perform well.

To help tease out the specific benefits of stencil vectors, we run benchmarks on two additional implementations:

- The **extra-tag stencil-vector HAMT** implementation is the same as the **stencil-vector HAMT** implementation, but the object's type tag holds the vector length (like a plain vector) and the mask is kept as an additional word after the tag word. This variant exposes the trade-off between using extra space to store a vector length versus having to perform a popcount operation to determine its length.
- The **write-barrier stencil-vector HAMT** implementation is the same as the **stencil-vector HAMT** implementation, but operations to initialize a stencil vector use the same write-barrier implementation that updating a plain vector would use.[7] This implementation nevertheless refrains from first filling a stencil vector with temporary values, because allocation and initialization are atomic.

For all of the HAMT implementations, keys and values get separate mask bits within a node in much the same way that CHAMP gives subtrees and key–value pairs separate mask bits. That separation is useful in Racket, because *sets* are an especially common use of persistent maps in the macro expander, and a set has no need of a value with each key. Hash codes are sometimes stored alongside the key in Racket's HAMTs, depending on the table's equality predicate; `eq?`-based (pointer equality) table and `eqv?`-based (number equality) tables do not store the hash code, because those equivalence predicates are always fast, while `equal?`-based (structural equality) tables store the hash code by grouping it with

[7] Chez Scheme's write-barrier implementation uses a combination of a sequential store buffer (SSB) and card marking. The end of the current nursery allocation area is used as the SSB. SSB writes are filtered inline when the written value is a fixnum. When the SSB fills up (i.e., when the backward-moving SSB pointer meets the meets the forward-moving allocation pointer), when a new nursery allocation area is created, or when a garbage collection starts, the SSB is flushed by marking cards. Flushing filters writes where the current value does not refer to a newer generation than the written address. Cards with marks are chained in a linked list, and each card also has summary generation information [11] so that the linked lists can be generation-specific; the garbage collector checks only cards relevant to generations being collected.

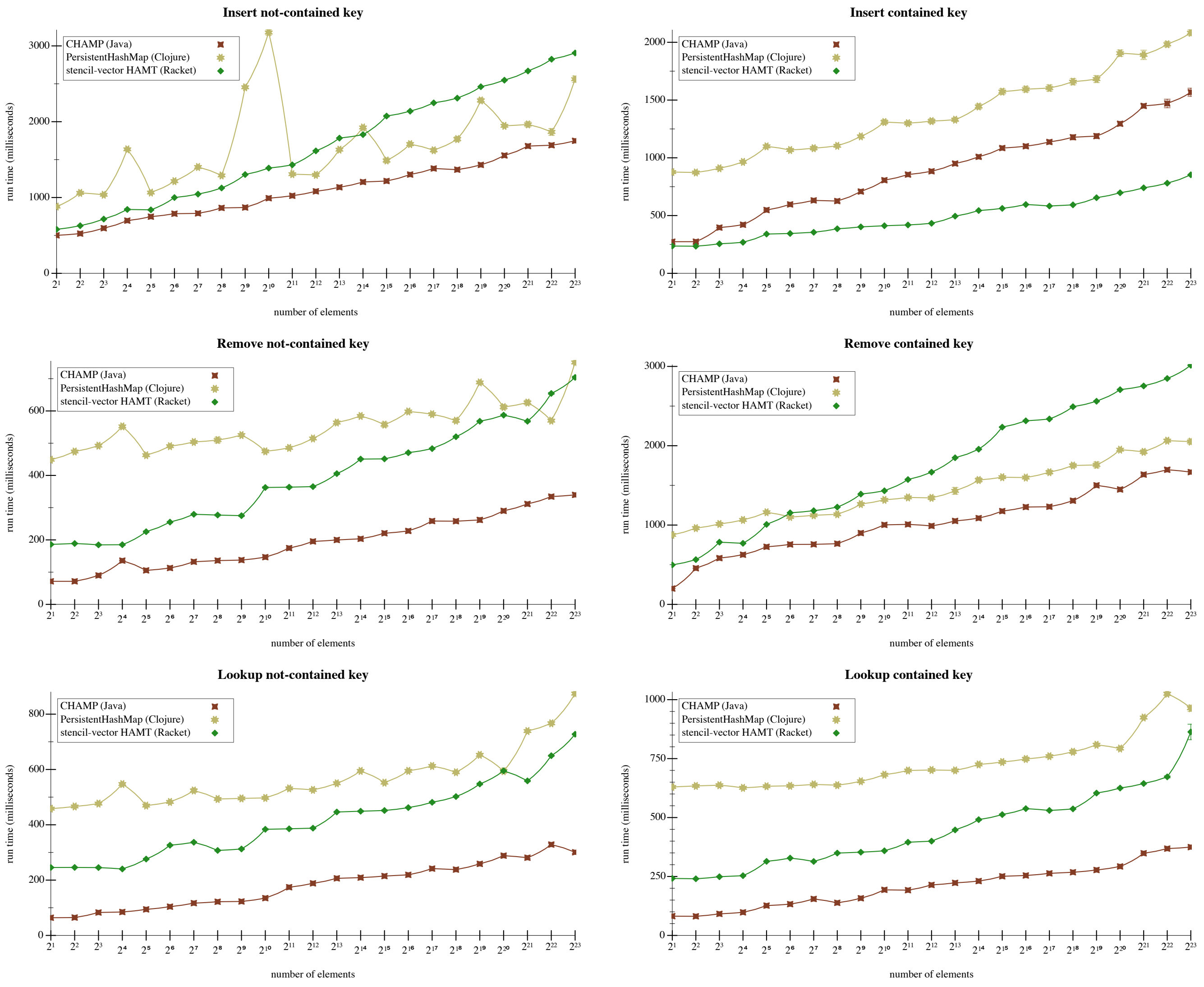

Figure 3. Run-time performance of Racket HAMTs, CHAMP, and Clojure's PersistentHashMap; lower is better

the value in a cons cell. To focus on HAMT-representation performance and not hash-function performance, we consider only eq?-based tables with keys mapped to values.

In absolute terms, our benchmarks will show that the five persistent-map implementations for Racket all perform similarly, at least for small maps. For large maps, Patricia tries tend to be slower due to their less-compact representation. Stencil-vector HAMTs are the most compact. Stencil-vector maps end up with run-time performance about on par with other approaches while being the most compact; that combination has made them the best choice for Racket overall.

7.2 Memory Usage

Figure 4 (a) shows how the different layout strategies affect memory use for maps. The reported sizes are for 1000 maps, each with a different set of 200 random keys. All keys and values are fixnums, so the total payload size is 1000×200×2×8 bytes, which is 3.05 MB; the unshaded portion of each plot shows that payload size (the same across implementations), while the shaded portion shows overhead beyond that payload size. The **stencil-vector HAMT** implementation uses 70% less space than the **Patricia trie** implementation and 25% less space than the **vector HAMT** implementation. Using a mask in place of a separate vector length saves 7% compared to **extra-tag stencil-vector HAMT**.

Figure 4 (b) and (c) show different ways of measuring how the implementation choice for maps affects overall memory use in applications. In (b), we measured the peak memory use encountered while loading the standard Racket library from

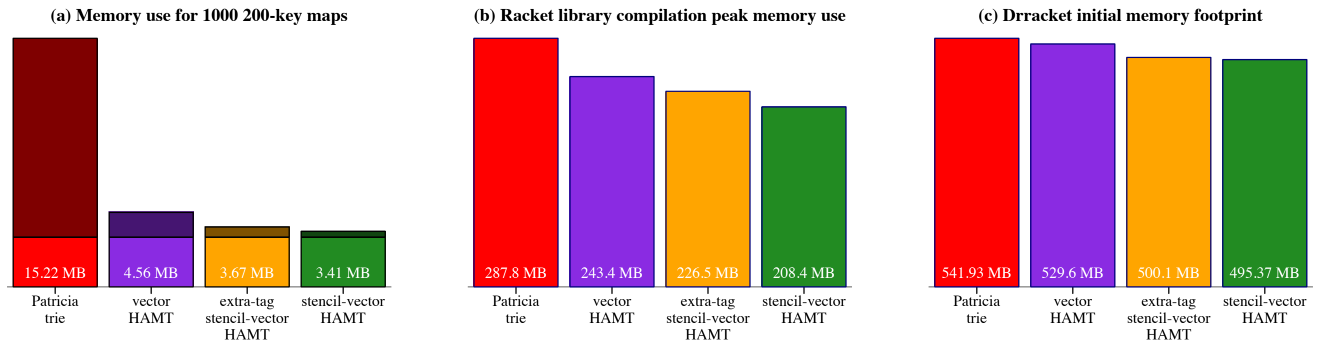

Figure 4. Memory-use benachmark and application results; each plot is on a different scale, and lower is better

source,[8] where roughly 40% of the time is macro expansion, and macro expansion involves many maps used to represent scope sets [4]. Figure 4 (c) shows the initial memory footprint of DrRacket,[9] which is the development environment for Racket that is itself implemented in Racket. DrRacket uses maps for many purposes, including representing syntax objects in macros loaded from the standard library. Overall, the differences in memory use within (b) and (c) are smaller than in (a), because they measure overall application memory use that includes data not related to maps, but the difference is still measurable and significant.

For loading the Racket library (b), avoiding an extra tag word in stencil-vector HAMT nodes matters more compared to other cases, because macro expansion tends to create and retain many 1- and 2-key maps. The memory use of the **stencil-vector HAMT** implementaiton is almost 10% lower than the **extra-tag stencil-vector HAMT** version, while being 25% lower than the **Patricia trie** version. A 10% savings in peak memory use becomes even more significant when scaling up to building a full Racket distribution, where complex macros and layers of languages push overall peak memory use to around 1GB.

For DrRacket (c), the **Patricia trie** implementation uses about 45MB more memory than **stencil-vector HAMT**, and **vector HAMT** uses about 35MB more memory. The **Patricia trie** and **vector HAMT** memory uses are more similar to each other than **stencil-vector HAMT** because, again, most maps in DrRacket are small, but there are enough small maps to have a significant effect on DrRacket's overall memory use. Using a mask in place of a separate vector length saves about 5MB compared to **extra-tag stencil-vector HAMT**.

[8]`racket -W debug@GC:major -cl racket`, where `-W debug@GC:major` reports peak and total allocation on exit, and `-cl` loads the library named after the flag from source.

[9]As reported by `(dump-memory-stats)` in the interactions window after starting DrRacket with `racket/base` as the default language, pausing long enough for the documentation link to appear before hitting Enter.

7.3 Run-Time Performance

To check the run-time performance of different persistent-map implementations, we use benchmarks that measure individual operations plus the Racket macro expander and a compiler pass as applications.

7.3.1 Benchmark Programs.

The benchmark programs measure the run-time performance for insertion, deletion, and lookup operations on maps of sizes 2^x, $x \in \{1, ..., 23\}$. The setup step for each benchmark includes filling the map with randomly generated numbers as keys, then invoking each operation with 8 random keys. All keys are constrained to Racket's fixnum range, and each key is mapped to itself as a value. All maps use pointer equality as the comparison function. (Racket's hashing function for `eq?` treats a fixnum as its own hash code.) Two kinds of arguments are considered and measured separately: keys that are contained in the map (already with the same value, in the case of insertion), and keys that are not in the map.

We repeat each operation 2 million times on each of 8 arguments for a single benchmark run. We perform each run 20 times and report the average of the measurements; the plots include bars to show the standard error, but usually the error bars are much smaller than the shape used to represent a measurement. On Racket, we prefix each run with a garbage collection, use the `time` form around the run, and report real time. For Java and Clojure, we use the Java Microbenchmarking Harness (JMH) configured to run 5 warm-up iterations and 20 measurement iterations in "Average Time" mode, and we also configure it to run the garbage collector before each invocation. We measured performance using Racket v8.0.0.3 and macOS Big Sur (version 11.2.3) on a 2.4GHz 8-Core Intel Core i9 processor with 32GB RAM. Popcount operations use the processor's `POPCNT` instruction.

7.3.2 Benchmark Results.

Figure 5 shows benchmark results for Racket map implementations. The **Patricia trie** results are sometimes truncated to avoid scaling the plot so that the other results collapse together. Generally, performance is similar across all implementations for small maps. However, other applications take advantage of larger maps, and even the macro expander can create large scope sets in certain unusual programs. Thus, the overall best result for Racket is to minimize memory use as long as performance does not suffer, and the **stencil-vector HAMT** implementation achieves that overall goal.

The **Patricia trie** implementation performs well when inserting a new key or removing an existing key from a map. Lookup is slower for large trees, primarily because a small branching factor compared to HAMTs makes the tree deeper. Inserting an already contained key and removing a not-contained key is slower for the same reason: the tree must be explored to more depth to discover that no change is needed, and then the absence of change must be detected on each step as recursion unwinds back through the explored spine.[10]

The **vector HAMT** implementation performs well for operations that do not change the map, including lookup, attempting to insert a key that is already in the map, or attempting to remove a key that is not in the map. The implementation is around 30% slower than the **stencil-vector HAMT** implementation (which takes advantage of atomic update to avoid a write barrier), while it performs similarly to the **write-barrier stencil-vector HAMT** implementation. For larger maps, the **vector HAMT** implementation performs better than **write-barrier stencil-vector HAMT** by avoiding some popcount steps and other bitwise operations. Similarly, the **vector HAMT** implementation is the fastest for lookup by a small margin, likely again by avoiding a small amount of arithmetic.

The **stencil-vector HAMT** implementation finds a happy medium. Whether for operations that modify the map or leave it unchanged, performance is very close to the fastest implementation, although it is rarely the fastest. This performance, combined with the fact that **stencil-vector HAMT** maps are the most compact, justifies the choice to use this implementation in the current version of Racket.

The **extra-tag stencil-vector HAMT** implementation results show that **stencil-vector HAMT** pays a price for compactness. The **extra-tag stencil-vector HAMT** implementation is faster by about 5%, particularly for operations that change the tree. The faster performance is related to the fact that fewer popcount operations are needed for allocating new nodes and for garbage-collecting live nodes. Even for operations that do not change the map, the mask field of a node in the **extra-tag stencil-vector HAMT** variant can be accessed without bitwise arithmetic to extract the mask from a tag word, which is why lookup performance is also faster.

The **write-barrier stencil-vector HAMT** implementation demonstrates the benefit of specializing atomic, functional updates to avoid write-barrier machinery, as it performs substantially worse than **stencil-vector HAMT** for operations that change the map. Avoiding a write barrier in the latter implementation improves run-time performance by about 30%. For operations that do not change the map, the implementations are the same, and they perform the same.

7.3.3 Application Results.

Figure 6 shows how the different persistent-map implementations affect different parts of Racket's performance (i.e., treating Racket's implementation as an application of persistent maps). Each plot shows the average of five runs, and the shaded portion at the left of each bar shows how much of the time can be attributed to garbage collection.

One of the most performance-sensitive uses of persistent maps within Racket is the macro expander, which uses maps to represent scope sets for hygienic expansion. Figure 6 (a) shows the time required to load Racket's large standard library from source—the same task as Figure 4 (b), where roughly 40% of the time to load involves macro expansion. Loading from source performs about 16 million insert operations on persistent maps, 3.4 million remove operations, and 53 million lookup operations, mostly as part of the macro expander. (Loading the same library in its compiled form performs only 64 thousand, 600, and 178 thousand operations, respectively.) Persistent maps are used for various purposes in the expander and compiler, but the vast majority of persistent-map operations while loading from source are applied to small scope sets.

The **stencil-vector HAMT** implementation performs the best overall for macro expansion. The variation in times for different persistent-map implementations is about 1.5 seconds, which is in line with the variation in benchmark performance for individual operations in figure 5: the benchmark variation for small tables tends to be around 0.2-0.5 seconds, and the application involves 4.5 times as many operations as one benchmark run. The garbage-collection time for macro expansion *increases* with compact stencil-vector HAMTs compared to other implementations, which is counter-intutive; it turns out that maintaining a smaller memory footprint triggers an extra major collection—5 major collections instead of 4 for the other implementations—but each per-collection time is smaller.

Figure 6 (b) shows the time required for a "schemify" step in the Racket compiler, which converts from Racket's core language to Chez Scheme. For this measurement, the schemify step is repeated 100 times on the (already expanded) implementation of the Racket macro expander itself. Schemify uses a persistent map to represent the lexical binding environment, which is a mapping from symbols to compile-time

[10]For the purposes of benchmarking, we adjusted the original Patricia trie implementation in Racket to avoid reallocating spine nodes when an operation does not change a tree. That adjustment makes insertion and deletion operations that change the tree around 15% slower, while it make operations that do not change a tree faster by 25-30%.

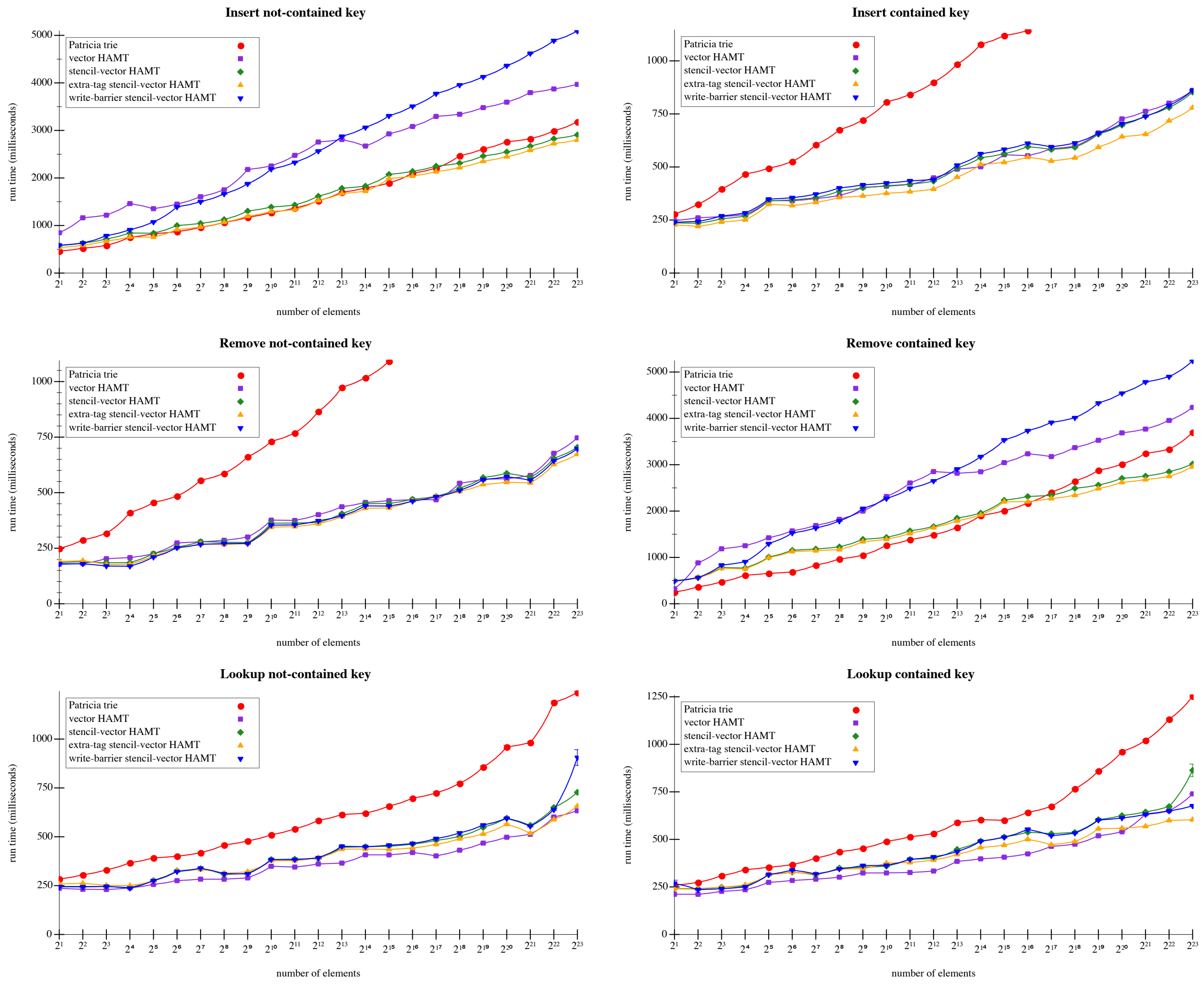

Figure 5. Run-time performance of Racket map implementations on 8×2 million operations; lower is better

information that is extended as the compiler descends into subexpressions that bind variables. Schemify also uses a persistent map for a table describing about 1000 primitive functions (although there is no particular advantage to persistence in that case, since the set of primitive functions does not change). Running schemify 100 times on the expander implementation performs about 1.5 million insert operations and 67 million lookup operations on persistent maps (with no remove operations).

The **stencil-vector HAMT** implementation performs the best overall for schemify. Compared to the expander, schemify performs many more lookup than insert operations. The results in figure 5 suggest that **vector HAMT** should therefore perform well, while **Patricia trie** will be slower, and that is indeed the result shown in Figure 6 (b). Nevertheless, the number of insertions combined with the garbage-collection differences still gives **stencil-vector HAMT** an edge over **vector HAMT** in this application.

8 Conclusion

Our experiments with persistent-map implementations for Racket confirm previous work in favor of using HAMTs. Furthermore, we have shown that *stencil vectors* are a useful primitive construct to improve the memory and run-time performance of HAMTs through a modest and maintainable investment at the level of the compiler and runtime system. Our measurements are specific to Racket, but other dynamic-language implementations use a similar strategy

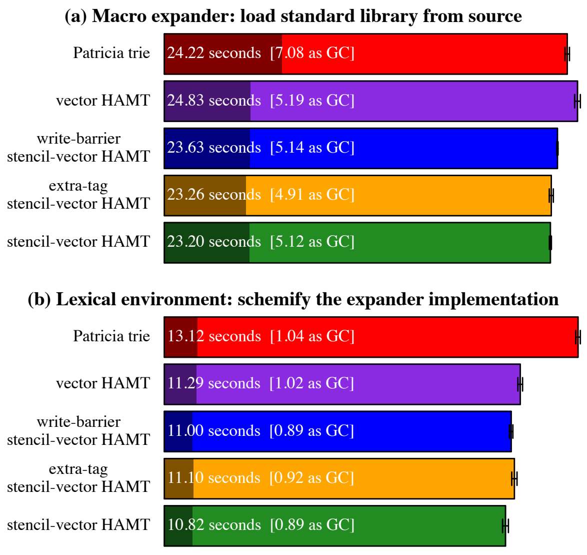

Figure 6. Performance of applications; lower is better

for representing values, and we expect that adding a stencil-vector datatype to those implementations would provide similar benefits.

Acknowledgments

Thank you to the anonymous reviewers for their feedback and suggestions on the paper's organization. Thanks also to Michael Steindorfer for further corrections and suggestions. This work was supported by the National Science Foundation.

References

[1] Phil Bagwell. Ideal Hash Trees. École polytechnique fédérale de Lausanne, LAMP-REPORT-2001-001, 2001.

[2] Rene De La Briandais. File Searching Using Variable Length Keys. In *Proc. Western Joint Computer Conference*, pp. 295–298, 1959. https://doi.org/10.1145/1457838.1457895

[3] R. Kent Dybvig, David Eby, and Carl Bruggeman. Don't Stop the BIBOP: Flexible and Efficient Storage Management for Dynamically Typed Languages. Indiana University Computer Science Department, 400, 1994.

[4] Matthew Flatt. Binding as Sets of Scopes. In *Proc. Principles of Programming Languages*, 2016. https://doi.org/10.1145/2837614.2837620

[5] Matthew Flatt, Caner Derici, R. Kent Dybvig, Andrew W. Keep, Gustavo E. Massaccesi, Sarah Spall, Sam Tobin-Hochstadt, and Jon Zeppieri. Rebuilding Racket on Chez Scheme (Experience Report). In *Proc. International Conference on Functional Programming*, 2019. https://doi.org/10.1145/3341642

[6] Edward Fredkin. Trie Memory. *Communications of the ACM* 3(9), pp. 490–499, 1960. https://doi.org/10.1145/367390.367400

[7] Rich Hickey. A History of Clojure. In *Proc. History of Programming Languages*, 2020. https://doi.org/10.1145/3386321

[8] Witold Litwin, Marie-Anne Neimat, and Donovan A Schneider. LH: Linear Hashing for Distributed Files. In *Proc. International Conference on Management of Data*, pp. 327–336, 1993. https://doi.org/10.1145/170036.170084

[9] Donald Morrison R. PATRICIA—practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM* 15, pp. 514–534, 1968. https://doi.org/10.1145/321479.321481

[10] Chris Okasaki and Andy Gill. Fast Mergeable Integer Maps. In *Proc. Workshop on ML*, pp. 77–86, 1998.

[11] John H. Reppy. A High-Performance Garbage Collector for Standard ML. Technical memorandum, AT&T Bell Laboratories, 1993.

[12] Michael J Steindorfer and Jurgen J Vinju. Optimizing Hash-Array Mapped Tries for Fast and Lean Immutable JVM Collections. In *Proc. International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 783–800, 2015. https://doi.org/10.1145/2814270.2814312

[13] Michael J Steindorfer and Jurgen J Vinju. To-Many or To-One? All-in-One!: Efficient Purely Functional Multi-maps with Type-Heterogeneous Hash-Tries. In *Proc. Programming Language Design and Implementation*, 2018. https://doi.org/10.1145/3192366.3192420

[14] Michael J Steindorfer and Jurgen J Vinju. Towards a Software Product Line of Trie-Based Collections. In *Proc. Generative Programming: Concepts and Engineering*, 2018. https://doi.org/10.1145/2993236.2993251

[15] Johan Tibell. Faster Persistent Data Structures Through Hashing. 2011. https://johantibell.com/files/galois-2011.pdf