Compiler and Runtime Support for HAMTS

Sona Torosyan
University of Utah

UUCS-21-003

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

19 April 2021

Abstract

Many immutable collection data structures of functional programming languages including Racket, Scala, and Clojure are implemented as Hash Array Mapped Tries (HAMTs). This data structure provides efficient lookup, insertion, and deletion operations and has a small memory footprint. Various design changes have been implemented since the first introduction of HAMTs that further improve memory footprint and runtime performance. However, these HAMT implementations still keep redundant data in the trie node and do not fully address the cost of cooperating with garbage collection when initializing or updating nodes. A stencil vector is a new data structure built into Racket's compiler and runtime system. Its as an intermediate field of HAMTs results in better performance and smaller memory use for persistent sets and maps compared to previous implementations.

COMPILER AND RUNTIME SUPPORT FOR HAMTS

by

Sona Torosyan

A Senior Thesis Submitted to the Faculty of
The University of Utah
In Partial Fulfillment of the Requirements for the Degree

Bachelor of Computer Science

School of Computing
The University of Utah
April 2021

Approved:

19-APR-2021

Matthew Flatt, PhD
Thesis Faculty Supervisor

/

H. James de St. Germain, PhD
Director of Undergraduate Studies
School of Computing

/

Mary Hall, PhD
Director
School of Computing

Abstract

Many immutable collection data structures of functional programming languages including Racket, Scala, and Clojure are implemented as Hash Array Mapped Tries (HAMTs). This data structure provides efficient lookup, insertion, and deletion operations and has a small memory footprint. Various design changes have been implemented since the first introduction of HAMTs that further improve memory footprint and runtime performance. However, these HAMT implementations still keep redundant data in the trie node and do not fully address the cost of cooperating with garbage collection when initializing or updating nodes. A stencil vector is a new data structure built into Racket's compiler and runtime system. Its use as an intermediate field of HAMTs results in better performance and smaller memory use for persistent sets and maps compared to previous implementations.

Contents

List of Figures **5**

List of Tables **6**

1 Introduction **7**

2 Background **9**

2.1 Previous Work 9

2.2 HAMT Implementation 10

3 Stencil Vectors **12**

3.1 Implementation 12

3.2 Stencil Vector Functions 17

4 Performance and Memory Benchmarks **19**

4.1 Memory Benchmarks 20

4.1.1 Methods 20

4.1.2 Memory Benchmarking Results 20

4.1.3 Real World Memory Measurement Experiment 21

4.2 Performance Benchmarks 21

4.2.1 Methods 21

4.2.2 Performance Benchmarking Results 22

5 Conclusion **33**

6 Acknowledgements **33**

References **34**

Appendices **35**

A Performance Benchmarks in Racket **35**

B Performance Differences of Patricia Trie Implementations and Stencil Vector–Based HAMTs **39**

List of Figures

1 Insertion of A, B, C, and D objects into a HAMT [8] 10

2 Memory diagram for a fixnum 12

3 Memory diagram for (a) pair and (b) vector 13

4 Memory diagram for a stencil vector 15

5 Performance benchmarking results of persistent map implementations in Racket for insertion operation 23

6 Performance benchmarking results of persistent map implementations in Racket for deletion operation 24

7 Performance benchmarking results of persistent map implementations in Racket for lookup operation 25

8 Runtime performance of regular stencil vectors compared to stencil vectors with write barriers and non-compact stencil vectors . 28

9 Performance benchmarking results of stencil vector–based HAMT, CHAMP, and Clojure's PersistentHashMap for insertion operation 30

10 Performance benchmarking results of stencil vector–based HAMT, CHAMP, and Clojure's PersistentHashMap for deletion operation 31

11 Performance benchmarking results of stencil vector–based HAMT, CHAMP, and Clojure's PersistentHashMap for lookup operation 32

12 Performance benchmarking results of stencil vector–based HAMT, Patricia trie, and updated Patricia trie with reuse check for insertion operation 39

13 Performance benchmarking results of stencil vector–based HAMT, Patricia trie, and updated Patricia trie with reuse check for deletion operation 40

List of Tables

1 Example hash codes of A, B, C, and D objects 10

2 Memory benchmarking results for persistent map implementations in Racket 20

3 DrRacket memory use differences based on persistent map implementation 21

1 Introduction

Almost every software application highly depends on some type of collection data structure for the implementation of its core functionality; therefore, the efficiency of various applications depends on the design and performance of the underlying data structures that they use. This makes the research aimed to understand and improve the implementations of collection data structures important.

In many programming languages, including Racket, Haskell, JVM-based languages such as Scala, Clojure, and functional constructs of Java, HAMTs are widely used for implementing persistent collection data structures such as sets and maps. This efficient, trie-based data structure was first proposed and implemented in C++ by Bagwell [1]. One of the main advantages of a HAMT over an array-based encoding is that it avoids table resizing and null references while keeping the trie node representation small. Since the initial introduction of HAMTs, many changes have been proposed and implemented that reduce memory footprint or improve the runtime performance of the overall data structure. However, even with these implementation improvements, there is still redundant data stored in the trie node that increases overall memory. Furthermore, runtime performance can be improved by considering how and when garbage collection is invoked during various HAMT update operations.

A stencil vector is a new data structure implemented in Racket as an intermediate part of HAMTs. The small implementation design of a stencil vector allows building it into the compiler and runtime system as HAMT support, thus, leaving more complicated details of HAMT implementation outside of the compiler. Its use as a HAMT node results in a smaller memory footprint and more efficient runtime performance for persistent sets and maps compared to other implementations.

This new data structure is at the implementation stage and has not been sufficiently researched yet. This thesis describes the implementation details of stencil vectors and their use as HAMT nodes. Additionally, it provides performance and memory evaluation for persistent maps represented as stencil vector–based HAMTs compared to other map representations including Patricia tries and other HAMT implementations.

2 Background

2.1 Previous Work

A trie is a tree data structure for strings where nodes of the tree are for shared prefixes instead of the entire string keys, the edges represent the characters of the string, and the position of the node is used to identify the associated key. This data structure was first implemented by Briandais [2] and named by Fredkin[3]. For the HAMT data structure, which is built based on the idea of tries, the bits of the hash codes of elements serve as the strings.

In 2001, Bagwell[1] combined partitioning based on the hash code with the main principles of Linear Hash presented by Litwin, Neimat, and Schneider [5] that solves collision management and storage growth. The original implementation by Bagwell has a smaller memory footprint than other tries and guarantees an upper bound of $O(log_{32}(n))$ for lookup, insertion, and deletion.

Bagwell's mutable HAMT implementation was later used to implement functional immutable HAMT by Rich Hickey, the lead-developer of Clojure [4]. There are two possible memory layout choices for HAMT. One of the approaches comes from Bagwell's original proposal. In this design, used in Clojure, the values of the nodes are included in the node itself, while with the second approach, used in Scala, the value is stored in a leaf node.

One of the recent improvements for HAMT implementations on JVM is proposed by Steindorfer and Vinju [8]. The authors call the new data structure Compressed Hash-Array Mapped Prefix-tree (CHAMP). It improves locality and makes sure the tree remains in canonical and compact representation form after deletion. Compared to HAMTs of Scala and Clojure, it has a smaller memory footprint due to a compact data layout for internal trie nodes. CHAMP also increases locality by reordering the references in a trie node at the cost of more bit arithmetic and reduces memory by avoiding empty slots in the array.

2.2 HAMT Implementation

In a typical implementation of a HAMT, each entry of a table is either a key-value pair or a node object that can hold references to a certain number of child nodes. The number of possible child nodes depends on a branching factor which is usually 32. Figure 1 shows how an empty HAMT with branching factor 32 changes with the insertion of four different objects A, B, C, and D [8]. Sample hash codes of these objects are provided in Table 1.

Object	Hash code in decimal	Hash code in binary	Hash code in base 32
A	32	0...00000.00010.00000	010...
B	2	0...00000.00000.00010	200...
C	4098	0...00100.00000.00010	204...
D	34	0...00000.00001.00010	210...

Table 1: Example hash codes of A, B, C, and D objects

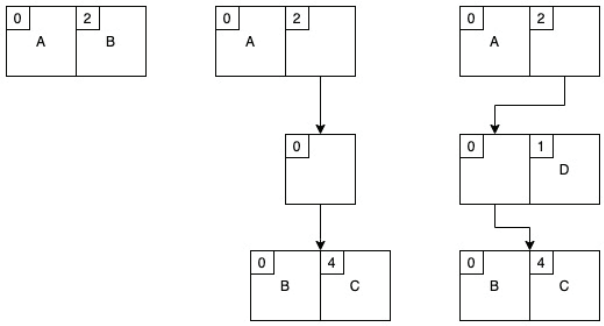

Figure 1: Insertion of A, B, C, and D objects into a HAMT [8]

Instead of storing null pointers for non-existing child nodes, the node object has a 32-bit bitmap where each set bit corresponds to an existing child node or a key-value pair. In Figure 1, the position of each node in the bitmap is marked by the numbers in the top left. The use of a bitmap reduces the space for a non-existing entry to a single bit. Along with the bitmap, the node object has an untyped array whose length equals the number of set bits in the bitmap and whose elements point to either subtries or key-value pairs. To find the

associated element that a bit represents in the array, a mapping function is implemented that involves counting the number of set bits in the bitmap [1]. In some implementations, such as C++, the least significant bit of the pointer is used as a bit flag to identify whether the node is a subtrie or a key-value pair. Other choices of differentiating between sub-nodes and key-value pairs include dynamic checks such as the *instanceof* operator in Java [1, 8].

If a HAMT is used to represent a map whose keys have associated internal values, the size of the array is doubled, and the values are stored next to the keys. However, this implementation leads to empty slots next to subtrie references in the array. C/C++ implementations of HAMTs eliminate this wasted space by using union types. For JVM languages, this issue is fixed by introducing a second bitmap and grouping the elements of the array into key-value pairs and subtrie references. This implementation improvement also eliminates the use of dynamic checks, such as *instanceof* operator, for identifying subtries and key-value pairs [8].

Additionally, some implementations of HAMT store values in leaf nodes as opposed to storing them next to subtrie nodes internally. While this approach increases the memory used by the data structure, it allows storing additional information in the nodes, such as memoized hash codes of elements used for update operations. This design results in better runtime performance, especially for operations that would otherwise recalculate the hash codes.

3 Stencil Vectors

3.1 Implementation

Overall, the node representation remains the largest factor affecting the memory footprint of the HAMT data structure; therefore, finding a compact node representation is important. Racket's HAMT implementation uses stencil vectors for representing nodes. This data structure is implemented in Chez Scheme, a Scheme variant used to build the most recent Racket compiler and Racket's runtime system. Using stencil vectors as HAMT nodes improves memory usage and overall performance of the data structure.

In Chez Scheme, the type of an object is identified through the lower bits of the pointer, which are otherwise wasted due to allocation alignment. For example, on a 64-bit platform, the lower 3 bits of a pointer are used to indicate whether it refers to one of the most common objects such as fixnums, pairs, and symbols, or a general object. For a fixnum, which is an exact integer that does not require allocation for computations in Racket, the rest of the bits are used to store the value, as shown in Figure 2. For most of the other objects, the rest of the bits store the object address.

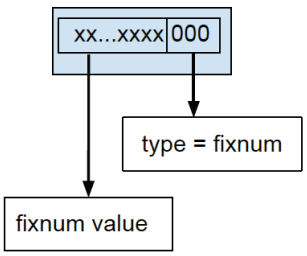

Figure 2: Memory diagram for a fixnum

All other general types of objects share a single bit pattern for the low bits,

and their types are further refined by using a tag word. The tag words are stored at the start of the object to which a pointer refers.

Figure 3 shows memory diagrams for a pair and a regular vector on a 64-bit platform. The pair is identified by the low bits *001* of a pointer, and the content is stored at the next 8-byte aligned address. For the vector, the lowest 3 bits are *111* indicating a general typed object whose type is determined by a tag word.

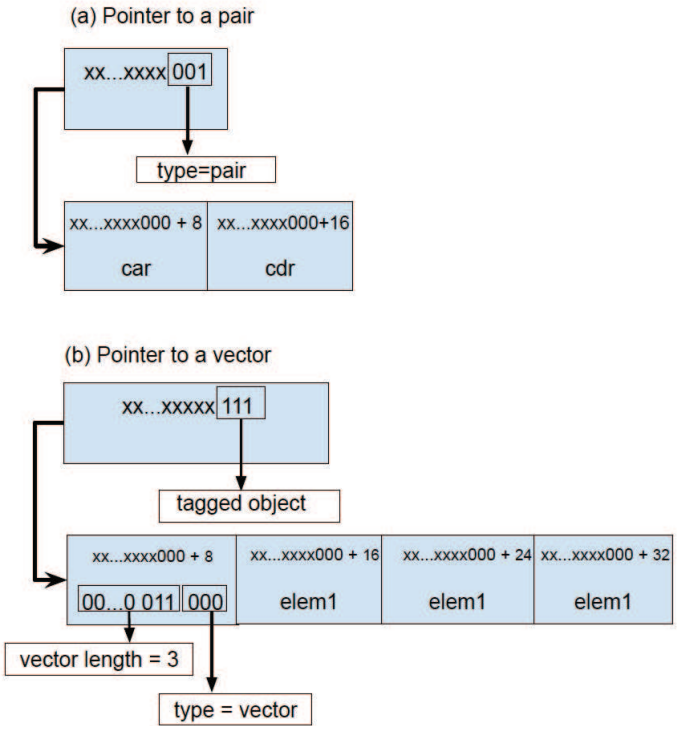

Figure 3: Memory diagram for (a) pair and (b) vector

As the lower bits of pointers identify the types of common objects, the lower bits of a tag word are used to identify the type of a general object. However, in contrast to pointers that only use the three lowest bits for identifying the type, the number of bits used for differentiating the type in a tag word varies. Furthermore, the rest of the bits in a tag word can be used to store more

information about the object such as its length. For example, if the low bits of a tag identify an object as a string or a vector, the rest of the bits are used to store the object length. In Figure 3, the tag word of a vector is stored at the next 8-byte aligned address. Its lowest 3 bits *000* indicate a vector, while the rest of the bits store the length of the vector. This information is then used in functions returning the length of an object and garbage collection.

A stencil vector combines a vector and a mask in a compact representation. Similar to regular vectors, stencil vectors are Scheme objects that are identified based on the low bits of a tag word. In addition, the tag word for this object type stores the mask for the vector which is also used to calculate the vector length. Figure 4 shows a memory diagram for a stencil vector. The lowest 3 bits of the pointer *111* indicate a general object, and the type is determined through the tag word. The low bits *011110* in the tag indicate a stencil vector, while the rest of the bits are used for the mask of the stencil vector. The number of set bits in the bitmask indicates the number of elements in the vector. In the example given in Figure 4, the mask has two set bits which means the length of the vector is two.

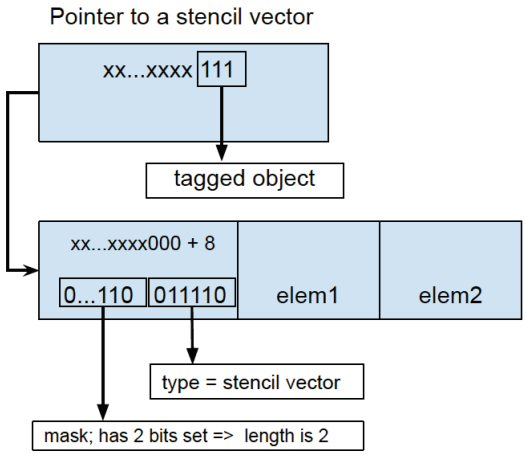

Figure 4: Memory diagram for a stencil vector

When a stencil vector is used to represent a HAMT node, the first elements are for any child nodes, followed by keys and values.

In addition to supporting a compact representation, a stencil vector's limited size and bitmask support a natural API for a functional update, which cooperates better with the garbage collector than allocating a vector and mutating its individual elements. This efficient functional update allows avoiding the need for a write barrier. In generational garbage collection, the write barrier is used to indicate if there are any objects in the old generation with references to objects in the new generation. This information is then used to prevent garbage collection for objects that have references from the old generation memory. In general, for immutable objects, the overhead of the garbage collection is decreased, since mutable objects increase the chance of having references from objects in the older generation memory to the younger one.

For stencil vectors, the tag word used to identify its type is also used to determine the vector length. The number of set bits in the mask, which is stored in the tag word, is the length of the stencil vector. This additional

information about the object is used by the garbage collector during traversal. In addition, the update operation of a stencil vector does not cause references from the old generation memory to the new one, which helps to avoid a write barrier.

3.2 Stencil Vector Functions

Stencil vectors are vectors that use a mask fixnum to determine their size. The length of a stencil vector, retrieved through *(stencil-vector-length v)* function where v is a stencil vector, is limited to the number of bits in a fixnum whose two's complement representation fits into 29-30 or 60-62 bits plus a sign bit depending on the architecture. The vector length limitation is due to the possible number of bits in a stencil vector mask retrieved through *(stencil-vector-mask-width)* which cannot be more than the number of bits in a fixnum.

To create a stencil vector of a given length and content the function *(stencil-vector mask value ...)* can be called. Here, *mask* must be a non-negative fixnum and its number of set bits must be equal to the number of values. Additionally, the number of bits must be no more than *stencil-vector-mask-width*.

For an existing stencil vector, we can get its mask through *(stencil-vector-mask v)* and its values through *(stencil-vector-ref v n)* where v is the vector and n is the index of the object to return. n must be a non-negative fixnum less than the length of vector v. While the number of valid positions in a stencil vector is determined by its mask, and each element in the vector has its corresponding bit in the mask, position n in *stencil-vector-ref* is the position in the vector itself and not in the mask. A bit position can be converted to an index using *(fxpopcount (fxand (stencil-vector-mask v) (fx- bit 1)))* calculation where v is the vector and *bit* is the position in the mask.

Listing 1 provides an example of how these functions can be called. Line 1 illustrates creating a vector $v1$ with mask 1011 and three elements a, b, and c. The following lines show how to retrieve its length (line 2) and mask (line 3). Finally, lines 4-6 show retrieving the elements based on their positions in the vector.

```
1 (define v1 (stencil-vector #b1011 'a 'b 'c))
2 (stencil-vector-length v1) ; => 3
3 (stencil-vector-mask v1)   ; => #b1011
4 (stencil-vector-ref v1 0)  ; => 'a
5 (stencil-vector-ref v1 1)  ; => 'b
6 (stencil-vector-ref v1 2)  ; => 'c
```

Listing 1: Creating a stencil vector with three elements

A stencil vector can be updated using *(stencil-vector-update v remove-bits add-bits values ...)* function where v is the vector to be updated. The update function returns a new stencil vector that contains all elements from v in their relative positions except those identified by *remove-bits* fixnum. *remove-bits* must be a subset of the mask of v for the operation to be successful. In addition, the new vector contains the new values at positions determined by *add-bits* fixnum which must not overlap the subtraction of *remove-bits* from the mask of v. Also, the number of provided values must match the number of set bits in *add-bits*. The mask of this new vector is the mask of v minus *remove-bits* plus *add-bits*.

Listing 2 shows an example of how *stencil-vector-update* function can be used. The update function in line 2 creates a new vector by removing the second element of vector $v1$ defined in line 1 and adding a new element x. The updated vector's mask is $1011 - 0010 + 0100 = 1101$ (line 3). Lines 4-6 show the elements of the updated vector.

```
1 (define v1 (stencil-vector #b1011 'a 'b 'c))
2 (define v2 (stencil-vector-update v1 #b0010 #b0100 'x))
3 (stencil-vector-mask v2)   ; => #b1101
4 (stencil-vector-ref v2 0)  ; => 'a
5 (stencil-vector-ref v2 1)  ; => 'x
6 (stencil-vector-ref v2 2)  ; => 'c
```

Listing 2: Updating a stencil vector

4 Performance and Memory Benchmarks

Our benchmarks show that HAMTs based on stencil vectors perform the best on average and have the smallest memory footprint among the implementations that we have tried for persistent maps in Racket, including Patricia tries, regular vector–based HAMTs, and HAMTs based on two different variants of stencil vectors described below. We implemented these stencil vector variants to evaluate the effects of avoiding write barriers during functional update and compact representation of stencil vectors.

In order to demonstrate the effects of stencil vector's compact representation on performance and memory, we implemented a version of stencil vectors that uses the rightmost bits of the tag word to store the length of the vector, while the bitmask is stored in the first slot of the vector, and the update operation still avoids write barriers. Additionally, to explore the effects of write barriers on runtime performance, we implemented a version of stencil vectors that uses a write barrier when updating the vector and compared its performance to that of stencil vectors which avoid write barriers.

The implementation of hash tables as Patricia tries is similar to Haskell's Data.IntMap implementation of hash tables, which uses integers for keys. This data structure was first implemented by Morrison [6] and then used by Okasaki and Gill [7] to represent a finite map with integer keys.

We also compared stencil vector–based HAMTs to Compressed Hash-Array Mapped Prefix-tree (CHAMP) in Java [8] and PersistentHashMap in Clojure to provide some evidence that the implementation performs comparably to the state of the art.

We used a machine with macOS Big Sur (version 11.2.3), 32GB RAM, and 2.4GHz 8-Core Intel Core i9 processor for all benchmarking experiments.

4.1 Memory Benchmarks

4.1.1 Methods

To evaluate the memory improvements of stencil vector–based HAMTs compared to other HAMT implementations and Patricia tries, we implemented microbenchmarks for measuring the memory footprints of persistent maps. The implementation of memory benchmarks does not involve extra libraries; instead, we used *current-memory-use* and *collect-garbage* functions available in Racket.

The microbenchmarks create an empty array of 1000 elements which is then populated by persistent maps of 200 elements. The memory microbenchmarks are invoked using 4 different implementations of persistent maps: stencil vector–based HAMTs, regular vector–based HAMTs, Patricia tries, and non-compact stencil vector–based HAMTs.

4.1.2 Memory Benchmarking Results

As shown in Table 2, the persistent map implementation with the smallest memory is represented by stencil vector–based HAMTs, followed by non-compact stencil vector–based HAMTs, then regular vectors, and, finally, Patricia tries. Compared to Patricia tries, stencil vector–based HAMTs improve the memory use by around 70%. Additionally, the difference between stencil vector–based HAMTs and the non-compact variant of stencil vectors is about 7%.

Persistent Map Implementation	**Used Memory (MB)**
Stencil vector–based HAMT	3.41
Non-compact stencil vector–based HAMT	3.67
Regular vector–based HAMT	4.55
Patricia Trie	15.20

Table 2: Memory benchmarking results for persistent map implementations in Racket

4.1.3 Real World Memory Measurement Experiment

Additionally, to determine how various implementation choices for persistent maps affect a real world application, we measured the memory use of DrRacket with the abovementioned implementations of persistent maps. The measurement results, which are consistent with the microbenchmarking results, are shown in Table 3. The implementation with stencil vector–based HAMTs results in the smallest memory use, while Patricia trie based implementation uses around 45MB more memory. Additionally, the non-compact version of stencil vector–based HAMTs results in 5MB more memory use compared to regular stencil vector HAMTs.

Persistent Map Implementation	**Total Memory Used by DrRacket (MB)**
Stencil vector–based HAMT	495.37
Non-compact Stencil Vector–based HAMT	500.10
Regular vector–based HAMT	529.60
Patricia Trie	541.93

Table 3: DrRacket memory use differences based on persistent map implementation

4.2 Performance Benchmarks

4.2.1 Methods

We measure the runtime performance for insertion, deletion, and lookup operations on hash tables of sizes $2^x, x \in \{1, ..., 23\}$. This size range was used by Bagwell [1] and Stenidorfer, Vinju [8] to measure the performance of HAMTs. The setup for each benchmark includes filling the collection with randomly generated numbers, then invoking each operation with 8 random parameters to measure the runtime. Two kinds of arguments are considered and measured separately - elements that are already contained in the data structure and elements that are not in the data structure. In order to have numerically comparable timing re-

sults, we repeat each operation 2,000,000 times. For performance measurement, we use Racket's *time* function that collects timing information for a procedure application. The results collected through this function include CPU and real time required to evaluate the operation for given arguments and CPU time spent on garbage collection in milliseconds. We run each benchmark 20 times and consider the average of the measurements for the evaluation. Additionally, we implemented similar benchmarks in Java for CHAMP implementation and Clojure's PersistentHashMap using Java Microbenchmarking Harness (JMH). JMH is configured to run 5 warmup iterations and 20 measurement iterations on Average Time mode, as well as to run garbage collector before each invocation.

The Racket source code of performance benchamrking is provided in Appendix A.

4.2.2 Performance Benchmarking Results

Figures 5, 6, and 7 show the performance microbenchmarking results for insertion, deletion, and lookup operations for 5 different implementations of persistent maps: stencil vector HAMTs, Patricia tries, regular vector HAMTs, HAMTs that use stencil vectors with write barriers as nodes, and HAMTs that use non-compact stencil vectors as nodes. The average relative standard deviation is less than 3% for all persistent map implementations in Racket and less than 1% for JMH benchmarks. The results of the latter are shown separately in Figures 9, 10, and 11.

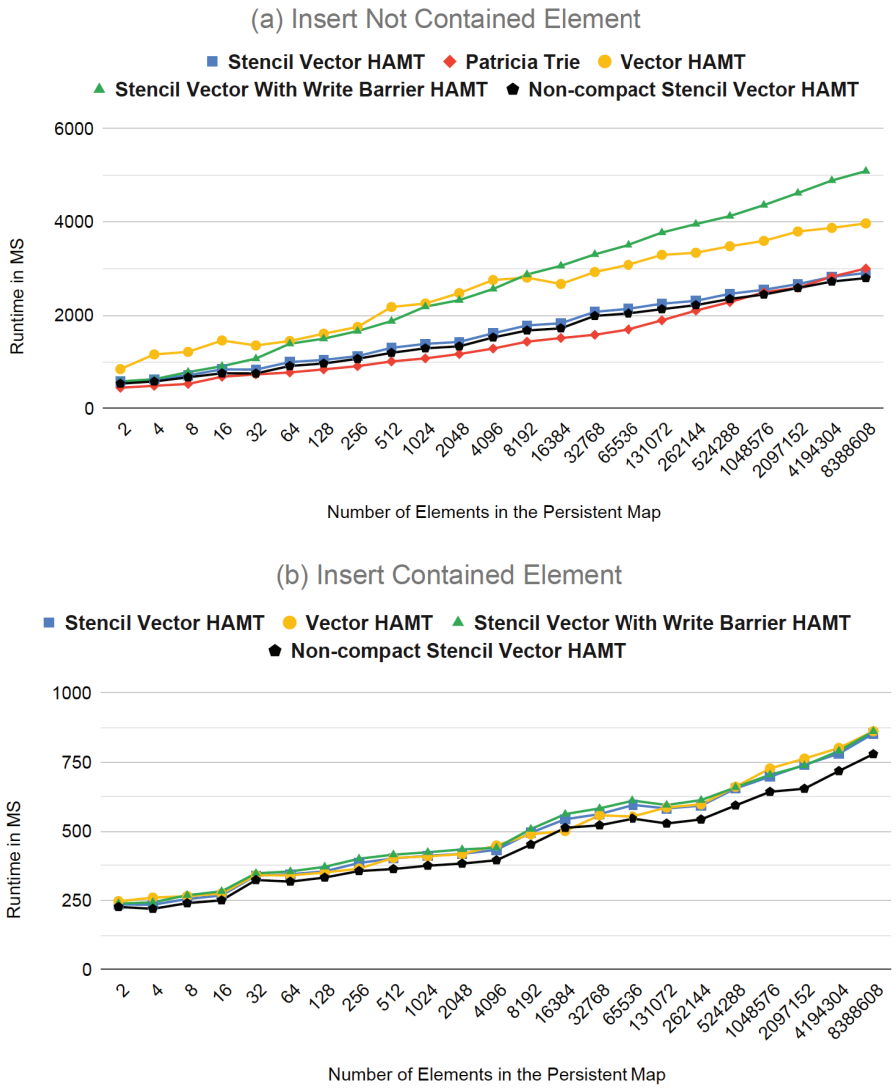

Figure 5: Performance benchmarking results of persistent map implementations in Racket for insertion operation

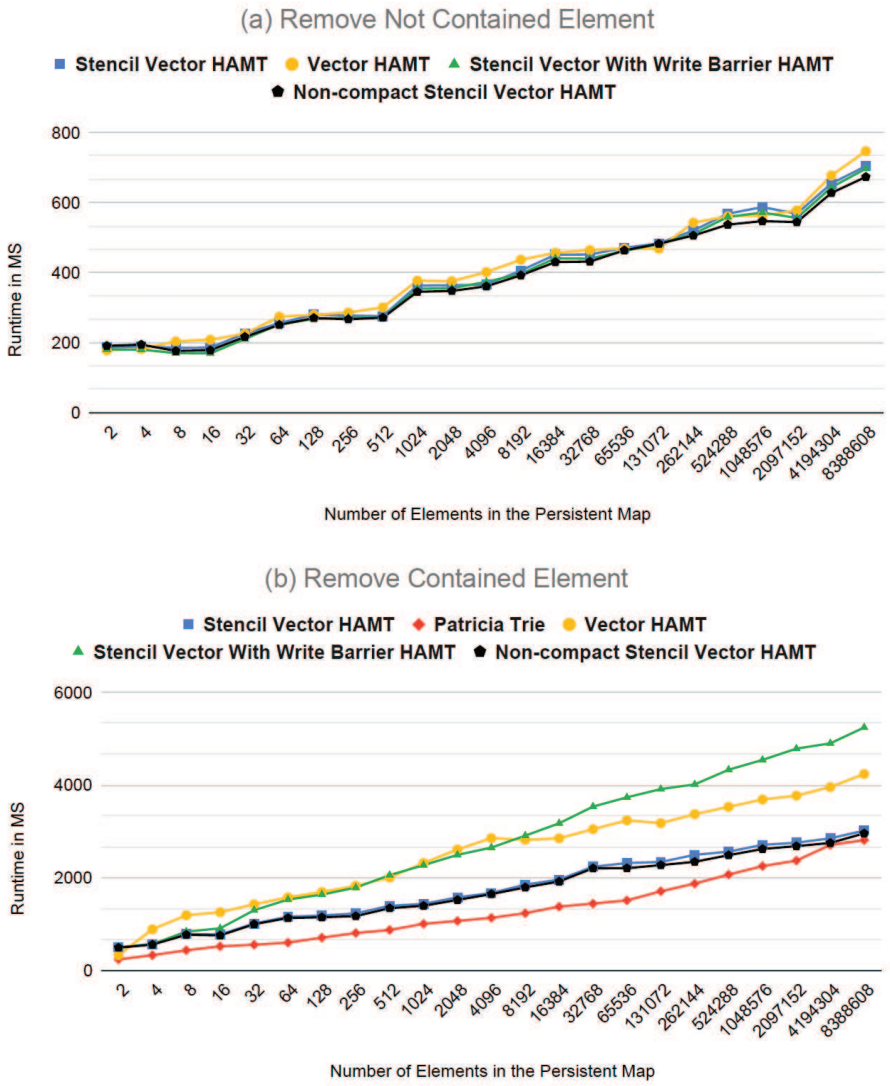

Figure 6: Performance benchmarking results of persistent map implementations in Racket for deletion operation

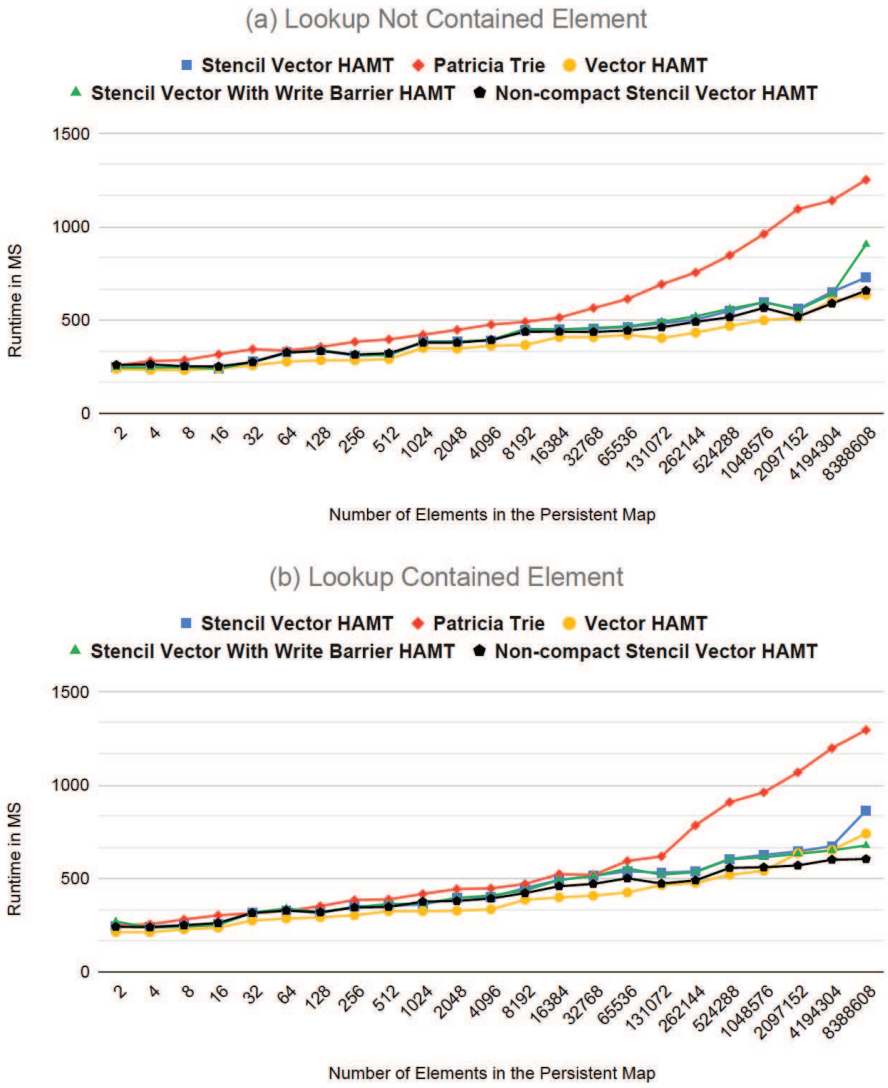

Figure 7: Performance benchmarking results of persistent map implementations in Racket for lookup operation

Performance Comparisons to Regular Vector–Based HAMTs. Stencil vector–based HAMTs perform significantly better compared to regular vector–based HAMTs for inserting new elements and removing existing ones since these operations involve garbage collection. In particular, stencil vector HAMTs perform better than regular vector HAMTs by about 35% for insertion operation and by 27% for removal operation as shown in Figure 5 and 6. For inserting already contained elements in the map and removing elements that are not in the map, both versions of HAMT perform similarly.

For both cases of lookup operation, regular vector HAMTs perform better than stencil vector HAMTs by about 8% as shown in Figure 7.

Performance Comparisons to Patricia Tries. The only operations when Patricia tries perform better are insertion of new elements and removal of existing elements as shown in Figures 5, 6, and 7. For these operations, Patricia tries perform better by 15% and 30%, respectively. However, for both cases, the difference is much less when there are more than 1 million entries, and the time spent on garbage collection increases. In particular, the performance difference is reduced to less than 5% for inserting new elements and 13% for removing contained elements.

When inserting elements that already exist in the map, stencil-vector HAMTs perform on average 60% better than Patricia tries. This result is due to stencil vector HAMTs checking whether the key is already mapped to the given value, in which case the update is skipped, while the Racket implementation of Patricia tries does not check for repeated key-value pairs. Similar results are observed for removal of not contained elements. While adding the reuse check to the Patricia implementation of update operations improves the performance of Patricia tries for inserting already existing elements and removing non-existing ones by 30% and 25%, respectively, it adds a 15% penalty to the actual update opera-

tions that are initially implemented to be simple and fast. These benchmarking results are shown in Figures 12 and 13 in Appendix B.

For both cases of lookup operation, stencil vector–based HAMTs perform better than Patricia tries by 21% and 15%, respectively, as shown in Figures 7(a) and 7(b).

Performance Comparisons to Stencil Vectors with Write Barriers. HAMTs based on stencil vectors with write barriers perform similarly for insertion of an already existing element, deletion of a not contained element, and both cases of lookup when compared to HAMTs based on stencil vectors that avoid write barriers.

However, for operations that update the HAMT, stencil vectors without write barriers perform significantly better. When inserting new elements, avoiding the write barrier allows to improve runtime performance by about 30%. Additionally, as the number of elements increases, so does the performance difference. For example, as shown in Figure 8, for maps that have more than 2^{17} elements stencil vector HAMTs that avoid write barriers perform better by more than 40%.

The results are similar for the removal of existing elements.

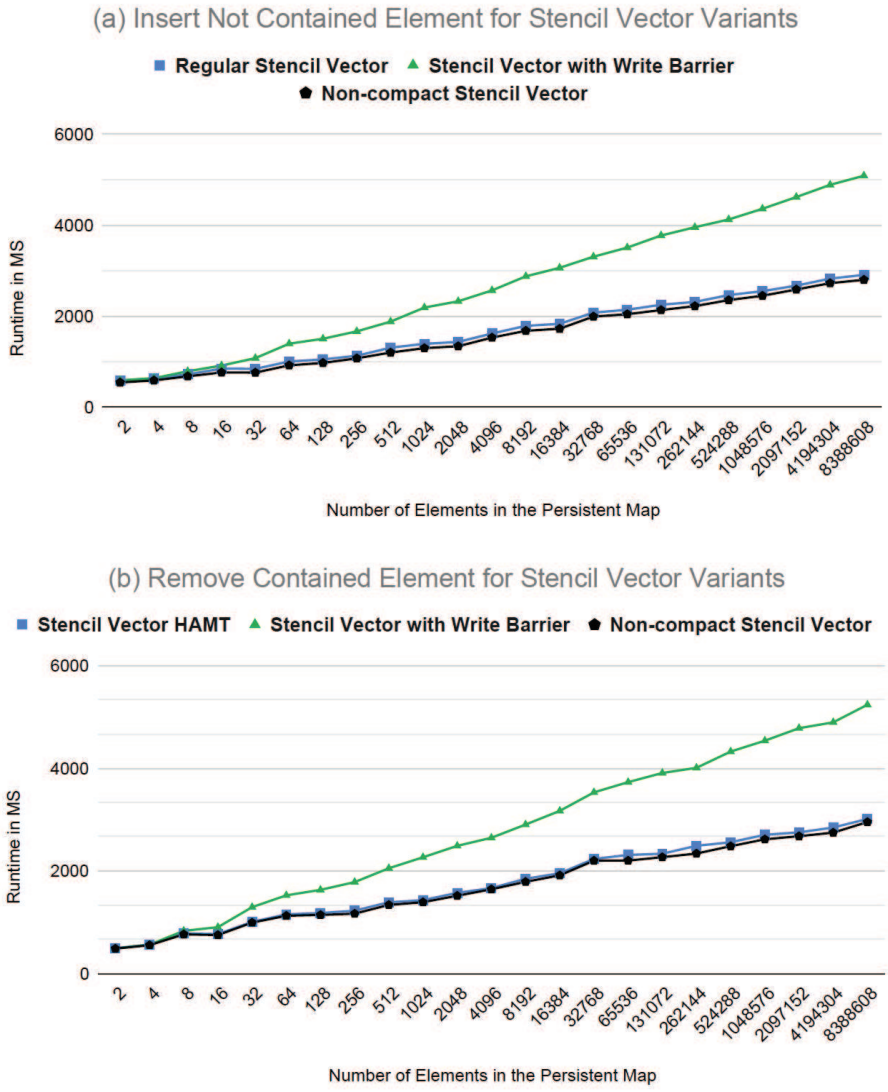

Figure 8: Runtime performance of regular stencil vectors compared to stencil vectors with write barriers and non-compact stencil vectors

Performance Comparisons to Non-Compact Representation of Stencil Vectors. Compared to the non-compact variant of stencil vectors, regular stencil vector–based HAMTs performance is slightly slower for some of the operations. Given that the length of the vector is directly stored in the tag word for the non-compact stencil vector, this variant of stencil-vectors is better by about 6% for insertion of new elements and 8% for insertion of already existing elements. For the rest of the operations, the difference is less than 5%.

Performance Comparisons to CHAMP and Clojure's PersistentHashMap. The results of the additional benchmarks implemented in Java for CHAMP and Clojure's PersistentHashMap, while from different platforms, show that the stencil vector implementation of HAMTs performs comparably to the state of the art. As shown in Figures 9, 10, and 11, the only cases when stencil vector–based HAMTs perform worse than Clojure's PersistentHashMap are for inserting a new element in maps with more than 2^{11} elements and removing an existing element from maps with more than 2^5 elements.

Additionally, stencil vector–based HAMTs perform the best for insertion of already contained elements. In particular, stencil vector HAMTs perform better than CHAMP by around 42% and 65% compared to Clojure's PersistentHashMap as shown in Figure 9(a). For the rest of the operations, performance difference compared to CHAMP in Java varies between 30% (for insertion of new elements) and 58% (for lookup). On average, the recent HAMT implementation for JVM languages that has better performance than Clojure's PersistentHashMap for all operations and Scala's immutable.HashMap for all operations except for removal of not contained elements, performs better than stencil vector–based HAMTs in Racket by a factor of 2.

Compared to Clojure's PersistentHashMap, stencil vector HAMTs perform better by around 30% for inserting new elements and deleting not contained

elements for maps with less than 2^{11} elements. For lookup operation, stencil vector HAMTs perform better than Clojure's PersistentHashMap by 40% for contained elements and by 28% for not contained elements. Similar to other operations, stencil vector HAMTs perform better especially for smaller sizes.

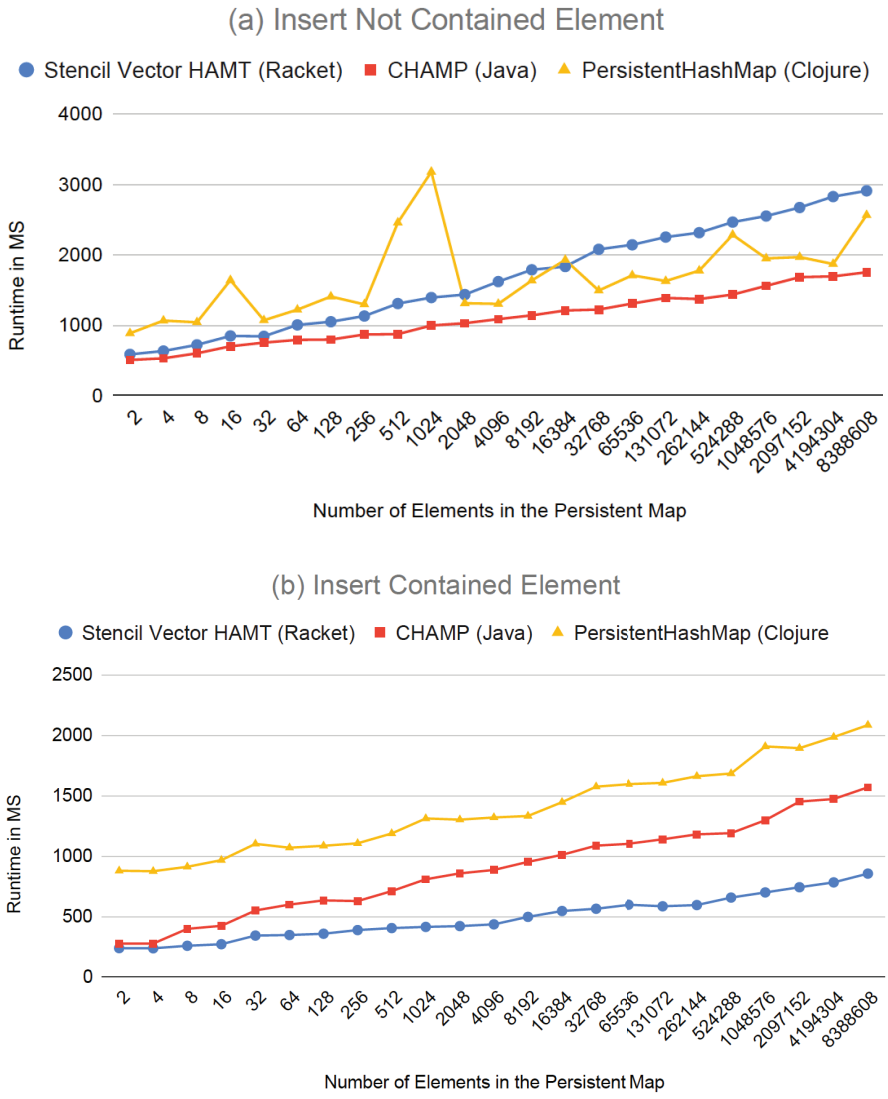

Figure 9: Performance benchmarking results of stencil vector–based HAMT, CHAMP, and Clojure's PersistentHashMap for insertion operation

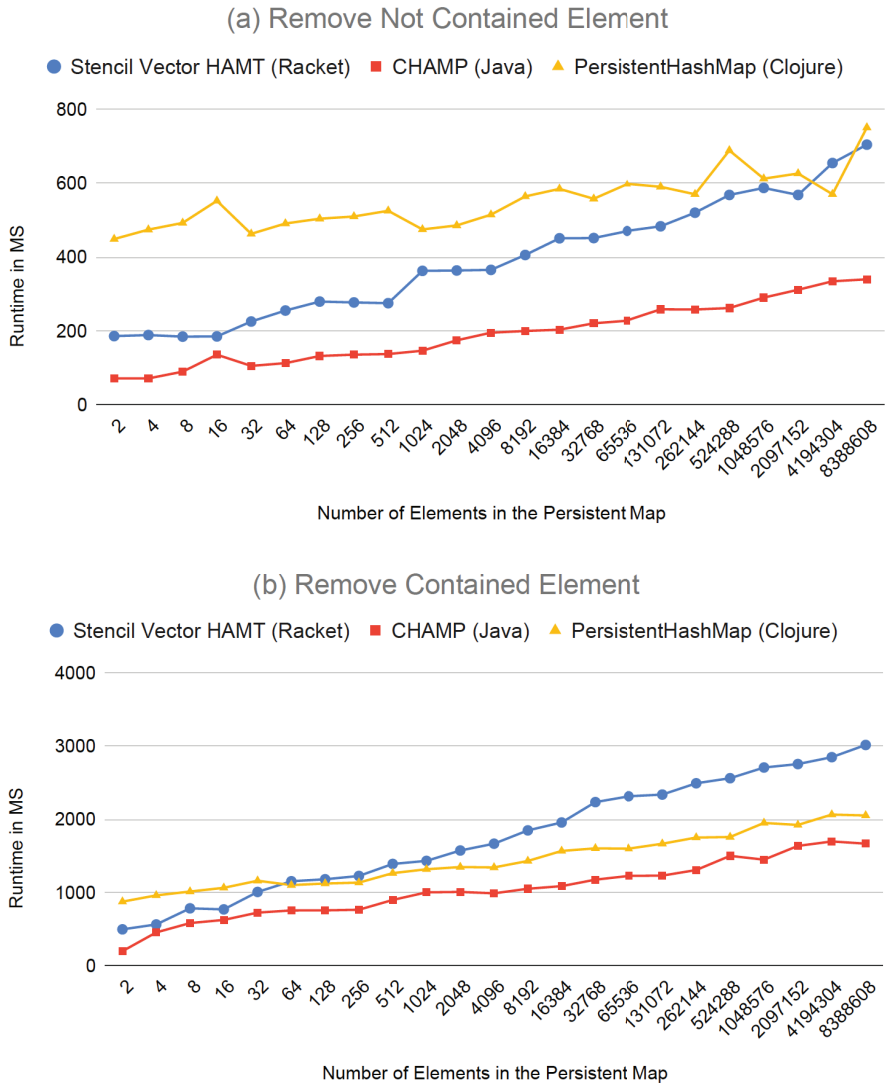

Figure 10: Performance benchmarking results of stencil vector–based HAMT, CHAMP, and Clojure's PersistentHashMap for deletion operation

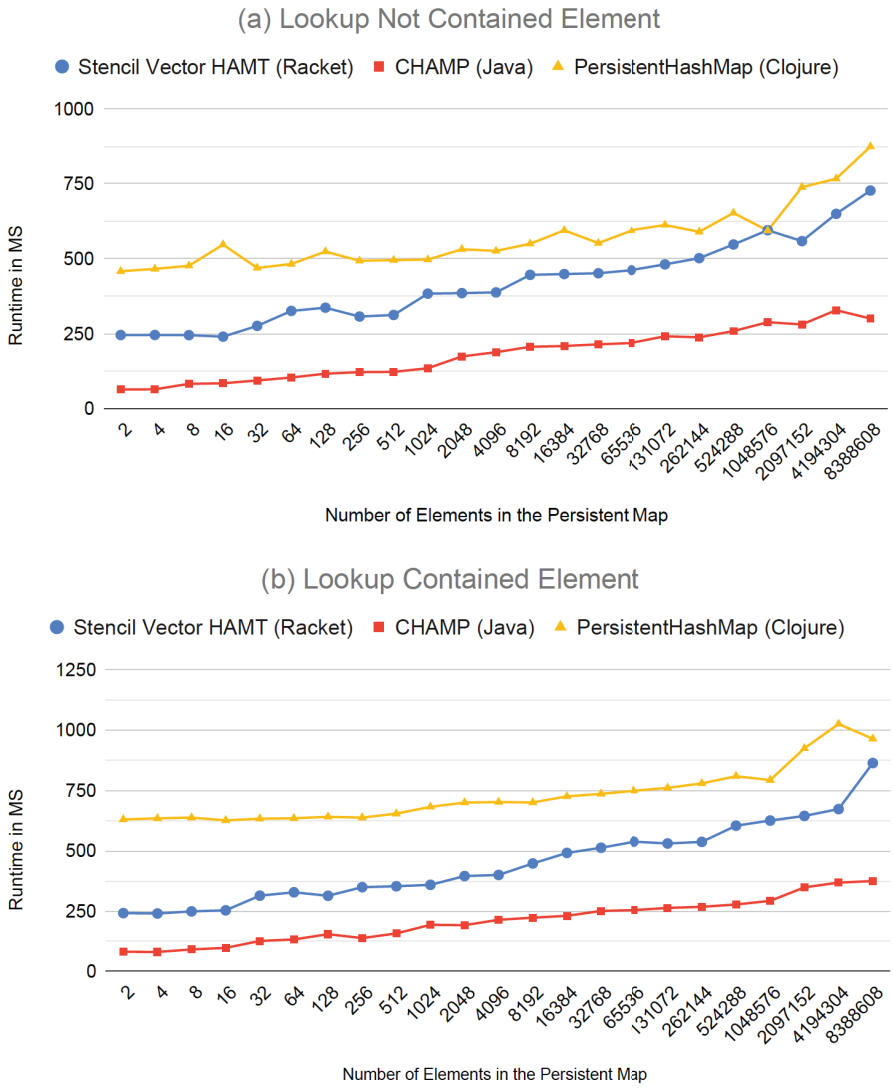

Figure 11: Performance benchmarking results of stencil vector–based HAMT, CHAMP, and Clojure's PersistentHashMap for lookup operation

5 Conclusion

We described a new intermediate data structure, stencil vector, that is currently implemented in Racket and built into the runtime system and compiler. It provides a smaller and more efficient representation for HAMT nodes compared to other node representations such as regular vectors. The evaluation of the new data structure shows that the performance of updating persistent maps that are represented by stencil vector HAMTs is improved compared to other HAMT implementations due to the data structure's cooperation with the garbage collector. Additionally, our memory microbenchmarking results have shown that stencil vector–based HAMTs result in smaller memory compared to Patricia trie and regular vector–based HAMTs implementations of persistent maps.

6 Acknowledgements

This work was supported by funding from the Undergraduate Research Opportunities Program at the University of Utah.

References

[1] Phil Bagwell. Ideal hash trees. Technical report, Ecole polytechnique fédérale de Lausanne, 2001.

[2] Rene De La Briandais. File searching using variable length keys. In *Papers presented at the the March 3-5, 1959, western joint computer conference*, pages 295–298, 1959.

[3] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[4] Rich Hickey. The clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages*, pages 1–1, 2008.

[5] Witold Litwin, Marie-Anne Neimat, and Donovan A Schneider. Lh: Linear hashing for distributed files. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 327–336, 1993.

[6] Donald R Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM (JACM)*, 15(4):514–534, 1968.

[7] Chris Okasaki and Andy Gill. Fast mergeable integer maps. In *Workshop on ML*, pages 77–86, 1998.

[8] Michael J Steindorfer and Jurgen J Vinju. Optimizing hash-array mapped tries for fast and lean immutable jvm collections. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 783–800, 2015.

Appendices

A Performance Benchmarks in Racket

```
#lang racket/base
(require racket/set racket/vector)

(define fixedSeedMersennePrime 2147483647)
(define seedForNotContained 12745)
(define I 20)
(define sizeOfTest 8)
(define Q 2000000)
(define maxNumber 4294967087)


(define-syntax times
  (syntax-rules ()
    [(_ e)
     (let loop ([v #f] [i I])
       (if (zero? i)
           v
           (loop (begin
                   (collect-garbage 'major)
                   (time e))
                 (sub1 i))))]))

;;create a map from random elements
(define (createMapRandom size seed)
  (random-seed seed)
  (let loop ([ht (hasheq)] [i size])
    (if (zero? i)
        ht
        (let ([val (random maxNumber)])
          (if (hash-ref ht val #f)
              (loop ht i)
              (loop (hash-set ht  val  val)
                    (sub1 i)))))))

;;create a vector of random elements of length 'size'
(define (vectorRand size seed)
  (let ([newVec (make-vector size)])
    (random-seed seed) ;;reset random
    (for ([ind (in-range size)])
```

```
40        (vector-set! newVec ind (random maxNumber)))
41      newVec))
42
43
44  ;;create a vector of length 'size'
45  ;;using only 'firstN' elements of 'vc'
46  (define (getSubsetVector vc size firstN)
47    (let ([newVec (make-vector size)])
48      (for ([ind (in-range size)])
49        (vector-set! newVec ind
50                     (vector-ref vc (modulo ind firstN))))
51      newVec))
52
53
54
55  (define vector8NotContained (vectorRand 8 seedForNotContained))
56  (define vector2NotContained (getSubsetVector vector8NotContained 8 2))
57  (define vector4NotContained (getSubsetVector vector8NotContained 8 4))
58
59  (define vector8Contained (vectorRand 8 fixedSeedMersennePrime))
60  (define vector2Contained (getSubsetVector vector8Contained 8 2))
61  (define vector4Contained (getSubsetVector vector8Contained 8 4))
62
63
64  (define (chooseSize x isNotContained)
65    (if isNotContained
66        (cond
67          [(equal? x 2) vector2NotContained]
68          [(equal? x 4) vector4NotContained]
69          [else vector8NotContained])
70        (cond
71          [(equal? x 2) vector2Contained]
72          [(equal? x 4) vector4Contained]
73          [else vector8Contained])))
74
75
76
77  ;;____benchmarks_____
78
79  ;;insert not contained
80  (displayln "insert not contained")
81  (for ([sz (in-range  1 24)])
82    (displayln (expt 2 sz))
83    (let ([ht (createMapRandom (expt 2 sz) fixedSeedMersennePrime)])
84      (let([notContained (chooseSize (expt 2 sz) #t)])
85        (times
```

```
86          (for ([i (in-range Q)])
87            (let loop ([v #f] [i sizeOfTest])
88              (if (zero? i)
89                  v
90                  (loop (hash-set ht (vector-ref notContained
91                                                 (- sizeOfTest i))
92                                  (vector-ref notContained
93                                                 (- sizeOfTest i)))
94                        (sub1 i)))))))))
95
96  ;;insert contained
97  (displayln "insert contained")
98  (for ([sz (in-range  1 24)])
99    (displayln (expt 2 sz))
100   (let ([ht (createMapRandom (expt 2 sz) fixedSeedMersennePrime)])
101     (let ([contained  (chooseSize (expt 2 sz) #f)])
102       (times
103        (for ([i (in-range Q)])
104          (let loop ([v #f] [i sizeOfTest])
105            (if (zero? i)
106                v
107                (loop (hash-set ht (vector-ref contained
108                                               (- sizeOfTest i))
109                                (vector-ref contained
110                                               (- sizeOfTest i)))
111                      (sub1 i)))))))))
112
113 ;;remove not contained
114 (displayln "remove not contained")
115 (for ([sz (in-range 1 24)])
116   (displayln (expt 2 sz))
117   (let ([ht (createMapRandom (expt 2 sz) fixedSeedMersennePrime)])
118     (let ([notContained (chooseSize (expt 2 sz) #t)])
119       (times
120        (for ([i (in-range Q)])
121          (let loop ([v #f] [i sizeOfTest])
122            (if (zero? i)
123                v
124                (loop (hash-remove ht (vector-ref notContained
125                                                  (- sizeOfTest i)))
126                      (sub1 i)))))))))
127
128
129 ;;remove contained
130 (displayln "remove contained")
131 (for ([sz (in-range 1 24)])
```

```
132   (displayln (expt 2 sz))
133   (let ([ht (createMapRandom (expt 2 sz) fixedSeedMersennePrime)])
134     (let ([contained  (chooseSize (expt 2 sz) #f)])
135       (times
136        (for ([i (in-range Q)])
137          (let loop ([v #f] [i sizeOfTest])
138            (if (zero? i)
139                v
140                (loop (hash-remove ht (vector-ref contained
141                                                  (- sizeOfTest i)))
142                      (sub1 i)))))))))
143
144
145 ;;lookup not contained
146 (displayln "lookup not contained")
147 (for ([sz (in-range 1 24)])
148   (displayln (expt 2 sz))
149   (let ([ht (createMapRandom (expt 2 sz) fixedSeedMersennePrime)])
150     (let ([notContained (chooseSize (expt 2 sz) #t)])
151       (times
152        (for ([i (in-range Q)])
153          (let loop ([v #f] [i sizeOfTest])
154            (if (zero? i)
155                v
156                (loop (hash-ref ht
157                                (vector-ref notContained
158                                            (- sizeOfTest i))
159                                #f)
160                      (sub1 i)))))))))
161
162 ;;lookup contained
163 (displayln "lookup contained")
164 (for ([sz (in-range 1 24)])
165   (displayln (expt 2 sz))
166   (let ([ht (createMapRandom (expt 2 sz) fixedSeedMersennePrime)])
167     (let ([contained  (chooseSize (expt 2 sz) #f)])
168       (times
169        (for ([i (in-range Q)])
170          (let loop ([v #f] [i sizeOfTest])
171            (if (zero? i)
172                v
173                (loop (hash-ref ht
174                                (vector-ref contained
175                                            (- sizeOfTest i))
176                                #f)
177                      (sub1 i)))))))))
```

B Performance Differences of Patricia Trie Implementations and Stencil Vector–Based HAMTs

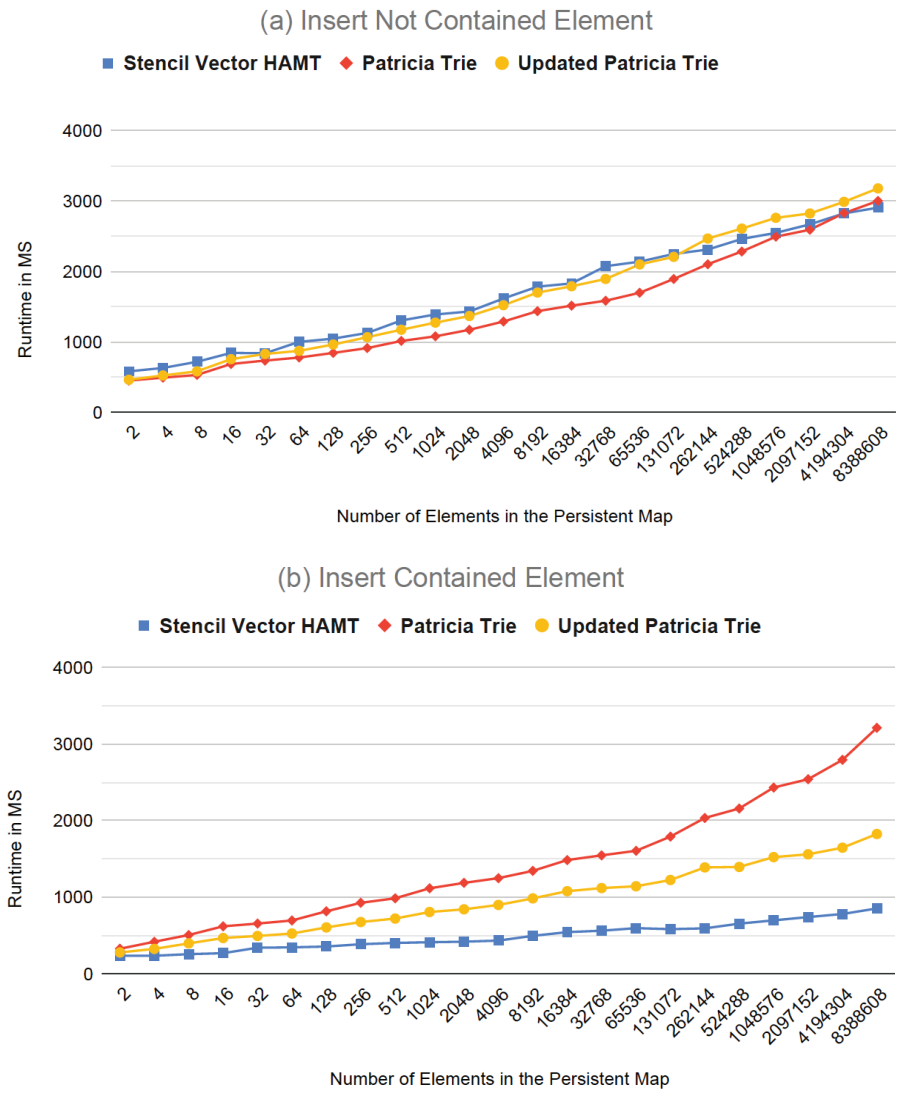

Figure 12: Performance benchmarking results of stencil vector–based HAMT, Patricia trie, and updated Patricia trie with reuse check for insertion operation

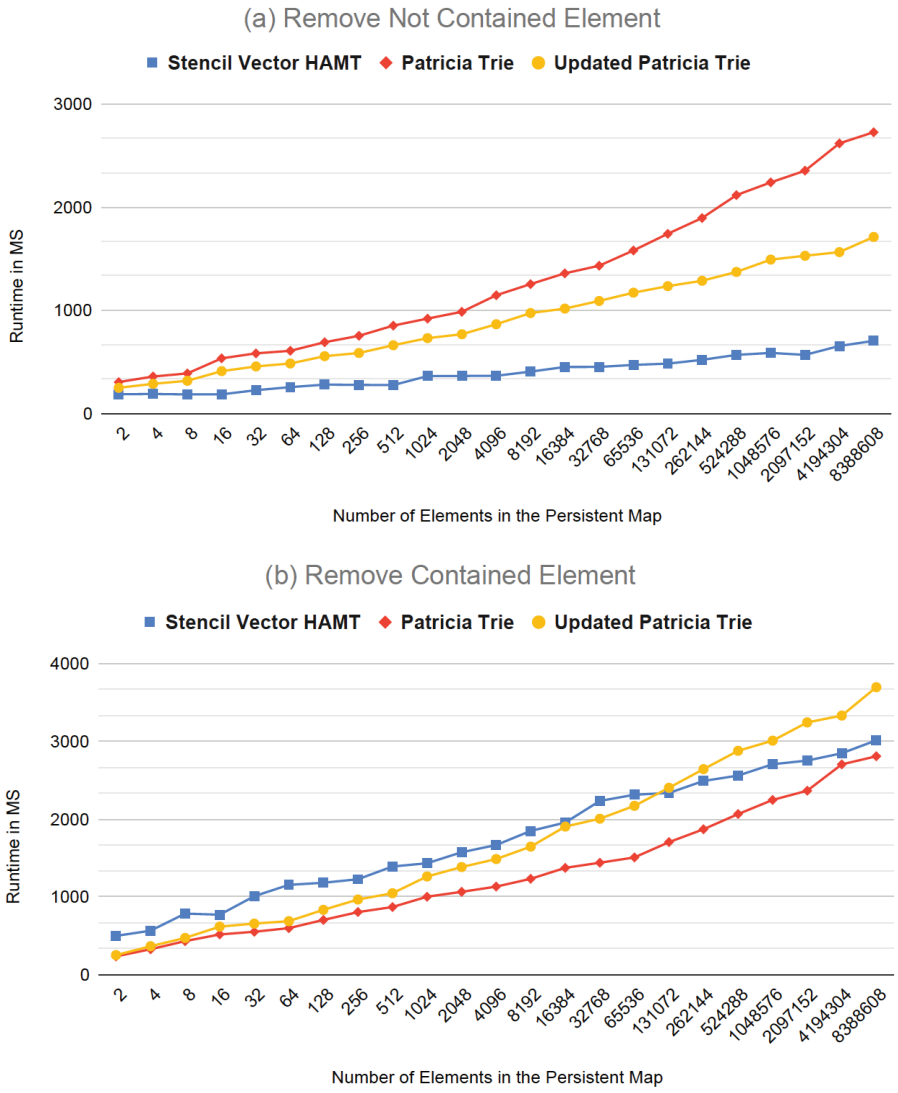

Figure 13: Performance benchmarking results of stencil vector–based HAMT, Patricia trie, and updated Patricia trie with reuse check for deletion operation