The "Jump-counting" Skipfield Pattern

A replacement for boolean skipfields yielding O(n) time complexity for iteration over unskipped elements

Matthew Bentley

mattreecebentley@gmail.com

Abstract. This document describes a numeric pattern for indicating inactive (skippable) items in data sequences, and methods for adjusting that pattern based on the insertion and erasure of items, to provide a more computationally-efficient alternative to boolean skipfields in computer science applications, particularly in the implementation of data containers and game engines.

Keywords: boolean, erasure field, skip field, computational efficiency, game dev, containers, performance

1 Introduction

In many areas of computer science, but in particular within video game engines, there exist scenarios where boolean fields are used to indicate items which are no longer active or usable instead of actually erasing or removing these elements, as erasure tends to come with one or more side-effects depending on the type of data container involved. Subsequently, during iteration and processing, this boolean field is checked so as to skip over and ignore the elements flagged as inactive. A simple example could be a raw array of data for individual enemy objects in a video game with an additional boolean field named "active" in each object, where the "active" flag is set to false when that enemy object is destroyed in the game. Objects with a false "active" flag would then be skipped over for subsequent processing.

The advantages of such a technique are (a) with a primitive structure such as an array, actual removal of an element from active memory may not be possible, and (b) if a more complex structure designed for iterative speed, such as a dynamic array or queue is used, actual erasure of elements typically causes strong performance penalties [1, Standard Template Library, p. 164] due to the reallocation of subsequent elements in order to preserve data contiguity, the process of which also (c) invalidates pointers to data within the container [2, Sequential Containers, p. 485]. This last point is of concern within highly modular or object-oriented code, of which game engines are often comprised. To work around these problems the aforementioned active/inactive boolean skipfield is often implemented and pointer/iterator validity thereby preserved, along with iterative processing speed.

(Note: highly-contiguous storage containers such as C++'s std::vector, std::deque and arrays tend to be preferred in high-performance scenarios over non-contiguous storage methods such as linked-lists and maps, despite the fact that the latter do not tend to have the above side-effects when erasing. This is due to the poor CPU cache performance associated with non-contiguous memory storage during iteration on modern CPU's [4, p.44].)

This technique is simple and often effective, but can be inefficient for any container with large numbers of consecutive "inactive" elements, as iteration over these requires checking a boolean field for each inactive element. An ideal solution would simply skip from one "active" element to the next in a single step. As soon as some elements in the container are made inactive, the time complexity for iterating from one active element to the next becomes O(random) instead of O(1), as there is no way for the program to know how many inactive elements are present between two active elements without consulting the boolean field for each and every element. Depending on the number and location of container erasures there could be as few as 1, or as many as "container size - 2", boolean checks between each "active" element in the container.

A jump-counting skipfield negates the inefficiencies of this approach, giving O(1) amortized time complexity for linear iterative traversal from one active element to the next in a container, without creating significant additional computational overhead. It is a numeric sequence which indicates both when to skip over elements and how many elements to skip over in any sequence of elements. It is most obviously useful in implementing a pointer-stable/iterator-stable data container, where it can indicate sequences of erased elements to skip over without necessitating reallocation of non-erased elements and thus causing performance degradation and index/pointer invalidation, as described above. It can also be directly implemented in any computational engine without a container object, for example to replace the boolean field which is described above as an indication of video game objects which are destroyed.

Another application is the indication of temporarily-inactive objects, a specific example for which is explored in more detail in section 5, "Additional areas of application and manipulation". Ultimately the jump-counting pattern, as with a boolean pattern, can be used to describe any binary aspect of any sequence of objects, regardless of whether that binary aspect is active/inactive, gray-scale/colored, living/dead or non-erased/erased. What it facilitates is the skipping over of elements which fit one of the two states. Hence an alternative name for the pattern is a 'Theyaton' skipfield, an acronym for 'Traversal of homogeneous elements yielding amortized time = O(n)', where any one of the binary states named above describes an aspect by which the non-skipped elements can be deemed homogeneous. For the purposes of simplicity we will discuss the jump-counting skipfield pattern in the context of implementation within a data container where a skipped element indicates an erased element and a non-skipped element indicates a non-erased element.

There is a simple variant of the pattern which allows only for the erasure of elements, and an advanced pattern which allows for both erasure of elements and reuse of erased-element locations upon subsequent insertion. Colony[1], the data container for which this pattern was developed, utilizes linked chains of increasingly-large element memory blocks with accompanying advanced jump-counting skipfields. This allows the container to maintain pointer/iterator stability during both insertion and erasure while maintaining fast iterative performance, as well as allowing it to reuse erased element memory space. But a programmer implementing a custom solution (using arrays with skipfields, for example) might, depending on their circumstances, choose the simple pattern as it has a very small erasure performance advantage.

1.1 Pre-existing Research

A review of the literature has shown no similar pre-existing work on replacements for boolean skipfields. Exhaustive searching of scholarly journals across 8 high-profile computer science journal aggregators did not yield similar patterns nor did it show any existing research into the use of boolean fields to indicate element erasure in data sets. One possible explanation for this is that interaction between video game development, where the boolean technique described is often used, and academia, is limited. Another explanation may be that this is a development which primarily arises in environments with a focus on high performance, where the most optimal solutions are typically hardware-bound and as such have a lesser focus on solutions for arbitrary hardware and generalized computational science.

2 The Simple Jump-counting Pattern

2.1 Basic notation

A jump-counting skipfield is always an array (or extensible container) of an unsigned integer type. This type must be of sufficient bit-depth to describe the number of elements that can be potentially stored within the memory block it is associated with. So a block large enough to store 65535 elements would require a skipfield made up of 16-bit unsigned integers, while a memory block large enough to store 4294967295 elements would require a skipfield of 32-bit unsigned integers. All non-erased elements are notated with zero, so if you had a set of ten non-erased elements in a memory block, the block's corresponding skipfield (S) would be:

$S = (0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0)$

Any non-zero value indicates erasure. If a singular element is erased and has no consecutive erased elements it is notated with 1:

[1] http://www.plflib.org/colony.htm

$S = (0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0)$

From this point onward we will refer to locations within the skipfield as "nodes" and sequences of consecutively erased elements (as notated in the skipfield) as "skipblocks". Below we show a skipfield attached to a data container where the 3rd, 4th, 5th and 6th elements have been erased. The first node indicating a skipblock (eg. the first "4" in the example below) is called the "start node". The last node in that skipblock is called the "end node" (eg. the last "4" in the example below). The values of both start and end node are set to the number of elements which need to be skipped. In the solo erasure example above, the "1" is both the start and end node of a skipblock.

$S = (0\ 0\ 4\ 0\ 0\ 4\ 0\ 0\ 0\ 0)$

2.2 Iteration

To iterate across a sequence of elements utilizing a jump-counting skipfield to identify erased elements, an iterator must keep track of both the element it's pointing to and that element's corresponding skipfield node. The skipfield node's index will always correspond to the element's index ie. element[5] is always associated with skipfield[5]. Because of the index association an iterator only technically need keep track of the index number (i), but for performance reasons we might choose an implementation using both a pointer to the element array (e) and a pointer to the skipfield array (s).

If we use the pointer approach, to advance an iterator by 1 we would:

1. increment both the element index (e) and the skipfield node index (s) by 1, then
2. add the value of the number pointed to by the skipfield pointer ($*s$ - the size of the additional jump) to both the element pointer and to the skipfield pointer itself.

ie.:
$s = s + 1$
$e = e + 1 + *s$
$s = s + *s$

Or in C++:
```
e += 1 + *(++s);
s += *s;
```

Alternatively if we use the index number iterator approach, we can:

1. increment the index number (i) by one, then

2. add the value of the skipfield at index i (S_i) to the index number (i).

ie.:
$i = i + 1$
$i = i + S_i$

Or in C++
```
++i;
i += S[i];
```

Regardless of whether an index or pointer implementation is utilized, the value of the number stored in the skipfield node is the number of additional units to advance by in a single iterative step. This is also equivalent to the number of consecutive erased elements at that point in the skipfield. For the purpose of simplicity, we will discuss only the index implementation of the jump-counting pattern for the remainder of this document. Below is an example of a single iterative step over the skipfield example shown earlier:

(Notes: in all examples, ↓ indicates the skipfield node corresponding to the current index number i. Index enumeration starts from 1, to make the examples easier to understand, rather than from 0, as would be implemented in most programming languages. Lastly, we will skip the S = () array notation for all subsequent examples for the purpose of visual clarity.)

Before incrementing:

↓
0 0 4 0 0 4 0 0 0 0
($i = 1$)

Increment by 1:

$\underline{i = i + 1}$

$i = 2, S_i = 0$

$\underline{i = i + S_i}$

$i = 2, S_i = 0$

After incrementing:

```
  ↓
0 0 4 0 0 4 0 0 0 0
```

Increment by 1 again:

$\underline{i = i + 1}$

$i = 3, S_i = 4$

$\underline{i = i + S_i}$

$i = 7, S_i = 0$

After incrementing again:

```
            ↓
0 0 4 0 0 4 0 0 0 0
```

This also works when reverse-iterating, in which case you:

1. decrement the index (i) by one, then
2. subtract the value of the skipfield node at index (S_i) from the index.

Once again the second number subtracted is the number of elements skipped in that decrement. Here is an example using the same skipfield pattern as above:

Before decrementing:

```
              ↓
0 0 4 0 0 4 0 0 0 0
```
($i = 8$)

Decrement by 1:

$\underline{i = i - 1}$

$i = 7, S_i = 0$

$\underline{i = i - S_i}$

$i = 7, S_i = 0$

After decrementing:

```
            ↓
0 0 4 0 0 4 0 0 0 0
```

Decrement by 1 again:

$\underline{i = i - 1}$

$i = 6, S_i = 4$

$\underline{i = i - S_i}$

$i = 2, S_i = 0$

After decrementing again:

```
  ↓
0 0 4 0 0 4 0 0 0 0
```

2.3 Erasure

If an element is erased we check the nodes to the left and to the right of its corresponding skipfield node:

*(note: * indicates nodes whose values we are assessing)*

```
      * ↓ *
0 0 0 0 0 0 0 0 0 0
```

If there is an erased node to the left, that node is the end node of a skipblock on the left. If there's an erased node to the right, that node is the start node of a skipblock on the right. The central premise of erasure within a jump-counting skipfield is that both start and end nodes count the number of nodes to skip, so if we know where the start node is, we also know where the end node is and vice-versa. We can update both if we know the value of one.
Once we have checked the value of the nodes to the left and right, there are four scenarios:

1. both left and right are zero
2. left is non-zero and right is zero
3. left is zero and right is non-zero
4. both left and right are non-zero

We will now examine how each of those scenarios is to be handled.

Scenario 1: Both left and right are zero

We set the value of the current skipfield node to 1. This indicates a single element erasure with no consecutive erased elements. No further action is required.

(note: in this context ↓ indicates the skipfield node corresponding to the container element we're erasing)

Before erasure:

```
            ↓
0 0 0 0 0 0 0 0 0 0
```

Erasure:

$S_i = 1$

After erasure:

```
            ↓
0 0 0 0 0 0 1 0 0 0
```

Scenario 2: Only left is non-zero

If only the left-hand node is non-zero, we set the value of the current node to the value of the left-hand node, then increment it by 1. The current node is now the new end node for the preceding skipblock. We then subtract the value of the left-hand node from the current node's index to find the index of the start node's, then set the start node's value to the current node's value (or increment the start node by one). We do not change the value of the left-hand node (the prior end node).

Before erasure:

```
            ↓
0 0 0 3 0 3 0 0 0 0
```

Erasure:

$S_i = 1 + S_{i-1}$
$j = i - S_{i-1}$
$S_j = S_i$

After erasure:

```
            ↓
0 0 0 4 0 3 4 0 0 0
```

Scenario 3: Only right is non-zero

If only the right-hand node is non-zero, we make the current node the start node of the subsequent skipblock by making it equal to the right-hand node, then incrementing it by 1. Then we add the right-hand node's value to the current node's index to find the end node, and set the value of the end node to the value of the current node.

Before erasure:

```
    ↓
0 0 0 2 2 0 0 0 0 0
```

$i = 3$

Erasure:

$S_i = 1 + S_{i+1}$
$j = i + S_{i+1}$
$S_j = S_i$

After erasure:

```
    ↓
0 0 3 2 3 0 0 0 0 0
```

Scenario 4: Both left and right are non-zero

If both left-hand and right-hand nodes are non-zero, then the current node is between two separate skipblocks, one to the left and another to the right. In this case we begin by subtracting the left-hand node's value from the current node's index number to find the left skipblock's start node's index. We increment the value of that start node by one plus the value of the node to the right of the current node. Then we add the same right-hand node's value to the current node's index to find the index of the end node for the skipblock on the right. Lastly we set the value of that end node to the same value as the left skipblock's start node.

Before erasure:

  ↓
2 2 0 3 0 3 0 0 0 0

Erasure:

$\underline{j = i - S_{i-1}}$

$j = 3 - S_2$

$\underline{k = i + S_{i+1}}$

$k = 3 + S_4$

$\underline{S_j = S_j + 1 + S_{i+1}}$

$S_1 = S_1 + 1 + S_4$

$\underline{S_k = S_j}$

$S_6 = S_1$

After erasure:

  ↓
6 2 0 3 0 6 0 0 0 0

2.4 Summary

All of these operations are straightforward but the result does not support the reuse of memory space from previously-erased elements. This is what the advanced pattern facilitates. Because the simple pattern involves fewer calculations than the advanced pattern however, it could be slightly faster in a use-case where reuse of memory space is either not useful or not practicable.

3 The Advanced Jump-counting Pattern

3.1 Basic notation

This does not differ from the simple pattern, except for the fact that the nodes between the start and end node in a skipblock cannot be zero and instead follow a sequential increment-by-one pattern as the example below demonstrates:

0 0 4 2 3 4 0 0 0 0

The formation of this pattern takes place in section 3.3 below, while its purpose becomes apparent in the subsequent restoration or "reuse" section, 3.4. What the above pattern communicates is the index of any given node within the skipblock itself, which also tells us where to find the the start node of the skipblock. The node with a value of 3 in the above example is the third in the skipblock, meaning the start node is 2 nodes away. From the value of the start node we can also find the end node, and as we will see, that information is sufficient for us to be able to update the entire skipblock.

3.2 Iteration

The procedures for iteration do not differ between the simple and advanced patterns, for either increments or decrements.

3.3 Erasure

Erasure in the advanced pattern involves slightly more work than the simple pattern but for the most part is similar. As in the simple pattern, if an erasure is made to the current element, we check the skipfield nodes to the left and to the right of the corresponding skipfield node.

Scenario 1: Both left and right are zero

As with the simple pattern, we set the current node's value to 1.

Scenario 2: Only left is non-zero

Once again the procedure is exactly the same as in the simple pattern. We set the value of the current node to the value of the left-hand node, then increment by 1. We then subtract the value of the left-hand node from the current node's index number to find the skipblock's start node, then set the start node to the same value as the current node.

Scenario 3: Only right is non-zero

Here we make the current node equal to the right-hand node then increment it by 1, and it becomes the start node of the skipblock to the right. So far this is the same as the simple pattern, but instead of finding the end node and updating it as we would in the simple pattern, we update the values of each node to the right of the current node, starting from a value of 2 and incrementing by 1 per node, stopping either when a node with a zero value or the end of the skipfield is reached.

Before erasure:

    $\downarrow$
0 0 0 3 2 3 0 0 0 0

Erasure:

$S_i = 1 + S_{i+1}$
$S_{i+1} = 2$
$S_{i+2} = 3$
$S_{i+3} = 4$

After erasure:

    $\downarrow$
0 0 4 2 3 4 0 0 0 0

We can now see the increment-by-one pattern noted earlier.

Scenario 4: Both left and right are non-zero

As with the simple pattern, we subtract the value of the left-hand node from the current node's index number to find the index of the left skipblocks start node . We increment this start node by 1 plus the value of the current node's right-hand node. Now we deviate from the simple pattern; instead of updating the end node of the left-hand skipblock, we take the value of the left-hand node, store it (x) and increment it by 1. Then, starting from the current node and continuing to the right, we set every node to x, incrementing x by 1 for each node, stopping when either a node with a value of zero or the end of the skipfield is reached.

Before erasure:

```
    ↓
2 2 0 3 2 3 0 0 0 0
```

Erasure:

$j = i - S_{i-1}$
$S_j = S_j + 1 + S_{i+1}$
$x = 1 + S_{i-1}$
$S_i = x$
$S_{i+1} = x + 1$
$S_{i+2} = x + 2$
$S_{i+3} = x + 3$

After erasure:

```
    ↓
6 2 3 4 5 6 0 0 0 0
```

Again we can see the increment-by-one pattern, which will become useful in the reuse section below.

3.4 Reuse of erased-element memory

If we want to insert into the container and reuse memory space from an element that was previously erased, we need a mechanism for updating the skipfield to reflect this change to the non-erased/erased status of the element, while simultaneously keeping the skipfield valid for use during iteration. Consider the following example: I wish to reuse the memory space of a previously-erased element at index 4 in a given container, and to indicate this in the skipfield I naively set the element’s corresponding skipfield node to zero. Here is what will happen:

Before reuse:

```
      ↓
6 2 3 4 5 6 0 0 0 0
```

(Incorrect) reuse:

$S_i = 0$

After reuse:

$$6\ 2\ 3\ \overset{\downarrow}{0}\ 5\ 6\ 0\ 0\ 0\ 0$$

With this result, if you subsequently iterated over the data from the beginning, the reused element's location would be skipped over. And if you began iteration from the reused element's location, you would skip over non-erased elements. Here's an example of the latter using the result above:

Before increment:

$$6\ 2\ 3\ \overset{\downarrow}{0}\ 5\ 6\ 0\ 0\ 0\ 0$$

$i = 4$

Increment by 1:

$\underline{i = i + 1}$

$i = 5$

$\underline{i = i + S_i}$

$i = 10$

After increment:

$$6\ 2\ 3\ 0\ 5\ 6\ 0\ 0\ 0\ \overset{\downarrow}{0}$$

This is obviously incorrect. To correctly update the skipfield when reusing erased locations we need to check the value of the skipfield nodes both to the left and right of the skipfield node corresponding to the erased element whose location we want to reuse, similar to the process we use during erasure. For example:

$$0\ 0\ 0\ 4\ \overset{*}{2}\ \overset{\downarrow}{3}\ \overset{*}{4}\ 0\ 0\ 0$$

As noted in section 3.1, for any skipblock, the value of any node after the start node indicates that node's index within the skipblock, which enables us to find the start node from any node in the skipblock. Once we know the start node's value we can also find the end node's value, and thereby correctly update the entire skipblock. Once again we have four scenarios: either both left and right node values are zero, left is non-zero and right is zero, left is zero and right is non-zero, or both left and right are non-zero. Here's how we handle those scenarios in this case:

Scenario 1: Both left and right are zero

We set the value of the current node to zero. No further updates are necessary.

Scenario 2: Only left is non-zero

This indicates that the current node is the end node of a skipblock on the left. In this case we take the current node's value and store it (x), subtract 1 from x, then subtract x from the current node's index to find the index of the skipblock's start node. We then set the start node's value to x and the current node's value to zero.

Before reuse:

0 0 0 4 2 3 $\overset{\downarrow}{4}$ 0 0 0

$i = 7, S_i = 4$

Reuse:

$x = S_i - 1$
$j = i - x$
$S_j = x$
$S_i = 0$

After reuse:

0 0 0 3 2 3 $\overset{\downarrow}{0}$ 0 0 0

Another example - before reuse:

0 2 $\overset{\downarrow}{2}$ 0 0 0 0 0 0 0

$i = 3, S_i = 2$

After reuse:

```
  ↓
0 1 0 0 0 0 0 0 0 0
```

Scenario 3: Only right is non-zero

This indicates that the current node is the start node of a skipblock continuing to the right. We set the value of the right-hand node to the value of the current node minus 1, then set the current node's value to zero. Then, starting with the node after the right-hand node, we update the values of all subsequent nodes to the right, starting with a value of 2 and incrementing by 1 for each node, until we reach either the end of the skipfield or a zero value. Example:

Before reuse:

```
      ↓
0 0 0 4 2 3 4 0 0 0
```

Reuse:

$S_{i+1} = S_i + 1$
$S_i = 0$
$S_{i+2} = 2$
$S_{i+3} = 3$

After reuse:

```
      ↓
0 0 0 0 3 2 3 0 0 0
```

Scenario 4: Both left and right are non-zero

In this scenario the current node is inside a skipblock, but neither at the beginning nor the end of it. The end result of the operation must be to split this singular skipblock into two. To do so we combine the procedures for the "only left is non-zero" and "only right is non-zero" scenarios. For the first phase of the transformation we store the current node's value as x and subtract 1 from x. We then subtract x from the current node's index to find the index of the start node, then set the value of the right-hand node to the value of the start node, minus the value of the current node.

Before:

```
        ↓
6 2 3 4 5 6 0 0 0 0
```

$i = 4, S_i = 4$

Reuse (first phase):

$\underline{x = S_i - 1}$

$x = 4 - 1$

$\underline{S_{i+1} = S_{i-x} - S_i}$

$S_5 = S_1 - 4$

After first phase:

```
        ↓
6 2 3 4 2 6 0 0 0 0
```

In the second phase we set the start node’s value to x and the value of the current node to zero.

Reuse (second phase):

$\underline{S_{i-x} = x}$

$S_1 = 3$

$\underline{S_i = 0}$

$S_4 = 0$

After second phase:

```
      ↓
3 2 3 0 2 6 0 0 0 0
```

The right-hand node is now the start node of a new skipblock continuing to the right, thus we have split the original skipblock into two. In the third phase we update the values of each node to the right of the right-hand node, beginning with a value of 2 and incrementing by 1 per node, until we either reach the end of the skipfield or a zero value. If the value of the node to the right of the right-hand node is zero or beyond the end of the skipfield, no update occurs. In the case of the above example, only a single update would take place:

Reuse (third phase):

$S_{i+2} = 2$

After third phase:

```
      ↓
3 2 3 0 2 2 0 0 0 0
```

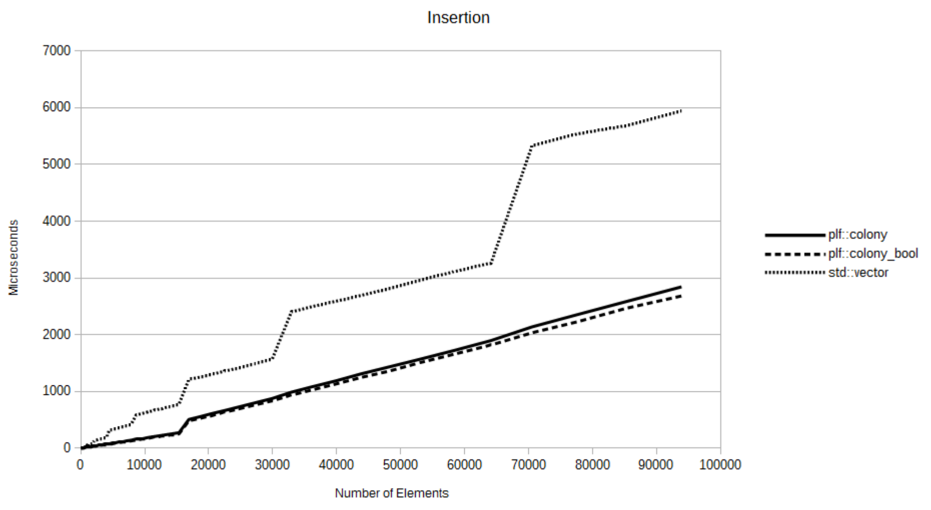
Reuse of the erased node is complete at this point and the skipfield can be iterated over correctly in both directions.

4 Performance results

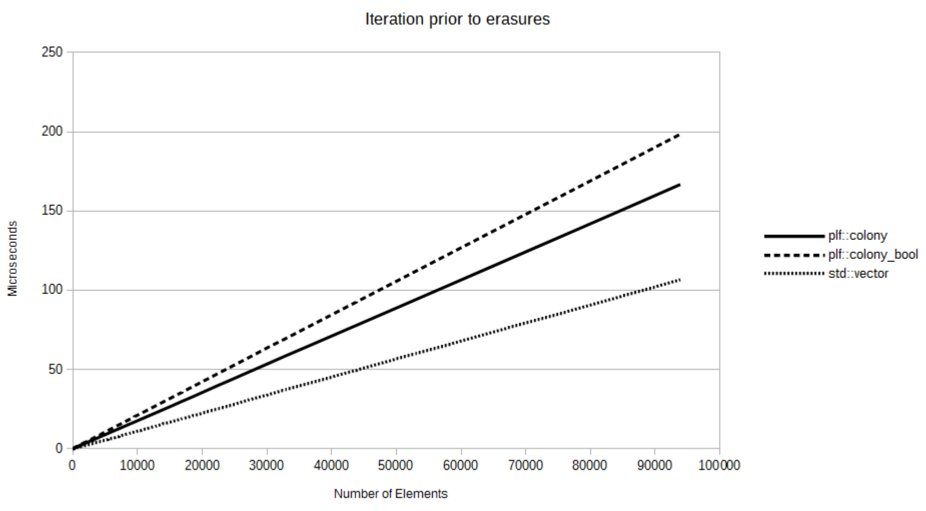
In this section we will benchmark (in C++) the performance of a colony container using a boolean field to indicate erasure, versus a colony container using a jump-counting skipfield to indicate erasure, and as a reference point, a std::vector. For brevity we will henceforth refer to the colony using the boolean skipfield as a “boolean colony”, and the colony using the jump-counting skipfield as a “jump-counting colony”. The std::vector will exhibit slow erasure and insertion speeds due to its need to reallocate elements to ensure element contiguity. Both colony types will exhibit fast insertion (due to the utilization of a multiple-memory-chunk allocation strategy, which does not necessitate the reallocation of elements once capacity is exceeded) and erasure times (due to utilization of a skipfield rather than reallocating all elements after the erased element) but corresponding slower iteration times (due to the partially non-contiguous element allocation caused by the multiple memory chunks, and the skipfield use).

For both colonies, each internal memory chunk has its own individual skipfield. The size of the boolean colony’s internal memory chunks is limited only by the amount of memory available to the system. The jump-counting colony’s skipfield consists of a series of 16-bit unsigned integers, and this reduces the maximum size of a colony’s internal memory chunks to 65535 elements each. Reducing the skipfield bit-depth means that more of the skipfield can then fit into the CPU’s cache at once, which generally increases performance. The boolean colony’s skipfield consists of a series of 8-bit unsigned integers, as this is the lowest integer size that a x86 CPU can access without bit manipulation (which would result in decreased performance on this architecture).

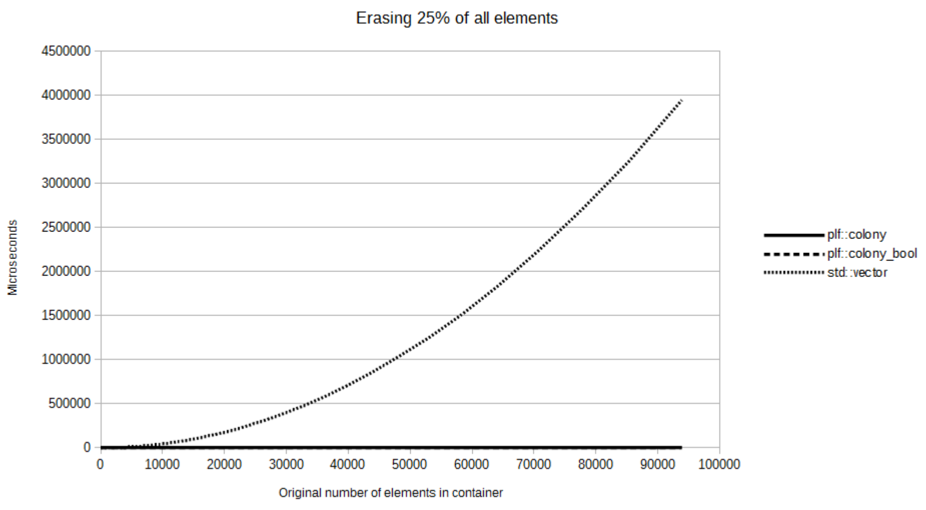
The test system is an Intel E8500 CPU with 8GB ram, running GCC 5.1 x64 as the compiler. Build settings are ”-O2;-march=native;-std=c++11;-fomit-frame-pointer”. Results for Visual Studio 2013 are similar. Tests are based on a sliding scale of number of runs vs number of elements, so a test using only 10 elements in a container will use 100000 runs and average the results, whereas a test with 100000 elements will use 10 runs and average the results. This tends to give reliable averages without requiring overly long test times. The insertion test measures inserting single elements sequentially, as opposed to inserting multiple elements at once. In the erasure tests we iterate through the container elements and erase single elements at random. A C++ “remove_if” pattern might be more common in this test if we were solely processing the std::vector, but such a pattern makes no difference to colony erasure performance, and so is omitted.

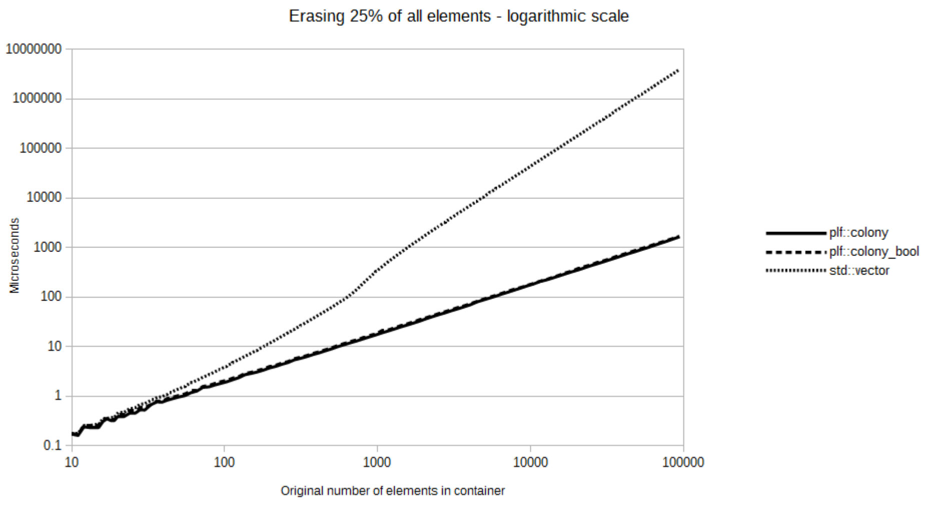


From the insertion results we can see that both colonies perform significantly better than std::vector, and the colony with the boolean skipfield slightly outperforms the colony with the jump-counting skipfield, due to its simpler procedures. At this point no erasures have occurred to either container, hence no changes to the colony skipfields have taken place. Below are the iteration results:

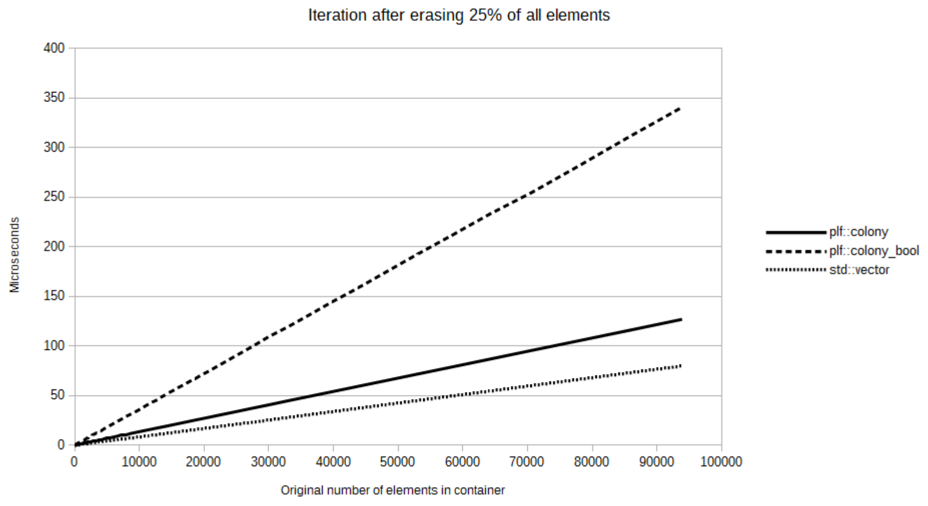


From this we can see that the std::vector comes in first, iterating approximately twice as fast as both colonies on average, with the jump-counting colony second and the boolean colony third, 15% slower than the jump-counting colony on average. At this stage the difference in speed between the two colonies is primarily due to the more complicated branching code necessary to iterate over the boolean skipfield compared to the simple addition needed for iteration over the jump-counting skipfield. Now we will measure erase performance when iterating over the containers and erasing 25% of all elements at random:



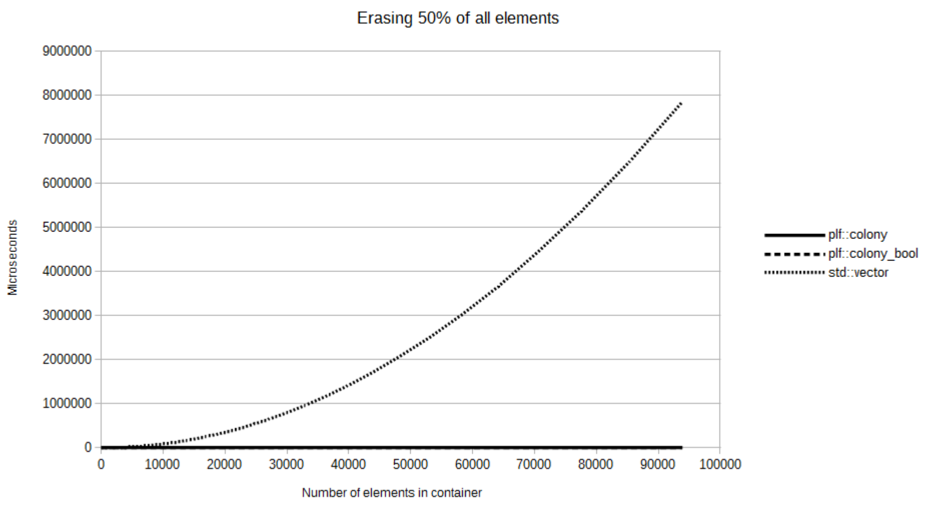


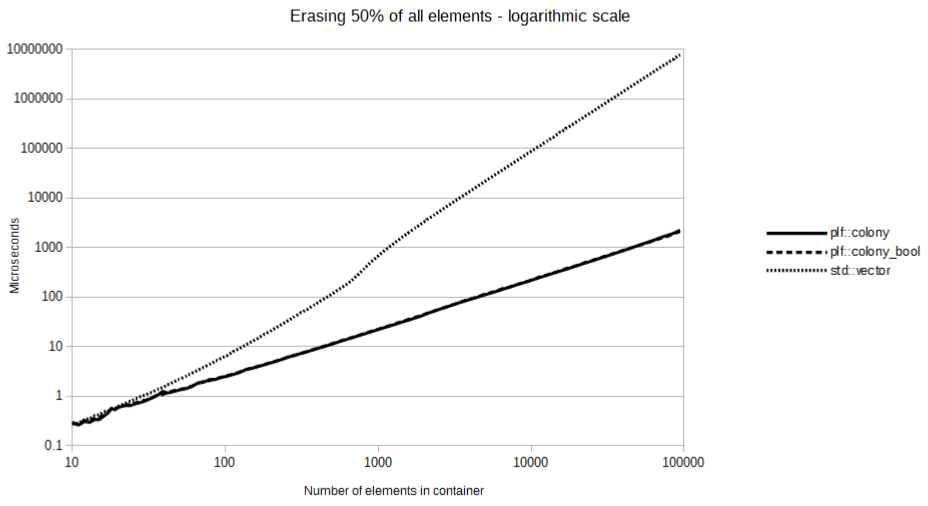
std::vector has predictably poor erasure performance due to its need to reallocate all subsequent elements after each erased element, in order to preserve element contiguity, and due to the random erasure pattern. Both colony containers perform equally well due to their lack of reallocation, requiring only a fraction of std::vector’s duration to complete. Now let’s see how iteration performance has altered:

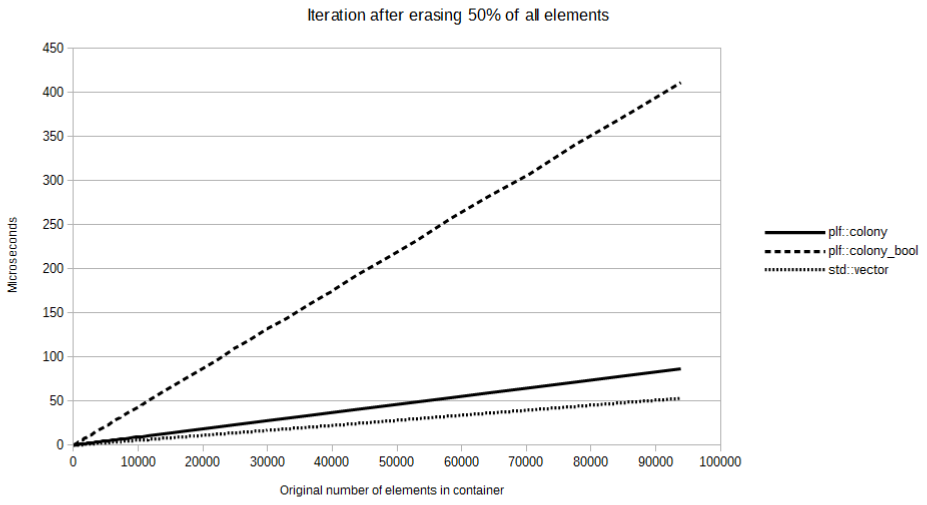


Immediately we can see a performance advantage for the jump-counting colony over the boolean colony. The jump-counting colony’s performance increases due to the reduced number of elements and skipfield nodes it is required to read, but the boolean colony’s performance almost halves compares to the iteration results prior to erasures. There are two reasons for this. Firstly, the boolean colony must check the value of each skipfield node to determine the erased status of the corresponding element, and hence has no ability in the case of a run of erased elements to directly skip from one non-erased element to the next. The jump-counting colony has this ability, and so its iteration speed is greatly increased.

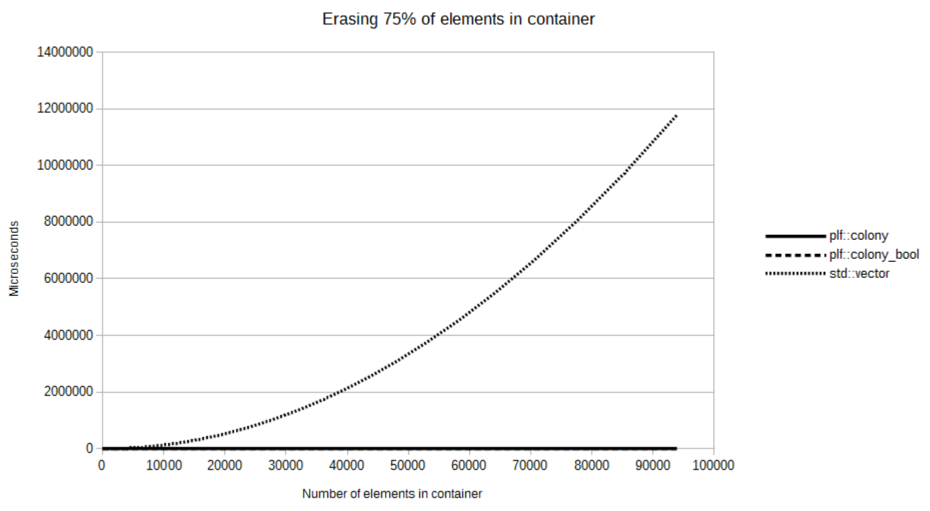
The second reason is that in the first iteration test there were no erased elements. And so, while the boolean colony had branching code (to enable skipping any individual element when its corresponding skipfield node indicated erasure), the CPU's branch prediction could lower the performance cost of the branching code significantly [5]. This is because there was only one path for the branch to take at this point due to a lack of erasures. But with 25% of all elements erased, the iterator will encounter a skipped element 25% of the time, also rendering the branch prediction which the CPU makes wrong 25% of the time and dramatically increasing the performance loss from the same branching code. To further explore the impact of these factors on performance, we will now increase the erasure percentage to 50% of all elements in the original containers:

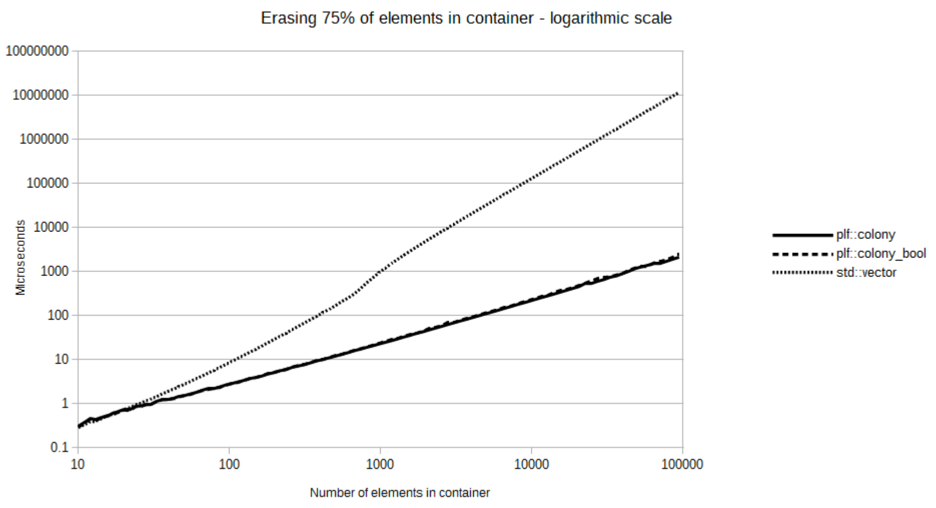


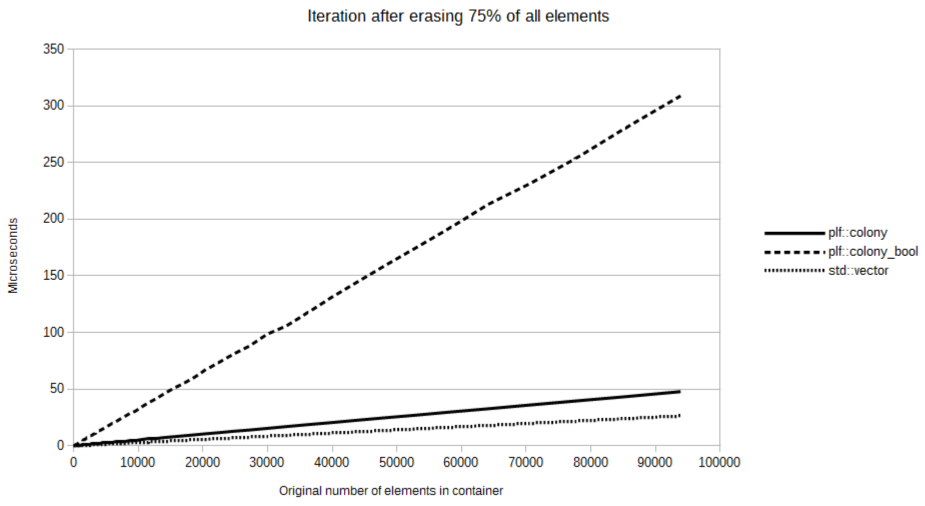




Here the boolean colony’s performance is worse than in the 25% erasure iteration benchmark, because with 50% of all elements erased at random there is no opportunity for the CPU’s branch prediction to be correct any more than 50% of the time. By contrast the jump-counting colony’s iteration performance continues to scale proportionally to the number of erasures, as does std::vector’s. Now we will erase 75% of all elements in the original containers:


Erasing 75% of elements in container
14000000
12000000
10000000
8000000
6000000
4000000
2000000
0
Microseconds
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
Number of elements in container
plf::colony
plf::colony_bool
std::vector


Erasing 75% of elements in container - logarithmic scale
100000000
10000000
1000000
100000
10000
1000
100
10
1
0.1
Microseconds
10
100
1000
10000
100000
Number of elements in container
plf::colony
plf::colony_bool
std::vector



Both the jump-counting colony and std::vector’s performance continues to scale proportionally to the number of erasures. The boolean colony in this case performs slightly better than in the 25% erasure iteration test, the reason being that with 75% erased elements there are 25% non-erased elements, so the CPU branch prediction can work just as effectively, but in this case only 25% of the original elements are having their values read, as opposed to the 75% being read in the 25% erasure iteration test.

Overall we can see that both the ability to skip directly from each active element to the next, and the lack of reliance on CPU branch prediction, play large roles in the performance of a jump-counting pattern compared to a simple boolean pattern. On a processor without branch prediction, we can expect the boolean pattern to perform worse in the majority of cases, with average performance being closer to that of the 50% erasure iteration test.

5 Additional areas of application and manipulation

While this document’s primary focus is on the application of a jump-counting skipfield in the context of a data container, in this section we will outline some other potential applications.

5.1 Binary Erasure Channels

It is unlikely that this pattern would yield any bandwidth benefits over a standard Binary Erasure Channel [7] in most scenarios, except in the case of signals where erasure is frequent but tends to occur in runs longer than 2. In this case a compressed simple jumping-counting pattern block as described in section 5.3 below might reduce the overall bandwidth necessary to transmit erasure information for a given block of received data. However this is a highly-studied area of information theory and hence it is likely that a more optimal and generic solution is available.

5.2 Available/busy unit indication

As noted earlier in section 1, the skipfield pattern can be used to iterate over any sequence of elements where the decision to skip a given element depends on the answer to a specific boolean question, regardless of whether that question is "is the object black or white" or "is the object erased or not", or "is the object available or unavailable", etcetera. It is reasonable to assume that there exist real world scenarios where data in a sequence might not actually be erased but instead be made *temporarily* inactive or unavailable, and where the application is required to iterate over the currently-active or available data in a performance-friendly fashion. The advanced jump-counting skipfield pattern is a good fit for such usage cases, enabling fast iteration over individual objects while not significantly impacting the performance of object activation/deactivation when compared to a boolean field.

Take for example a large data center, with perhaps 1,000,000 hard drives in various configurations, all being accessed asynchronously by various sources. To achieve maximum performance and lowest-possible latency, we would ideally want to assign each task to a hard drive which is not currently being utilized, rather than adding requests to a queue for a particular hard drive; though the latter may become necessary if the data center's system was to be saturated with requests. The timing for when two similar requests actually complete on each individual hard drive is also asynchronous, due to the nature of hard drive storage. For this reason a first-in-first-out strategy for determining available hard drives is not technically feasible, and we would instead be inclined to use a randomly-updatable skipfield, to indicate available/busy status for each hard drive.

If we use a simple boolean field to indicate which hard drives are currently available or busy, this presents an immediate problem: for any given request, we could be making anywhere between 1 and 999,999 boolean field checks in order to find an available hard drive. This is obviously insufficient as: (a) the time complexity is O(random), making latency highly variable and, (b) the solution is slow. If we use an advanced jump-counting skipfield pattern instead, we will only ever require a single field check to find the next available hard drive, in other words an operation with O(1) time complexity. And while the operations to switch any given skipfield node between "busy" and "available" will take longer when compared to a boolean state change, the overall time-saving will still be in the factors of 10, as the benchmarks in section 4 show, due to the increased speed of finding an available drive with a jump-counting skipfield.

In extreme cases where almost all of the available hard drives are busy, the above solution could result in slow state change operations, as depending on the location of the changed node and the saturation of requests, anywhere between 1 and 999,999 skipfield nodes might need to be updated. Because these updates don't involve significant branching, it is expected that the performance cost of this would still be low. But to ameliorate any performance detriment we could split the original group of 1000000 drives into smaller clusters of 65536 drives and create individual skipfields for each group. This would reduce the necessary number of potential skipfield writes required for any given state change operation (from busy to available or from available to busy), and also reduce the necessary skipfield bit-depth from 32-bit to 16-bit, resulting in potentially faster writes and reads as more of the skipfield will fit into the CPU’s cache. The jump-counting colony container used in the benchmarks within section 4 also uses this strategy.

This increases the potential number of reads from the skipfield groups (in order to find an available hard drive) from 1 to between 1 and 16. But it reduces the maximum potential number of writes to the skipfield (in the event of a hard drive state change) from between 1 and 999,999 32-bit writes, to between 1 and 65535 16-bit writes. To further reduce the potential number of reads down to 2, we could also apply a jump-counting skipfield to the collection of groups itself, using the binary state of “all drives busy” / “some drives available”. Thus the first read would determine the first group with available drives, and the second would find the first available hard drive within that group. This subdivision could be taken further still, creating 3907 groups containing 256 hard drives each, using 8-bit skipfields within the groups to indicate available drives, and a 16-bit skipfield across the groups to indicate whether any given group has an available drive.

This last subdivision would keep the number of reads necessary to find an available drive at 2, but would reduce the potential number of writes required for a drive state change from between 1 and 65535 16-bit writes, to between 1 and 256 8-bit writes (and 1 potential 16-bit write if the entire group is saturated). The greater the subdivision, the more operations can occur simultaneously within the entire system, because each group has its own skipfield which is independent of the other groups. Therefore mutexes [6] can be applied to individual groups rather than to the data structure as a whole when processing multiple hard drive requests synchronously. Benchmarking would be necessary to find the optimal strategy in any given case and on any given platform.

A keen observer might note that a similar strategy could be applied to a boolean skipfield, by subdividing recursively into 16 groups of 65535 drives, then dividing each group into 256 sub-groups of 256 drives, and using boolean flags at both the group and subgroup levels to indicate which groups/sub-groups have available drives left. This approach does not have quite the same level of read-reduction as the jump-counting skipfield approach, reducing the maximum number of reads to find an available drive from 999,999 32-bit reads to 528 8-bit reads (16 (group level) + 256 (sub-group level) + 256 (drive level)). But it also does not have the write costs of the jump-counting approach; a maximum of 3 8-bit writes is necessary for a drive state-change in a boolean skipfield using 3 levels of subdivision, versus a potential maximum of 256 8-bit writes for a single drive state change in a jump-counting skipfield.

However the benchmarks in section 4 above clearly show that even when erasing 75% of all available elements at random for large sets of data, the performance impact is low and does not affect overall erasure performance compared to a boolean skipfield. Moreover the benchmarks also show that the branching necessitated by a boolean skipfield has a significant impact on performance even when the number of reads is low. Based on this evidence, the performance advantage would still belong to the jump-counting skipfield.

5.3 Compression

In cases where the jump-counting pattern is used to indicate a binary aspect of elements which is not erasure/non-erasure, there could conceivably be a necessity to store the skipfield somewhere other than in active memory, for example, when exiting a program. In this case we might choose, either for performance reasons or due to storage limitations, to compress the skipfield pattern as opposed to storing it raw. Any skipfield can be seen as a series of alternating runs of active (zero) and inactive (non-zero) elements ie. a non-zero run always follows a zero run, and vice-versa. In the context of a jump-counting skipfield the length of a run will never be larger than the largest possible number afforded by the bit-depth of the skipfield, as that number must match the maximum number of elements in the sequence as described in section 2.1.

Hence, one compression strategy might take the form of a series of unsigned integers of the same bitdepth as the original skipfield. The first number in the compressed skipfield indicates whether the first run in the sequence is a zero or non-zero run, accordingly noting either 0 or 1. Each subsequent number notes the number of elements in the alternating zero/non-zero runs respectively. Taking the following skipfield:

3 2 3 0 2 2 0 0 0 0 3 2 3

We would compress by starting with "1" to indicate the non-zero nature of the first run, then follow with the length of each run in sequence. We end up with:

1 3 1 2 4 3

This is obviously a form of run-length encoding albeit without the need to specify the value of the run whose length we are denoting (aside from the first run's value). We can perform subsequent compression on this field which would typically result in greater compression than if we had simply compressed the original field. For example, the sequence above could subsequently then be Huffman-coded[8] to produce a compressed skipfield of 12 bits. Because the original sequence in this case is so short, once we added a Huffman table for decompression, the resulting compressed pattern would be larger than if we had simply binary-encoded the original field as a boolean sequence with a bit-depth of 1. But for any field of substantial length, the Huffman-encoded solution would be smaller.

As an extreme example, consider a data set with 100,000 elements and alternating patterns of 50 erased and 50 non-erased elements. Using the above technique, the original skipfield of 100,000 integers would compress to 2001 unsigned integers, each one (other than the initial zero/non-zero indicator) equal to 50, which Huffman-coding would further reduce to 2001 bits. Compared to an equivalent binary-encoded 1-bit boolean field of the original field, this is a reduction of 98%, and a run-length encoding pass would reduce this further still. Such a test is an atypical and artificial pattern which may not reflect real-world usage, but shows the maximum potential effectiveness of the above compression strategy.

5.4 Parallel Computing

The jump-counting algorithms are principally serial in design and as such are unlikely to be of significant use in parallel architectures such as CUDA[9]. Data processing in such environments is typically performed on many elements at once rather than iterating over single elements, and single iteration is where the jump-counting skipfield's performance advantages are found. But neither does the pattern necessarily hinder parallel usage. As an example, if one were to use an advanced jump-counting skipfield instead of a boolean skipfield to indicate inactive data in a parallel-processing environment, this would not prevent a "compact" operation utilising an exclusive[10] or inclusive[11] scan. One would merely predicate any non-zero value in the skipfield as a zero, and any zero value as a one in the context of the scan. (note: the simple jump-counting pattern would not be usable in this context as nodes within skipblocks in the simple pattern may contain zero values).

6 Summary

Given an appropriate use case, implementing a "jump-counting" rather than a boolean skipfield will reduce iteration times substantially where multiple consecutive skipped elements are common. While the pattern will typically exhibit better performance in a multiple-memory-block situation where block-sizes can be attenuated to allow for a smaller bit-depth in the skipfield nodes (for example, lowering maximum memory-block size to 65535 elements allows for the use of a 16-bit skipfield), it can be expected to still show strong performance benefits with larger memory blocks requiring higher bit-depths, if multiple consecutive erased elements are common.

The pattern can be applied in any scenario where a singular binary aspect of each element in a sequence decides whether or not to skip that element. This can include available/busy, erased/non-erased or any other two-value choice. It is expected to be of greatest use in video game engines, where boolean fields are traditionally use in this capacity. The cost for erasure is low and very close to that of a boolean skipfield's in the majority of scenarios. In scenarios where element erasure is infrequent, a boolean skipfield utilizing a lower bit-depth in greater simplicity of development. However, for the majority of game development scenarios, element insertion, erasure and iteration over elements are among the most common per-frame activities, and so any substantial performance increase in those areas can be expected to yield overall performance improvements.

Acknowledgements

I would like to thank both Dr Gisela Klette of The Auckland University of Technology and Dr Robert Hurley for their invaluable advice and support in the critiquing and editing of this document.

References

1. Bulka, Dov, and David Mayhew., *"Efficient C++: performance programming techniques."*, Addison-Wesley Professional (2000).
2. Marc Gregoire., *"Professional C++"*, John Wiley & Sons (2014)
3. Ghosh, Somnath, Margaret Martonosi, and Sharad Malik., *"Cache miss equations: An analytical representation of cache misses."*, Proceedings of the 11th international conference on Supercomputing. ACM. (1997).
4. Goldthwaite, Lois., *"Technical report on C++ performance."*, ISO/IEC PDTR 18015 (2006).
5. Arun Kejariwal, Center for Embedded Computer Systems University of California (Irvine, USA), Alexander V. Veidenbaum, Alexandru Nicolau, Xinmin Tian, Milind Girkar, Hideki Saito, Utpal Banerjee, *"Comparative architectural characterization of SPEC CPU2000 and CPU2006 benchmarks on the intel Core 2 Duo processor"*, Embedded Computer Systems: Architectures, Modeling, and Simulation, 2008. SAMOS 2008. (2008)
6. Courtois, Pierre-Jacques, Frans Heymans, and David Lorge Parnas, *"Concurrent control with 'readers' and 'writers'."*, Communications of the ACM 14.10, pp 667-668. (1971)
7. David J. C. MacKay, *"Information Theory, Inference, and Learning Algorithms"*, Cambridge University Press. (2003)
8. D.A. Huffman, *"A Method for the Construction of Minimum-Redundancy Codes"*, Proceedings of the I.R.E., pp 10981102. (1952)
9. Nvidia, *"C. U. D. A. Programming guide."*, (2008).
10. Blelloch, Guy E., *"Prefix sums and their applications."*, (1990).
11. Hillis, W. Daniel, and Guy L. Steele Jr. , *"Data parallel algorithms."*, Communications of the ACM 29.12 (1986): 1170-1183.