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but in particular within game engines, there exist scenarios where boolean fields are used to indicate items which are no longer active or useable, instead of actually erasing or removing these elements, as erasure tends to come with one or more side-effects depending on the type of data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 of this is having a raw array of data for individual enemy objects in a video game, with an additional boolean field named "active" in each object, setting the "active" flag to false when that enemy object is destroyed in the game. Subsequent to this action, in terms of iteration that object would be skipped during 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 vector or deque is used, actual erasure of elements typically causes strong performance detriments [1, Standard Template Library, p. 164] due to the reallocation of subsequent elements in order to preserve data contiguousness, the process of which also (c) invalidates pointers to data within the container [2, Sequential Containers, p. 485]. This last point is of high 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 stability thereby preserved, along with 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 worthwhile, simple, but inefficient for any container with large amounts of consecutive "inactive" elements, as iteration over these requires checking a boolean field for each inactive element, with no means by which to simply skip from each "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 table 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 a thousand, 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 detriment 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 status, greyscale/coloured status, living/dead status or non-erased/erased status. 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 homogenous 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 homogenous. But 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, for the majority of the remainder of this document.

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 which this pattern was developed for, utilises linked chains of increasingly-large element memory blocks with accompanying advanced jump-counting skipfields which allows the container to maintain pointer/iterator stability during both insertion and erasure, and to reuse erased element memory space, while maintaining fast iterative performance. But any programmer implementing a custom solution (using arrays with skipfields, for example) might choose the simple pattern, depending on circumstance, as it has a very small erasure performance advantage.

1.1 Pre-existing Research

An extensive review of the literature has shown no similar pre-existing work on replacements for boolean skipfields. An exhaustive search of scholarly journals across 8 high-profile computer science journal aggregators did not yield any similar patterns, nor did it show any existing research into the use of boolean fields to indicate element erasure in data sets rather than the more common removal of elements from data sets and consolidation of that set. One possible explanation for this is that interaction between 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, which typically are hardware-bound and as such have a lesser focus on generalized solutions for arbitrary hardware and 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 integer type must be of a sufficient bitdepth to describe the number of elements that can be potentially stored within the memory block it is associated with. So a memory 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:

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

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

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

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

From this point onwards 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 set of elements utilising a jump-counting skipfield to identify erased elements, an iterator must keep track of both the current element it's pointing to and the current 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 needs to keep track of the index number (i), but for performance reasons we might choose to implement using both a pointer to the element array (e) and a pointer to the skipfield array (s).

If we use the pointer approach, to iterate an iterator forward 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 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 elements to skip in a single iterative step. This number is equivalent to the number of consecutive erased elements at that point in the skipfield. For the remainder of this document we will discuss the jump-counting pattern utilizing an index implementation for the purposes of simplicity. 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 at the 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 the skipfield node corresponding with the element being erased:

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

```
    * | *
0 0 0 0 0 0 0 0 0 0
```

If there is an erased node on the left, that node is the end node of a skipblock on the left. If there's an erased node on 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 on 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, or
4. both left and right are non-zero.

We now examine how each of those scenarios is 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 start node's index, and 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 number to the right 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. Subsequently 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 in the middle of two separate skipblocks, one on the left and another on the right. We subtract the left-hand node's value from the current node's index number to find the left skipblock start node's index. We increment the value of that start

node by 1 plus the value of the current node's right-hand node. Then we add the right-hand node's value to the current node's index to find the right skipblock end node's index, and 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 principles are straightforward but they do not support reuse of memory space from previously erased elements, which is what the advanced pattern facilitates. Because the simple pattern involves fewer calculations than the advanced pattern, it could be slightly faster in the 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 1" pattern as the example below demonstrates:

0 0 4 2 3 4 0 0 0 0

The formation of this pattern takes place in the erasure section below, while it's purpose becomes apparent in the subsequent restoration or "reuse" section. What it shows is the index of any given node within the skipblock, which tells us where to find the start node. 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 that information is enough to be able to update the entire skipblock, as we shall see.

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 node corresponding to the element we want to erase.

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 subsequent 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:

```
    |
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:

```
    |
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 left skipblock start node's index. 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. 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 see the "increment-by-1" pattern which becomes 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: if I want 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 happens:

Before reuse:

```
      |
6 2 3 4 5 6 0 0 0 0
```

(Incorrect) reuse:

$S_i = 0$

After reuse:

```
      |
6 2 3 0 5 6 0 0 0 0
```

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

Before increment:

```
      |
6 2 3 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 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 2 3 4 0 0 0
```

As noted earlier, 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 subsequently we can 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

If only the left node is non-zero, this indicates 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 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 0 0 0 0
```

Another example - before reuse:

```
  |
0 2 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

If only the right node is non-zero, this indicates 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, and 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, starting with a value of 2 and incrementing by 1 per 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 the skipblock. The end result of the operation must be to split the singular skipblock into two skipblocks. To do so we combine the behaviour 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, and set 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, starting from the node after the right-hand node, update the values of each node, starting with a value of 2 and incrementing by 1 per node, till either we reach the end of the skipfield or a zero value. If the value of the node after the right-hand node is either zero or beyond the end of the skipfield, no update occurs. For the example above, only a single update would take place at this point.

Reuse (third phase):

$S_{i+2} = 2$

After third phase:

```
      |
3 2 3 0 2 2 0 0 0 0
```

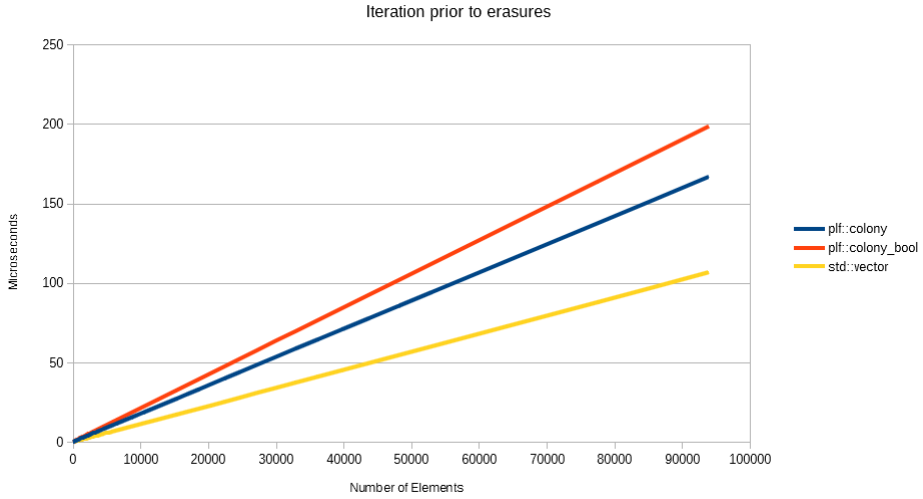
Reuse of the erased node is complete at this point and the skipfield can be iterated over correctly in both forward and reverse directions.

4 Performance results

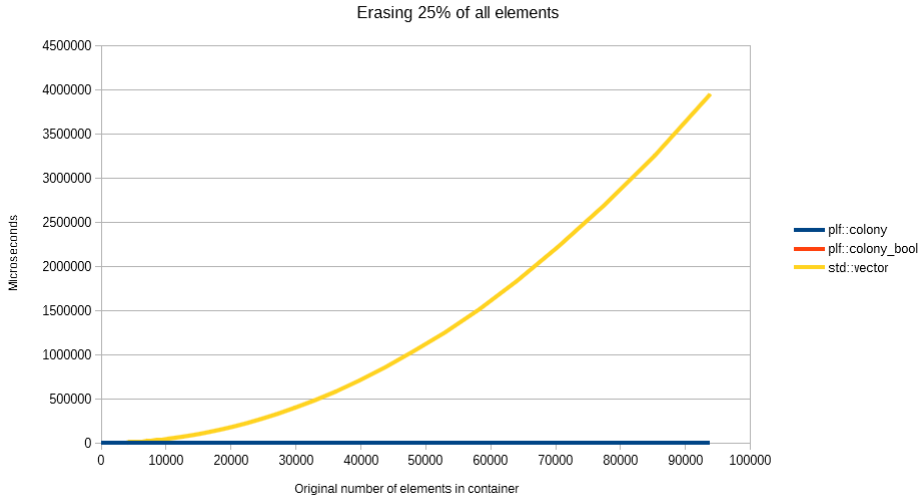
Below we benchmark in C++ the results for a colony container using a boolean field to indicate erasure, versus a colony container using a jump-counting skipfield to indicate erasure, and as a reference point, a std::vector. The std::vector will have slow erasure and insertion speed due to it's need to reallocate subsequent elements to ensure element contiguity, but fast iteration speed due to it's entirely contiguous element storage, which gives performance similar to a regular array. Both colony types will have fast insertion and erasure times but corresponding slower iteration times.

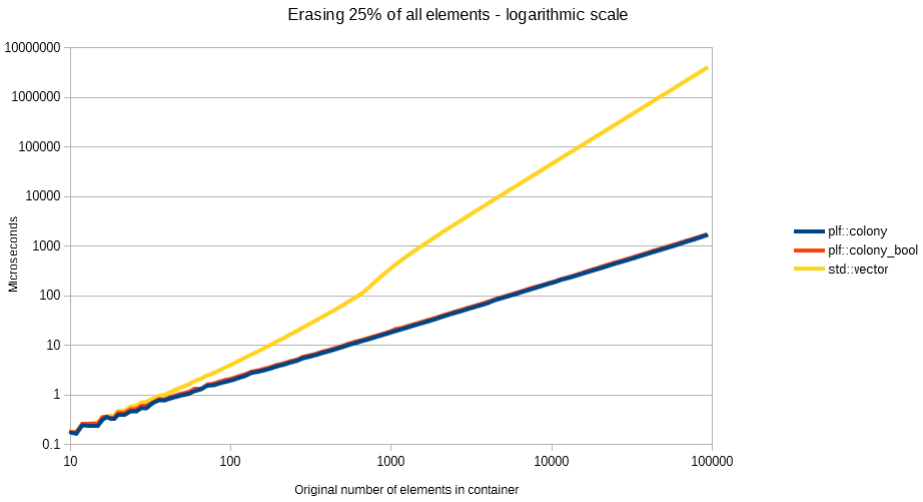
The test setup is an Intel E8500 CPU, 8GB ram, running GCC 5.1 x64 as compiler. Build settings are "-O2;-march=native;-std=c++11;-fomit-frame-pointer". Results for Visual Studio 2013 are similar. Tests are based on a sliding scale of number of runs vs number of elements, so a test using only 10 elements 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 overly lengthening test times.

We will ignore insertion benchmarks as the performance differences are not substantial between the boolean colony and the jump-counting colony, and focus on iteration and erasure results. In the erasure tests we are iterating through the container elements and erasing at random, rather than utilizing a C++ "remove_if" erasure pattern, as might be more common or useful if we were solely processing the std::vector. Here are the iteration results prior to any erasures:

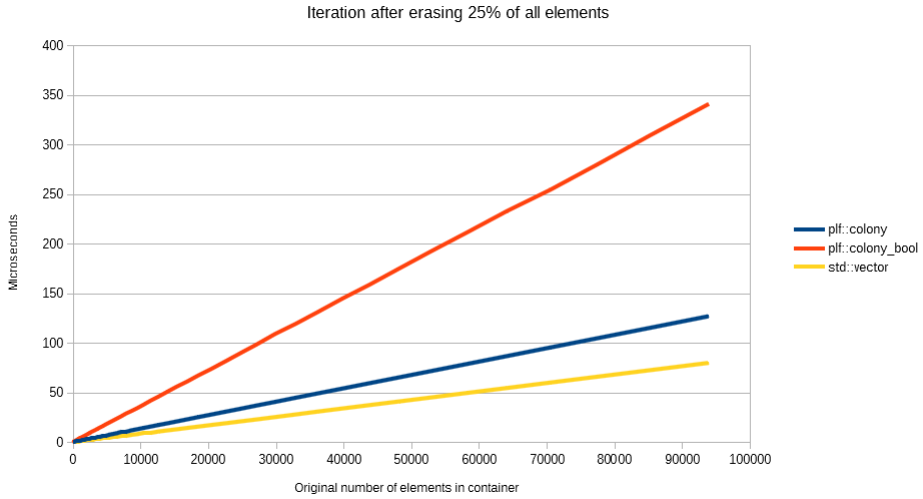


From this we can see that the std::vector comes in first, with the jump-counting colony second and the boolean colony close behind. At this stage the difference in speed between the last two is likely down to the need for branching in the boolean iteration compared to the simple addition necessary for the jump-counting iteration. Now we will measure erase performance when iterating over the containers and erasing 25% of all elements at random:





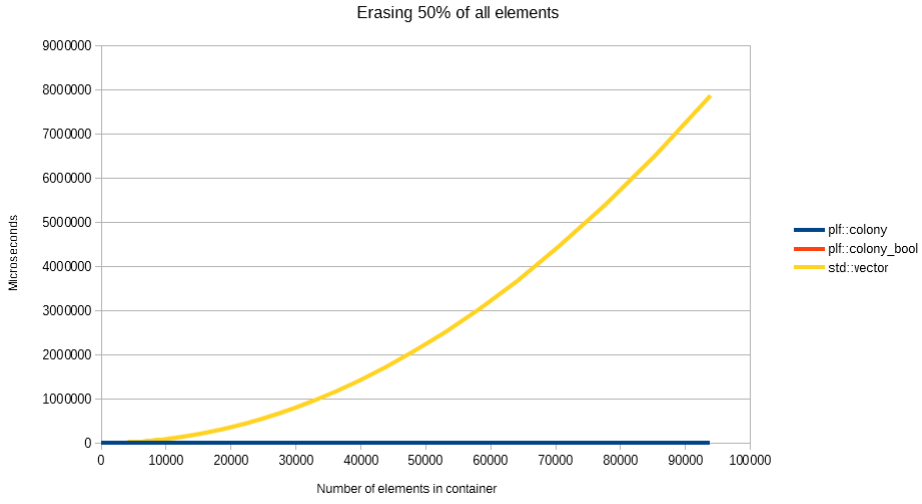
Std::vector has predictably poor erasure performance due to it's need to reallocate all subsequent elements after the erased element, in order to achieve element contiguousness for iteration speed, and due to the random erasure pattern. Both colony containers perform equally well due to their lack of reallocation, requiring a fraction of std::vector's time. Now let's see how the performance of iteration over the elements has altered:

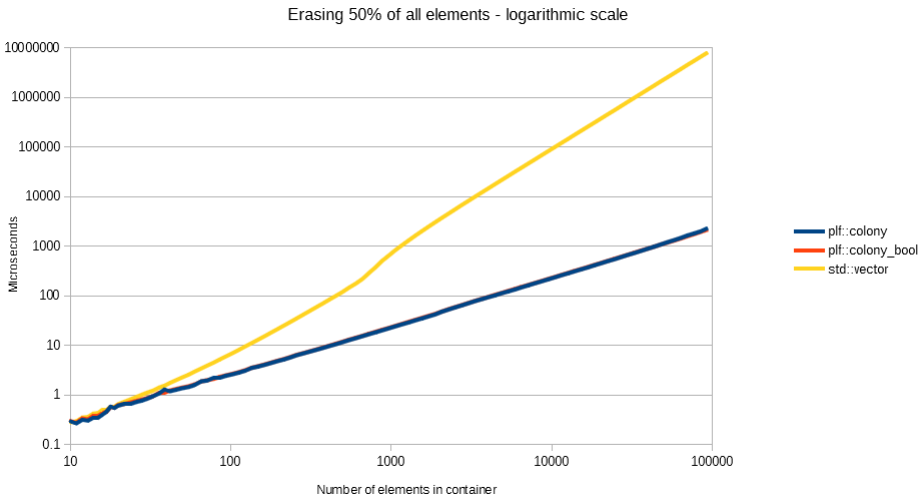


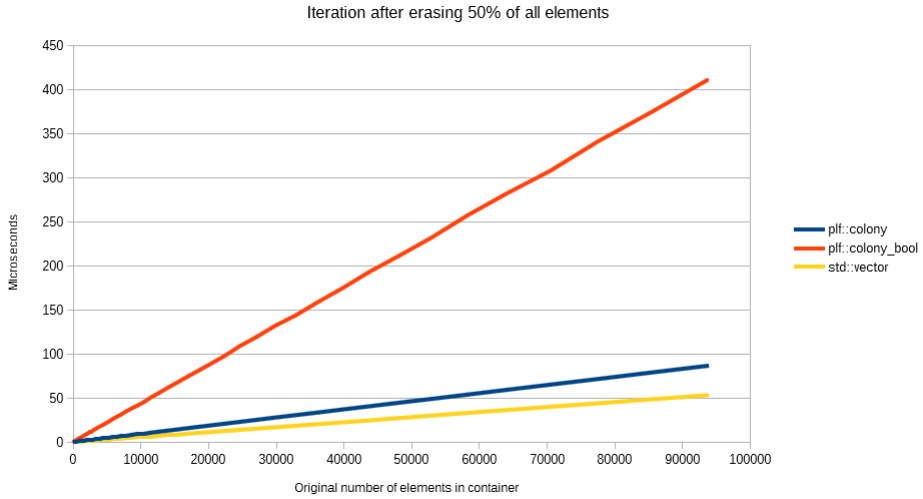
Immediately we see a performance advantage for the jump-counting colony over the boolean colony. The jump-counting colony's performance increases due to the fewer number of elements to read, but the boolean colony's performance almost halves. There are two reasons for this; firstly, the boolean colony 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 simply skip from one non-erased element to the next non-erased element. The jump-counting colony has this ability, and so iteration speed is greatly increased in this way.

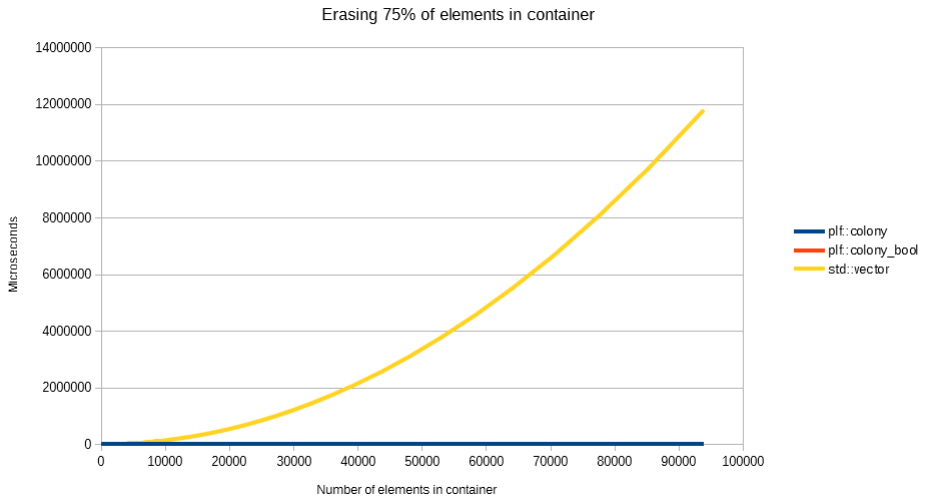
Secondly, in the first iteration test there were no erased elements and hence while the boolean colony had branching code (to enable skipping of an individual element when it's corresponding skipfield node indicated it was erased), the CPU's branch prediction could lower the performance cost of the branching code [5], minimizing cache misses, as there was only one path for the branch to take at this point, due to lack of erasures. But with 25% of all elements erased, the iterator will detect a skipped element 25% of the time, the default branching that the CPU will predict is therefore wrong 25% of the time, which dramatically increases the cost of the branching code. We will now show this effect further by erasing 50% of all elements in the original containers at random:

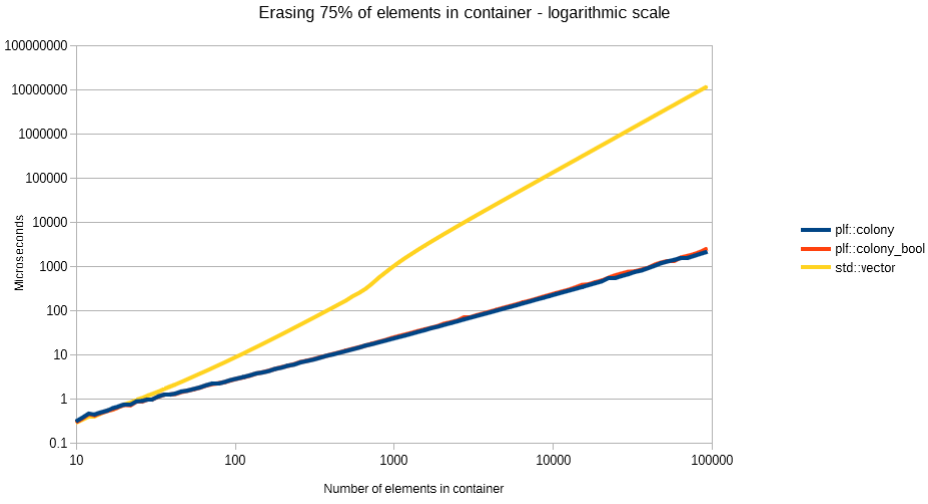


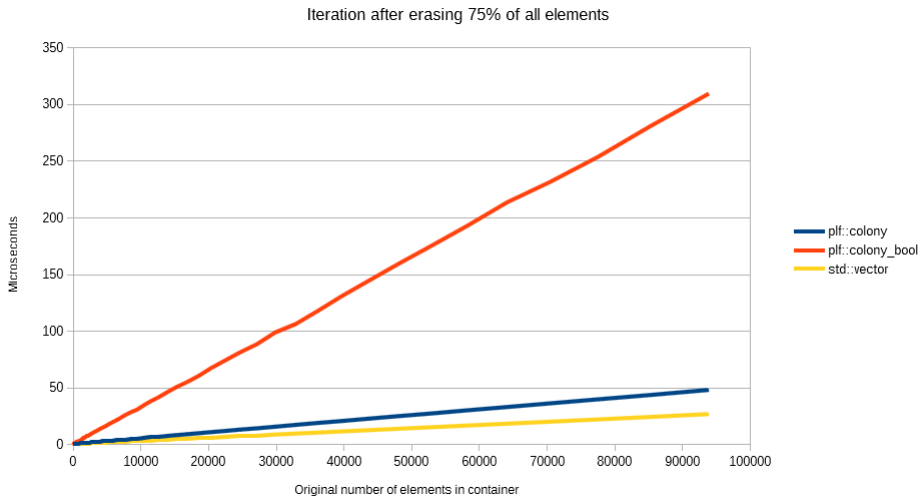




Here the boolean colony's iteration performance is worse than the 25% erasure performance, because for 50% randomized erasures there is no chance for the CPU's branch prediction to be correct any more than 50% of the time. The jump-counting colony's iteration performance continues to scale proportionally to the number of erasures, as does the std::vector's. Now we erase 75% of all elements in the original containers:


Erasing 75% of elements in container
Microseconds
14000000
12000000
10000000
8000000
6000000
4000000
2000000
0
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
Microseconds
100000000
10000000
1000000
100000
10000
1000
100
10
1
0.1
10
100
1000
10000
100000
Number of elements in container
plf::colony
plf::colony_bool
std::vector



Both the jump-counting colony and the std::vector continue 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 likely reason being that at this point the CPU branch prediction can work at least as adequately as it did with the 25% erasure iteration, but unlike that benchmark, in the case iterating when 75% of all elements have been erased, only 25% of the original elements are having their values read, as opposed to the 75% of all original elements having their values read, in the 25% erasure iteration test.

Overall we can see that not only the lack of branch code but also the lack of reliance on CPU branch prediction plays a large role in the performance of a jump-counting pattern, over a simple boolean pattern. On a CPU without branch prediction, we can expect the boolean pattern to perform worse in the majority of cases, with the performance being closer to the 50% erasure iteration test. The rest of the performance loss is made up of the number of reads necessary for the boolean skipfield, versus a jump-counting skipfield.

5 Additional areas of application and manipulation

While this document focuses primarily on the application of a jump-counting skipfield in the context of a data container, here we focus on 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 [6] 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 further in the

section 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 computer science and it is likely that more optimal and generic solutions to this problem have already been discovered.

5.2 Available/busy unit indication

As noted earlier, 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 non-erased", or "is the object available/unavailable", etcetera. It is rational to assume that there are situations 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 fits well for this domain, enabling fast iteration while not affecting the speed of activating/deactivating items significantly, compared to a boolean field.

Take for example a large-scale data center, with maybe 1000000 hard drives in various configurations, all being accessed by various sources asynchronously. 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 cue for a particular hard drive; though the latter may become necessary if the data center's system was saturated with requests. When these requests actually complete on each individual hard drive is also asynchronous; two similar requests are not guaranteed to take the same length of time to complete, 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 possible, and we would likely use a randomly-updatable skipfield instead, 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 a problem: for any given request, we could be making anywhere between 1 and 999999 boolean field checks 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. Subsequent data requests to the system will be delayed while the first request completes it's search for available drives and updates the field, further increasing latency in the request pipeline. If we use an advanced jump-counting skipfield pattern instead, we will only ever require 1 check to find the next available hard drive. It becomes an O(1) operation, and while the operations to switch a given skipfield node between "busy" and "available" take longer compared to a boolean state change, as the benchmarks above show the overall time-saving will be in the factors of 10.

In extreme cases, where almost all of the available hard drives are busy, the above solution could result in slow state change operations, as anywhere between 1 and 999999 skipfield nodes might need to be updated depending on the location of the changed node and the saturation of requests. To ameliorate

this performance detriment we could split the original group of 1000000 drives into smaller clusters of 65536 drives, and create individual skipfields for each of these groups. This reduces the necessary number of potential skipfield writes required for any given state change operation (from busy to available or from available to busy), and reduces the necessary size of the skipfield bitdepth from 32-bit to 16-bit, resulting in potentially faster writes and reads, as more of the skipfield will fit in a CPU's cache. The colony container used in the benchmarks above 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 999999 32-bit writes, to between 1 and 65535 16-bit writes. To reduce the potential number of reads down to 2, we can apply a jump-counting skipfield to the groups themselves, using the binary state of "all drives busy"/"some drives available". Therefore the first read would determine the first group with available drives, and the second read would find the first available hard drive within that group. This subdividing could be applied even further, 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 reduces the potential number of writes necessary for a drive state change, down 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 is independent in terms of it's skipfield and therefore mutexes can be applied to individual groups rather than to the data structure as a whole, when processing multiple hard drive requests at once. Benchmarking would be necessary to find the optimal strategy.

An observer might note that a similar strategy could be applied to a boolean skipfield, subdividing recursively into 16 groups of 65535 drives, then dividing each group into 256 sub-groups of 256 drives, using boolean flags at both 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, instead reducing the maximum number of reads to find an available drive down from 999999 32-bit reads to 528 8-bit reads (16 (group level) + 256 (sub-group level) + 256 (drive level)). But they might further note that 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 the section above clearly show that even when erasing 75% of all available elements at random for large sets of data, the statistical probability of large numbers of writes during erasure for a jump-counting skipfield is quite low and does not affect overall erasure performance compared

to a boolean skipfield. And the benchmarks also show that the probability of large numbers of reads during iteration for a boolean skipfield is quite high. Based on this, the performance advantage would overwhelmingly belong to the jump-counting skipfield.

5.3 Compression

In a context 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 to compress the skipfield pattern as opposed to storing it raw, for storage limitations or performance reasons. 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 bitdepth of the skipfield, because the bitdepth matches the greatest possible number of elements.

Hence, one compression strategy takes 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 without the need to specify the value of the data whose run we are enumerating (asides from the initial starting value). We can perform subsequent compression on this field which would typically result in more compression than if we had compressed the original field. For example, the sequence above could subsequently then be huffman-coded[7] to produce a compressed skipfield of 12 bits. Because the original sequence in this case is so short, once we add a huffman table for decompression, the resulting compressed signal 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, take the last test detailed in the performance section of this document, with 100000 elements and alternating patterns of 50 erased

and 50 non-erased elements. The original skipfield of 100000 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 even further. Though this test is an atypical and artificial pattern, which may not reflect real-world usage in any particular scenario, it at least shows the potential extent of overall compression.

5.4 Parallel Computing

The jump-counting algorithms are principally serial in design and as such are unlikely to find significant application in parallel architectures such as CUDA[8]. Data processing in those environments is typically performed on many elements at once rather than progressing iteratively through those elements, and progressive iteration is where the jump-counting skipfield's performance advantages are held. 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[9] or inclusive[10] 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 context, applying a jump-counting skipfield rather than using a boolean skipfield will reduce iteration times substantially where multiple consecutive skipped elements are common, and can be considered in any area of development where a boolean skipfield might typically be used. While the pattern will typically have better performance in a multiple-memory-block situation where the block-sizes can be attenuated to allow for a smaller bitdepth in the skipfield nodes (for example, lowering maximum memory-block size to 65535 elements would allow the use of a 16-bit skipfield), it can be expected to still have a strong performance impact with larger memory blocks requiring higher bitdepths, if multiple consecutive erased elements are common.

The pattern can be applied to any scenario where a singular binary aspect of each element in a sequence of elements decides whether or not to skip that element. This can include available/busy status, erased/non-erased status or any other two-value choice. It is expected to be of most use in video game engines, where boolean fields are traditionally use in this capacity. The cost for erasure is low and very close to a boolean skipfield's in the statistical majority of scenarios. In scenarios where element erasure is infrequent, a boolean skipfield utilising a

lower bit-depth may result in better performance as well as greater simplicity of development. However for the majority of game development scenarios, element insertion, erasure and iteration over elements are some of the most common per-frame activities, so any substantial performance increase in those areas can be expected to yield overall performance improvements.

Acknowledgements

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

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. David J. C. MacKay, *"Information Theory, Inference, and Learning Algorithms"*, Cambridge University Press. (2003)
7. D.A. Huffman, *"A Method for the Construction of Minimum-Redundancy Codes"*, Proceedings of the I.R.E., pp 10981102. (1952)
8. Nvidia, *"C. U. D. A. Programming guide."*, (2008).
9. Blelloch, Guy E., *"Prefix sums and their applications."*, (1990).
10. Hillis, W. Daniel, and Guy L. Steele Jr. , *"Data parallel algorithms."*, Communications of the ACM 29.12 (1986): 1170-1183.