Linear Hashing Based Hash Tables for the CPH STL

Anders Thøgersen and Jørgen Havsberg Seland

Abstract. We describe a modular hash table implementation for the CPH STL that utilizes universal classes of hash functions and the linear hashing method of table dynamization. We analyze and compare the performance of our implementation to that of established STL hash table implementations, and discuss possible bottlenecks and future optimizations. Although the empirical results for total execution time are inferior to the competitors, our results position linear hashing based hash tables as an interresting alternative when stable access times and space-efficiency is desired. We also recommend further work on linear hashing, as we believe its cache characteristics may outweigh the sheer simplicity of competing methods on modern computer systems.

1. Summary

The purpose of this paper is to present and evaluate a hash table design for the CPH STL. The first section covers the necessary theoretical background, surveying relevant topics within hash table algorithms and universal hashing. As our goal is comparative performance analysis, a number of common optimizations are also covered. The following section details the requirements that the design must fulfill, both those posed by official standards and additional ones set forth by the authors and by the CPH STL. We then proceed to suggest a design that fulfills these requirements, which is then described in the subsequent three sections, covering first the interfaces of the various classes that comprise the hash table, then covering in more detail how these entities interact to provide the functionality exposed by the aforementioned public interfaces, and finally describing in detail critical points in the implementation of our three candidate tables. Two of these are based on the linear hashing method [30, 27], while the third is based on the design used in [21].

The second part of the paper covers the comparative performance analysis, in which our three candidates are compared with publicly available STL hash table implementations using synthetic benchmarks based on data sets with varying random and non-random distributions. We then conclude by suggesting future directions for completing the design.

CPH STL Report 2005-4. June 2005

2. Background

> "From a practical standpoint, the most important hash technique invented in the late 1970's is probably the method that Witold Litwin called linear hashing."
> — Donald E. Knuth [25]

Our primary motivation came from the need of a flexible hash table implementation for the CPH STL. When we began this project there already existed an implementation of a `hash_map`, however the other associative containers that use a hash table where missing. We found it an interesting task to try to add a hash table general enough to be used for all of the associative containers: `hash_map`, `hash_set`, `hash_multimap`, and `hash_multiset`, as well as being flexible enough to accommodate varied user-supplied hash table algorithms.

The choice of linear hashing as the main focus of this report came initially from its space usage characteristics. The typical hash table implementation uses a standard `vector` which utilizes a doubling technique for memory allocation and deallocation. As described in [26] the doubling vector typically wastes space proportional to the amount of elements stored in the vector. We found an allocation scheme that did not waste so much space attractive, providing a more elegant solution to the dynamization problem than doubling and halving the allocated space. Precedence in linear hashing based implementations was also lacking, P. J. Plauger's *incremental hashing* implementation[1] in the Dinkumware STL being the only general purpose implementation to our knowledge, so part of the purpose of this report is to explore the suitability of this method for a general-purpose hash table.

2.1 Usage Profiles

A general purpose hash should support common usage profiles and be extendible to support other more specific needs. Here we will try to identify typical settings in which a hash table might be used so that specific extension of the hash table will not be necessary in the common case.

- Compilers often use a hash table for symbol table lookup.
- DNS servers might use a hash table for mapping host names to IP addresses.
- Databases typically use hash tables heavily both for performing queries and for accessing blocks on disk.

We have identified the following three settings in which it might be desirable to specialize a hash table:

[1] The incremental hashing method was described in a paper published by P. J. Plauger in 1998, according to [2]. Despite intensive research, we have been unable to obtain a copy of this paper, and our only source of information has therefore been second-hand sources and the STL source code from the Microsoft Platform SDK. Due to obvious copyright issues, we have been restrictive in the utilization of the latter.

- *Lookup-intensive*: A DNS server using a hash table for hostname lookup is an example of a setting in which `lookup` is the most common operation. Insertion and deletion might occur, but do not constitute a significant amount of the operations.
- *Insert-intensive*: Duplicate detection in a scenario where few duplicates exist is an insert-intensive application. If many duplicates exists, the application may instead be considered lookup-intensive, given the definition of the `insert` operation (see the next section).
- *Responsiveness*: In a real time system it is important that the characteristics of operations on the hash table do not display too much variability.
- *Average*: This setting represents the "typical" usage settings. It is assumed that all operations are equally likely. Note that this is the only situation where `delete` operations may constitute a large amount of the operations, given that there may be no more successful `delete` operations than there are successful `insert` operations.

2.2 Definition of a hash table

A hash table is a data structure that supports the following basic operations:

`insert(e)` Insert an *element* `e` into the data structure, if the element has not yet been inserted[2].

`delete(e)` Delete the element `e`.

`lookup(e)` Retrieve the element `e`.

The above set of operations is similar to those supported by a dictionary, but unlike a dictionary implemented using an ordered list or a tree, elements are not ordered within the data structure, and operations such as finding the predecessor or successor are not directly supported. Conversely, a hash table typically provides constant or amortized constant access time to elements which is much better than what a list or tree can provide. This is achieved by storing the elements in a table with the help of a *hash function* $h(e)$, which maps elements in the possible set of elements U to indexes often using a modulus operation to wrap the function values to fit the table size. We refer to the value of the hash function as the *hash value* of the element, and denote the set of such values M. Equality of elements is determined by an *equality function*, which may be true only if the elements compared have the same hash value.

It can occur that different elements map to the same index causing a *collision*. To resolve this problem a *collision-resolution method* is employed. The different collision-resolution methods can be divided into two groups: *Chaining* and *open addressing*. In this report we focus on collision resolution by chaining which is a method for resolving collisions by keeping a linked list at each index in the hash table. A given index in the hash table and its

[2] Later sections introduce the concept of a *multimap* where duplicate elements are allowed.

corresponding data structure for holding additional elements is referred to as a *bucket*.

The open addressing approach to collision resolution differs from the chaining method in that colliding elements are kept within the hash table. When a collision occurs an attempt to find an unoccupied index is done via method known as *probing*.

A *load factor* is used as a measure of how many elements the hash table contains in comparison to the size of the hash table, and is commonly denoted α.

2.3 Chaining versus open hashing

The two dominant methods of resolving collisions in a hash table are chaining and open hashing. A brief description of the two will be given here. For more in-depth information on collision resolution methods see [8, 25].

2.3.1 Open hashing

With open hashing all elements are stored within the hash table. A consequence of this is that the load factor will always lie between 0 and 1. To resolve a collision there exist different methods of probing for a free index within the hash table. Two of these are *linear probing* and *quadratic probing*. Linear probing is simple, but suffers from *primary clustering*. Quadratic probing is considered better because it does not suffer from primary clustering in the same way as linear probing does, but instead suffers from *secondary clustering*.

A hash table using the open addressing collision resolution method has a good performance when inserting elements. Deletions however are not as effectively handled as in a chained hash table because deletion requires the elements to be moved backwards through the probe sequence which may be a very complex operation, depending on the probe method. Because of this, one often just marks the elements as deleted and move the non-deleted elements to a new table at regular intervals, an operation called *rehashing*.

2.3.2 Chaining

Chaining is the simplest collision resolution method. The basic idea is to extend each index in the hash table with a linked list allowing multiple values to arrive at the same index. Such an extended index is commonly referred to as a *bucket*, and the linked list as a *chain*. This makes it necessary to search for a given element in the linked list when performing a `lookup(k)` operation.

The load factor of a hash table using chaining is an estimate of the average list length. The speed of operations on hash tables decreases when the load factor increases because the linked lists that have to be searched increase in size. Worst case running time of a lookup operation on a hash table using

chaining is $\mathcal{O}(n)$ which is highly improbable as all elements would have to land in the same bucket for this to happen. It is often assumed that the hash function produces indexes that are evenly distributed across the table, independent of where other elements have hashed to. This is the assumption of *simple uniform hashing*, which allows us to estimate the upper bound on a lookup to be proportional to the average list length. The following section on *universal hash functions* describes a method for fulfilling this requirement with non-random element distributions.

Most existing STL implementations use a hash table with chaining because chaining is a conceptually simple and robust strategy, and does not suffer from the deletion inefficiencies of open addressing.

2.4 Universal Hash Functions

The concept of *universal classes of hash functions* was first introduced by Carter and Wegman [7]. A class of hash functions is considered *universal* if the probability of collision between the hash values of two elements for a function h randomly chosen from the class is less than or equal to $\frac{1}{|M|}$, where M is the set of possible hash values [8]. More precisely, a class is considered (c,k)-universal, if, for k distinct elements $x_0, x_1, ..., x_{k-1}$ and indexes $l_0, l_1, ..., l_{k-1}$, $\mathbf{Prob}\{h(x_i) = l_i, \text{for } 0 \leq i < k\} \leq \frac{c}{|M|^k}$ [11]. Intuitively, this refers to an expected upper bound on the probability that k distinct elements collide. A member of such a class will henceforth be referred to as a *universal hash function*.

If $k = 2$, the notation is sometimes shortened to "c-universal" [21], and this is the notation we will continue to use in this paper. Feasible solutions for functions with $k > 2$ exist, and have desirable properties with regard to the variability of the collision probability (see for instance [35]), but are outside the scope of our paper.

2.4.1 Benefits

The benefit of using a universal hash function is that the index distribution becomes independent of the input, at least in theory, and thus makes the theoretical analysis valid for most inputs, not just those that are uniformly random. An application using a hash table based on universal classes of hash functions is therefore guaranteed to avoid worst-case performance in most situations, and it is impossible to create a perfect adversary[3] without knowledge of the internal state of the application. This has important consequences for DoS[4] attacks, and is therefore an important requirement of our implementation, as otherwise the applications based on the implementation would all be vulnerable to such adversaries. It is important to note, however, that worst case performance may still occur, even though this is highly un-

[3] A dataset that causes worst-case performance every time.
[4] Denial of Service.

likely, so we do not escape the $\mathcal{O}(n)$ bound on operation complexity. Section 2.6.3 on linear hashing covers a method for further reducing this risk.

2.4.2 Examples

In [21] definitions of two very efficient universal classes of hash functions are given. The first one is a class of multiplicative hash functions determined by the two parameters 2^k and 2^l, containing 2^k hash functions of the form

$$h_a(x) = \frac{(ax \bmod 2^k)}{2^{k-l}},$$

where a is an odd integer in the range $[1, 2^k - 1]$. As the operands of the modulus and division operations are powers of two, bitwise *and* and *shift* operations may be used. This function is proved to be 2-universal [12].

The other function is based on random lookup tables, and is well suited for longer keys, such as strings. The original version is 1-universal, and uses one random lookup table for each character position, requiring $\mathcal{O}(\ell 2^k)$ storage, where 2^k is the size of the alphabet (typically 2^8 for character strings), and ℓ is the length of the longest string. However, there is experimental evidence that one may use the same lookup table for more than one character position without significant reduction in performance, although this modification invalidates the theoretical results [31].

2.5 Linear Hashing

Linear hashing is a solution to the dynamization problem of hash tables. Conventional hash table algorithms tend to disregard the requirement that a table should grow dynamically to accommodate newly inserted elements, and shrink if the allocated storage is no longer needed. The common solution to this problem is to create a new hash table twice the size when the existing one is full, and to insert all the existing elements in this new table, keeping α below some constant limit.

Linear hashing works by adding single buckets to the table when it fills up, and relocating approximately half of the entries hashing to an existing bucket to this new bucket, an operation referred to as *bucket splitting*. Conversely, contraction is achieved by a *bucket joining* operation. This produces a linear growth and reduction in table size, hence the name.

2.5.1 Precedence

The concept of linear hashing was first introduced in [27], in the form of an external hashing algorithm designed for use in file systems. It was later extended by Per-Åke Larson in [30] to hash tables in main memory.

Larson uses gradual expansion and contraction, but this is implemented using a memory allocation method that does not scale, and is therefore unsuitable as a general-purpose implementation.

2.5.2 Collision-resolution method

Both the variants of linear hashing that we have seen resolve collisions by chaining. Even though using open hashing for collision resolution is possible, it is unsuitable, as more advanced (and thus efficient) probing methods complicate deletion, which, in opposition to the doubling method, cannot be avoided in linear hashing, as deletion is a vital component of the bucket splitting operation.

In the following, we will therefore assume collision resolution with chaining.

2.5.3 Bucket splitting

Bucket splitting is performed by assigning a new hash function to buckets that contain more than a certain amount of elements. The elements in the bucket are relocated to fit this new hash function possibly moving them to other buckets. With a suitable choice of hash function, this will bring the number of elements in the bucket down below the limit. However, to make bucket splitting practical certain limitations have to be made on how and when buckets are split, and also on the choice of hash functions for split buckets.

In [30] the hash functions are constructed by masking off a given number of bits from a common, underlying hash function, and the new hash function is chosen by adding one more bit to the old one. This has the desirable property that only the most significant bit separates the values of the hash functions, effectively limiting the possible destination buckets to the split bucket and to the bucket 2^{k-1} buckets further up the table, where k is the index of the added bit.

Which bucket is split, is decided by a pointer that starts out pointing at the first bucket. When the load factor of the table exceeds a predefined amount, the table is expanded by one bucket, and the bucket designated by the pointer is split, moving on average half of its elements to the new bucket. Finally, the pointer is set to point to the next bucket in the table (see figure 1).

When all the buckets in the initial table have been split, the pointer is moved back to point to the first element, the old hash function is discarded, and the whole process is repeated. At this point, the table has doubled in size, compared to the original table (see figure 2).

This method maintains the invariant that all buckets below the pointer may be accessed by the new hash function. If the element hashes to a bucket below the pointer by the old function, the new hash function should be used instead. Thus a constant amount of storage is required to determine which function to use for each bucket.

Additionally, this method limits the number of active hash functions to two, as all buckets must be split (and therefore be assigned the new hash function) before any buckets are split a second time.

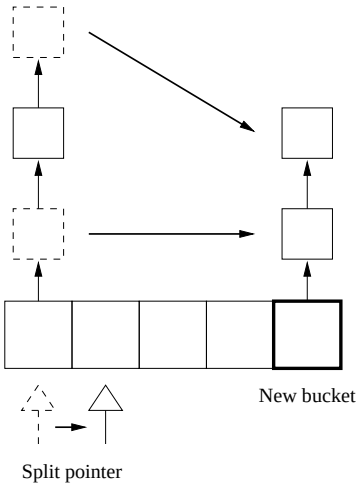

Figure 1. The bucket split operation.

2.5.4 Time Complexity

In the following we will refer to Larson's original method of linear hashing as *the one-table method*. A complexity analysis is given in [30], in which the complexity analysis for regular hash tables using collision-resolution by chaining given in [8] is extended by modeling the effect of bucket splitting on the table load factor.

The model introduces the factor x, which is the fraction of the buckets

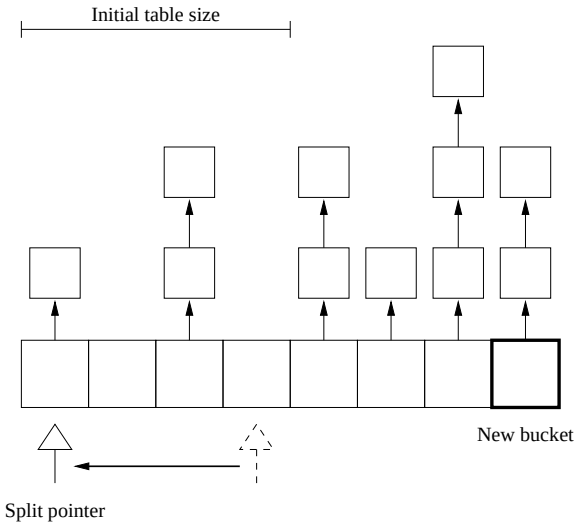

Figure 2. Pointer reset on split of final bucket in initial table.

in the initial table that have been split. During the course of expansion, x cycles in the range $[0, 1[$, with the minimum value just after pointer reset[5]. Based on x, a formula for the load factor of split and unsplit buckets is derived, and is used to determine the expected lengths of successful and unsuccessful searches. Larson states that these vary from the minimal $1 + \frac{\alpha}{2}$ for successful and α for unsuccessful searches when $x = 0$ and $x \to 1$, to the maximal $1 + \frac{9}{8}\frac{\alpha}{2}$ and $\frac{9}{8}\alpha$ at $x = \frac{1}{2}$, following the formulas $1 + \frac{\alpha}{4}(2 + x - x^2)$ and $\frac{\alpha}{2}(2 + x - x^2)$, respectively. Note that the minimal list length is the same as with a regular, fixed table.

Linear hashing also introduces an extra cost associated with the bucket split and join operations. Larson only models the first, which is incurred at a fraction $\frac{1}{\alpha}$ of the insert operations when ignoring deletions. The average number of extra computations, given the expected list length of $\alpha(1 + x)$, is $1 + x$, or 1.5 on average across a complete table doubling.

2.6 The Two-Table Method

A variation of Larson's scheme is achieved by using two hash tables instead of one, where each uses its own hash function, and the second table is twice as large as the first one. When splitting buckets, the split bucket is emptied rather than halved, and all the items are transferred to the buckets in the other table (figure 3). When contracting, the reverse of this operation is performed. When the split pointer arrives at the end of the first table, this table is discarded, the second table takes the place of the first, a new table twice as large as the second is allocated, and a new hash function is assigned (figure 4).

This method is less space efficient, as we are allocating memory for buckets which are not used: The buckets below the split pointer in the first table are always empty. There are, however, some interesting benefits to this version of linear hashing.

2.6.1 Load Factor

The load factor of the two-table method is not immediately obvious. Using the number of elements divided by the total number of buckets does not give a realistic picture of the probability of collisions, as only one third of the buckets are accessible when the split pointer is in the initial position, and almost two thirds just before it is reset. Furthermore, this size measure will not increase linearly, and the load factor will not drop when buckets are split, except when a new table is allocated. We therefore introduce the concept of *effective load factor*, which scales the impact of a bucket by its availability.

In the initial position, the second table is unavailable. As buckets are split, more and more of the hits to the first table are forwarded to the second, until the entire second table becomes available as the first one is discarded.

[5] Note that x never reaches 1 as the pointer is reset when the last bucket in the initial table is split.

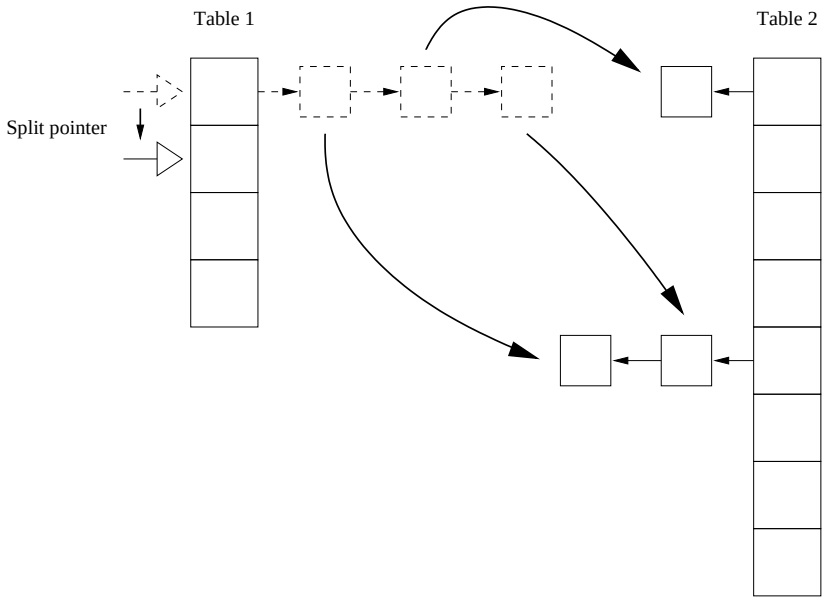

Figure 3. Bucket split operation using the two-table method.

Similarly, less and less of the first table becomes available as the split pointer moves towards the end. We introduce $p \in [0, T_1[$ to mean the index of the split pointer, where T_1 is the size of the first table ($T_2 = 2T_1$). The fraction of the second table available through the first is given by $\frac{p}{T_1}$, and the fraction of the first table available is given by $\frac{T_1 - p}{T_1}$. The effective table size, when the individual sizes are weighed by their availability, becomes

$$\frac{T_1 - p}{T_1} T_1 + \frac{p}{T_1} T_2 = T_1 + p.$$

The effective load factor is consequently given by $\alpha = \frac{n}{T_1 + p}$, which grows linearly as the table expands.

2.6.2 Space Utilization

Having established an effective table size measure, we can proceed to compute the space overhead of the two-table method. We denote the load factor at which contraction is triggered by α_{min}, and the corresponding load factor for expansion by α_{max}. This limits the range of the effective load factor to

$$\alpha_{min} \leq \frac{n}{T_1 + p} \leq \alpha_{max},$$

and the number of elements to

$$\alpha_{min} (T_1 + p) \leq n \leq \alpha_{max} (T_1 + p).$$

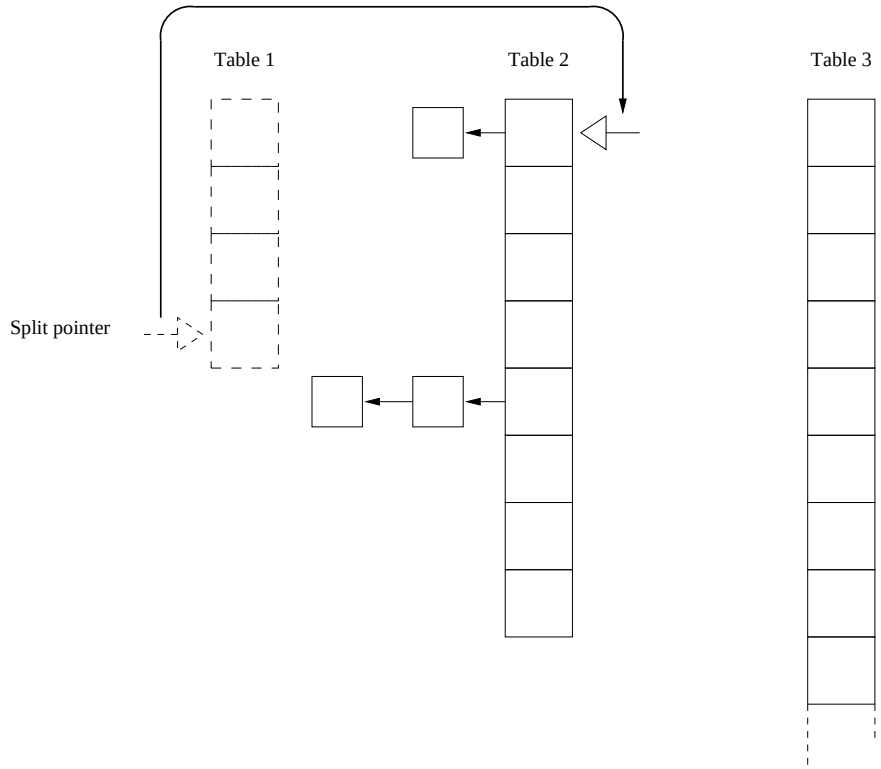

Figure 4. Pointer reset operation using the two-table method.

Given that p cycles between 0 and $T_1 - 1$, and that $\alpha_{min} \leq \alpha_{max}$, this limits n to

$$\alpha_{min} T_1 \leq n \leq \alpha_{max} \left(2T_1 - 1\right) \leq 2\alpha_{max} T_1.$$

The actual space usage for the tables is $T_1 + T_2 = 3T_1$, and thus the proportion w of allocated to used space is $\frac{3T_1}{n}$. If we divide the previous equation by the actual space usage and invert the result, we get

$$\frac{3}{\alpha_{min}} \geq \frac{3T_1}{n} \geq \frac{3}{2\alpha_{max}},$$

which places w within the range $\left[\frac{3}{2\alpha_{max}}, \frac{3}{\alpha_{min}}\right]$. If we assume one machine word of overhead for each bucket, and another two for each element, we arrive at

$$(w+2)\, n = \left[\left(\frac{3}{2\alpha_{max}} + 2\right) n, \left(\frac{3}{\alpha_{min}} + 2\right) n\right]$$

machine words of overhead which for load factor limits of 2.5 and 5 gives $[2.3n, 3.2n]$.

2.6.3 Using Multiple Hash Functions

An interesting feature of the two-table method is that it is possible to change to a different member of a class of universal hash functions when allocating a new table. The only change to the bucket splitting operation is that the elements may now hash to more than two destinations. Bucket joining, on the other hand, is made much more complicated.

In general when joining a bucket with index b, we need to find the set of elements $E = \{e_0, e_1, ..., e_{n-1} \| h_1(e_i) = b\}$, where h_1 is the hash function of the first table. The index of the buckets where the elements may be found may be computed by the formula

$$h_2(h_1^{-1}(b)), \tag{1}$$

where h_2 is the hash function of the second table and h_1^{-1} is the inverse of h_1.

When the tables share the same underlying hash function, we implicitly rely on the fact that alternative functions derived by bit selections are easily inverted: We may easily compute the buckets where we may find the elements that go into the bucket we are "joining" by looking at the indexes b and $b+T_1$, where T_1 is the size of the first table.

Unfortunately, (1) is not computable for hash functions in general, and poses a significant problem when switching between arbitrary underlying hash functions. We therefore need a different way to address the elements that go into the destination bucket.

One solution to the problem, is to maintain a latent list for each split bucket in the first table. When splitting a bucket, the elements are distributed into their respective buckets in the second table, but are also kept in a list in the original bucket (see figure 5). If bucket joining is required, we may simply "reactivate" the latent list, and the bucket is joined in $\mathcal{O}(1)$ time. This is achieved by maintaining two sets of linked list pointers, one for each table's buckets, and choosing which one is used when traversing the list based on which table's buckets we are accessing the list through.

There are two penalties using this method: There is the space overhead of maintaining two sets of linked list pointers, but the major penalty comes from having to maintain the latent list after it has been deactivated. Elements that hash to a split bucket must be inserted both into that bucket and into the relevant bucket in the second table.

The benefit of switching hash functions is that we further reduce the risk of using a family member that causes poor performance: If we should be so unfortunate as to choose an inefficient family member, this will be remedied with high probability upon the next table change.

Switching of hash functions may of course also be utilized when using the conventional table doubling method of dynamization.

2.7 Computations of general hash table metrics

In this section, we'll develop formulae for

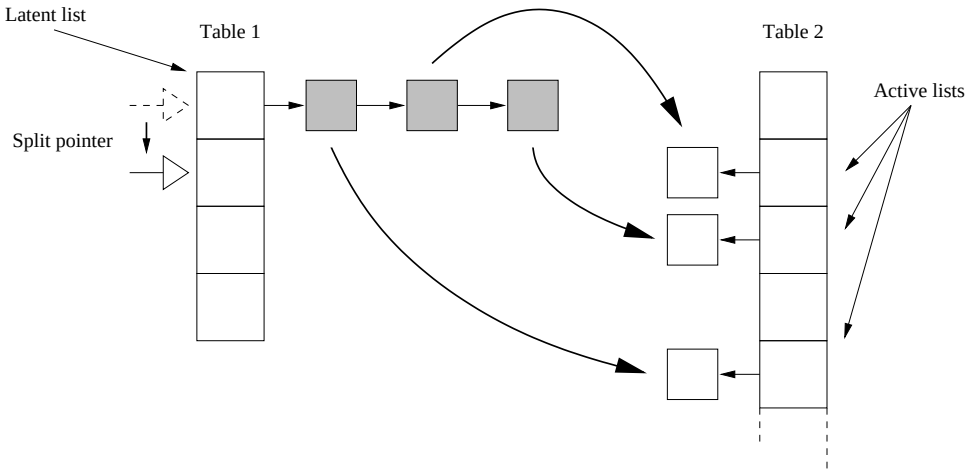

Figure 5. The bucket split operation when using latent lists to support change of hash functions. Note that the split may now distribute the elements across more than two buckets.

- expected maximum list length,
- the number of non-empty buckets and
- the distance between these,

given a c-universal hash function.

2.7.1 Expected maximum list length

In Carter and Wegman's original paper on universal hashing [7], an expected upper bound on the number of collisions for 1-universal hash functions is established. The quantity $\delta_f(x, S)$ is defined as "the number of collisions between the element x and the set of elements S, which is a subset of U".

If S is taken to be the set of elements inserted into the table before an operation on x, the expected maximum number of elements to traverse is $1 + \delta_f(x, S)$ (x plus the number of elements that collide with x), which, for 1-universal functions, is bounded by $1 + \frac{|S|}{|M|}$. The expected maximum list length after n insertions is therefore $1 + \frac{n}{m}$ (where $n = |S|$ and $m = |M|$). For a c-universal function, the expression becomes $1 + \frac{cn}{m}$.

2.7.2 Number of non-empty buckets

If we assume that no more than $1 + \frac{cn}{m}$ elements may be found in each bucket, the minimum number of non-empty buckets is

$$\left\lceil \frac{n}{1 + \frac{cn}{m}} \right\rceil = \left\lceil \frac{nm}{m + cn} \right\rceil .$$

The meaning of the above formula may be seen if the table is visualized as an array of boxes that each may contain $1 + \frac{cn}{m}$ elements. The above

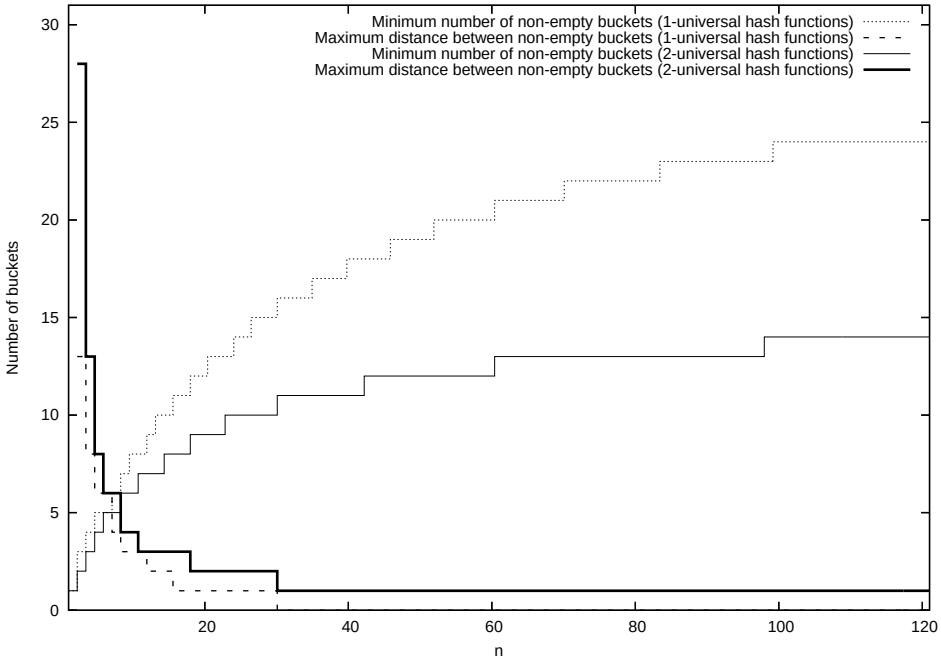

Figure 6. Plot of number of non-empty buckets and the distance between them for a table of size 30.

expression then gives the minimal number of boxes required for n elements. The expression approaches $\frac{m}{c}$ as n approaches infinity.

2.7.3 Maximum distance between non-empty buckets

When implementing iteration in tables without storing information about the next and previous non-empty buckets, the maximum distance between non-empty buckets is of interest, as an overhead is introduced into the iteration process proportional to this distance. This quantity is given by the formula

$$\frac{m - \left\lceil \frac{nm}{m+cn} \right\rceil}{\left\lceil \frac{nm}{m+cn} \right\rceil - 1}$$

which computes the number of empty buckets divided by the number of spaces between non-empty buckets. Figure 6 shows a plot of the expected quantities for a table of size 30. By looking at the limit for the number of non-empty buckets, we may expect the distance to approach zero and one for 1- and 2-universal hash functions, respectively.

2.8 Previous work on hashing for the CPH STL

The following design ideas were presented in the previous implementation of a hash map for the CPH STL [5].

2.8.1 Chunking

The idea behind chunking is to let the nodes of the linked lists consist of not just one element, but continuously allocated *chunks* of a fixed number of elements. This should theoretically give a significant improvement in the cache miss ratio. When traversing a linked list, nodes are randomly dispersed across main memory. With chunking instead a linear memory block is searched. This is faster as it enables prefetch and reduces cache fragmentation, but it comes at the cost of some memory overhead, although the overhead has been reduced as fewer pointers are needed per element.

2.8.2 Saving the hash value

Storing the calculated hash values before the modulus operation together with the elements in the hash table proved to be a much better optimization than the use of chunking. This is because the amount of element comparisons during `lookup` operations is drastically reduced by first comparing the hash values. When using data types such as strings this is potentially a big advantage, but may not be an advantage at all when using simple data types such as integers, because integer comparison is cheap compared to string comparison.

2.8.3 Results

Surprisingly, the expected benefits from cache effects were missing when the implementation was tested. The hash table performed worse than the SGI implementation despite the theoretical cache advantages [5, page 7]. In the fourth last paragraph on page 8 it is mentioned that they have tried to exchange the chunk-based implementation with a simple linked list, and that this brought the run time down on par with that of the SGI hash table. This indicates that the the problem has to do with the chunking optimization, but it was not possible to point a finger at the exact problem. A close look at the figures on page 7 reveals that the curves seem to share a cyclic pattern. On page 6 this pattern is attributed to "noise". We believe that it could also be an indication of a problematic implementation detail. We have been unable to find the exact cause of these fluctuations, although we have considered many possible factors.

The most important result of the report is that storing the hash values to reduce key comparisons resulted in a significant performance boost.

2.9 Alternative storage for linear hashing

We have found an alternative vector implementation that reduces the amount of data copying named *space efficient singly resizable array* [22]. The vector implementation described therein fits nicely with the strategy of gradually expanding the hash table as in linear hashing. A similar data structure, the *space efficient doubling tree*, has been implemented for the CPH STL [26], which has made it easy for us to make experiments using this storage model. This data structure will from here on be referred to as a SED tree.

The SED tree wastes $\mathcal{O}(\sqrt{n})$ machine words as opposed to the doubling vector's $\mathcal{O}(n)$. Furthermore, the SED tree does not perform data copying on expansion.

2.10 Iterator validity

The C++ standard defines *iterator validity* by requiring that an operation that maintains iterator validity does not cause iterators to become *singular*[6]. The CPH STL definition of iterator validity is [20]:

> An iterator and the element pointed to live in a close symbiosis; when the element is moved, the iterator may become invalid if it is not updated accordingly. A data structure is said to provide *iterator validity* if the iterators to its elements are kept valid at all times independent of the element moves.

We have found that iterator validity across all operations comes at the cost of both a time and a space overhead depending on the specific setting and the specific implementation of iterator validity used. We have considered two possible methods of implementing iterator validity, which we will describe here in increasing order of complexity. None of these have been implemented.

2.10.1 Reference counting

It is possible to implement iterator validity by maintaining a *reference count* within each element. Each time an iterator enters a specific element, the reference count of the element is incremented and each time an iterator leaves an element the reference count is decremented. When the reference count becomes zero it is guaranteed that no iterators are referring to this element. In addition, it is necessary to be able to indicate if an element has been deleted. There is the possibility that the time complexity of iteration becomes linear in the size of the hash table if many elements in sequence become deleted while having a nonzero reference count.

An implementation of iterator validity using this method causes a small time overhead for each iteration step because of the increment and decrement of the reference count. Also a significant amount of extra memory is needed because every single compartment will need a reference count. The amount of

[6] The term "singular" is defined in the C++ standard by the example of an uninitialized pointer. We therefore interpret a singular iterator as one not pointing to a valid element.

extra memory needed depends on the data type used to track the reference count. Typically a single byte will be sufficient as this allows up to 255 iterators to point to the same compartment without loss of iterator validity.

An advantage of using a reference count is that it does not require changes to the storage model as opposed to the following method.

2.10.2 Persistent data structures

Our second and more complex solution to the problem of iterator validity redefines the semantics of a hash table. The resulting hash table would implement *partial persistence* allowing a user to access previous versions of the hash table. The idea was inspired by the presentation of persistent data structures found in [6]. It follows from this paper that it is possible to add partial persistence to a hash table implementation incurring only a constant time overhead for updating the structure. The addition of persistence would make it possible to have iterators that point to different versions of the data structure. The definition of what iterator validity entails is redefined however. All versions of a structure using partial persistence may be inspected, but modification access is allowed only to the most recent version of a data structure. A *fully persistent* hash table would implement iterator validity without this restriction, but we know of no currently existing methods feasible for a high-performance data structures.

2.11 Common optimizations

We here present possible optimizations, some of which are implemented.

2.11.1 Sparse tables

A *sparse table* is a table that compresses empty locations. This is realized by maintaining a fixed size vector of bits and dynamically allocating a block of memory for storing the actual elements. A set bit at a given index into the bit vector indicates that there exists an element at that index. More information on sparse tables can be found in [10, "sparsetable" documentation section].

Sparse tables are used by the Google sparse hash table implementation [10] to reduce the amount of memory used. This also allows much lower load factors using the same amount of space. Sparse tables could easily be adopted for use with linear hashing to achieve the same effects. Especially the two-table method could benefit from using sparse tables.

2.11.2 Storing hash values

Storing of hash values has additional benefits to those described above. The time taken to calculate the hash value of a string of bytes may be expressed as $\mathcal{O}(\ell c)$ where ℓ is the length of the string and c the amount of times the hash value is calculated. When storing the hash value we reduce this expression to $\mathcal{O}(\ell + c)$.

If we retain the same underlying hash function when performing bucket splitting or rehashing to a larger table (depending on the method of dynamization employed), storing of the underlying hash function's hash values (before masking them) has the benefit that the hash function needs not be reevaluated. Without storing of hash values, the hash function will have to be recomputed for every existing entry[7]. Storing both the underlying and the masked hash functions will in most cases be unnecessary, as the former may be derived from the latter by a bitwise `and` operation, which is commonly inexpensive.

Storing of the hash values of the underlying hash function can also save time in the equality function. This approach is similar to that taken in the Rabin-Karp string matching algorithm [8, section 32.2]. When performing a `lookup` operation, a linear search of the elements with the same index is performed, and a comparison has to be performed for every entry in the list. With stored hash values available, we may use these as a filter, and only invoke the equality function whenever the hash values match, as the elements may be equal only if the hash values are equal. We may, as in Rabin-Karp, expect *spurious matches*, but given the characteristics of a universal hash function, we may expect their number to be negligible[8] (as is done in [5]). An important detail is that in addition to saving calls to the equality function, this approach also avoids accessing the actual data of most of the elements. This may prevent a cache miss per element in the list.

The overhead of storing the underlying hash function value will commonly be one machine word per element inserted.

2.11.3 Ordered buckets

If an ascending order is imposed on the elements within each bucket, it becomes possible to search for an element by using a less-than comparison function instead of an equality function to compare elements. This is done by iterating through the list of elements until the current element becomes not-less than the element we are searching for. When this happens, the last element checked must be the element searched for if the element exists in the hash table. If not, the search is unsuccessful.

If the ascending order of elements within a bucket list is determined from the value of a stored hash value it is possible to still parameterize the hash table template class with an equality function, while internally using less-than for comparing the stored hash values when performing a lookup.

The advantage of ordering elements within a bucket is that the average search time for unsuccessful searches is almost halved. As a search is performed on every insert and delete, the benefit is not limited to the lookup

[7] Note that this is not the case with contraction, as the new index may be found in the least significant bits of the current index.

[8] An empirical test of the previously mentioned random lookup table hash function (using only four tables) on a list of English words revealed around 135 spurious matches in 212,000 elements.

operation.

Keeping the ordering on insert is inexpensive as the space before the element where the search terminated is a valid insert location.

2.11.4 Using sentinels for tight inner loops

This optimization is a slight change to the method used when searching for an element within a bucket's list. The requirement for this optimization is that a *sentinel element* terminates the list being searched.

The element that is searched for is first copied into the sentinel, and then the search of elements within the bucket begins. When the element we are searching for is found we have either found the actual element or arrived at the sentinel. This can be checked by comparing the address of the element found with that of the sentinel.

This strategy allows for a tight inner loop for traversing the bucket list when searching for an element because it is not necessary to check for the end of the bucket.

3. Methodology

This section is meant as a brief overview of how our work has been conducted and to the tools that have been used. We did not complete the work that the following sections describe in the same strict order in which they are presented, but rather iterated repeatedly through the stages they represent. We found that iterating between design and implementation was helpful because it allowed us to see parts of the design in action before working on the parts on which we were unsure.

3.1 Design

A blackboard, a piece of chalk and knowledge of UML are useful tools to have when discussing how to develop software. During the design process we often returned to the blackboard to elaborate on specific issues regarding parts of our design, and found this very helpful. Also developing small independent programs to test a new idea, or as a means of gaining insight into a specific problem proved helpful.

We initially started the work on our design by drawing *CRC cards* [4] to provide an overview of the class collaborations, but soon we moved over to use a blackboard instead. We found it comfortable to discuss our design at the blackboard, and most of our design was made by drawing and discussing in front of one.

3.2 Testing

All code that has been written for our hash table implementation has been tested with the CPPUnit testing framework [14]. Once we became familiar

with writing tests using CPPUnit it was a small task to add a new test for a newly developed module.

We have not taken measures to guarantee full test coverage of all the code that has been written, instead the central functionalities of program parts were tested. We believe that the tests cover a broad perspective. In a final version, however measures to guarantee full test coverage should be taken, for instance using tools such as `gcov` [16].

3.3 Profiling

In order to determine which functions that are taking up the most time we have used the `gprof` [15] profiler. We have also briefly run tests using the `Tau` profiler, which looks promising.

We intended to use the results of profiling runs to optimize specific parts of the code, but we have left many optimizations undone in favor of other tasks.

4. Requirements

In this section we detail the requirements that our design must fulfill. We divide these into those posed by the C++ standard, additional requirements posed by the CPH STL, requirements that follow from our desire to provide a modular design, and finally a few requirements that improve the scalability of the resulting design.

4.1 C++ Standard Requirements

We have not covered every requirement posed for a standards-compliant C++ STL container, as this is a huge task, far outside the scope of this project. We have, however, tried to cover all the requirements that will significantly shape the implementation. The following sections list the requirements we have found.

It should be noted that the structures `hash_map`, `hash_set`, `hash_multimap` and `hash_multiset` (hereafter referred to as `hash_*`) are not part of the official standard. This is mainly due to the fact that they were not taken into the standard from the beginning, and that subsequent proposals to add them have arrived out of sync with updates to the standard. An amended version of a proposal to add hash tables to the standard [2] is scheduled for inclusion in the upcoming *Library Technical Report* (TR), which functions as a gateway for additions to the standard. Inclusion may, at the time of writing, still be several years in the future, and we have therefore chosen to follow this specification only partially.

4.1.1 Exception Safety

STL containers provide different exception safety guarantees for different operations. Most operations make guarantees that the data structure is not

corrupted, and that no memory leaks occur [2]. The requirements on whether the user should be able to determine the resulting state of the data structure, that is, if the operation should either succeed or have no effect, is different based on the operation and container in question. One example is the hash table `insert` operation. If hash functions are allowed to throw exceptions, and if the insert operation causes a rehash (we are assuming dynamization by table doubling), then some items may already have been moved, and moving them back would require re-invoking the hash function, which could potentially cause another exception. We are left in a situation where we would have to discard one or more elements to reestablish a consistent state [2].

The proposal lists only the basic exception guarantees, but this is amended by [3, section 6.7], which states that in addition to `clear`, `erase`, `swap` and the single-element `insert`, the `rehash` function should provide a success-or-no-operation guarantee unless the exception is thrown by the hash function or the equality function. Section 4.1.6 describes how this latter requirement translates to our solution.

4.1.2 Code Conventions

The code conventions established by Bjarne Stroustrup [33] appear to be a guideline for most STL implementations. We have not been able to find an official code conventions specification, and have therefore decided to follow the style of Bjarne Stroustrup's book.

4.1.3 Naming

Austern's proposal suggests the names `hash_*` and `unordered_*`. As the first suggestion would cause compatibility problems with existing non-standard implementations [23, section 6.1, subsection "Hash Tables"], we use the second naming convention.

The proposal also introduces the name `hasher` for the hash function. This is superior to `hash_function` in that confusion with other types of function objects is avoided, and that it is oriented towards the action of the hash function (i.e. "to hash").

The name `data_type` is used for satellite data, but as existing `map` implementations have already introduced the name `mapped_type` we will refrain from introducing a new name and use this instead.

4.1.4 Iteration Time Complexity

The time complexity of the `++` and `--` operators of an iterator should be $\mathcal{O}(b)$, where b is the number of buckets in the table. As stated in [3, section 6.14], the Dinkumware `hash_*` implementation provides $\mathcal{O}(1)$ iteration at the potential cost of linear time insertions in a sparsely populated table. Examination of the source code reveals that this is done by keeping track

of adjacent buckets, and updating this information on insertion. A different method for providing $\mathcal{O}(1)$ iteration is suggested in the proposal, where the contents of the table are kept in a single doubly linked list in addition to maintaining lists for each bucket [2, section F: Iterators]. This method is detailed further in our linear hash table design (section 7.3.1).

4.1.5 Comparison Operation

An unordered associative container may be implemented using either an equality function or a less-than function. The latter may be used to provide ordered bucket chains, as stated in section 2.11.3. The proposal allows for both solutions.

4.1.6 The Bucket Interface

The design suggested in [2] exposes an interface for analyzing table performance. This is managed through an interface for accessing and iterating over individual buckets, and by exposing a `rehash` function that may be used to adjust the number of buckets in the container

We choose not to use this interface for the following reasons: It moves the responsibility of what we consider typical "housekeeping" operations over to the user, but more importantly, it encourages external management of the table's load factor, which may prevent novel implementations from managing rehashing in a way that is better suited to the type of memory management being used, as is the case with linear hashing. Furthermore, it requires that the user knows how to compute a suitable bucket count in relation to the number of elements that are going to be inserted.

Another important limitation of providing a bucket interface is that implementations using different collision resolution methods will become impractical or at least very difficult to implement. Seeing the superior performance of Google's hash table implementations, this seems to be a counter-productive constraint to pose.

Our suggestion is to provide a `reserve` function that may be used to prepare the data structure for insertion of a large number of elements, and that takes the desired number of elements instead of the desired bucket count. This also improves conformance with the other container structures.

We propose that performance analysis and repair may be provided by two alternative means, depending on the internals of a hash table implementation:

1. For tables that use a bucket-based strategy, the bucket interface may be exposed by an internal module providing the actual storage functionality, which is made accessible to the user through the method `storage_policy`.
2. For tables with more esoteric management schemes, the analysis and repair functionality may be implemented as part of the internal module. We believe this to be a more viable approach as such functionality (max

chain length statistics etc.) will typically benefit greatly from knowledge of the specifics of the implementation.

See section 4.3.3 for more details on these internal modules.

4.1.7 Collision Resolution Method

The only constraint on the choice of collision resolution method in [2] is the bucket interface. Having chosen not to use this, we are left with a free choice of how to handle the problem of collision resolution. Current non-standard implementations use different solutions: the Dinkumware and SGI implementations use chained hashing, whereas Google's implementations use open hashing.

As our main focus is on linear hashing, we choose collision resolution by chaining, as this places us closer to those implementations that have the most widespread use.

4.2 CPH STL Requirements

4.2.1 Iterator Validity and Complexity

It is an intention of the CPH STL to provide stronger guarantees on iterator validity than what is provided by the C++ standard.

The only formal requirement to iterator validity is that non-const iterators for associative containers should be *mutable*, meaning that modification of the part of an element that does not affect the equality- and hash functions should not invalidate the iterator. Although not stated as a formal requirement, the guarantees provided by the SGI implementation are referred to: Deletions invalidate only iterators pointing to the deleted element.

The bound on iteration complexity is further narrowed down to $\mathcal{O}(1)$ [20].

4.2.2 Bidirectionality of Iterators

Iterators to unordered associative containers in the CPH STL are required to be bidirectional [20]. This may seem counterintuitive with an unordered container, but situations may be envisioned where algorithms are implemented that require working through the data in both directions (for instance, wrapping the iterator in a stream interface with putback functionality).

The costs of bidirectionality are covered in sections 7.2.2 and 7.3.1.

4.3 Modularity Requirements

As the ideal hash table implementation may differ depending on the intended usage profile, being able to replace parts of the implementation at minimal cost would be a highly desirable feature. We have therefore posed a number of modularity requirements on the design.

4.3.1 Traits and Policies

Traits and *policies* are much used design patterns in existing STL implementations. The distinction between the two is not entirely clear-cut, but a traits object is generally a collection of operational parameters, whereas policies typically consist of a set of operations [36], and are reminiscent of the *strategy* design pattern [17].

4.3.2 Specialization Traits

One goal that the modularization must fulfill, is to provide the four required interfaces (`unordered_map`, `unordered_set`, `unordered_multimap`, `unordered_multiset`) with a minimal amount of code. For this to be achieved, a number of template parameters will typically be be closely tied to the chosen interface, and these four "collections" may be grouped using traits classes.

4.3.3 Storage Policies

Another goal of the modularization is to be able to modify the algorithm used to store the hash table to suit the current usage profile. This is most easily achieved by using a policy class for each algorithm. We will refer to these as *storage policies*.

A storage policy must encapsulate the differences between the possible storage schemes that fit the required interfaces. The primary schemes that should be covered, are the traditional table doubling methods and the two previously described linear hashing methods. The use of alternative collision resolution methods should not be hampered, and the design should at least allow for collision resolution by chaining and by open addressing. On the other hand, a storage policy must provide an interface that is flexible enough not to become a bottleneck.

Finally, a storage policy should be simple to implement, with a minimal number of required functions, as this will lower the cost of creating custom tailored implementations where needed, and will extend the usefulness of the design.

4.4 Scalability Requirements

In addition to those requirements posed by the various standards, and those incurred by the choice of a modular design, a few additional requirements are needed to ensure the scalability of the design.

4.4.1 The size of the hash value data type

Most commonly, hash functions will be designed to return an integer type the size of a pointer in the current machine (`size_t`) to be able to index the maximum allocatable table size. For instance, on an AMD64 machine, the

address space is 64-bit and consequently tables with more than 2^{32} indexes may be allocated. To be able to address a complete table, a 64-bit valued hash function is required.

In some scenarios this would introduce an unnecessary memory overhead, as many applications would never have tables of this size. The ability of using smaller hash value types would therefore be desirable, and would not be limited to a hash value size of 32 bits: In a situation where the set of possible keys were smaller than 2^{16}, but where the set of used keys was much smaller, a regular array would waste significant amounts of space, and one would benefit from a specialized hash value type.

Conversely, one might conceive of a specialized application where, on a machine with an address space of 32 bits or less, one would want to construct a storage policy that enabled *external storage*, and thus would only be limited by the address space of the file system. In this case the hash value type would be required to be *larger* than the size of a pointer.

4.4.2 Optional storage of hash values

As previously mentioned, the storage of hash values may provide a large performance benefit, but does not improve neither speed nor storage efficiency if the size of the data that the equality function operates on, is equal to or smaller than the size of the hash value. Additionally, in certain scenarios the user may want to trade speed for space and disable the storage of hash values.

5. Interfaces

The modularity requirements posed dictate that the hash table implementation should be a composite. Furthermore, we wish to use a storage policy with a simplified interface, and as the external interface of the `unordered_*` classes is quite complex, this indicates that some sort of wrapping is required. We must therefore operate with both internal and external interfaces. A structural overview of the classes may be seen in figure 7.

The `hash_table` invokes all methods on the internal interfaces that are needed to provide the functionality that is exposed by in the external interface.

Note that the interfaces are just abstractions, as the overhead of virtual function calls prevents using actual interface classes in the implementation.

5.1 External interfaces

The external interface is the API that an end user will see and is largely identical to the one specified in the proposal [2].

The four different associative containers interfaces are: `unordered_map`, `unordered_set`, `unordered_multimap` and `unordered_mutiset`. The `*_map` versions of these containers have in common that they all store both a key and

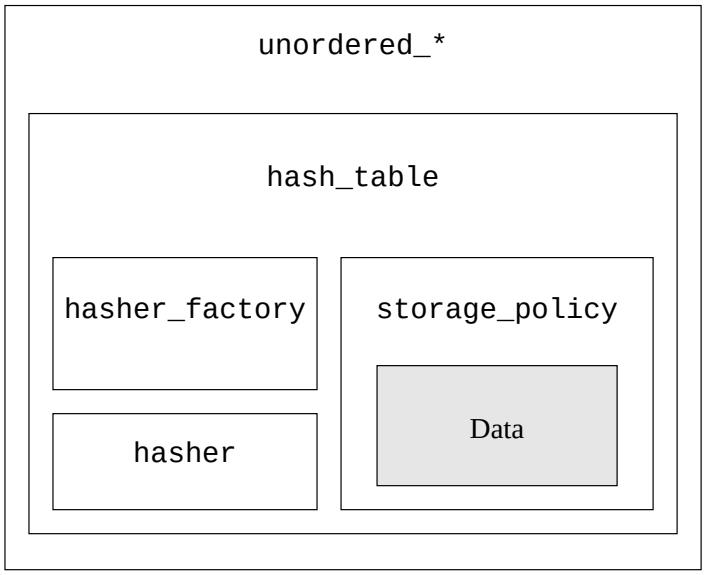

Figure 7. An overview of the structure of the `unordered_*` classes.

satellite data, whereas the `*_set` containers store only a key. The `multi_*` containers have in common duplicate keys to be stored.

The `*_map` containers define the `value_type` to be a pair of key and data, where the `*_set` containers define the `value_type` as being only a key.

As mentioned in section 4.1.6 we have chosen not to implement the bucket interface and we therefore leave out the `local_iterator` type and the `bucket_count`, `max_bucket_count`, `bucket` and `bucket_size` methods. Instead we introduce the `reserve` and `storage_policy` methods. To support universal classes of hash functions, we allow parameterization by `hasher_factory` rather than `hasher`. Finally, we introduce a `min_load_factor` to schedule contraction. This may be 0.

5.2 Internal interfaces

The internal interfaces together contain definitions of the functionality of all the `unordered_*` containers.

5.2.1 The storage policy interface

The storage policy is the most central part of our hash table implementation. All memory allocation responsibility is placed herein as are the internal versions of the three central hash table methods `lookup`, `insert_in_bucket` and `delete_from_bucket`[9]. A diagram of the one-table storage policy is

[9] Arguably, more appropriate names for these methods would be `insert_at_index` and `delete_at_index`.

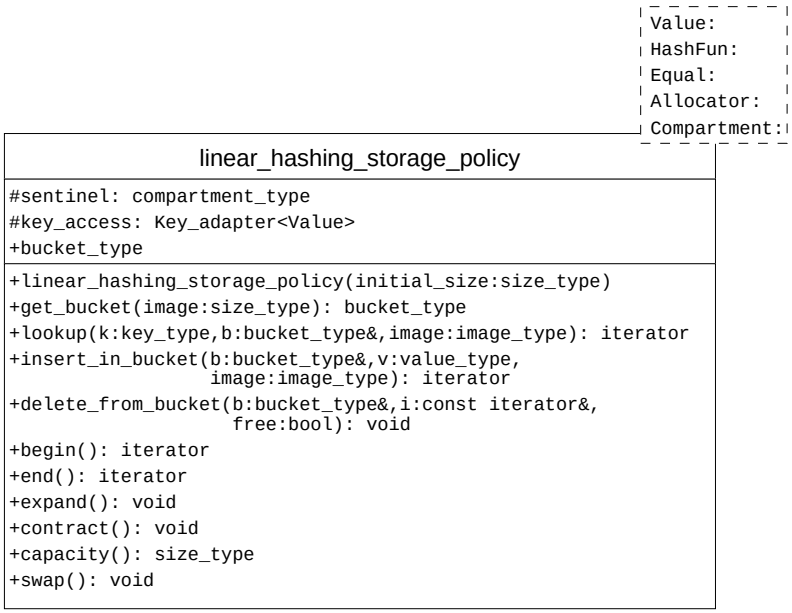

Figure 8. The storage policy interface. The values shown in the box in the upper right corner are template parameters.

shown in figure 8.

Also shown on the figure is the sentinel, the key access and the definition of the bucket type which are all specific to the linear hashing storage policy. The methods, however, are those that represent the storage policy interface. These are the methods that are needed as a minimum to implement a storage policy class. A short description of the storage policy interface follows.

The `Compartment` is the basic data type stored within the hash table. Besides the `value_type`, it may contain pointers for implementing the linked list and a stored hash value, which in this context is referred to as the *image*. The sentinel is a special compartment that is stored within the base storage policy class. Special use of the sentinel is described further in section 2.11.4. Note that `Compartment` is a specific feature of the `linear_hashing_storage_policy` and is not required by the `storage_policy` interface.

The `get_bucket` function is a utility function to return a reference to a bucket within the `bucket_vector` to which a given hash value maps.

Expansion and contraction of the hash table is done by the methods `expand()` and `contract()`. This functionality could have been provided through a `resize(n)` method, but it would limit require a mechanism by which the user of the storage policy could compute bucket counts that fit its allocation strategy. The `capacity()` function is also placed within the `storage_policy` interface. This effectively places all memory management logic within the same class which makes it possible to handle all aspects of the memory allocation in a custom fashion.

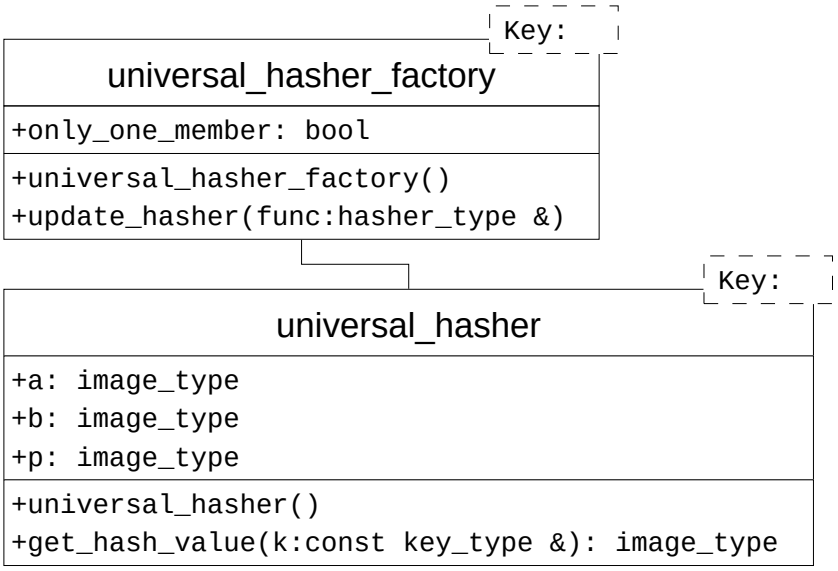

Figure 9. The hasher factory is used to change the hasher.

To support the `swap` method in the external interface an equivalent function must be added to the internal interface. This is not currently implemented. It must also be possible to delete an element both by using an iterator, and by using the element's key. Only the former has been added, but the latter may be solved by including a `delete` method which takes an iterator, to which the iterator-based `erase` calls are delegated.

The implementation of `equal_range` is detailed further in the next section (see figure 11).

5.2.2 Hasher factory

We have chosen to use a factory pattern [17] in the implementation of our hash functions: Instead of letting the hash table receive a hash function as template argument it is given a hasher factory class that represents a class of hash functions. The hash table has an instance of both a hasher and a hasher factory, which are allocated "inline" as part of the hash table instance avoiding one level of indirection when evaluating the function.

The hasher must be initialized by the hasher function factory before it can be used. This initialization consists of modifying internal data elements within the hasher that are used to calculate the actual hash value. This design enables switching of hash functions while resizing or expanding the hash table.

A potential user might wish to add a specific singular hash function and might not be able to or be unwilling to express this hash function as being part of a class of functions. It must therefore be possible to distinguish be-

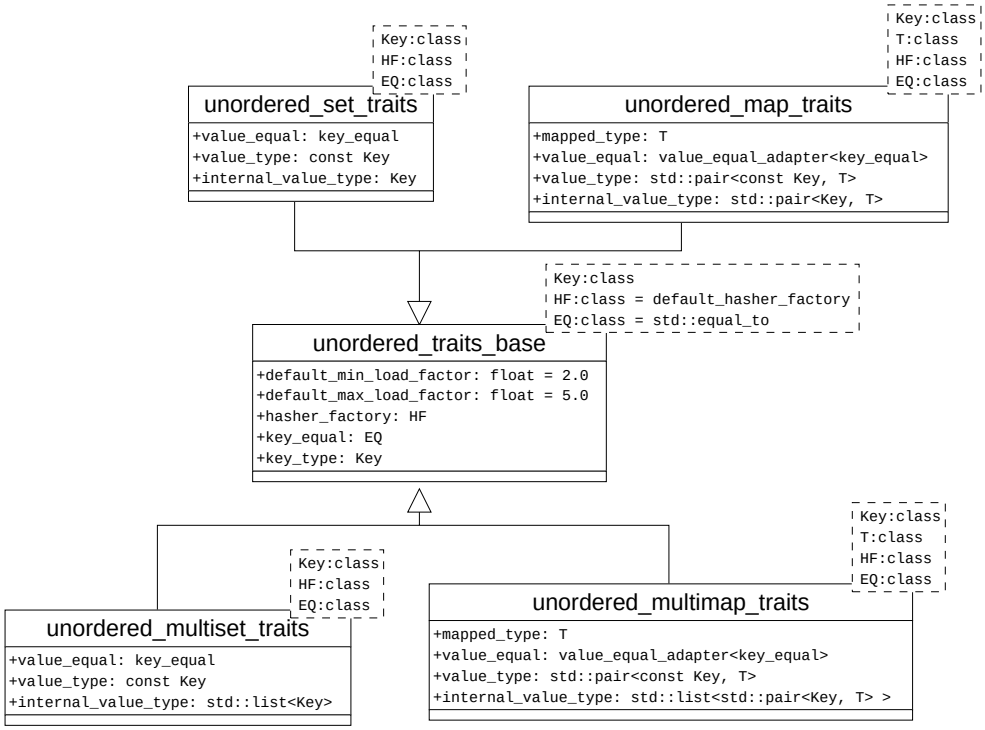

Figure 10. The four traits classes.

tween hasher factories implementing only a single function and hash factories implementing several. For this we have added a boolean data field to each factory, which can be used by the storage policy class to determine what kind of hash function factory that is in use.

6. Collaborations

Having described the interfaces of the various components, we move on to the details of how the various interfaces are realized.

The implementation is centered around the class `hash_table`. By using specialization through four different traits classes (figure 10) `hash_table` provides functionality for all the `unordered_*` interfaces.

The default load factor limits, hasher factory class, equality function and key type fields are shared between the four specializations. The default load factor limits directly correspond to α_{min} and α_{max} mentioned earlier. The values are set to values that give a reasonable tradeoff between speed and memory consumption given our current implementations (see 8.4.2). The `hasher_factory_type` is a class implementing the `hasher_factory` interface. The equality function is a standard predicate, and we have by default used the `equal_to` function object template class from the `functional` STL

header.

6.1 The element type

The `key_type` field is the part of the type `value_type` that is shared between both sets, maps, multisets and multimaps, and is the part that the equality- and hash functions operate on. To conform with `map` and `set`, an equality function for values is also introduced. The `unordered_map` and `unordered_multimap` classes introduce the `mapped_type` to represent satellite data.

6.1.1 Constness

The difference in constness between `internal_value_type` and `value_type` (note that the `Key` part is always non-const in the internal element type while being const in the externally available element type) is easily solved using a cast of the references passed from the storage policy's methods. Iteration, however, requires a more extensive solution. Using the decorator design pattern, `const_key_iterator` implements all the methods of a bidirectional iterator by sub-classing a regular bidirectional iterator and casting the results from `internal_value_type` to `value_type`. All methods being inline, the costs of this wrapper are exclusively compile-time, unless a non-optimized build is performed.

6.1.2 Encapsulation

All traits classes define the `value_type` and `internal_value_type` types, that encapsulate the internal and external element type of the hash table. The distinction between the two comes from the requirement of the element type used by the storage policy to be *assignable*. This is required if storage policies need to copy elements, for instance when performing expansion.

The `value_type` is not available to the storage policy, and the `internal_value_type` is kept opaque to it. This has the desirable consequence that the same storage policy implementation may be used for all four specializations. There is, however, one exception to this rule: When performing the `lookup` operation, the storage policy requires access to the key part of the `internal_value_type` to be able to pass it to the equality function. To avoid introducing dependencies on the structure of `value_type`, we introduce the class `key_adapter`, which function is to retrieve key part of the element.

An alternative to letting the hash table know the internals of `value_type`, would be to use an element type that was opaque even to the hash table, and whose key was implicitly defined as the part that affected the results of the equality- and hash functions. This is a more flexible solution as it allows an arbitrary part of the user's data to function as the key but has two important disadvantages:

1. It will differ from the interfaces of the standard STL associative containers (`map` explicitly defines a `key_type` and a `mapped_type`).
2. It does not aid in enforcing the very important restriction that the part of the element that affects the result of the equality- and hash functions may not be modified after the element has been inserted into the hash table. If this restriction is not heeded, elements may become lost in the table as their position may no longer correspond with their hash value, and the storage policy will consequently be looking in the wrong place. The solution of having an explicit and constant key part of the `value_type` is not a complete solution, but requires considerable work to circumvent[10].

6.2 Multiset and multimap

Having an element type that is opaque to the storage policy, implementing the `unordered_multi_*` container functionality is straightforward. The `unordered_multimap_traits` and `unordered_multiset_traits` classes define `internal_value_type` as a list of `value_type` entries, the effect being that all equal elements may be stored in the same storage policy element. Any duplicates are invisible to the storage policy, and duplicates do not increase the load factor. Additionally, if storing of hash values is used, only one of these will be stored per unique key. Iteration is handled through the adapter `twolevel_iterator` which is initialized with an iterator to a list of lists. Each of these lists are in turn traversed resulting in complete enumeration of every element in the table.

If an additional constraint on the type of the key and the functionality of the equality function is met, an additional optimization may be employed. The constraint is as follows:

> The equality function should be true iff the keys being tested are actually the same object.

If key equality is synonymous with key identity, then only one key is required. The internal element type may be reduced to a key and a list of satellite data, or just a key and a counter in the case of `unordered_multiset`. This optimization has not been implemented, as we have not found a way of determining whether a given equality function has the required properties, and because the `unordered_multimap` solution is incompatible with the established interface: When dereferencing an iterator, we should obtain a reference to an element. However, the `value_type` reference may not in this case be obtained by simply extracting a `value_type` object from the storage policy element, but will have to be assembled from the `key_type` and `mapped_type` parts. The only way of allowing the element to remain mutable will be to create a pair of references to the key and satellite data, but this will not match the expected return type, which is still `value_type`. We have not been able to find a solution to this problem.

[10] One possibility would be to let the key be a pointer into the mapped data, and let the equality- and hash functions act on this data.

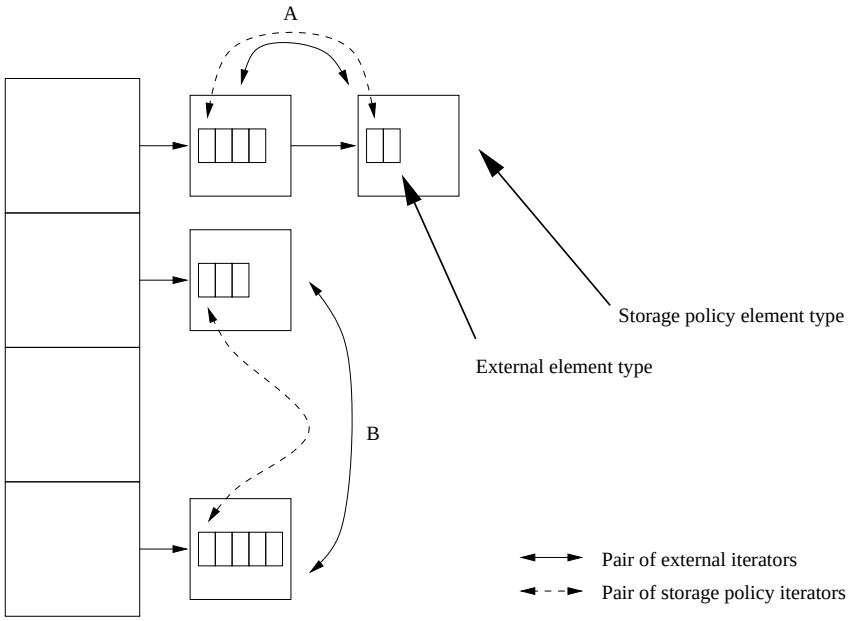

Figure 11. Implementation of `equal_range`. The figure shows retrieval of iterators for two consecutive storage policy elements (A), and two storage policy elements in separate buckets (B). The dashed lines display the iterator pairs that will be exposed to the user, which will be instances of `twolevel_iterator`.

In addition to the standard hash table functions the `unordered_multi_*` containers should support the `equal_range(k)` function. This function returns a pair of iterators defining a range of elements that equal the given key. For the unique element containers, the implementation is trivial. For multiset and multimap the solution is also fairly simple: The start of the range may be retrieved by a standard `lookup` operation. The element one beyond the end of the range (the second iterator in the returned pair) may then be found by iterating one element further within the storage policy (see figure 11). Note that the approach is not limited to bucket-based storage policies, as long as the iteration order is reliable.

6.3 Reverse iterators

We have solved the requirement of reverse iterators by using the STL standard wrapper class `reverse_iterator`[11].

Implementing the `rend` iterator is not straightforward in many data structures, as it represents the element just before the beginning of a sequence (opposite what is stated in [32, page 373]). The wrapper solves this by main-

[11] Previously, a standard wrapper called `reverse_bidirectional_iterator` was available. However, this was deprecated as the standard dictates that unused template functions should not be instantiated, and that the extra functionality provided in `reverse_iterator` therefore would not pose a problem [36, page 44].

taining an internal iterator referring to the element before the element that it appears to refer to, and invoking the post-decrement operator on this iterator within the dereference operator (thus when it appears to refer to `rend`, it is actually referring to `begin`). The upshot of this implementation is that the decrement operator is invoked twice on the wrapped iterator. Because of the complexity of its decrement operator, this has adverse consequences for our chained hashing storage policy implementation (see section 7.2.2). A design solution that would enable the storage policy to suggest an alternative reverse iterator would be beneficial.

7. Implementations

The one-table version of a linear hash table was the first one we implemented using our modular design. It is the implementation that is most true to Larson's original idea of linear hash tables and therefore represents the main focus of this report.

The two-table linear hash table implementation is an attempt to change the allocation strategy of the one-table version in order to provide for the possibility of using different hash functions for each of the tables. It also demonstrates that it is possible to change to a quite different implementation with relatively few lines of code.

The chained hash table was inspired by [21] but extended by using a copy-minimizing extension- and contraction algorithm and using only one sentinel instead of one per bucket.

The chained hash table implementation provides a good baseline against which our modular design can be tested. It is very similar to the SGI implementation and any significant differences in behavior should therefore be fairly easy to explain.

7.1 Common features

All three of our hash table implementations define a *compartment* as the basic unit of storage, which contains the `value_type` and pointers to implement either a singly or a doubly linked list. The enhancement of saving the calculated hash value is used in all three implementations and this hash value is also stored within the compartment.

We envisioned that a gain in performance could be achieved if we could limit data copying by using the `realloc` system call[12] when expanding a hash table using a doubling vector. To our disappointment we found that it is not possible to get access to the `realloc` system call using the standard C++ allocator. We considered a special allocator which would implement this, but found that we would loose portability in doing this, as reallocation support in the standard allocator is non-standard. In [33] it is mentioned

[12] The `realloc` system call attempts to resize a memory block in-place, and will only allocate and copy to a different block if resizing is impossible.

that users wishing access to realloc-like semantics should use a standard vector, but we have been unable to determine if this means that the use of a standard vector is equivalent to the `realloc` system call.

7.2 The chained hash table

7.2.1 Storage

The allocation of the hash table holding the buckets uses a standard doubling vector. The first element is stored in the bucket vector. Subsequent elements are appended to the first element to form a singly linked list of elements. This strategy introduces extra special cases for the `delete` and `insert` operations.

The `insert` operation has been defined in two different versions. One is part of the `storage_policy` interface, the other is for internal use in the `expand` method of the storage policy. The internal version is for inserting existing compartments into buckets. The need of an internal insert method can be illustrated as follows: When inserting an existing compartment as the first element into an empty bucket, the compartment's data must be copied into the vector. The source compartment may safely be deleted after this copy operation because a copy has been made. This is not the case when attempting to insert an element into a bucket that already contains elements. In this case the element may simply be appended to the existing elements by changing the pointers in the linked list.

Some copying is prevented on expansion by expanding the vector and relocating only those elements that now hash to the upper half of the table. Resizing the vector may of course cause all of the buckets to be copied, but this will be a far more cache-efficient copy operation than if it was interleaved with other code, as is the case when copying each bucket independently.

The optimization of providing tight inner loops by using a sentinel described in section 2.11.4 has also been implemented.

7.2.2 Iterators

The implementation of backwards iteration in the chained hash table is very inefficient. This is because the chains connecting elements are singly linked which means that there is no easy way to find the previous element.

In the this case the previous non-empty bucket must be located. When this bucket is found the bucket chain must be traversed to find the element just before the element to which the iterator was previously referring.

We therefore do not recommend iterating backwards through the chained hash table. Fortunately use of backward iteration is not commonly needed.

7.3 Linear hash tables

The one-table and the two-table implementations of linear hash tables share a significant amount of code. The two differ only in their bucket vector orga-

nization. In the following we are therefore referring to both implementations unless otherwise noted.

7.3.1 The table chain

The linear hash table storage policies both define a bucket chain as a segment of a circular doubly linked list containing all compartments in the hash table. The buckets consist of two pointers that delimit a section of the *table chain*. The table chain always contains at least one element, the sentinel, which functions as a place holder to which other compartments may be appended. Using a sentinel proved to be a useful optimization because the special case when the bucket chain becomes empty is avoided. The primary motivation for implementing a table chain is that it makes it possible to iterate over all elements in the hash table with a very small amount of overhead.

We initially had some problems because we had defined the two pointers within a bucket in a manner similar to how the `begin()` and `end()` iterators are defined. The pointer designating the end of the bucket actually referred to the element just beyond the last element. This meant that the end pointer of one bucket and the start pointer of the next bucket coincided. We found that this definition of a bucket actually produced fewer special cases but had the flaw that access to the previous bucket was required when deleting a bucket's first element. Consequently, it performed worse than the alternative implementation where both pointers within a bucket were inclusive. We noted a 16% gain in performance on an initial test of insertion using the inclusive bucket definition.

We do not avoid the bucket access problem entirely, as it should be possible to delete an element by using an iterator referring to it. The problem is that it will become necessary to modify the bucket when the element at the start or at the end of the bucket chain is deleted, and the referred element does not give direct knowledge of which bucket this element lies within. In such cases the bucket in question must be found using the element's hash value, thereby adding to the cost of the delete operation.

7.3.2 Storage

The two-table implementation uses two STL doubling vectors for implementing the two tables as described in section 2.6 with the exception that instead of also allocating a new table, the empty table is resized and reused. This can be easily implemented by calling `swap`.

As mentioned in section 2.9 we use a *space efficient doubling tree* to allocate the bucket vector in the one-table implementation.

7.4 Other parts

7.4.1 Hasher factories

We have implemented two universal classes of hash function which have been used for benchmarking. The `string_table_hasher` is equivalent to the random lookup table function described in section 2.4.2, while the `universal_hasher` is from [29].

7.4.2 Randomization

Due to the varying reliability of the `rand()` functions across different architectures, we have implemented a pseudo random number generator as described by Knuth [24]. The random numbers are used in the hasher factories when switching hash functions.

7.5 Implementation deficiencies

This is a short overview of missing parts of our implementation.

7.5.1 Evaluation of the hash function

The actual calculation of the hash value is not currently done in the storage policy class, which prevents implementation of hash function switching on expansion and contraction. This could potentially be solved by passing the hasher factory template parameter on to the storage policy, and adding a `get_image` parameter to the storage policy interface, which would be delegated to the contained hasher.

7.5.2 Excluding the hash value

The storing of the hash value is a trade for speed at the cost of space. In the current implementations it is not possible to specify that the hash value should not be stored. This is sub-optimal, and we have discussed a couple of methods that would make it possible to not save the hash value.

It could be argued that if the key used in a specific instantiation of a hash table is a simple data type which is the same size or smaller than the stored hash value, there is no benefit gained in storing the hash value and therefore this situation should be automatically detected so that storing of the image is be left out. It should also be possible to explicitly specify if the image should left out or included provided the key is not a pointer or a reference type.

The design of how to optionally leave out the hash value has not been successfully completed. Our proposed solution to this problem is to use a construction similar to that used for the `value_type`. A class similar to the `key_adapter` could be used for taking out the stored hash value from a compartment when this is needed.

8. Benchmarks

We here present an overview of our benchmark configuration and a representative subset of the benchmark results. Including all benchmarks would not be justifiable, because there are much too many (~9000). All benchmarks have been run with the *benz* tool [34, 19] using our three hash table implementations and the existing Google and SGI hash table implementations.

8.1 Input data

The three basic hash table operations insert, lookup and delete and also forward iteration, have been benchmarked with variable sized strings, character arrays of fixed size (10 bytes), and machine word sized data (integers).

Random input data is used as well as data aimed at being representative of the kinds of data that could be used in realistic setting. The following lists detail the kind input data used:

Strings

paths Variable sized strings collected with the `locate` Unix program.

words A large word list collected from Unix dictionaries and text files from `http://www.gutenberg.org`.

random Printable iso8859-1 strings generated using a pseudo random number generator.

Integers

genes Human genome data [9] converted to binary data by representing amino acids with 2 bits.

random Marsaglia random number collections [28].

8.2 Configuration

The metrics that we have chosen to focus on for each operation follow from the specification of usage profiles from section 2.1. They are total execution time, response time (standard deviation of execution time per operation), amount of memory allocated, cache accesses, and cache miss ratio. Some of these metrics required the PAPI [13] extensions.

Not all combinations of the four above mentioned operations and these metrics make sense. Measuring memory allocation for an operation like iteration is not interesting because the amount of memory allocated remains constant. A number of unwanted measurements have therefore been excluded. A diagram of the actual benchmarks run can be seen in figure 12.

All of the benchmark programs were compiled with `gcc` using the `-O2` optimization flag.

8.3 Machines

The following two machines have been used to run the benchmarks. Both machines have a *cache line size* (often referred to as *block size*) of 64 bytes.

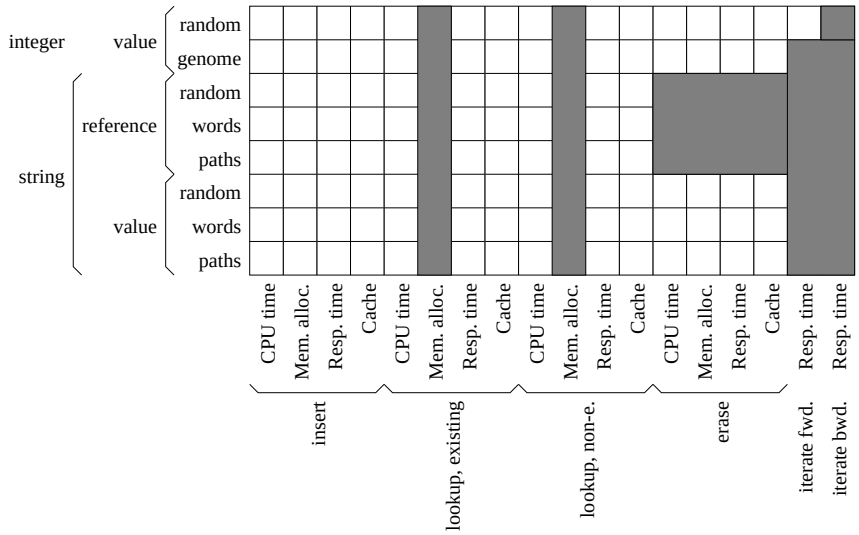

Figure 12. The benchmark schedule. Skipped combinations are grayed.

8.3.1 Pentium 4

All benchmarks that have been run on a Pentium 4 have been run on one of the student servers at DIKU. The hardware specifications of these machines are identical and we will therefore refer to them as "the Pentium 4 machine" in the following sections. The machines were all running gentoo Linux 2.6.10, and the CPU specifications were as follows [18]:

CPU	2 x Intel Pentium 4 CPU 3.00GHz
RAM	2 GB
L1 Cache	12 K- decoded μ Ops execution trace + 16KB data cache, 4-way
L2 Cache	1 MB, 8-way

8.3.2 AMD 64

This machine was running gentoo Linux 2.6.9 with PAPI extensions and a minimum amount of running processes. CPU specifications were found in [1]

CPU	1 x AMD64 2GHz 3000+
RAM	512 MB
L1 Cache	64 KB data and instruction cache, 2-way
L2 Cache	512 KB, 16-way

8.4 Measurements

Extensions to the benchmark tool were written to support CPU cycle count measurements and memory allocation. A special allocator was used for the

latter extension. The results of benchmark runs were collected in a set of files enabling later extractions and analysis by for instance by using `gnuplot`.

8.4.1 Limitations

The PAPI extensions were not available on the Pentium 4 and therefore some measurements are missing from benchmarks that were run on this machine. Moreover, it is not possible to instantiate the Google hash tables with a custom allocator wherefore no memory allocation data has been gathered for these implementations.

We have consistently used the `string_table_hasher` (random table lookup method) for string data and the `universal_hasher` (the method of [29, section 4]) for integer data.

Only forward iteration has been benchmarked because neither the Google nor SGI hash tables support reverse iteration.

8.4.2 Load factors

The Google hash tables set the maximum load factor to 0.8. The SGI hash tables use a strategy that effectively limits the maximum load factor to 1. We initially used a maximum load factor of 5, but have later changed this to 1 to be able to compare the behavior of our implementations better to that of the other hash tables. Using a load factor of 1 our hash tables were able to compete with the others in most of the benchmarks.

8.5 Results

The results of the benchmarks are presented below.

8.5.1 Throughput

We have measured the throughput of operations using two methods: CPU time which is the total number of seconds taken to perform a given amount of operations, and also the number of CPU clock-cycles used for each operation.

Our hash tables perform quite well on `insert` operations as can be seen on figure 14. The chained hash table performs particularly well in this benchmark which can be attributed to the fact that the first compartment is kept within the bucket vector and thus may prevent one cache miss. Insertion of character arrays is not as efficient, probably due to the price of copying larger data blocks into this first compartment and the resulting thrashing that this introduces. A similar behavior is exhibited with delete operations (see figure 13).

Because of the cache access saved by storing hash values we see an increase in performance compared to the competing implementations when the keys are accessed by reference as opposed to by value (also displayed in figure 14).

Much of the good performance exhibited on insertion and deletion is connected to the performance of the `lookup` operation. In these benchmarks

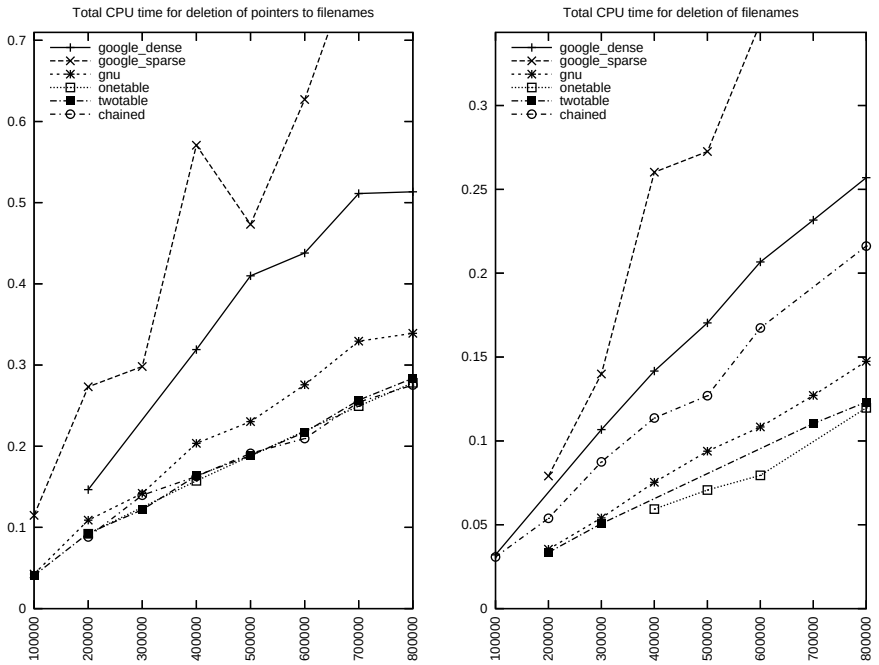

Figure 13. Illustration of the performance penalty of the chained implementation having to copy large datatypes into the bucket vector. The figure shows pointers to filenames (4 bytes, left) versus actual filenames (10 bytes, right). The other implementation stay mostly in place relative to each other, while the chained implementation makes a jump in execution time. (The missing points to the left on both figures are caused by benz trying to remedy the low precision of the `clock()` call by re-running the benchmark until 90% of the measurements come within 20% of the mean, but giving up if this cannot be done within 100 measurements.)

the `lookup` operation within the `insert` operation is always an unsuccessful search. Conversely, the `delete` operations all start by performing a `lookup` on an existing element. More elements will have to be traversed on average when performing a lookup of an element that does not exist as the entire linked list will have to be traversed.

The `lookup` operation exhibits the best performance in the chained hash table and the worst in the linear hash tables (see figure 15). We believe that this is caused by a differences in the method by which the two linear hash tables and the chained hash table search for an element within a bucket. Often lookups in the chained hash table occur within the bucket vector of which parts are likely to be stored in the L1 or L2 cache. This hash table also uses a tight inner loop as described in section 2.11.4, causing further speedup. The linear hash tables implement a doubly linked list and therefore have one more pointer per compartment than the chained hash table. We think that this may be exhausting the cache resources more quickly, which will contribute to a performance loss.

As expected, the open hashing methods suffer from unevenly distributed

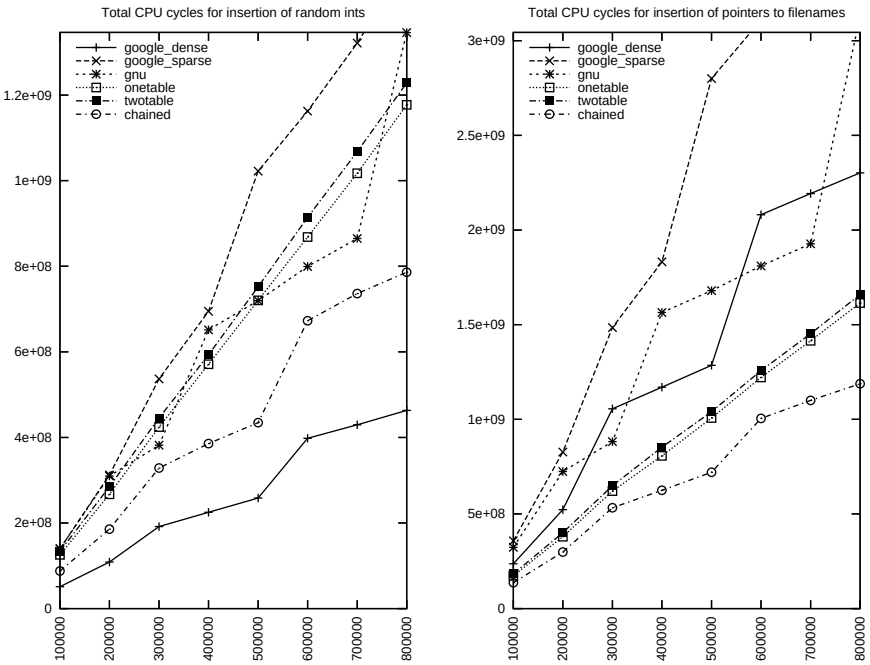

Figure 14. These figures illustrate the effect of storing hash values: Our hash tables perform satisfactorily when accessing the key by value (left), but are superior when the key is accessed by reference (right). In both figures, the maximum load factor is 1. The measurements were performed on the AMD64.

data on deletion (see figure 16).

8.5.2 Response time

Because the buckets are defined as a section of a circular linked list containing all elements, iteration on the linear hash tables exhibits the least amount of variability. Iterating over the linear hash tables is not as fast as iterating over the chained- and Google hash tables which all benefit from cache prefetch, as they traverse the elements in order of increasing addresses (see figure 17). (Note that this is not true of the chained hash table if the chains are longer than one element.)

For the `lookup` and `delete` operations a higher load factor generally introduces a higher amount of variability because the linked lists become longer. For the `insert` operation on the linear hash, the inverse is true, as a low load factor incurs bucket splitting on a larger proportion of the insertions.

8.5.3 Allocation

With a max load factor limit of five the chained hash table takes up the least memory, followed by the one- and two-table implementations, respectively.

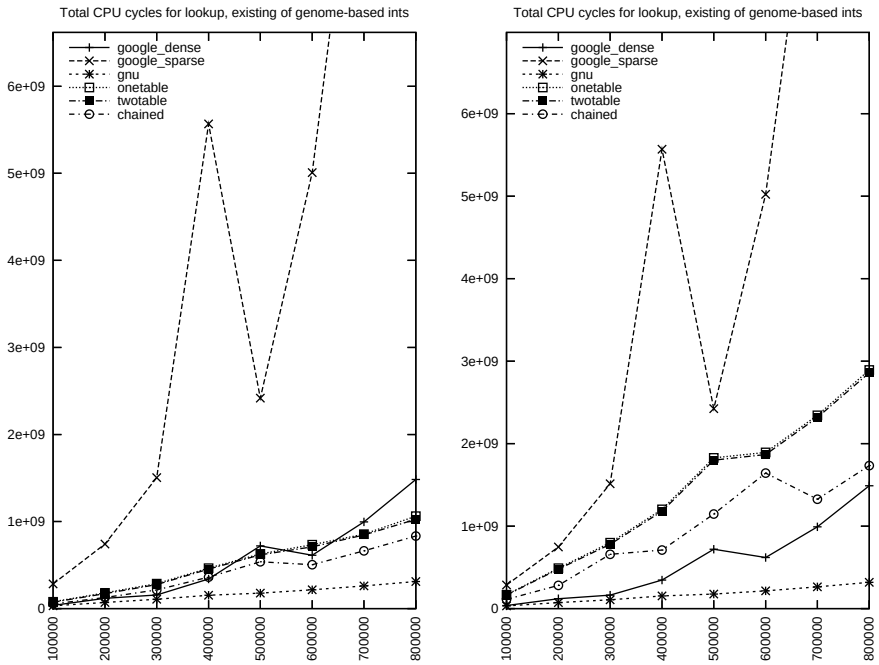

Figure 15. Lookup operations for existing elements[14] with a maximum load factor of 1 (left) and 5 (right).

This advantage is lost when decreasing the maximum load factor as can be seen on figure 18. The chained and one-table implementations now use about the same amount of memory on average. The drop in load factor affects the chained implementation more severly because its buckets are bigger.

Note that the one-table implementation allocates precisely the same amount of memory per element regardless of how many elements it contains. Also, it is worth noting that in the lower left figure the chained hash table appears to be using an amount of memory that is linearly proportional to the number of elements. This behaviour is caused by the expand operation "pulling in" and deallocating most of the externally allocated compartments when expanding the bucket vector, as the chains naturally become shorter. This dampens the effect of expanding the bucket vector, and reduces the "spikes" to the point where they almost disappear.

9. Discussion

9.1 A comment on measurement accuracy

Initially, the clock cycle count measurements were included only for measuring variability in execution time per operation. We then found that we could easily use this an alternative means of measuring total execution time, for which the "CPU time" benchmark was originally intended. This uncov-

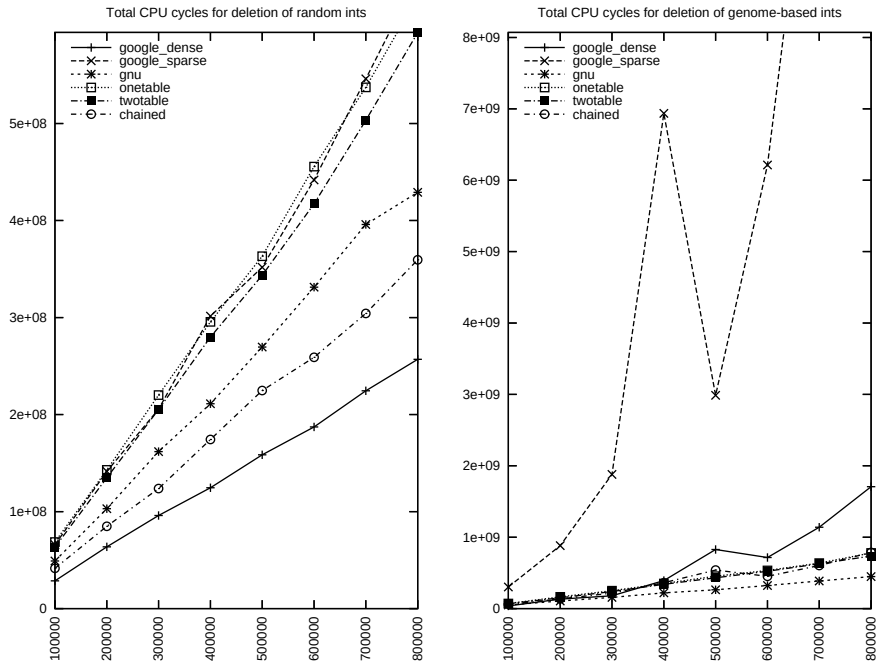

Figure 16. These figures illustrate the performance issues with open hashing and deletion. When the data are evenly distributed, the performance is on par with chaining-based implementations (left), but there is a significant penalty associated with uneven distributions (right). The measurements were performed on the AMD64.

ered some surprising discrepancies, for which the exact reasons are yet to be covered. We do, however, have two possible hypotheses.

Measurements have indicated that there are only 100 distinct values per second available from the `clock()` system call used to measure CPU time. Additionally, the counter accessed by `clock()` is maintained by the operating system's scheduling algorithm, which may or may not be designed to provide precise accounting.

On the other hand, the clock cycle count measurements are taken on a single operation basis, which introduces PAPI calls within the inner loop of the benchmark. An additional run is performed where only the PAPI code is executed, which is subtracted from the first, but we suspect that more subtle effects may occur. One of these is that the small delay introduced between each hash table operation may allow cache prefetch to function more efficiently, as there is more time for the L1 and L2 caches to fetch data. We have not been able to determine if this is the case.

9.2 On the successfulness of our design

As expected, insertion time in the linear hash table based implementations had a very low variability. However, the low cost of these insertions was

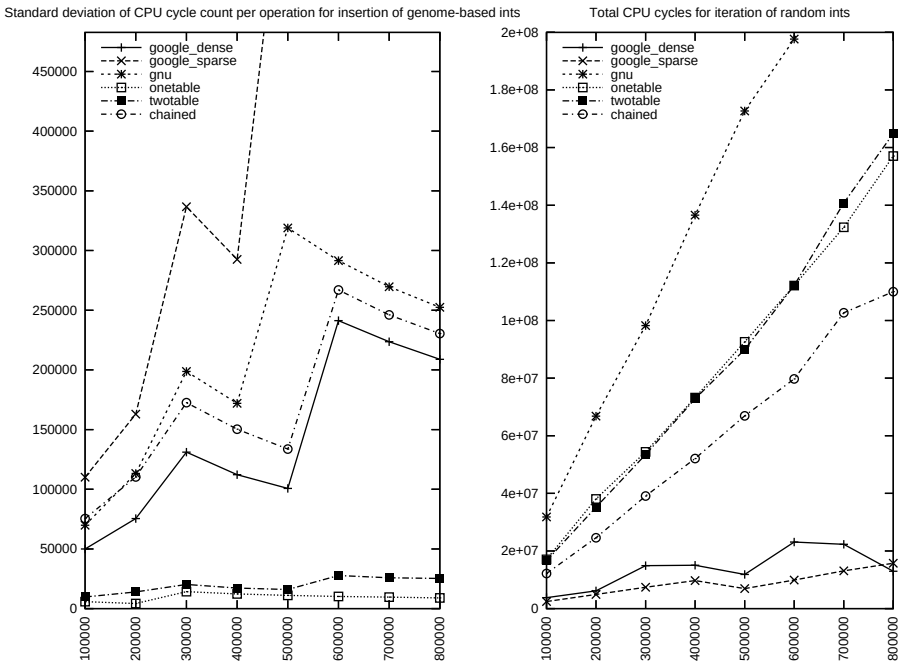

Figure 17. The linear hashing based implementations have very low variability on insert operations (left), while open hashing has a major cache advantage when it comes to iteration (right). Maximum load factor is 1 in both figures.

surprising, especially at a maximum load factor of 1, where an expansion operation is triggered on every insertion. We relate this to the lowered thrashing caused by performing gradual data copying.

On the other hand, the lookup operations were unexpectedly slow in both the linear and chained hash table implementations. The exact cause of this is still unknown, but we suspect that the large size of our compartment structure compared to that of the rivaling implementations may exhaust cache resources more rapidly. Further work should be done in this area.

Another surprise, was the performance of iteration when not using a table chain. The table chain guarantees constant time iteration, but the cache benefit of accessing the elements by increasing memory addresses dwarves this advantage. Our computations regarding expected distance between non-empty buckets also appear to hold, as we do not see a significant reduction in iteration performance in the chained hash, even when the load factor is high (and the cache advantage therefore is smaller).

We conclude that the storage of hash values is still a sound optimization, as our implementations show a smaller performance penalty on access by reference as opposed to access by value than the competitors. However, we have not performed measurements of which beneficial effect is more important: The reduction in hash function evaluation time, or the reduction

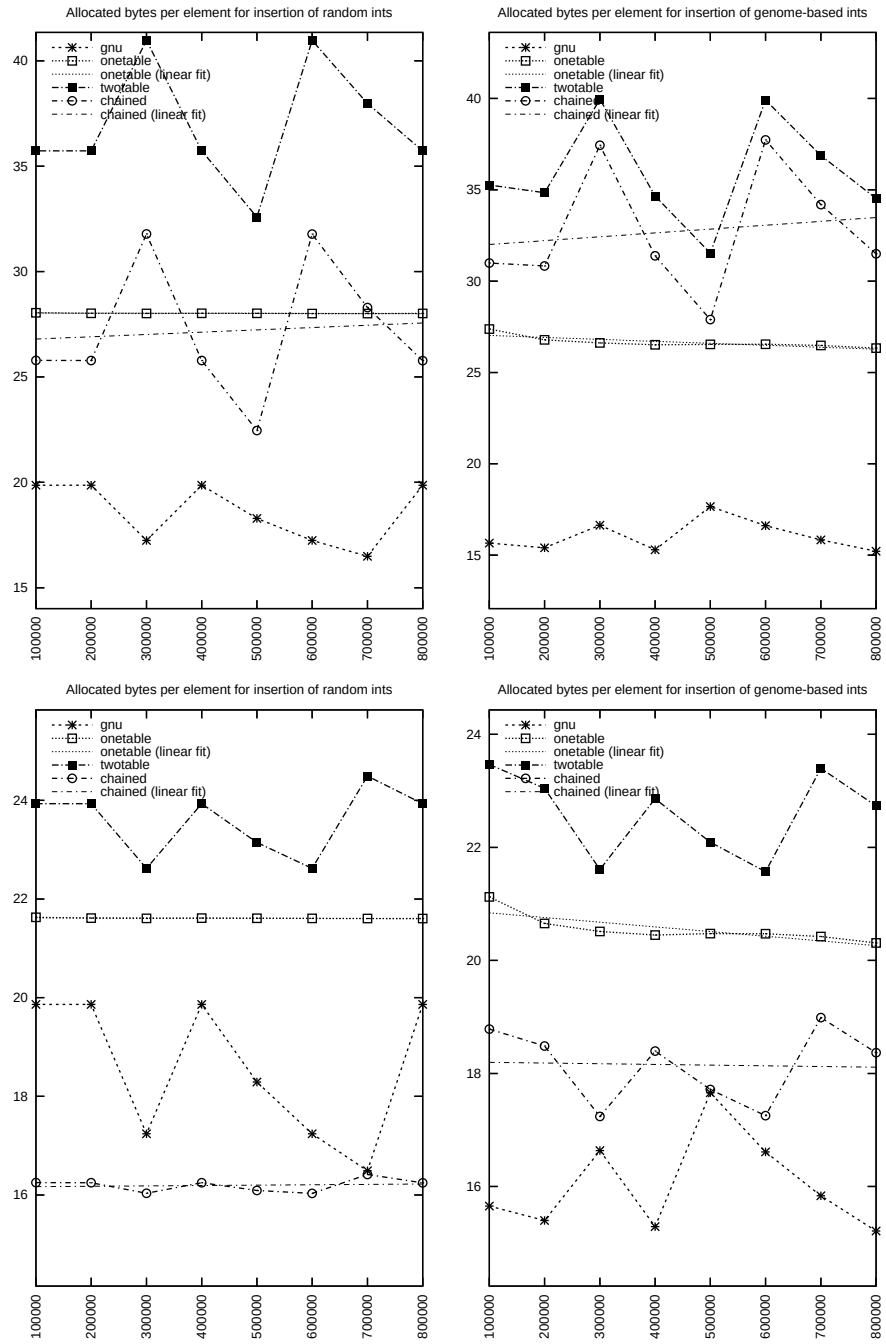

Figure 18. A number of interresting features are displayed in these memory allocation figures: The chained implementation suffers from increased waste when the distribution is uneven (right), but is superior at a maximum load factor of 5 (bottom). However, all our hash table implementations are fairly expensive at a maximum load factor of 1 (top). The one-table implementation, due to the use of a space-efficient data structure, is almost constant in its allocation per element. The two-table and chained implementations use the same expansion scheme and therefore follow the same pattern. The fall in the righmost one-table curves is caused by a small number of duplicates in the genome input data. The measurements were performed on the Pentium 4, which has a pointer size of 4 bytes.

in the number of cache misses. Also, the memory consumption penalty is significant.

Finally, the storage of the first compartment within the bucket vector that is used in the chained hash is a good memory consumption optimization if the distribution is even, but looses much of its advantage when buckets are left empty. It is also vulnerable performance-wise when the element type is large because of increased copying.

9.2.1 Future work

We have performed a broad array of benchmarks, but additional measurements should be done to clear up some unresolved issues.

We expect that our varied distributions give a fairly realistic picture of real-world performance, but we do not vary our operation distribution, and because of this, non-synthetic benchmarks should also be run. This we did not do because of time concerns.

No measurements of chain lengths were performed, and our assumptions regarding chain lengths are still just based in theory.

As noted above, a benchmark should also be performed to determine whether the storage of hash values benefits cache performance or hash function evaluation time the most.

In this report, we have also suggested two potential optimizations that were not implemented, but which should be part of a final implementation. The first of these is extending the design to allow for leaving out the hash value from the compartment structure on small key types. This removes a potential handicap that our implementations suffer from in those scenarios. Secondly, we expect that keeping the chains in sorted order would significantly increase both insertion and lookup performance in those cases where the element in question does not already exist in the data structure.

10. Conclusion

The purpose of this paper was to present and evaluate a hash table implementation for the CPH STL. We have covered and adapted the necessary theory, and the two-table variant of linear hashing has been introduced, which facilitates the interesting theoretical concept of switching between universal hash functions while expanding a table.

A modular design has been presented, which provides a more adaptive alternative to the one currently proposed by STL standard, allowing the user to freely choose the policies that best suit his or her specific application requirements, and permitting a wider variety of implementations.

The resulting implementation was compared to those of Google and the SGI STL in varied synthetic benchmarks, and was found to be competitive both in memory consumption, performance, and response time measurements. Additional areas of improvement were indicated.

The linear hashing algorithms that we have presented are found to be superior when it comes to throughput on insertions and response time on insertions and deletions. The chaining implementation dominates when memory consumption and accumulated throughput is considered. In certain scenarios the open addressing-based implementations prove to be exceptionally efficient. We relate this advantage to its cache friendliness.

We strongly believe that a modular design is the only choice that provides optimal performance in diverse scenarios, as no current algorithms provide a perfect solution in all situations. The linear hashing algorithm coupled with the table chain structure does however provide a good all-round solution, and would be a suitable default algorithm in such a design. We conclude by mentioning that a modular design depends on a good compiler. Considering the impressive ongoing progress in compiler technology, we do not count this as a roadblock.

Acknowledgements

We would like to thank our supervisor Jyrki Katajainen for being a great inspiration and source of ideas.

References

[1] AMD, *AMD Athlon 64 Product Data Sheet*, Webpage (2004). Available at `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24659.PDF`.

[2] M. Austern, N1456: A proposal to add hash tables to the standard library (revision 4), *C++ Standards Committee Papers* (2003). Avaliable at `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1456.html`.

[3] M. Austern, N1712: Library extension technical report — issues list, *C++ Standards Committee Mailings* (2004). Avaliable at `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1712.pdf`.

[4] D. Bellin and S. S. Simone, *The CRC Card Book*, Addision Wesley (1997).

[5] C. Boesgaard and J. C. Poulsen, CPH STL – hash map, CPH STL Report 2001-5, DIKU, University of Copenhagen (2001).

[6] G. S. Brodal, Partially persistent data structures of bounded in degree with constant update time., *Nordic Journal of Computing* **3** (1996), 238–255.

[7] J. L. Carter and M. N. Wegman, Universal classes of hash functions, *Journal of Computer and System Sciences*,18 (1979), 143–154.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd Edition, MIT Press (2001).

[9] Deloukas et al., *Human Genome Project*, Webpage (2000). Gene data available at `http://www.gutenberg.org`.

[10] C. DiBona et al., *Google sparse hash*, Webpage (2005). Available at `http://goog-sparsehash.sourceforge.net`.

[11] M. Dietzfelbinger, Universal hashing and k-wise independent random variables via integer arithmetic without primes, *Lecture Notes In Computer Science; Vol. 1046, Proceedings of the 13th Annual Symposion on Theoretical Aspects of Computer Science*, Springer-Verlag, London (1996), 569–580.

[12] M. Dietzfelbinger et al., A reliable randomized algorithm for the closest-pair problem, *Journal of Algorithms* **25** (1997), 19–51.

[13] J. Dongorra et al., *PAPI — Performance Application programming interface* (2005). Available at `http://icl.cs.utk.edu/papi`.

[14] M. Feathers et al., *CppUnit Wiki*. Available at `http://cppunit.sourceforge.net/cgi-bin/moin.cgi/FrontPage`.

[15] J. Fenlason, R. Stallman, et al., *GNU gprof — The GNU Profiler* (1997). Available at `http://www.gnu.org/software/binutils/manual/gprof-2.9.1/html_mono/gprof.html`.

[16] The Free Software Foundation, *gcov — a Test Coverage Program* (2000). Available at `http://gcc.gnu.org/onlinedocs/gcc-3.3.5/gcc/Gcov.html`.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*, Addison-Wesley Longman Publishing Co., Inc., Boston (1995).

[18] Intel Corporation, *Intel Pentium 4 Processor — Product Overview*, Webpage (2005). Available at `http://intel.com/design/pentium4/prodbref`.

[19] J. Katajainen, *Brief introduction to the Benz benchmarking tool* (2003). Available at `http://www.cphstl.dk/Presentation/4th-STL-workshop/First-four-months/Jyrki-26.05.2003.pdf`.

[20] J. Katajainen, The cost of iterator validity, *CPH STL Presentations* (2004). Available at `http://www.cphstl.dk/Presentation/Gothenburg-2004/Jyrki-12.05.2004.ps`.

[21] J. Katajainen and M. Lykke, Experiments with universal hashing, Technical report, Copenhagen, Denmark (1996).

[22] J. Katajainen and B. B. Mortensen, Experiences with the design and implementation of space-efficient deques, *Lecture Notes in Computer Science* **2141**, Springer-Verlag, Berlin/Heidelberg (2001), 39–50.

[23] R. Klarer, *N1459: Minutes of J16 Meeting No. 36/WG21 Meeting No. 31, April 7-11*, Webpage (2003). Avaliable at `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1459.html`.

[24] D. E. Knuth, Uniform random numbers, *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*, Addison-Wesley Longman Publishing Co., Inc., Boston (1997), 10–40.

[25] D. E. Knuth, Hashing, *The art of computer programming, volume 3: (2nd ed.) sorting and searching*, Addison Wesley Longman Publishing Co., Inc., Redwood City (1998), 513–558.

[26] M. D. Kristensen, Vector implementation for the CPH STL, CPH STL Report 2004-2, DIKU, University of Copenhagen (2004).

[27] W. Litwin, Linear hashing: A new tool for file and table addressing, *Sixth International Conference on Very Large Data Bases, October 1-3, 1980, Montreal, Quebec, Canada, Proceedings*, IEEE Computer Society (1980), 212–223.

[28] G. Marsaglia, *The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness*, Webpage (1995). Available at `http://stat.fsu.edu/pub/diehard/`.

[29] P. B. Miltersen, Universal hashing, *Lecture notes in "Pearls of Theory" course*, University of Aarhus (1998).

[30] P.-Å. Larson, Dynamic hash tables, *Commun. ACM* **31**,4 (1988), 446–457.

[31] P. K. Pearson, Fast hashing of variable-length text strings, *Commun. ACM* **33**,6 (1990), 677–680.

[32] P. J. Plauger, A. A. Stepanov, M. Lee, and D. R. Musser, *The C++ Standard Template Library*, Prentice Hall PTR, Upper Saddle River (2001).

[33] B. Stroustrup, *The C++ Programming Language*, 3rd Edition, Addison-Wesley (2004).

[34] C. U. Søttrup and J. G. Pedersen, CPHSTL's benchmark verktøj, CPH STL Report 2003-1, DIKU, University of Copenhagen (2003).

[35] M. Thorup and Y. Zhang, Tabulation based 4-universal hashing with applications to second moment estimation, *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms* (2004), 615–624.

[36] D. Vandevoorde and N. M. Josuttis, *C++ Templates: The complete guide*, Addison-

Wesley Longman Publishing Co., Inc., Boston (2002).

AppendixA. Source Code Listings

AppendixA.1 Reading the source code

Appendix A is divided into the four sections: Hash table core classes, storage policies and adapters, decorators and utilities, of which we will give a short overview here.

AppendixA.1.1 Hash table core classes

These are the classes that encapsulate the storage policies. Eventually, instantiation of `hash_table` will be wrapped in `unordered_*` classes.

`traits.h` This file contains definitions of the traits classes that are used for the four different associative containers.

`hash_table.h` Contains the hash table class that is used by all of the storage policies and associative containers.

`hasher_factory.h` Herein definitions of hashers and hasher factories are found. there is also an example of a single function factory.

AppendixA.1.2 Storage policies

The files in this section implement the memory allocation strategies used by the three hash table implementations, and the definitions of their compartments and iterators.

`linear_hashing_storage_policy.h` Defines a base class used by both the one-table and the two-table hash tables.

`onetable_linear_hashing_storage_policy.h` The central part of the one-table implementation.

`twotable_linear_hashing_storage_policy.h` The central part of the two-table implementation.

`linear_hashing_compartment.h` In this file the basic compartment used by both of the linear hash table implementations is defined. It implements a doubly linked list for bidirectional iteration.

`linear_hashing_compartment_iterator.h` This file contains the iterator used by the one-table and the two-table hash tables.

`chained_hash_storage_policy.h` This is the central part of the chained hash table definition.

`chained_hash_compartment.h` This is the compartment used by the chained hash table. It implements a singly linked list.

`chained_hash_compartment_iterator.h` The iterator used by the chained hash table.

AppendixA.1.3 Adapters, decorators and utilities

A collection of utility classes.

`key_adapter.h` This is the utility used to extract the key part from a `internal_value_type`. For the `unordered_multimap` and `unordered_multiset` to be implemented, this will have to be extended to handle the lists of `internal_value_type` that these will use.

`const_key_iterator.h` This is a definition of the external iterator which makes the key const.

`twolevel_iterator.h` This is the definition of the iterator that will be used by the `unordered_multimap` and `unordered_multiset`.

`emulist.h` Contains a class that emulates a list of elements without size. To be useful, it should be rewritten to emulate a list of equivalent elements.

`random.h` This contains a reliable random number generator used by the hasher factories.

AppendixA.2 Hash table core classes

AppendixA.2.1 Traits classes (`traits.h`)

```
1  #include <functional>
2  #include <list>
3
4  #include "emulist.h"
5  #include "hasher_factory.h"
6
7  #ifndef _TRAITS_H_
8  #define _TRAITS_H_
9
10 /// @todo Implement
11 template <class Foo>
12 class value_equal_adapter {};
13
14 template <class Key, class HF = universal_hasher_factory<Key>,
15     class EQ = std::equal_to<Key> >
16 struct unordered_map_traits_base {
17     static const float default_min_load_factor;
18     static const float default_max_load_factor;
19
20     typedef Key key_type;
21
22     typedef HF hasher_factory_type;
23     typedef EQ key_equal;
24
25     /// @todo Place this where it fits
26     //typedef key_adapter<value_type> key_adapter;
27 };
28
29 template <class Key, class HF = universal_hasher_factory<Key>,
30     class EQ = std::equal_to<Key> >
31 struct unordered_set_traits : public unordered_map_traits_base<Key, HF
      , EQ> {
32     /// @todo See if there's a simpler way around the "key_equal is
          implicitly
33     /// a typename" warning.
34     typedef typename unordered_map_traits_base<Key, HF, EQ>::key_equal
           value_equal;
35     typedef const Key value_type;
36     typedef Key internal_value_type;
37
38     enum { MULTI = false };
```

```
39 };
40
41 template <class Key, class HF = universal_hasher_factory<Key>,
42     class EQ = std::equal_to<Key> >
43 struct unordered_multiset_traits : public unordered_map_traits_base<
       Key, HF, EQ> {
44     /// @todo See if there's a simpler way around the "key_equal is
           implicitly
45     /// a typename" warning.
46     typedef typename unordered_map_traits_base<Key, HF, EQ>::key_equal
           value_equal;
47
48     /// @todo See if the emulist should be defined based on the
           storage policy's
49     /// size_type
50     typedef emulist<size_t> datalist_type;
51     typedef const Key value_type;
52     typedef std::pair<Key, datalist_type> internal_value_type;
53
54     enum { MULTI = true };
55 };
56
57 template <class Key, class T, class HF = universal_hasher_factory<Key
       >,
58     class EQ = std::equal_to<Key> >
59 struct unordered_map_traits : public unordered_map_traits_base<Key, HF
       , EQ> {
60     typedef T mapped_type;
61
62     /// @todo See if there's a simpler way around the "key_equal is
           implicitly
63     /// a typename" warning.
64     typedef value_equal_adapter<typename unordered_map_traits_base<Key
           , HF, EQ>::key_equal>
65         value_equal;
66     typedef std::pair<const Key, T> value_type;
67     typedef std::pair<Key, T> internal_value_type;
68
69     enum { MULTI = false };
70 };
71
72 template <class Key, class T, class HF = universal_hasher_factory<Key
       >,
73     class EQ = std::equal_to<Key> >
74 struct unordered_multimap_traits : public unordered_map_traits_base<
       Key, HF, EQ> {
75     typedef T mapped_type;
76
77     /// @todo See if there's a simpler way around the "key_equal is
           implicitly
78     /// a typename" warning.
79     typedef value_equal_adapter<typename unordered_map_traits_base<Key
           , HF, EQ>::key_equal>
80         value_equal;
81     typedef std::pair<const Key, T> value_type;
82
83     /// @note Ideally we'd want this to store only one key, but this
           prevents
84     /// us from implementing a standard iterator.
85     typedef std::list<std::pair<Key, T> > datalist_type;
86     typedef datalist_type internal_value_type;
87
88
89     enum { MULTI = true };
90 };
```

```
91
92 template <class Key, class HF, class EQ>
93 const float unordered_map_traits_base<Key, HF, EQ>::
       default_min_load_factor = 2.0f;
94 template <class Key, class HF, class EQ>
95 const float unordered_map_traits_base<Key, HF, EQ>::
       default_max_load_factor = 5.0f;
96
97 #endif //_TRAITS_H_
```

AppendixA.2.2 Hash table (`hash_table.h`)

```
1  #ifndef _HASH_TABLE_H_
2  #define _HASH_TABLE_H_
3
4  #include <memory>
5  #include <utility>
6
7  #include "onetable_linear_hashing_storage_policy.h"
8  #include "twotable_linear_hashing_storage_policy.h"
9  #include "chained_hash_storage_policy.h"
10 #include "twolevel_iterator.h"
11 #include "const_key_iterator.h"
12
13 #include "hasher_factory.h"
14 #include "key_adapter.h"
15 #include "mr_trace.h"
16
17 /**
18  * Internal hash table class
19  *
20  * @note The default arguments for compartment and storage policy has
        non-const
21  * Key parameters. This is because we must be able to assign copy
        value_t's
22  * internally.
23  *
24  * @todo Find the proper type for the allocator; it'll only be a dummy
         type, but
25  * we should use existing container conventions as guidelines.
26  *
27  * @todo Find a way of supporting search optimization by sorting the
        buckets:
28  * This requires a less-operation instead of an equal-operation.
29  *
30  * @note When implementing multi_*, create two classes, one for map
        and one for
31  * set that can act as data lists, and which contain methods for
        inserting and
32  * deleting elements. In multi_set, they will modify a counter, and
        will not
33  * have arguments. In multi_map, they'll actually insert and delete
        elements.
34  * The delete methods are only valid within an iteration context. The
        data
35  * classes should therefore have iterator types. These iterator types
        must be
36  * available to the storage policy, as the storage policy iterator
        must use
37  * these internally.
38  *
39  * @todo See if one can specialize a template on void.
40  *
41  * @todo Remove allocator argument to the hash table. Only storage
        needs it.
```

```
42   *
43   * @note The compartment type is no longer required for all storage
         policies,
44   * and may be omitted in user-defined implementations.
45   */
46  template <class Traits,
47      class Allocator = std::allocator<typename Traits::
            internal_value_type>,
48      class StoragePolicy = twotable_linear_hashing_storage_policy<
49          typename Traits::internal_value_type,
50          typename Traits::hasher_factory_type,
51          typename Traits::key_equal,
52          Allocator,
53          linear_hashing_compartment<typename Traits::
                internal_value_type> > >
54  class hash_table {
55  private:
56      typedef hash_table<Traits, Allocator, StoragePolicy> this_type;
57      typedef typename StoragePolicy::iterator internal_iterator;
58
59  public:
60      typedef typename Traits::key_type key_type;
61      /// @todo Handle situation where this type is missing (i.e. all *
            _set
62      /// structures)
63      typedef typename Traits::mapped_type mapped_type;
64      typedef typename Traits::value_type value_type;
65
66      /// @todo Define these using the allocator object and rebind.
67      typedef typename Traits::value_type &reference;
68      typedef const typename Traits::value_type &const_reference;
69      typedef typename Traits::value_type *pointer;
70      typedef const typename Traits::value_type *const_pointer;
71
72      typedef StoragePolicy storage_policy;
73
74      typedef typename StoragePolicy::bucket_type bucket_type;
75      typedef typename StoragePolicy::size_type size_type;
76
77      typedef const_key_iterator<value_type, reference, const_reference,
78          pointer, const_pointer, internal_iterator> iterator;
79
80      //typedef twolevel_iterator<internal_iterator,
81      //  typename Traits::datalist_type> iterator;
82
83      // See [Plauger et al., 2000, p44] for why we do not use
84      // reverse_bidirectional_iterator
85      typedef std::reverse_iterator<iterator> reverse_iterator;
86
87      /// @todo Define const iterators in the storage policies, or
            define a const
88      /// iterator wrapper.
89      typedef iterator const_iterator;
90      typedef reverse_iterator const_reverse_iterator;
91
92      typedef typename Traits::key_equal key_equal;
93      typedef typename Traits::value_equal value_equal;
94
95      typedef typename Traits::hasher_factory_type hasher_factory_type;
96      typedef typename hasher_factory_type::hasher_type hasher_type;
97      typedef typename hasher_type::image_type image_type;
98
99  private:
100     typedef typename Traits::internal_value_type internal_value_type;
101
```

```
102     key_adapter<internal_value_type> key_access;
103
104     storage_policy storage;
105
106     float _min_load_factor;
107     float _max_load_factor;
108
109     /// @note If the max load factor is above 1.0, then size_type will
            be
110     /// insufficient to represent a full table.
111     size_type element_count;
112     size_type compartment_count;
113     size_type min_compartment_count;
114     size_type max_compartment_count;
115
116     hasher_factory_type hasher_factory;
117     hasher_type hasher;
118
119 public:
120     /// @todo Remove, outdated.
121     hash_table(size_type initial_size = 4,
122         float min_load_factor = Traits::default_min_load_factor,
123         float max_load_factor = Traits::default_max_load_factor) :
124         storage(initial_size),
125         _min_load_factor(min_load_factor),
126         _max_load_factor(max_load_factor),
127         element_count(0),
128         compartment_count(0),
129         min_compartment_count((size_type) (initial_size *
                min_load_factor)),
130         max_compartment_count((size_type) (initial_size *
                max_load_factor))
131     {
132         hasher_factory.update_hasher(hasher);
133     }
134
135     hash_table(size_type initial_size, const hasher_type &hasher,
136         const key_equal &key_equal) {
137         /// @todo Implement
138     }
139
140     hash_table(size_type initial_size, const hasher_type &hasher) {
141         /// @todo Implement
142     }
143
144     /* remmed - cause ambiguity with the deprecated constructor
145     hash_table(size_type initial_size) {
146         /// @todo Implement
147     }
148
149     hash_table() {
150         /// @todo Implement
151     }
152     */
153
154     template <class InIt>
155     hash_table(InIt i, InIt j, size_type initial_size, const
            hasher_type &hasher,
156         const key_equal &key_equal) {
157         /// @todo Implement
158     }
159
160     template <class InIt>
161     hash_table(InIt i, InIt j, size_type initial_size, const
            hasher_type &hasher) {
```

```
162         /// @todo Implement
163     }
164
165     template <class InIt>
166     hash_table(InIt i, InIt j, size_type initial_size) {
167         /// @todo Implement
168     }
169
170     template <class InIt>
171     hash_table(InIt i, InIt j) {
172         /// @todo Implement
173     }
174
175     /*
176     hash_table(const this_type &b) {
177         /// @todo Implement
178     }
179     */
180
181     this_type &operator =(const this_type &b) {
182         /// @todo Implement
183     }
184
185     inline size_type count() const {
186         return element_count;
187     }
188
189     /**
190      * Return number of elements in the table.
191      *
192      * This function is deprecated: Use count() instead.
193      *
194      * @return Number of elements in the table.
195      */
196     inline size_type size() const {
197         return element_count;
198     }
199
200     /// @todo Implement multi* varieties: They return only an iterator
         .
201     std::pair<iterator, bool> insert(const value_type &t) {
202
203         TR("hash_table::insert()");
204
205         /// @todo Don't access t.first directly, use key_access.
206         image_type image = hasher.get_hash_value(t.first);
207         bucket_type *b = &(storage.get_bucket(image));
208
209         // Check if the element's already in the table.
210         /// @todo Don't access t.first directly, use key_access.
211         internal_iterator exists = storage.lookup(t.first, *b, image);
212
213         // Element with key t.first exists in the hash table.
214         /// @todo Implement multi* insertions. Remember to increase
              element
215         /// count, and not compartment count.
216         if (exists != storage.end())
217             return std::make_pair(iterator(exists), false);
218
219         element_count++;
220
221         // Expand if required
222         /// @note It is important to expand before inserting, as
              otherwise the
```

```
223         /// iterator obtained on insertion will no longer be a valid
                return
224         /// value.
225         compartment_count++;
226         if (compartment_count > max_compartment_count) {
227             storage.expand();
228 
229             // Recompute limits from new compartment count
230             min_compartment_count = (size_type) (_min_load_factor *
                    storage.capacity());
231             max_compartment_count = (size_type) (_max_load_factor *
                    storage.capacity());
232 
233             /// @hack Try to avoid this double bucket retreival; it is
                     currently
234             /// required as the bucket may have become invalid after
                    an expand.
235             b = &storage.get_bucket(image);
236         }
237 
238         // Insert new element
239         internal_iterator pos = storage.insert_in_bucket(*b, t, image)
                ;
240 
241         // bool is true if the element did not exist before and was
                actually
242         // inserted
243         return std::make_pair(iterator(pos), true);
244     }
245 
246     iterator insert(iterator r, const value_type &t) {
247         /// @todo Implement
248     }
249 
250     template <class InIt>
251     void insert(InIt i, InIt j) {
252         /// @todo Implement
253     }
254 
255     size_type erase(const key_type &k) {
256 
257         TR("hash_table::erase()");
258 
259         image_type image = hasher.get_hash_value(k);
260         bucket_type &b = storage.get_bucket(image);
261         internal_iterator c = storage.lookup(k, b, image);
262 
263         if (c == storage.end())
264             return 0;
265 
266         element_count--;
267 
268         /// @note The multi_set will be much more efficient if we are
                aware in
269         /// some way of the internal structure of the compartment. But
                 we
270         /// probably do already have that information:
271         /// multi_map.value_t = std::pair<key, list<data> >
272         /// multi_set.value_t = std::pair<key, size_type>
273         /// This makes the current implementation work, except that
                the return
274         /// value should be calculated from list.size and data,
                respectively.
275         storage.delete_from_bucket(b, c);
276 
```

```
277             // Contract if required
278             compartment_count--;
279             if (compartment_count < min_compartment_count) {
280                 storage.contract();
281 
282                 // Recompute limits from new compartment count
283                 min_compartment_count = (size_type) (_min_load_factor *
                        storage.capacity());
284                 max_compartment_count = (size_type) (_max_load_factor *
                        storage.capacity());
285             }
286 
287             return 1;
288         }
289 
290         void erase(iterator r) {
291             /// @todo Implement
292         }
293 
294         void erase(iterator r1, iterator r2) {
295             /// @todo Implement
296         }
297 
298         void clear() {
299             /// @todo Implement
300         }
301 
302         iterator find(const key_type &k) {
303 
304             TR("hash_table::find()");
305 
306             image_type image = hasher.get_hash_value(k);
307             bucket_type &b = storage.get_bucket(image);
308             return iterator(storage.lookup(k, b, image));
309         }
310 
311         const_iterator find(const key_type &k) const {
312             /// @todo Implement
313         }
314 
315         size_type count(const key_type &k) {
316             /// @todo Implement
317         }
318 
319         std::pair<iterator, iterator> equal_range(const key_type &k) {
320             /// @todo Implement, although only for multi* varieties
321         }
322 
323         std::pair<const_iterator, const_iterator> equal_range(const
                key_type &k) const {
324             /// @todo Implement, although only for multi* varieties
325         }
326 
327         inline iterator begin() {
328             return iterator(storage.begin());
329         }
330 
331         const_iterator begin() const {
332             /// @todo Implement
333         }
334 
335         inline iterator end() {
336             return iterator(storage.end());
337         }
338 
```

```
339         const_iterator end() const {
340             /// @todo Implement
341         }
342 
343         reverse_iterator rbegin() {
344             return reverse_iterator(end());
345         }
346 
347         const_reverse_iterator rbegin() const {
348             /// @todo Implement
349         }
350 
351         reverse_iterator rend() {
352             return reverse_iterator(begin());
353         }
354 
355         const_reverse_iterator rend() const {
356             /// @todo Implement
357         }
358 
359         float load_factor() const {
360             return (float)compartment_count / storage.capacity();
361         }
362 
363         inline float max_load_factor() const {
364             return _max_load_factor;
365         }
366 
367         void max_load_factor(float z) {
368             /// @todo Implement
369         }
370 
371         inline float min_load_factor() const {
372             return _min_load_factor;
373         }
374 
375         void min_load_factor(float z) {
376             /// @todo Implement
377         }
378 
379         void reserve(size_type n) {
380             /// @todo Implement
381         }
382 
383         mapped_type& operator[](const key_type &k) {
384             value_type val;
385             /// @todo Implement
386         }
387 
388         void swap(this_type &b) {
389             /// @todo Implement
390         }
391 
392         bool operator==(const this_type &b) const {
393             /// @todo Implement
394             return false;
395         }
396 
397         bool operator!=(const this_type &b) const {
398             /// @todo Implement
399             return false;
400         }
401 
402         bool operator<(const this_type &b) const {
403             /// @todo Implement
```

```
404         return false;
405     }
406
407     bool operator>(const this_type &b) const {
408         /// @todo Implement
409         return false;
410     }
411
412     bool operator<=(const this_type &b) const {
413         /// @todo Implement
414         return false;
415     }
416
417     bool operator>=(const this_type &b) const {
418         /// @todo Implement
419         return false;
420     }
421
422     bool empty() const {
423         return element_count == 0;
424     }
425
426     const Allocator &get_allocator() const {
427         return storage.get_allocator();
428     }
429
430     key_equal key_eq() {
431         /// @todo Implement
432     }
433
434     value_equal value_eq() {
435         /// @todo Implement
436     }
437
438     size_type max_size() const {
439         /// @todo Implement
440     }
441
442     /// @hack To be able to graph the contents
443     storage_policy &storage_pol() {
444         return storage;
445     }
446 };
447
448 #endif //_HASH_TABLE_H_
449 // Local Variables:
450 // mode: c++
451 // c-basic-offset:4
452 // tab-width:4
453 // End:
454 /* $Id: hash_table.h,v 1.1.1.1 2005/06/16 13:51:20 lyhne Exp $ */
```

AppendixA.2.3 Hasher- and hasher factory implementations (hasher_factory.h)

```
1  /**
2   * $Id: hasher_factory.h,v 1.1.1.1 2005/06/16 13:51:20 lyhne Exp $
3   *
4   * @hack We're assuming size_t is the same as an unsigned long type.
5   */
6  #ifndef HASHER_FACTORY_H
7  #define HASHER_FACTORY_H
8
9  #include "random.h"
```

```
10 #include <string>
11 #include <limits.h>
12
13 /**
14  * Universal hash function parameters
15  */
16 #if BIT64
17  static const size_t UNIVERSAL_PRIME = 18446744073709551557UL;
18 #else
19  static const size_t UNIVERSAL_PRIME = 4294967291UL;
20 #endif
21
22 /* FNV hash function parameters */
23 #if BIT64
24  static const size_t FNV_PRIME = 1099511628211UL;
25  static const size_t FNV1_OFFSET_BASIS = 14695981039346656037UL;
26 #else
27  static const size_t FNV_PRIME = 16777619U;
28  static const size_t FNV1_OFFSET_BASIS = 2166136261U;
29 #endif
30
31 template<class Key>
32 class universal_hasher_base {
33 public:
34     /// @todo Make it work with a general image type
35     typedef size_t image_type;
36     typedef Key key_type;
37
38 protected:
39     template<class> friend class universal_hasher_factory;
40
41     // 1 <= a < p, 0 <= b < p
42     image_type a, b;
43
44     // > m
45     image_type p;
46
47 public:
48     /// @note These prime numbers are just below UINT_MAX
49     universal_hasher_base(image_type p = UNIVERSAL_PRIME) : p(p) {}
50 };
51
52 template<class Key>
53 class universal_hasher : public universal_hasher_base<Key> {
54 public:
55     /// @todo Find out why none of the remmed-out varieties work.
56     /// <http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_closed.html>
          mentions
57     /// it but doesn't really give an answer.
58     //using typename universal_hasher_base<Key>::image_type;
59     //using typename universal_hasher_base<Key>::image_type image_type
          ;
60     typedef typename universal_hasher::image_type image_type;
61
62     /**
63      * A note on the surprising "this->" parts in the implementation
          below:
64      *
65      * This is due to "two-stage name lookup" (implemented since gcc
          3.4, not
66      * implemented in MSVC as of 7.1). In detail, it can be summed up
          like this
67      * (from <http: *gcc.gnu.org/onlinedocs/gcc/Name-lookup.html#Name-
          lookup>):
68      *
```

```
69        *     In [get_hash_value], [a, b, p] is not used in a dependent
70        *     context, so the compiler will look for a name declared at
              the
71        *     enclosing namespace scope (which is the global scope here).
               It
72        *     will not look into the base class, since that is dependent
              and
73        *     you may declare specializations of Base even after
              declaring
74        *     Derived, so the compiler can't really know what i would
              refer
75        *     to. If there is no global variable i, then you will get an
76        *     error message.
77        *
78        *     In order to make it clear that you want the member of the
              base
79        *     class, you need to defer lookup until instantiation time,
              at
80        *     which the base class is known. For this, you need to access
               i
81        *     in a dependent context, by either using this->i (remember
              that
82        *     this is of type Derived<T>*, so is obviously dependent), or
83        *     using Base<T>::i.  Alternatively, Base<T>::i might be
              brought
84        *     into scope by a using-declaration.
85        *
86        * So, the alternatives are:
87        *
88        *     using universal_hasher::a;
89        *     ...
90        *
91        *     inline image_type get_hash_value(const Key k) {
92        *       return (this->a * k + this->b) % this->p;
93        *     }
94        *
95        *     inline image_type get_hash_value(const Key k) {
96        *       return (universal_hasher::a * k + universal_hasher::b) %
97        *         universal_hasher::p;
98        *     }
99        *
100       * See also bottom of <http: *gcc.gnu.org/bugs.html#nonbugs_cxx>.
101       *
102       * @return The "raw" image. %m will be done later.
103       */
104      inline image_type get_hash_value(const Key k) const {
105          return (this->a * k + this->b) % this->p;
106      }
107  };
108
109  template<>
110  class universal_hasher<std::string> : public universal_hasher_base<std
         ::string> {
111  public:
112      inline image_type get_hash_value(const key_type &k) const {
113          std::string::const_iterator i = k.begin();
114
115          /// @todo Optimize by using full machine word width? (Four
                byte-sized
116          /// characters at a time on 32-bit CPUs, eigth on 64-bit CPUs
                etc.)
117          image_type image = *i;
118          for (; i != k.end(); ++i) {
119              char c = *i;
120              image ^= ((a * c + b) % p);
```

```
121         }
122         return image;
123     }
124 };
125
126 template<>
127 class universal_hasher<const char *> :
128     public universal_hasher_base<const char *> {
129 public:
130     inline image_type get_hash_value(key_type k) const {
131         const char *i = k;
132         image_type image = *i;
133
134         for (; *i; ++i) {
135             char c = *i;
136             image ^= ((a * c + b) % p);
137         }
138         return image;
139     }
140 };
141
142 template<class Key>
143 class universal_hasher_factory {
144 private:
145 public:
146     // template <class> class hasher_type;
147     typedef universal_hasher<Key> hasher_type;
148     typedef typename hasher_type::image_type image_type;
149     typedef typename hasher_type::key_type key_type;
150
151     static const bool only_one_member = false;
152
153     universal_hasher_factory() {
154     }
155
156     void update_hasher(hasher_type &func) {
157         func.a = 1 + random::get_next() % (func.p - 1);
158         func.b = random::get_next() % func.p;
159     }
160 };
161
162 /* string_table_hasher_base
163  *----------------------------*/
164 template<class Key, size_t STRING_TABLE_SIZE>
165 class string_table_hasher_base {
166 public:
167     /// @todo Make it work with a general image type
168     typedef size_t image_type;
169     typedef Key key_type;
170
171  protected:
172     template<class, size_t> friend class string_table_hasher_factory;
173
174     // static const size_t length = sizeof(key_type);
175     static const size_t length = STRING_TABLE_SIZE;
176     image_type table[STRING_TABLE_SIZE][256];
177 };
178
179 template<class Key, size_t STRING_TABLE_SIZE>
180 class string_table_hasher :
181     public string_table_hasher_base<Key, STRING_TABLE_SIZE> {
182 public:
183     typedef typename string_table_hasher::image_type image_type;
184
185     inline image_type get_hash_value(const Key &k) const {
```

```
186         /// @hack Outputs more information that the assertion, avoids
                having to enter
187         /// the debugger.
188         if (string_table_hasher::length > STRING_TABLE_SIZE) {
189             cout << "string_table_hasher::length " <<
                    string_table_hasher::length << endl
190                  << " STRING_TABLE_SIZE " << STRING_TABLE_SIZE << endl
                        ;
191         }
192         assert(string_table_hasher::length <= STRING_TABLE_SIZE);
193 
194         image_type image = k;
195         size_t shift = k;
196         size_t bits  = k & 0xff;
197         size_t i=0;
198         while (i < string_table_hasher::length) {
199             image ^= string_table_hasher::table[i][bits];
200             bits = (shift << (8*i)) & 0xff;
201             ++i;
202         }
203         return image;
204     }
205 };
206 
207 template<size_t STRING_TABLE_SIZE>
208 class string_table_hasher<std::string, STRING_TABLE_SIZE> :
209     public string_table_hasher_base<std::string, STRING_TABLE_SIZE> {
210 public:
211     typedef typename string_table_hasher::image_type image_type;
212 
213     inline image_type get_hash_value(std::string k) const {
214         std::string::const_iterator i = k.begin();
215         image_type image = *i;
216         size_t j = 0;
217         while (i != k.end()) {
218             char c = (char)*i;
219             if (j >= STRING_TABLE_SIZE)
220                 j = 0;
221             /// @todo See if the index here becomes negative if c is
                    an
222             /// international character.
223             image ^= string_table_hasher::table[j][(int)c];
224             ++i, ++j;
225         }
226         return image;
227     }
228 };
229 
230 template<class T, size_t STRING_TABLE_SIZE>
231 class string_table_hasher<const T *, STRING_TABLE_SIZE> :
232     public string_table_hasher_base<const T *, STRING_TABLE_SIZE> {
233 public:
234     typedef typename string_table_hasher::image_type image_type;
235 
236     inline image_type get_hash_value(const T *k) const {
237         const T *i = k;
238         image_type image = *i;
239         size_t j = 0;
240         while (*i) {
241             /// @todo Use the whole element
242             char c = (char)*i;
243             if (j >= STRING_TABLE_SIZE)
244                 j = 0;
245             image ^= string_table_hasher::table[j][(int)c];
246             ++i, ++j;
```

```
247         }
248         return image;
249     }
250 };
251 
252 template<class Key, size_t STRING_TABLE_SIZE = 4>
253 class string_table_hasher_factory {
254 public:
255     typedef string_table_hasher<Key, STRING_TABLE_SIZE> hasher_type;
256     typedef typename hasher_type::image_type image_type;
257     typedef typename hasher_type::key_type key_type;
258 
259     static const bool only_one_member = false;
260 
261     string_table_hasher_factory() {
262     }
263 
264     void update_hasher(hasher_type &func) {
265         size_t i,j;
266         for (i=0; i < STRING_TABLE_SIZE; ++i)
267             for (j=0; j < 256; ++j)
268                 func.table[i][j] = (image_type) random::get_next();
269     }
270 };
271 
272 /**
273  * fnv1a hash factory. This has only one member.
274  * */
275 template<class Key>
276 class fnv1a_hasher_base {
277 public:
278     /// @todo Make it work with a general image type
279     typedef size_t image_type;
280     typedef Key key_type;
281     fnv1a_hasher_base() {}
282 
283 };
284 
285 template<class Key>
286 class fnv1a_hasher : public fnv1a_hasher_base<Key> {
287 public:
288     typedef typename fnv1a_hasher::image_type image_type;
289     typedef typename fnv1a_hasher::key_type key_type;
290     inline image_type get_hash_value(key_type k) const {
291         image_type image = (image_type) FNV1_OFFSET_BASIS;
292         for (size_t i=0; i < sizeof(key_type); ++i) {
293             image = image ^ ((k>>i) & 0xff);
294             image = image * FNV_PRIME;
295         }
296         // cout << "image " << image <<  endl;
297         return image;
298     }
299 };
300 
301 template<>
302 class fnv1a_hasher<std::string> : public fnv1a_hasher_base<std::string
    > {
303 public:
304     // typedef typename fnv1a_hasher::image_type image_type;
305     inline image_type get_hash_value(std::string k) const {
306         image_type image = (image_type) FNV1_OFFSET_BASIS;
307         std::string::const_iterator i = k.begin();
308         for (; *i; ++i) {
309             image = image ^ *i;
310             image = image * FNV_PRIME;
```

```
311         }
312         return image;
313     }
314 };
315 
316 template<class Key>
317 class fnv1a_hasher<const Key *> : public fnv1a_hasher_base<const Key
        *> {
318 public:
319     typedef typename fnv1a_hasher::image_type image_type;
320     // typedef typename fnv1a_hasher::key_type key_type;
321     inline image_type get_hash_value(const Key *k) const {
322         image_type image = (image_type) FNV1_OFFSET_BASIS;
323         for (; *k; ++k) {
324             image = image ^ *k;
325             image = image * FNV_PRIME;
326         }
327         return image;
328     }
329 };
330 
331 template<class Key>
332 class fnv1a_hasher_factory {
333 public:
334     typedef fnv1a_hasher<Key> hasher_type;
335     typedef typename hasher_type::image_type image_type;
336     typedef typename hasher_type::key_type key_type;
337 
338     static const bool only_one_member = true;
339 
340     fnv1a_hasher_factory() {
341     }
342     ///@todo this generates a warning
343     void update_hasher(hasher_type &func) {
344     }
345 };
346 #endif /* ifndef HASHER_FACTORY_H */
347 // Local Variables:
348 // mode: c++
349 // c-basic-offset:4
350 // tab-width:4
351 // End:
352 /* $Id: hasher_factory.h,v 1.1.1.1 2005/06/16 13:51:20 lyhne Exp $ */
```

AppendixA.3 Storage policies

AppendixA.3.1 Linear hashing storage policy base class (linear_hashing_storage_policy.h)

```
 1 /*
 2  * $Id: linear_hashing_storage_policy.h,v 1.1.1.1 2005/06/16 13:51:20
       lyhne Exp $
 3  *
 4  * TODO DAN EN-TABEL-UDGAVE MED DOUBLING TREE!!!
 5  *
 6  *
 7  *
 8  *
 9  * This is the storage part of the hash table which has the
       tailpointer
10  * inside the bucket chain.
11  *
12  */
13 #ifndef LINEAR_HASHING_STORAGE_POLICY_H
```

```
14 #define LINEAR_HASHING_STORAGE_POLICY_H
15
16 #include <memory>
17 #include <vector>
18 #include <cassert>
19
20 #include "linear_hashing_compartment.h"
21 #include "linear_hashing_compartment_iterator.h"
22 #include "hasher_factory.h"
23 #include "key_adapter.h"
24 #include "mr_trace.h"
25
26 template< class Value,
27           class HF,
28           class Equal,
29           class Allocator,
30           class Compartment >
31 class linear_hashing_storage_policy {
32 protected:
33     typedef Compartment compartment_type;
34
35 public:
36     typedef typename HF::hasher_type::image_type image_type;
37     typedef size_t size_type;
38     typedef Value value_type;
39
40     /// @hack size_type is not the proper difference type
41     typedef linear_hashing_compartment_iterator<value_type,
          compartment_type,
42         size_type> iterator;
43
44     struct bucket_type {
45         compartment_type *head;
46         compartment_type *tail;
47     };
48
49 protected:
50     /// @note "template" keyword is required to tell the compiler that
           rebind
51     /// is, in fact, a template. GCC pre 3.4 will accept the
          declaration without
52     /// it, but will not be in compliance with the C++ standard.
53     typedef typename Allocator::template rebind<bucket_type>::other
54         bucket_allocator_type;
55     typedef typename Allocator::template rebind<compartment_type>::
          other
56         compartment_allocator_type;
57     typedef typename key_adapter<Value>::key_type key_type;
58
59     size_type p;
60     size_type maxp; // m
61     compartment_type sentinel;
62
63     bucket_type empty_bucket;
64     Equal equals;
65
66     compartment_allocator_type compartment_allocator;
67     key_adapter<Value> key_access;
68
69     inline iterator insert_in_bucket(bucket_type &b, compartment_type
          *c) {
70         compartment_type *next, *prev;
71
72         if (b.head == &sentinel) {
```

```
73              // Bucket empty: Inserts the first element of the bucket
                    into the
74              // end of the table chain.
75              next = &sentinel;
76              prev = sentinel.prev;
77
78              b.head = c;
79              b.tail = c;
80
81          } else if (b.head == b.tail) {
82              // Bucket contains only one element: Inserts a subsequent
                    element
83              // at the end of the bucket chain.
84
85              next = b.tail->next;
86              prev = b.tail;
87
88              b.tail = c;
89          } else {
90              // Bucket contains multiple elements: Inserts a subsequent
                    element
91              // at the end of the bucket chain.
92
93              next = b.tail;
94              prev = b.tail->prev;
95          }
96
97          c->next = next;
98          c->prev = prev;
99
100         prev->next = c;
101         next->prev = c;
102
103         return iterator(c);
104     }
105
106 public:
107     linear_hashing_storage_policy(size_type initial_size) :
108         p(0),
109         maxp(initial_size),
110         sentinel(value_type(), size_type()) {
111
112         assert(sizeof(image_type) >= sizeof(size_type));
113
114         // Prototype bucket
115         empty_bucket.head = empty_bucket.tail = &sentinel;
116
117         // Table chain circular
118         sentinel.prev = sentinel.next = &sentinel;
119     }
120
121     inline iterator lookup(const key_type &k, bucket_type &b,
        image_type image) { // {{{
122
123         TR("lhsp::lookup()");
124
125         compartment_type *c = b.head;
126
127         if (c != &sentinel) {
128             while (true) {
129                 if ((c->image == image) &&
130                     equals(key_access.get_key(c->value), k)) {
131                     return iterator(c);
132                 }
133
```

```
134                 if (c == b.tail)
135                     break;
136
137                 c = c->next;
138             }
139         }
140         return end();
141     }
142
143     inline iterator insert_in_bucket(bucket_type &b, const value_type
            &value,
144         image_type image) {
145         TR("lhsp::insert_in_bucket()");
146
147         compartment_type *c = compartment_allocator.allocate(1);
148         compartment_allocator.construct(c, compartment_type(value,
                image));
149
150         return insert_in_bucket(b, c);
151     }
152
153     inline void delete_from_bucket(bucket_type &b, const iterator &c,
154         bool free = true) {
155
156         TR("lhsp::delete_from_bucket()");
157
158         compartment_type *prev = c.c->prev;
159         compartment_type *next = c.c->next;
160
161         // Splice out the compartment
162
163         // Safe: next and prev are either the sentinel or normal
                compartments
164         prev->next = next;
165         next->prev = prev;
166
167         // Update bucket pointers if needed
168
169         if (b.head == b.tail)
170             b.head = &sentinel;
171         else if (b.head == c.c)
172             b.head = next;
173         else if (b.tail == c.c)
174             b.tail = prev;
175
176         // Clean up
177         /// @hack This is required because of the
178         /// onetable_linear_hashing_storage_policy, as it uses
                delete_from_bucket
179         /// to move compartments between buckets.
180         if (free)
181             compartment_allocator.deallocate(c.c, 1);
182     }
183
184     inline iterator begin() {
185         TR("lhsp::begin()");
186
187         // If the table's empty, this will be the sentinel. Otherwise
                it will be
188         // the first element.
189         return iterator(sentinel.next);
190     }
191
192     inline iterator end() {
193         TR("lhsp::begin()");
```

```
194         return iterator(&sentinel);
195     }
196
197     /**
198      * Current capacity, for computing load factors.
199      */
200     inline size_type capacity() const {
201         return maxp + p;
202     }
203 };
204
205 #endif /* ifndef LINEAR_HASHING_STORAGE_POLICY_H */
206 /* $Id: linear_hashing_storage_policy.h,v 1.1.1.1 2005/06/16 13:51:20
    lyhne Exp $ */
```

AppendixA.3.2 One-table linear hashing storage policy (onetable_linear_hashing_storage_policy.h)

```
1  /**
2   * $Id: onetable_linear_hashing_storage_policy.h,v 1.1.1.1 2005/06/16
       13:51:20 lyhne Exp $
3   *
4   * This is the storage part of the hash table which has the
       tailpointer
5   * inside the bucket chain.
6   *
7   */
8  #include "Vector/Implementation/math_utils.h"
9  #include "Vector/Implementation/hat_iterator.h"
10 #include "Vector/Implementation/hashed_array_tree.h"
11 #include "cphstl/vector.h"
12 #include "linear_hashing_storage_policy.h"
13 #include "mr_trace.h"
14
15 #ifndef ONETABLE_LINEAR_HASHING_STORAGE_POLICY_H
16 #define ONETABLE_LINEAR_HASHING_STORAGE_POLICY_H
17
18 template< class Value,
19           class HF,
20           class Equal,
21           class Allocator = std::allocator<Value>,
22           class Compartment = linear_hashing_compartment<Value> >
23 class onetable_linear_hashing_storage_policy :
24     public linear_hashing_storage_policy<Value, HF, Equal, Allocator,
          Compartment> {
25
26 public:
27     // Bring up public base class types
28     typedef typename onetable_linear_hashing_storage_policy::
          bucket_type
29         bucket_type;
30     typedef typename onetable_linear_hashing_storage_policy::size_type
31         size_type;
32     typedef typename onetable_linear_hashing_storage_policy::
          image_type
33         image_type;
34
35 private:
36     // Bring up protected base class types
37     typedef typename onetable_linear_hashing_storage_policy::
          bucket_allocator_type
38         bucket_allocator_type;
39     typedef typename onetable_linear_hashing_storage_policy::
          compartment_type
```

```
40          compartment_type;
41
42      typedef cphstl::vector<bucket_type, bucket_allocator_type,
43          cphstl::hashed_array_tree<bucket_allocator_type> >
                bucket_vector_type;
44
45          using onetable_linear_hashing_storage_policy::p;
46          using onetable_linear_hashing_storage_policy::sentinel;
47          using onetable_linear_hashing_storage_policy::maxp;
48          using onetable_linear_hashing_storage_policy::empty_bucket;
49
50      bucket_vector_type buckets;
51
52  public:
53      onetable_linear_hashing_storage_policy(size_type initial_size) :
54          linear_hashing_storage_policy<Value, HF, Equal, Allocator,
                Compartment>(initial_size) {
55
56          /// @todo Implement resize in cphstl::hashed_array_tree
57          //buckets.resize(maxp * 2, empty_bucket);
58          buckets.reserve(maxp);
59          for (typename bucket_vector_type::size_type i = 0; i < maxp; i
                ++)
60              buckets.push_back(empty_bucket);
61      }
62
63
64      void expand() {
65          TR("otlhsp::expand()");
66
67          // Append extra bucket
68          buckets.push_back(empty_bucket);
69
70          /// @hack This should be possible to do before the push_back,
                but the
71          /// HAT moves things, which it wasn't supposed to do.
72          bucket_type &b = buckets[p];
73          compartment_type *c = b.head;
74
75          // Move entries from existing bucket (if not empty)
76          if (c != &sentinel) {
77
78              // Store tail as it may change as we remove compartments
79              compartment_type *tail = b.tail;
80              /// @todo Optimize by keeping iterator pointing to b and
                    other_b
81              // remmed - its back is broken
82              /// @todo Fix HAT.back(), it 1) does not compile, 2)
                    refers to the
83              /// element *one position beyond* the end of the container
                    .
84              //bucket_type &other_b = buckets.back();
85              bucket_type &other_b = buckets[maxp + p];
86
87              while (true) {
88                  compartment_type *next = c->next;
89
90                  // Assumes only two destination buckets.
91                  if (c->image & maxp) {
92                      delete_from_bucket(b, c, false);
93                      insert_in_bucket(other_b, c);
94                  }
95
96                  if (c == tail)
97                      break;
```

```
98
99                  c = next;
100             }
101         }
102         p++;
103
104         if (p == maxp) {
105             maxp *= 2;
106             p = 0;
107         }
108     }
109
110     void contract() {
111         TR("otlhsp::contract()");
112
113         if (p == 0) {
114             /// @hack Prevents maxp from going off the scale
115             if (maxp == 1)
116                 return;
117
118             maxp /= 2;
119             p = maxp - 1;
120         } else {
121             --p;
122         }
123
124         bucket_type &other_b = buckets[maxp + p];
125
126         // Move entries from existing bucket (if not empty)
127         if (other_b.head != &sentinel) {
128             bucket_type &b = buckets[p];
129
130             if (b.head == &sentinel) {
131                 // Destination bucket empty, do a brutal replace.
132                 // (This happens if items have been inserted into
                        other_b
133                 // while b was below p)
134                 b = other_b;
135             } else {
136                 compartment_type *prev, *next;
137
138                 // Splice other_b's chain out of the table chain
139                 other_b.head->prev->next = other_b.tail->next;
140                 other_b.tail->next->prev = other_b.head->prev;
141
142                 // Splice other_b's chain into b
143                 if (b.head == b.tail) {
144                     /// @todo Optimize by splicing b out instead.
145                     prev = b.tail;
146                     next = b.tail->next;
147
148                     // Expand b; it contained only one element and
                            thus had no
149                     // inside.
150                     b.tail = other_b.tail;
151                 } else {
152                     prev = b.tail->prev;
153                     next = b.tail;
154                 }
155
156                 prev->next = other_b.head;
157                 other_b.head->prev = prev;
158
159                 next->prev = other_b.tail;
160                 other_b.tail->next = next;
```

```
161             }
162         }
163
164         // Ditch extra bucket
165         buckets.pop_back();
166     }
167
168     inline bucket_type &get_bucket(image_type image) {
169
170         TR("otlhsp::get_bucket()");
171
172         size_type index = image & (maxp - 1);
173         if (index < p)
174             // Equivalent to index = image & (maxp * 2 - 1)
175             index |= image & maxp;
176
177         return buckets[index];
178     }
179 };
180
181 #endif /* ifndef ONETABLE_LINEAR_HASHING_STORAGE_POLICY_H */
182 /* $Id: onetable_linear_hashing_storage_policy.h,v 1.1.1.1 2005/06/16
    13:51:20 lyhne Exp $ */
```

AppendixA.3.3 Two-table linear hashing storage policy (twotable_linear_hashing_storage_policy.h)

```
1  /*
2   * $Id: twotable_linear_hashing_storage_policy.h,v 1.1.1.1 2005/06/16
      13:51:20 lyhne Exp $
3   *
4   * This is the storage part of the hash table which has the
      tailpointer
5   * inside the bucket chain.
6   *
7   */
8  #include "linear_hashing_storage_policy.h"
9  #include "mr_trace.h"
10
11 #ifndef TWOTABLE_LINEAR_HASHING_STORAGE_POLICY_H
12 #define TWOTABLE_LINEAR_HASHING_STORAGE_POLICY_H
13
14 template< class Value,
15           class HF,
16           class Equal,
17           class Allocator = std::allocator<Value>,
18           class Compartment = linear_hashing_compartment<Value> >
19 class twotable_linear_hashing_storage_policy :
20     public linear_hashing_storage_policy<Value, HF, Equal, Allocator,
          Compartment> {
21
22 public:
23     // Bring up public base class types
24     typedef typename twotable_linear_hashing_storage_policy::
          bucket_type
25         bucket_type;
26     typedef typename twotable_linear_hashing_storage_policy::size_type
27         size_type;
28     typedef typename twotable_linear_hashing_storage_policy::
          image_type
29         image_type;
30
31 private:
32     enum { FIRST, SECOND };
```

```
33
34      // Bring up protected base class types
35      typedef typename twotable_linear_hashing_storage_policy::
            bucket_allocator_type
36          bucket_allocator_type;
37      typedef typename twotable_linear_hashing_storage_policy::
            compartment_type
38          compartment_type;
39
40      typedef std::vector<bucket_type, bucket_allocator_type>
            bucket_vector_type;
41          using twotable_linear_hashing_storage_policy::p;
42          using twotable_linear_hashing_storage_policy::sentinel;
43          using twotable_linear_hashing_storage_policy::maxp;
44          using twotable_linear_hashing_storage_policy::empty_bucket;
45
46      bucket_vector_type buckets[2];
47
48  public:
49      twotable_linear_hashing_storage_policy(size_type initial_size) :
50          linear_hashing_storage_policy<Value, HF, Equal, Allocator,
                Compartment>(initial_size) {
51
52          // initialize the 2 tables.
53          buckets[FIRST].resize(maxp, empty_bucket);
54          buckets[SECOND].resize(maxp * 2, empty_bucket);
55      }
56
57      void expand() {
58          TR("ttlhsp::expand()");
59
60          bucket_type &b = buckets[FIRST][p];
61          compartment_type *c = b.head;
62          size_type mask = maxp * 2 - 1;
63
64          if (c != &sentinel) {
65              // Splice previous bucket chain out of the table chain
66
67              compartment_type *bucket_predecessor = b.head->prev;
68              compartment_type *bucket_successor = b.tail->next;
69
70              bucket_predecessor->next = bucket_successor;
71              bucket_successor->prev = bucket_predecessor;
72
73              /// @todo Optimize if we do not switch hash functions, as
                    we'll
74              /// only be accessing two buckets in the SECOND table (
                    although
75              /// the cache probably does an excellent job of that
                    already).
76              while (true) {
77                  size_type index = c->image & mask;
78                  compartment_type *next = c->next;
79
80                  insert_in_bucket(buckets[SECOND][index], c);
81
82                  if (c == b.tail)
83                      break;
84
85                  c = next;
86              }
87
88              // Mark bucket as empty
89              b.head = &sentinel;
90          }
```

```
91          p++;
92          if (p == maxp) {
93  #if TRACE
94              std::cout << "Resize!" << endl;
95  #endif // TRACE
96
97              maxp *= 2;
98
99              buckets[FIRST].swap(buckets[SECOND]);
100             buckets[SECOND].resize(maxp * 2, empty_bucket);
101
102             p = 0;
103         }
104     }
105
106     void contract() {
107         TR("ttlhsp::contract()");
108
109         if (p == 0) {
110             /// @hack Prevents maxp from going off the scale
111             if (maxp == 1)
112                 return;
113
114             maxp /= 2;
115             p = maxp - 1;
116
117             buckets[FIRST].swap(buckets[SECOND]);
118             buckets[FIRST].resize(maxp, empty_bucket);
119         } else {
120             --p;
121         }
122
123         /// @note Assumes that the buckets are separated
124         /// by the most significant bit only.
125         bucket_type &dst = buckets[FIRST][p];
126         bucket_type &src1 = buckets[SECOND][p];
127         bucket_type &src2 = buckets[SECOND][maxp + p];
128
129         // Move entries from existing buckets (if not empty)
130         // If only one bucket contains items, use that as the new
              bucket.
131         // If both contain items, splice the second one into the first
               one.
132         if (src1.head == &sentinel) {
133             // Only src2 has contents
134             // If src2.head == &sentinel, this acts as a no-op.
135             dst = src2;
136         } else {
137             // src1 has contents
138             dst = src1;
139
140             if (src2.head != &sentinel) {
141                 // Both src1 and src2 have contents: Splice them
                      together
142
143                 // Splice src2 out of the table chain
144                 src2.head->prev->next = src2.tail->next;
145                 src2.tail->next->prev = src2.head->prev;
146
147                 // Splice src2's chain into src1
148                 compartment_type *prev, *next;
149                 if (dst.head == dst.tail) {
150                     /// @todo Optimize by splicing src2 out instead.
151                     prev = dst.tail;
152                     next = dst.tail->next;
```

```
153
154                     // Expand src1; it contained only one element and
                            thus had no
155                     // inside.
156                     dst.tail = src2.tail;
157                 } else {
158                     prev = dst.tail->prev;
159                     next = dst.tail;
160                 }
161
162                 prev->next = src2.head;
163                 src2.head->prev = prev;
164
165                 next->prev = src2.tail;
166                 src2.tail->next = next;
167             }
168         }
169     }
170
171     inline bucket_type &get_bucket(image_type image) {
172         TR("ttlhsp::get_bucket()");
173
174         size_type table, index;
175
176         index = image & (maxp - 1);
177         if (index < p) {
178             // Equivalent to index = image & (maxp * 2 - 1)
179             index |= image & maxp;
180             table = SECOND;
181         } else {
182             table = FIRST;
183         }
184
185         return buckets[table][index];
186     }
187 };
188
189 #endif /* ifndef TWOTABLE_LINEAR_HASHING_STORAGE_POLICY_H */
190 /* $Id: twotable_linear_hashing_storage_policy.h,v 1.1.1.1 2005/06/16
       13:51:20 lyhne Exp $ */
```

AppendixA.3.4 Linear hashing compartment (linear_hashing_compartment.h)

```
1  /*
2   * $Id: linear_hashing_compartment.h,v 1.1.1.1 2005/06/16 13:51:20
        lyhne Exp $
3   *
4   * @todo Use image type from the hasher.
5   */
6
7  #ifndef COMPARTMENT_H
8  #define COMPARTMENT_H
9
10 template<class V>
11 class linear_hashing_compartment {
12 private:
13     typedef linear_hashing_compartment<V> this_type;
14
15 public:
16     typedef V value_type;
17
18     value_type value;
19     size_t image;
```

```
20      this_type *next;
21      this_type *prev;
22
23      /// @todo See if this construct, or possibly a triple is a
24      /// better construct for including image.
25      /// typedef std::pair<std::pair<const key, size_t>, data)
26
27      inline linear_hashing_compartment() {}
28      inline linear_hashing_compartment(value_type value, size_t image)
            :
29          value(value), image(image) {}
30  };
31
32  #endif /* ifndef COMPARTMENT_H */
33  // Local Variables:
34  // mode: c++
35  // c-basic-offset:4
36  // tab-width:4
37  // End:
38  /* $Id: linear_hashing_compartment.h,v 1.1.1.1 2005/06/16 13:51:20
        lyhne Exp $ */
```

AppendixA.3.5 Linear hashing iterator (linear_hashing_compartment_iterator.h)

```
1   /*
2    * $Id: linear_hashing_compartment_iterator.h,v 1.1.1.1 2005/06/16
         13:51:20 lyhne Exp $
3    *
4    *
5    * This is the storage part of the hash table which has the
         tailpointer
6    * inside the bucket chain.
7    *
8    */
9   #include <iterator>
10
11  #ifndef LINEAR_HASHING_COMPARTMENT_ITERATOR_H
12  #define LINEAR_HASHING_COMPARTMENT_ITERATOR_H
13
14  template <class Value, class Compartment, class Diff>
15  class linear_hashing_compartment_iterator :
16      public std::iterator<std::bidirectional_iterator_tag, Value, Diff>
            {
17  private:
18      template <class, class, class, class, class>
19          friend class linear_hashing_storage_policy;
20
21      typedef Compartment compartment_type;
22
23      compartment_type *c;
24
25  public:
26      // Bring up from base class
27      typedef typename linear_hashing_compartment_iterator::value_type
            value_type;
28
29      linear_hashing_compartment_iterator(compartment_type *c) : c(c) {}
30
31      linear_hashing_compartment_iterator() {}
32
33      inline value_type &operator *() {
34          return c->value;
35      }
```

```
36
37      inline value_type *operator ->() {
38          return &c->value;
39      }
40
41      inline const value_type *operator ->() const {
42          return &c->value;
43      }
44
45      inline bool operator == (const linear_hashing_compartment_iterator
            &b) {
46          return c == b.c;
47      }
48
49      inline bool operator != (const linear_hashing_compartment_iterator
            &b) {
50          return !operator == (b);
51      }
52
53      inline linear_hashing_compartment_iterator &operator ++ () {
54          c = c->next;
55          return *this;
56      }
57
58      inline linear_hashing_compartment_iterator &operator -- () {
59          c = c->prev;
60          return *this;
61      }
62
63      // const is to prevent misunderstandings when doing i++++
64      inline const linear_hashing_compartment_iterator operator ++ (int)
            {
65          linear_hashing_compartment_iterator old(*this);
66          c = c->next;
67          return old;
68      }
69
70      // const is to prevent misunderstandings when doing i++++
71      inline const linear_hashing_compartment_iterator operator -- (int)
            {
72          linear_hashing_compartment_iterator old(*this);
73          c = c->prev;
74          return old;
75      }
76  };
77
78  #endif /* ifndef LINEAR_HASHING_COMPARTMENT_ITERATOR_H */
79  /* $Id: linear_hashing_compartment_iterator.h,v 1.1.1.1 2005/06/16
        13:51:20 lyhne Exp $ */
```

AppendixA.3.6 Chained hash storage policy (chained_hash_storage_policy.h)

```
1   /*
2    * $Id: chained_hash_storage_policy.h,v 1.1.1.1 2005/06/16 13:51:20
         lyhne Exp $
3    *
4    * This is the storage part of the hash table based on a design by
5    * Jyrki Katajainen and Michael Lykke (Katajainen, J., Lykke, M.
         (1996):
6    * "Experiments with universal hashing", pp. 5-7).
7    *
8    * @hack As we don't have a special-purpose value to designate empty
```

```
 9   * compartments, we define a bucket to be empty if its first
       compartments next-
10   * pointer is NULL, instead of pointing to the sentinel. This will
       have to be
11   * checked as a special case.
12   */
13  #ifndef CHAINED_HASH_STORAGE_POLICY_H
14  #define CHAINED_HASH_STORAGE_POLICY_H
15
16  #include <iterator>
17  #include <algorithm>
18  #include <memory>
19  #include <vector>
20  #include <cassert>
21
22  #include "chained_hash_compartment.h"
23  #include "chained_hashing_compartment_iterator.h"
24  #include "hasher_factory.h"
25  #include "key_adapter.h"
26  #include "mr_trace.h"
27
28  // initial size allocated.
29  static const size_t INITIAL_SIZE = 4;
30
31  template< class Value,
32            class HF,
33            class Equal,
34            class Allocator = std::allocator<Value>,
35            class Compartment = chained_hash_compartment<Value> >
36  class chained_hash_storage_policy {
37  public:
38      typedef typename HF::hasher_type::image_type image_type;
39      typedef Value value_type;
40      typedef Compartment bucket_type;
41      typedef Compartment compartment_type;
42      typedef size_t size_type;
43
44      typedef chained_hash_storage_policy<Value, HF, Equal, Allocator,
            Compartment>
45          self_type;
46
47      // @TODO size_t is not the difference type
48      typedef chained_hashing_compartment_iterator<value_type,
            compartment_type, self_type, size_t> iterator;
49
50      typedef typename Allocator::template rebind<compartment_type>::
            other
51          compartment_allocator_type;
52      typedef std::vector<bucket_type, compartment_allocator_type>
            bucket_vector;
53      typedef typename bucket_vector::iterator bucket_iterator;
54
55  private:
56      template <class, class, class, class>
57          friend class chained_hashing_compartment_iterator;
58
59      typedef typename key_adapter<Value>::key_type key_type;
60
61  public:
62
63  private:
64  #if defined(GRAPH_ACCESS)
65      template <class> friend class chained_hash_grapher;
66  #endif // GRAPH_ACCESS
67
```

```
68
69      compartment_type sentinel;
70      bucket_vector buckets;
71      compartment_allocator_type compartment_allocator;
72      Equal equals;
73      key_adapter<Value> key_access;
74
75      static inline bool is_empty(bucket_type &b) {
76          return b.next == NULL;
77      }
78
79  public:
80
81      chained_hash_storage_policy(size_type initial_size = INITIAL_SIZE)
            :
82          sentinel(value_type(), size_type())
83      {
84          // Sentinel also functions as template for empty buckets
85          sentinel.next = NULL;
86          buckets.resize(initial_size, sentinel);
87      }
88
89      /**
90       * Expand by rehashing to a table that's twice as large
91       *
92       * Current code assumes that the hash function does not change, i.
             e. that
93       * the elements either keep their place or move old_size buckets
             up.
94       */
95      void expand() {
96          TR("chsp::expand()");
97
98          size_type old_size = buckets.size();
99          size_type new_size = old_size * 2;
100         buckets.resize(new_size, sentinel);
101
102         // Traverse existing elements and relocate those that do not
                fit. After
103         // that, contract the existing bucket chain.
104
105         // Important optimization: Attempt not to perform copying
                unless the
106         // chain ends up containing only one compartment.
107         for (size_type i = 0; i < old_size; i++) {
108             bucket_type &src = buckets[i];
109
110             // Check for empty bucket
111             if (!is_empty(src)) {
112                 // Splice the rest of the compartments out
113                 compartment_type *prev = &src;
114                 compartment_type *curr = src.next;
115
116                 while (curr != &sentinel)
117                 {
118                     compartment_type *next = curr->next;
119
120                     /// @todo there is one more bit in the image if we
                            are to
121                     /// move this compartment, so a test to see if
                            this bit is
122                     /// set should be sufficient.
123                     bucket_type &dst = get_bucket(curr->image);
124                     if (&dst != &src) {
125                         insert_in_bucket(dst, curr);
```

```
126                         prev->next = next;
127
128                         // Don't modify prev if we spliced the
                                compartment out
129                     } else {
130                         prev = curr;
131                     }
132
133                     curr = next;
134                 }
135
136                 // Handle first compartment
137
138                 // We only have to make modification if the first
                        compartment
139                 // moves. In that case, we'll perform a swap operation
                        .
140                 bucket_type &dst = get_bucket(src.image);
141                 if (&dst != &src) {
142                     // Allocate standalone compartment for the part-of
                            -table
143                     // compartment.
144                     /// @todo Optimize by avoiding double copy when we
                             first swap
145                     /// and then discover that the first compartment
                            was actually
146                     /// going to be the first compartment in the
                            destination, too.
147                     compartment_type *tgt = src.next;
148
149                     // The pointers will be overwritten on insert, so
                            don't
150                     // copy those.
151                     if (tgt != &sentinel)
152                     {
153                         // Reuse the compartment with which we're
                                going to
154                         // overwrite the one we're moving.
155                         std::swap(src.value, tgt->value);
156                         std::swap(src.image, tgt->image);
157
158                         src.next = tgt->next;
159
160                         // Insert into new bucket
161                         insert_in_bucket(dst, tgt);
162                     }
163                     else
164                     {
165                         // Insert into new bucket
166                         insert_in_bucket(dst, src.value, src.image);
167
168                         // Remove from old by overwriting
169                         // Copying the sentinel is OK.
170                         src = *src.next;
171                     }
172                 }
173             }
174         }
175     }
176
177     int contract() {
178         TR("chsp::contract()");
179
180         size_type old_size              = buckets.size();
181
```

```
182         if (old_size == 1)
183             return 0;
184 
185         size_type new_size             = old_size / 2;
186         size_type buckets_size_minus_1 = new_size - 1;
187 
188         // only change the top part of the hash table:
189         // between new_size and old_size -1
190         for (size_type i = new_size; i < old_size; i++) {
191 
192             bucket_type &src = buckets[i];
193 
194             if (is_empty(src))
195                 continue;
196 
197             /// @note We assume only the MSB separates this and the
                    new function.
198             /// This means that we have a guarantee that all the
                    compartments
199             /// that hit src will hit dst with the new function.
200             bucket_type &dst = get_bucket( src.image,
                    buckets_size_minus_1 );
201 
202             // Check for empty bucket
203             if (!is_empty(dst)) {
204 
205                 assert( &src != &dst );
206 
207                 // Find end of destination bucket
208                 compartment_type *curr = &dst;
209                 while (curr->next != &sentinel)
210                     curr = curr->next;
211 
212                 // Room for the compartment located in the vector
213                 // this allocation might be the one creating a strange
                        value
214                 curr->next = compartment_allocator.allocate(1);
215                 compartment_allocator.construct(curr->next,
                        compartment_type(src));
216 
217                 // curr = curr->next;
218 
219                 // // seems to be here that the strange value is
                        introduced
220                 // *curr = src;
221 
222             } else {
223 
224                 dst = src;
225             }
226         }
227 
228         buckets.resize(new_size);
229 
230         return 1;
231     }
232 
233     // for use with contract
234     inline bucket_type &get_bucket(image_type image, size_type
            buckets_size_minus_1) {
235         TR("chsp::get_bucket(image, buckets_size_minus_1)");
236         return buckets[image & (buckets_size_minus_1)];
237     }
238     inline bucket_type &get_bucket(image_type image) {
239         TR("chsp::get_bucket(image)");
```

```
240         return buckets[image & (buckets.size() - 1)];
241     }
242
243     /// @todo Set prev.
244     inline iterator lookup(const key_type &k, bucket_type &b,
            image_type image) {
245         TR("chsp::lookup()");
246
247         if (is_empty(b))
248             return end();
249
250         // Prepare search
251         key_access.set_key(sentinel.value, k);
252         sentinel.image = image;
253
254         // Perform search
255         compartment_type *c = &b;
256         compartment_type *prev = NULL;
257         while (!((c->image == image) && equals(key_access.get_key(c->
                value), k))) {
258             prev = c;
259             c = c->next;
260         }
261
262         // See if we hit the target or the sentinel
263         if (c != &sentinel) {
264             /// @hack We'll need cheaper access to an iterator
                    pointing to the
265             /// relevant bucket.
266             size_type index = &b - &buckets.front();
267
268             // assert( prev != NULL );
269
270             return iterator(this, buckets.begin() + index, c, prev);
271         } else {
272             return end();
273         }
274     }
275     /// @todo Make private
276     inline iterator insert_in_bucket(bucket_type &b, compartment_type
            *c) {
277         TR("chsp::insert_in_bucket(bucket_type␣&,␣compartment_type␣*)"
                );
278
279         /// @hack We'll need cheaper access to an iterator pointing to
                 the
280         /// relevant bucket.
281         size_type index = &b - &buckets.front();
282
283         // Bucket empty, copy into bucket
284         if (is_empty(b)) {
285             b.value = c->value;
286             b.image = c->image;
287             b.next = &sentinel;
288
289             compartment_allocator.deallocate(c, 1);
290
291             return iterator(this, buckets.begin() + index, &b, NULL);
292         } else {
293             // Insert as next-first element, to minimize copying.
294             c->next = b.next;
295             b.next = c;
296             return iterator(this, buckets.begin() + index, c, &b);
297         }
298     }
```

```
299
300     inline iterator insert_in_bucket(bucket_type &b, const value_type
            &value, image_type image) {
301         TR("chsp::insert_in_bucket(bucket_type␣&,␣const␣value_type␣&,␣
                image_type)");
302
303         /// @hack We'll need cheaper access to an iterator pointing to
                the
304         /// relevant bucket.
305         size_type index = &b - &buckets.front();
306
307         // Bucket empty, copy into bucket
308         if (is_empty(b)) {
309             b.value = value;
310             b.image = image;
311             b.next = &sentinel;
312
313             return iterator(this, buckets.begin() + index, &b, NULL);
314         } else {
315
316             compartment_type *c = compartment_allocator.allocate(1);
317             compartment_allocator.construct(c, compartment_type(value,
                    image));
318
319             // Insert as next-first element, to minimize copying.
320             c->next = b.next;
321             b.next = c;
322             return iterator(this, buckets.begin() + index, c, &b);
323         }
324     }
325
326     inline void delete_from_bucket(bucket_type &b, const iterator &i)
            {
327         TR("chsp::delete_from_bucket(bucket_type␣&,␣const␣iterator␣&)"
                );
328
329         // First in chain
330         if (!i.prev) {
331             assert(i.curr == &b);
332
333             // Copying the sentinel is OK.
334             compartment_type *next = i.curr->next;
335             b = *next;
336             if (next != &sentinel)
337                 compartment_allocator.deallocate(next, 1);
338         } else {
339             i.prev->next = i.curr->next;
340             compartment_allocator.deallocate(i.curr, 1);
341         }
342     }
343
344     /// @todo This is not pretty: an iterator is incremented till it
            meets a
345     /// valid bucket.
346     inline iterator begin() {
347         TR("chsp::begin()");
348
349         bucket_iterator it = buckets.begin();
350
351         while ( it != buckets.end() && it->next == NULL )
352             ++ it;
353
354         if ( it == buckets.end() )
355             return iterator(this, it, &sentinel, NULL);
356         else
```

```
357             return iterator(this, it, &(*it), NULL);
358     }
359
360     inline iterator end() {
361         TR("chsp::end()");
362         return iterator(this, buckets.end(), &sentinel, NULL);
363     }
364
365     /**
366      * Current capacity, for computing load factors.
367      */
368     inline size_type capacity() const {
369         return buckets.size();
370     }
371
372     /// @todo Maybe move the size into storage policy
373     // inline size_type size() const {
374         // return
375     // }
376 };
377
378 #endif /* ifndef CHAINED_HASH_STORAGE_POLICY_H */
379 // Local Variables:
380 // mode: c++
381 // c-basic-offset:4
382 // tab-width:4
383 // End:
384 /* $Id: chained_hash_storage_policy.h,v 1.1.1.1 2005/06/16 13:51:20
        lyhne Exp $ */
```

AppendixA.3.7 Chained hash compartment (chained_hash_compartment.h)

```
1  /*
2   * $Id: chained_hash_compartment.h,v 1.1.1.1 2005/06/16 13:51:20 lyhne
        Exp $
3   */
4
5  #ifndef CHAINED_HASH_COMPARTMENT_H
6  #define CHAINED_HASH_COMPARTMENT_H
7
8  template<class V>
9  struct chained_hash_compartment {
10 private:
11     typedef chained_hash_compartment<V> compartment_t;
12
13 public:
14     typedef V value_type;
15
16     value_type value;
17     size_t image;
18     compartment_t *next;
19
20     inline chained_hash_compartment(value_type value, size_t image) :
21         value(value), image(image), next(NULL) {}
22
23     inline chained_hash_compartment(const chained_hash_compartment &b)
           :
24         value(b.value), image(b.image), next(b.next) {}
25
26     inline chained_hash_compartment() { }
27
28     inline chained_hash_compartment &operator=(const
           chained_hash_compartment &b) {
```

```
29          value = b.value;
30          image = b.image;
31          next = b.next;
32          return *this;
33      }
34  };
35
36  #endif /* ifndef CHAINED_HASH_COMPARTMENT_H */
37  // Local Variables:
38  // mode: c++
39  // c-basic-offset:4
40  // tab-width:4
41  // End:
42  /* $Id: chained_hash_compartment.h,v 1.1.1.1 2005/06/16 13:51:20 lyhne
       Exp $ */
```

AppendixA.3.8 Chained hash iterator (chained_hash_compartment_iterator.h)

```
1   /*
2    * Date: 14-Jun-2005
3    * $Id: chained_hashing_compartment_iterator.h,v 1.1.1.1 2005/06/16
         13:51:20 lyhne Exp $
4    */
5   #ifndef CHAINED_HASHING_COMPARTMENT_ITERATOR_H
6   #define CHAINED_HASHING_COMPARTMENT_ITERATOR_H
7
8   template <class Value, class Compartment, class Table, class Diff>
9   class chained_hashing_compartment_iterator : public std::iterator<std
        ::bidirectional_iterator_tag, Value, Diff> {
10  public:
11      // Bring up from base class
12      typedef Value value_type;
13
14      typedef Table self_type;
15      typedef typename self_type::compartment_type compartment_type;
16      typedef typename self_type::bucket_iterator bucket_iterator;
17
18  private:
19      template <class, class, class, class, class>
20          friend class chained_hash_storage_policy;
21
22      self_type *table;
23      compartment_type *curr;
24      compartment_type *prev;
25      bucket_iterator bucket;
26
27  public:
28
29      chained_hashing_compartment_iterator(self_type *table,
            bucket_iterator bucket, compartment_type *curr,
            compartment_type *prev) :
30          table(table), curr(curr), prev(prev), bucket(bucket) {}
31
32      chained_hashing_compartment_iterator() { }
33
34      inline value_type &operator* () {
35          assert(bucket != table->buckets.end());
36          return curr->value;
37      }
38
39      inline value_type *operator ->() {
40          return &curr->value;
41      }
```

```
42
43      /**
44       * WARNING: If all elements are in one bucket, this operation is O
             (m)
45       * @note curr never initially points to an empty bucket.
46       */
47      inline chained_hashing_compartment_iterator &operator ++ () {
48
49          compartment_type *next = curr->next;
50
51          if (next == &table->sentinel) {
52
53              // Skip to next bucket
54              ++bucket;
55              prev = NULL;
56
57              // Skip all empty buckets
58              while (bucket != table->buckets.end() && bucket->next ==
                     NULL)
59                  ++bucket;
60
61              if ( bucket != table->buckets.end() )
62                  next = &(*bucket);
63
64          } else {
65
66              prev = curr;
67          }
68
69          curr = next;
70          assert( curr != NULL );
71          return *this;
72      }
73
74      inline const chained_hashing_compartment_iterator operator ++ (int
            ) {
75          chained_hashing_compartment_iterator tmp = *this;
76          ++ *this;
77          return tmp;
78      }
79
80      /**
81       * WARNING: This has a worst case complexity of O(n^2)
82       */
83      inline chained_hashing_compartment_iterator &operator -- () {
84
85          // First element in bucket
86          if (prev == NULL) {
87
88              // Skip to previous bucket
89              /// @note We do not check if bucket == begin() here, as
                     this is
90              /// something that the user should have done before
                     calling this
91              /// method.
92              -- bucket;
93
94              // Skip empty buckets
95              while (bucket != table->buckets.begin() && bucket->next ==
                      NULL)
96                  -- bucket;
97
98              curr = &(*bucket);
99
```

```
100             // Find curr and prev (last and next-last elements in
                    bucket)
101             prev = NULL;
102             while (curr->next != &table->sentinel) {
103                 prev = curr;
104                 curr = curr->next;
105             }
106
107         } else {
108
109             curr = prev;
110
111             prev = &(*bucket);
112
113             if (prev == curr) {
114                 // curr is first element in bucket, leave until next
                        time to
115                 // find next bucket.
116                 prev = NULL;
117             } else {
118                 while (prev->next != curr && prev->next != &table->
                        sentinel)
119                     prev = prev->next;
120             }
121         }
122
123         assert( curr != NULL );
124         return *this;
125     }
126
127     inline const chained_hashing_compartment_iterator operator -- (int
            ) {
128         chained_hashing_compartment_iterator tmp = *this;
129         -- *this;
130         return tmp;
131     }
132
133     inline bool operator == (const
            chained_hashing_compartment_iterator &i) {
134         // Works because if bucket == end(), then curr == sentinel.
135         return (bucket == i.bucket) && (curr == i.curr);
136     }
137
138     inline bool operator != (const
            chained_hashing_compartment_iterator &i) {
139         return !operator == (i);
140     }
141 };
142
143 #endif /* ifndef CHAINED_HASHING_COMPARTMENT_ITERATOR_H */
144 /* $Id: chained_hashing_compartment_iterator.h,v 1.1.1.1 2005/06/16
        13:51:20 lyhne Exp $ */
```

AppendixA.4 Adapters, decorators and utilities

AppendixA.4.1 Key adapter (`key_adapter.h`)

```
1  /**
2   * @file key_adapter.h
3   *
4   * Adapter for getting at the key part of a compartment value
5   */
6  #ifndef _KEY_ADAPTER_H_
7  #define _KEY_ADAPTER_H_
```

```
 8
 9  #include <utility>
10
11  template <class V>
12  class key_adapter {
13  public:
14      typedef V value_type;
15      typedef V key_type;
16
17      static inline const key_type &get_key(const value_type &v) {
18          return v;
19      }
20
21      static inline void set_key(value_type &v, const key_type &k) {
22          v = k;
23      }
24  };
25
26  template <class K, class D>
27  class key_adapter<std::pair<K, D> > {
28  public:
29      typedef K key_type;
30      typedef std::pair<K, D> value_type;
31
32      static inline const key_type &get_key(const value_type &v) {
33          return v.first;
34      }
35
36      static inline void set_key(value_type &v, const key_type &k) {
37          v.first = k;
38      }
39  };
40
41  #endif // _KEY_ADAPTER_H_
42  // Local Variables:
43  // mode: c++
44  // c-basic-offset:4
45  // tab-width:4
46  // End:
47  /* $Id: key_adapter.h,v 1.1.1.1 2005/06/16 13:51:20 lyhne Exp $ */
```

AppendixA.4.2 Const key iterator adapter (const_key_iterator.h)

```
1  #ifndef _CONST_KEY_ITERATOR_H_
2  #define _CONST_KEY_ITERATOR_H_
3
4  template<class V, class Vref, class CVref, class Vptr, class CVptr,
       class It>
5  class const_key_iterator : public It {
6  public:
7      typedef Vref reference;
8      typedef Vptr pointer;
9      typedef V value_type;
10
11     const_key_iterator(const It &i) : It(i) {}
12
13     const_key_iterator() : It() {}
14
15     inline Vptr *operator ->() {
16         return (Vptr)It::operator->();
17     }
18
19     inline CVptr *operator ->() const {
20         return (CVptr)It::operator->();
21     }
```

```
22
23      inline Vref operator*() {
24          return (Vref)It::operator*();
25      }
26
27      //inline CVref operator*() const {
28      //  return (CVref)It::operator*();
29      //}
30
31      inline bool operator == (const const_key_iterator &b) {
32          return It::operator==((const It &)b);
33      }
34
35      inline bool operator != (const const_key_iterator &b) {
36          return It::operator!=((const It &)b);
37      }
38
39      inline const_key_iterator &operator ++ () {
40          return (const_key_iterator &)It::operator++();
41      }
42
43      inline const_key_iterator &operator -- () {
44          return (const_key_iterator &)It::operator--();
45      }
46
47      inline const const_key_iterator operator ++ (int) {
48          return const_key_iterator(It::operator++(0));
49      }
50
51      inline const const_key_iterator operator -- (int) {
52          return const_key_iterator(It::operator--(0));
53      }
54  };
55
56  #endif // _CONST_KEY_ITERATOR_H_
57  /* $Id: const_key_iterator.h,v 1.1.1.1 2005/06/16 13:51:20 lyhne Exp $
        */
```

AppendixA.4.3 Two-level iterator (twolevel_iterator.h)

```
1   #include <iterator>
2   #include "emulist.h"
3
4   #ifndef _TWOLEVEL_ITERATOR_H_
5   #define _TWOLEVEL_ITERATOR_H_
6
7   template <class OuterValue, class OuterIt, class InnerIt>
8   struct twolevel_iterator_value_composer {
9       typedef typename OuterValue::value_type value_type;
10      typedef typename InnerIt::reference reference;
11      typedef typename InnerIt::pointer pointer;
12
13      static OuterValue &get_list(const OuterIt &outer) {
14          return *outer;
15      }
16
17      static inline reference get_value(const OuterIt &, const InnerIt &
            inner) {
18          return *inner;
19      }
20  };
21
22  /// @hack emulist's size_type should come elsewhere from
23  template <class Key, class OuterIt, class InnerIt>
```

```
24  struct twolevel_iterator_value_composer<std::pair<Key, emulist<size_t>
        >,
25      OuterIt, InnerIt> {
26
27      typedef const Key value_type;
28      /// @hack Should come from storage policy (the outer list type)
29      typedef Key &reference;
30      typedef Key *pointer;
31
32      static emulist<size_t> &get_list(const OuterIt &outer) {
33          return outer->second;
34      }
35
36      static inline reference get_value(const OuterIt &outer, const
            InnerIt &) {
37          return outer->first;
38      }
39  };
40
41  /**
42   * Iterator over elements in a list of lists, or more specifically: A
         list of
43   * pairs whose second elements are lists.
44   */
45  template<class OuterIt,
46      class InnerList,
47      class Composer = twolevel_iterator_value_composer<typename OuterIt
            ::value_type,
48          OuterIt, typename InnerList::iterator> >
49  class twolevel_iterator :
50      public std::iterator<std::bidirectional_iterator_tag,
51          typename Composer::value_type,
52          size_t,
53          typename Composer::pointer,
54          typename Composer::reference> {
55  public:
56      typedef OuterIt outer_iterator;
57
58  protected:
59      typedef twolevel_iterator<OuterIt, InnerList, Composer> this_type;
60
61      typedef InnerList list_type;
62
63      typedef typename list_type::iterator inner_iterator;
64      //typedef typename list_type::value_type inner_value_type;
65
66      inner_iterator inner;
67      outer_iterator outer;
68      outer_iterator end;
69
70      inner_iterator inner_begin, inner_end;
71
72      /// @hack Fake list for an iterator when outer does not point to a
            list.
73      /// Must be static to be able to compare different iterators.
74      static list_type fake_list;
75
76  public:
77      twolevel_iterator(outer_iterator outer, outer_iterator end) :
78          inner(outer != end ? Composer::get_list(outer).begin() :
79              fake_list.end()),
80          outer(outer),
81          end(end),
82          inner_begin(inner),
83          inner_end(outer != end ? Composer::get_list(outer).end() :
```

```
84              fake_list.end()) {
85      }
86
87      this_type &operator++ () {
88          ++inner;
89          if (inner == inner_end) {
90              ++outer;
91              if (outer != end) {
92                  inner = inner_begin = Composer::get_list(outer).begin
                        ();
93                  inner_end = Composer::get_list(outer).end();
94              } else {
95                  inner = inner_begin = inner_end = fake_list.end();
96              }
97          }
98          return *this;
99      }
100
101     this_type &operator-- () {
102
103         if (inner == inner_begin) {
104             --outer;
105             inner_begin = Composer::get_list(outer).begin();
106             inner = inner_end = Composer::get_list(outer).end();
107         }
108         --inner;
109         return *this;
110     }
111
112     const this_type operator++ (int) {
113         this_type tmp = *this;
114         ++*this;
115         return tmp;
116     }
117
118     const this_type operator-- (int) {
119         this_type tmp = *this;
120         --*this;
121         return tmp;
122     }
123
124     bool operator==(const this_type &b) {
125         return (outer == b.outer) && (inner == b.inner);
126     }
127
128     bool operator!=(const this_type &b) {
129         return !operator==(b);
130     }
131
132     typename Composer::reference operator*() {
133         return Composer::get_value(outer, inner);
134     }
135 };
136
137 template <class OuterIt, class InnerList, class Composer>
138 typename twolevel_iterator<OuterIt, InnerList, Composer>::list_type
139     twolevel_iterator<OuterIt, InnerList, Composer>::fake_list;
140
141 #endif //_TWOLEVEL_ITERATOR_H_
142 /* $Id: twolevel_iterator.h,v 1.1.1.1 2005/06/16 13:51:20 lyhne Exp $
    */
```

AppendixA.4.4 Multiset emulated list counter class (`emulist.h`)

```
1  #ifndef _EMULIST_H_
```

```
2  #define _EMULIST_H_
3
4  /**
5   * A fake list of empty items.
6   */
7  template <class Size>
8  class emulist {
9  public:
10     typedef Size size_type;
11     typedef void value_type;
12
13 private:
14     size_type element_count;
15
16 public:
17     class iterator {
18     private:
19         size_type index;
20
21     public:
22         inline iterator(size_type index) : index(index) {}
23
24         inline iterator &operator ++() {
25             index++;
26             return *this;
27         }
28
29         inline iterator &operator --() {
30             index--;
31             return *this;
32         }
33
34         inline bool operator==(iterator b) {
35             return index == b.index;
36         }
37     };
38
39     typedef iterator const_iterator;
40
41     inline emulist() : element_count(0) {
42     }
43
44     inline void clear() {
45         element_count = 0;
46     }
47
48     inline size_type size() {
49         return element_count;
50     }
51
52     static inline iterator begin() {
53         return iterator(0);
54     }
55
56     inline iterator end() const {
57         return iterator(element_count);
58     }
59
60     inline void push_back(void) {
61         element_count++;
62     }
63 };
64
65 #endif //_EMULIST_H_
```

AppendixA.4.5 Random number generator (`random.h`)

```
1  /*
2   * File: random.h
3   *
4   * This is a random number generator, which uses a long long int as
        base, and
5   * allows for construction of random number generators delivering
        other sizes
6   * as output.
7   */
8
9   /**
10   * @note on a 32 bit x86 machine long is 4 bytes and long long is 8.
11   *       on an amd64 both long and long long are 8 bytes.
12   */
13
14 #ifndef _RANDOM_H_INCLUDED
15 #define _RANDOM_H_INCLUDED
16
17 #include <time.h>
18
19 #ifdef __amd64__
20 #define INT_BITS 64
21 #else // __amd64__
22 #define INT_BITS 32
23 #endif // __amd64__
24
25 struct random_constants
26 {
27     /// @note sizeof(seed_type) is 8 on both 32 and 64 bit platforms
28     typedef unsigned long long seed_type;
29     typedef unsigned long value_type;
30
31     static const seed_type MULTIPLIER = 0x5DEECE66DULL;
32     static const seed_type ADDEND = 0xB;
33     enum { MASK_BITS = sizeof(seed_type) * 6 };
34     static const seed_type MASK = ((seed_type)1 << MASK_BITS) - 1;
35 };
36
37
38 class random : public random_constants {
39 private:
40     static seed_type seed;
41
42 public:
43     static inline void set_seed(seed_type seed)
44     {
45         random::seed = (seed ^ MULTIPLIER) & MASK;
46     }
47
48     static inline void randomize()
49     {
50         set_seed((seed_type)time(NULL));
51     }
52
53     /**
54      * Get the given number of bits of the next number in the random
            sequence
55      */
56     static inline unsigned long get_next(unsigned long bits)
57     {
58       seed = (seed * MULTIPLIER + ADDEND) & MASK;
59       return (unsigned long)(seed >> (MASK_BITS - bits));
60     }
```

```
61
62      /**
63       * Special case of get_next(sizeof(unsigned long) * 8), for speed.
64       */
65      static inline unsigned long get_next()
66      {
67        seed = (seed * MULTIPLIER + ADDEND) & MASK;
68  #if INT_BITS==32
69        return (unsigned long)(seed >> (MASK_BITS - (sizeof(unsigned
              long) * 8)));
70  #else // INT_BITS==64
71        // On a typical 64 bit machine, we don't have access to 128 bit
              types,
72        // and therefore we're splitting up the computation.
73        /// @todo Find a library method or other mechanism to do this,
              rather than
74        /// hardcoding per platform type.
75        seed_type seed1 = seed;
76        seed = (seed * MULTIPLIER + ADDEND) & MASK;
77        seed_type seed2 = seed;
78        return (seed1 >> 8) + (seed2 << 32);
79  #endif
80      }
81  };
82
83  #endif  // _RANDOM_H_INCLUDED
84  /* $Id: random.h,v 1.1.1.1 2005/06/16 13:51:20 lyhne Exp $ */
```