Lists Revisited: Cache Conscious STL Lists

LEONOR FRIAS
and
JORDI PETIT
and
SALVADOR ROURA
Universitat Politècnica de Catalunya

We present three cache conscious implementations of STL standard compliant lists. Up to now, one could either find simple doubly linked list implementations that easily cope with standard strict requirements, or theoretical approaches that do not take into account any of these requirements in their design. In contrast, we have merged both approaches, paying special attention to iterators constraints. In this paper, the competitiveness of our implementations is evinced with an extensive experimental analysis. This shows, for instance, 5-10 times faster traversals and 3-5 times faster internal sort.

Categories and Subject Descriptors: E.1 [**Data Structures**]: —*Lists, stacks, and queues*; F.2 [**Analysis of Algorithms and Problem Complexity**]: General; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Software libraries*; B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*

General Terms: Algorithms, Design, Experimentation, Performance

Additional Key Words and Phrases: Cache conscious, iterator, STL

1. INTRODUCTION

The Standard Template Library (STL) is the algorithmic core of the C++ standard library. The STL is made up of containers, algorithms and iterators. Containers consist on basic data structures such as lists, stacks, vectors, maps or sets. Algorithms include basic operations on containers such as sort, reverse or find. Iterators are a kind of high-level pointers used to access and traverse the elements in a con-

A previous version appeared at the Workshop on Experimental Algorithms (WEA) 2006.
This work has been supported by ALINEX project under grant TIN2005-05446.
L. Frias has been supported by grant 2005FI 00856 of the *Agència de Gestió d'Ajuts Universitaris i de Recerca* and the European Social Fund.
J. Petit has been supported by GRAMARS project under grant TIN2004-07925-C03-01.
S. Roura has been supported by AEDRI-II project under grant MCYT TIC2002-00190.
Author's addresses: Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, E-08034 Barcelona; e-mail: {lfrias,jpetit,roura}@lsi.upc.edu.

tainer and, therefore, iterators are the link between containers and algorithms. The C++ standard [International Standard ISO/IEC 14882 1998] not only specifies the functionality of these objects, but also their temporal and spatial efficiency, using asymptotic notation.

From a theoretical point of view, the knowledge required to implement the STL is well laid down on basic textbooks on algorithms and data structures; see e.g. [Cormen et al. 2001; Weiss 1998; Sedgewick 1998]. In fact, the design of current widely used STL implementations (including SGI, GCC, and VC++) is based on these. For instance, all these implementations use doubly linked list to implement lists and red-black trees to implement maps and sets.

Taking advantage of the underlying memory hierarchy of modern computers is a way to improve the performance of some data structures. Not in vain, in the last years the algorithmic community has realized that the old unitary cost memory model has turned inaccurate with the changes in computer architecture. This has raised an interest on cache conscious algorithms and data structures that take into account the existence of a memory hierarchy. The cache aware (see e.g. [Lamarca 1996; Sen and Chatterjee 2000]) and the cache oblivious model (see e.g. [Demaine 2002; Frigo et al. 1999]) are the main models in which studies are done. Unfortunately, in order to include and distribute these data structures in standard software libraries, these must conform to some technical requirements that are usually not addressed in their theoretical treatment.

In this paper we focus on lists, one of the most simple but essential objects in the STL. Specifically, we propose three standard compliant implementations of lists that try to exploit the cache hierarchy. In order to assess the goodness of our implementations, we have performed a set of experiments on several common settings. The results show that nice speedups are obtained compared to traditional doubly linked lists found for instance in the GCC STL implementation and in the LEDA library [Mehlhorn and Naher 1999].

The remainder of the paper is organized as follows. To begin with, in Section 2, we present some basic facts on STL lists. Then, in Section 3, we consider the behavior of classic doubly linked list implementations with regard to the cache memory. We follow with a background on previous work on cache conscious lists in Section 4. Then, we present our data structure. First, in Section 5 we discuss the main design issues. Second, in Section 6 we present an specific reorganization algorithm and in Section 7, further analyze. Third, in Section 8 we give an experimental evaluation. Finally, we present in Section 9 some conclusions.

2. STL LISTS

A list in the STL library is a generic sequential container that supports single insertion and deletion at any point together with forward and backward traversal. Additionally, there are operations to sort and splice the list, as well as some minor utilities. The standard requires that, on a list with n elements, all operations take $\mathrm{O}(1)$ time, with the exception of `sort()` that takes $\mathrm{O}(n \log n)$ time, and of `size()` and several other copy operations that must take $\mathrm{O}(n)$ time.

As it is mandatory in all the STL containers, individual elements in lists are addressed and traversed thanks to high-level pointers called *iterators*. The standard

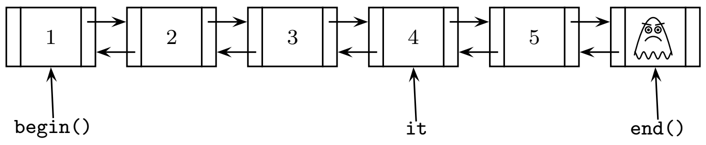


Fig. 1. Classical doubly linked implementation.

states that no operation can invalidate list iterators, that is, they must point to the same element after any operation has been applied (except after deletion, which invalidates the iterator). Besides, the number of iterators per list and per element is unbounded.

The following program illustrates how to use an STL list to solve the *Josephus problem*: Given a group of n men arranged in a circle under the edict that every k-th man will be executed going around the circle until only one remains, find the position in which one should stand in order to be the last survivor.

```
int survivor (int n, int k) {
    if (k==1) return n;
    list<int> L;
    for (int i = 1; i <= n; ++i) L.push_back(i);
    list<int>::iterator it = L.begin();
    for (int i = 1; i < n; ++i) {
        int c = (k - 2) % n;
        for (int j = 0; j < c; ++j) {
            ++it;
            if (it == L.end()) it = L.begin();
        }
        it = L.erase(it);
        if (it == L.end()) it = L.begin();
    }
    return *L.begin();
}
```

Further details and documentation on STL lists can be found in [Josuttis 1999].

3. CACHE BEHAVIOR OF DOUBLY LINKED LISTS

In order to implement a list fulfilling all the STL requirements, a classical doubly linked list suffices. Indeed, this is what all known STL implementations do. In this case, nodes store an element together with a pointer to the previous and the next node in the list. Moreover, list iterators are just pointers to these nodes. Because of the existence of the `begin()` and `end()` iterators, implementations include a fake node at the end of the list. Figure 1 depicts this classical solution and just following we give its simplified definition in C++.

```
struct node {
    node* next;                 struct iterator {
    node* prev;                     node* p;
    elem x;                     };
};
```

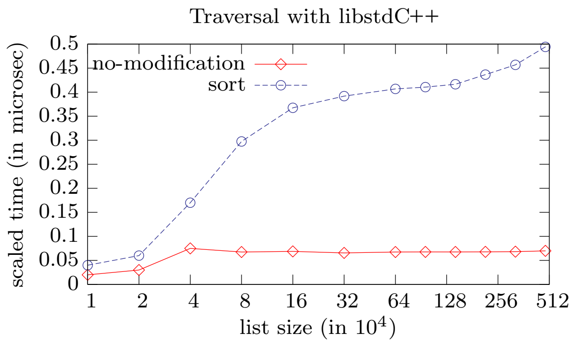


Fig. 2. Time measurements for list traversal before and after shuffling it. The vertical axis is scaled to the list size.

A key property of any pointer-based data structure as the one above is that even though the physical position of each element in the memory is permanent, its logical position in the list can be changed by modifying the pointers in the nodes. As a consequence, iterators are not affected by these logical movements and they always remain valid.

Furthermore, pointer-based data structures use *memory allocators* to allocate and deallocate their nodes. These allocators typically answer consecutive memory requests with consecutive addresses of memory (whenever possible). This is particularly important in the case of lists, because if elements are added at one end (and no other allocations are performed at the same time), there is a good chance that logically consecutive elements are also physically consecutive, which leads to a good cache performance when elements are traversed. However, if elements are inserted at random points or if the list is shuffled, logically consecutive elements will be rarely at physically nearby locations. Therefore, a traversal may incur in a cache miss per access, thus dramatically increasing the execution time.

In order to give evidence of the above statement, we have performed the following experiment with the GCC list implementation: Given an empty list, n random integers are pushed back one by one. Then, we measure the time to fully traverse it. Afterwards, we sort the list (thus randomly shuffling the links between nodes) and we measure again the time to fully traverse it. Traversal times before and after sorting the list are shown in Figure 2. Except for very small lists, it can be seen that the traversal of the shuffled list is about ten times slower than the traversal of the original list; and the only difference can be in the memory layout (and so, in the number of cache misses).

Taking into account that lists are used when elements are often reorganized (e.g. sorted) or inserted and deleted at arbitrary positions (e.g. the Josephus problem), the previous experiment shows that it is worth to try to improve the performance of lists using a cache conscious approach. Note that if a user only wished to perform operations at the ends, he would better have selected a vector, stack, queue or dequeue rather than a list.

4. PREVIOUS WORK

Memory hierarchies were introduced by computer architects to try to minimize the gap between access memory time and arithmetic operation time. Their effectiveness rely on the locality of data and programs. However, this is not a universal property as we have seen in Section 3 with pure pointer-based data structures.

This has motivated looking for alternative data structures that organize data in such a way that logical access patterns are related to physical memory locations. These are the so-called cache conscious data structures. Specifically, depending on whether they are designed for specific parameters of the cache hierarchy or not, they are respectively called cache aware or cache oblivious respectively.

A good summary of previous work on cache conscious lists can be found in [Demaine 2002] which we reproduce just following. The operations taken into account are traversal (as a whole), insertion and deletion. The cost is measured as the number of memory transfers. Then, let be n the list size and B be the cache line size. The *cache aware* solution consists on a partition of $\Theta(n/B)$ pieces, each between $B/2$ an B consecutive elements, achieving $O(n/B)$ amortized traversal cost and constant update cost. The *cache oblivious* solutions are based on the packed memory structure [Bender et al. 2002], basically an array of $\Theta(n)$ size with uniformly distributed gaps. To guarantee this uniformity updates require $O((\log^2 n)/B)$, which can be slightly lowered by partitioning the array in smaller arrays. Finally, *self-organizing structures* [Bender et al. 2002] achieve the same bounds as the cache aware but amortized. There, updates breaking the uniformity are allowed until the list is reorganized when traversed. However, note that this approach is not convenient in the case of STL lists because they are not traversed as a whole but step by step via iterators.

Therefore, theory shows that cache conscious lists fasten scan based operations and hopefully, do not rise significantly update costs compared to traditional doubly linked lists. However, none of the previous designs take into account common requirements of software libraries. In particular, combining iterator requirements and cache consciousness rules out some of the more attractive choices.

5. DESIGN OF CACHE CONSCIOUS STL LISTS

In this section, we present the main issues that appear in the design of cache conscious STL lists. Firstly, we present the problems to keep up with list iterator requirements as well as some hypotheses which have guide some of our choices. Then, we present the key ideas of our approach.

5.1 Iterator concerns and hypotheses on common list usages

As we have already seen, logical and physical locations of elements are not related in pointer-based structures. However, in cache conscious structures, these are heavily related to take advantage of cache hierarchies. As a consequence it is difficult to implement iterators trivially with pointers, because these should be kept coherent whenever a modification in the list affects the position of the pointed element. In fact, the main issue here is that an unbounded number of iterators can point to the same element. Besides, any arbitrary restriction on them would make the solution non standard compliant.

In order to overcome this difficulty, our design has been guided by some hypotheses on common list usages. In particular, from our experience as STL users, we can state that many uses of lists are in keeping with the following:

(1) The list is often modified at any position: insertions, deletions, splices.

(2) A particular list instance has typically only a few iterators on it.

(3) Many traversals are expected.

(4) The stored elements are not very big (e.g. integers, doubles, ...).

Let us briefly justify these points: Point 1 applies because if there were no updates at arbitrary points, the programmer would have selected another container rather than a list (an stack, a deque, a vector, ...). Point 2 raises from the observation that iterators are mainly declared as local variables to perform traversals or update the list. Truly, one can conceive creating arrays of iterators, but this is very rare. Point 3 is a direct consequence of Point 2 and the fact that elements in the list can only be accessed through iterators, which provide sequential access. Finally, Point 4 is a common assumption in the design of cache conscious structures that also appears implicitly or explicitly in general cache conscious data structures literature. Since this last condition can be checked at compilation time, a traditional implementation could be used instead (this could be neatly achieved with template specialization).

5.2 Our data structure

In the following, we present the main characteristics of our data structure. This will prove to be specially convenient when the hypotheses on common list usages hold, but, in any case, it shall always be compliant with the costs required by the STL even when the above conditions do not apply.

The core of the data structure is inspired by the cache aware solution previously mentioned in Section 4. Specifically, it consists in a doubly linked list of buckets. A bucket contains an small array of K elements, pointers to the next and previous buckets, and extra fields to manage the data in the array. This simple data structure ensures locality inside the bucket, but logically consecutive buckets are let to be physically far.

From now on, K will denote the *capacity* of the buckets, that is, the number of elements a bucket can hold. Moreover, we will say that the *occupancy* of a bucket is its number of elements divided by its capacity.

Finally, we must deal with the following issues:

(1) *Capacity of a bucket.* The capacity of a bucket has been fixed according to the outcome of the experiments. We will see it later.

(2) *Elements arrangement inside a bucket.* We devise three possible ways to arrange the elements inside a bucket:

— *Contiguous*: The elements are stored contiguously from left to right. Consequently, insertions and deletions must shift some elements towards their left or their right.

— *With gaps*: Elements are stored from left to right, but gaps between elements are allowed. Gaps though do not improve the worst case with respect to the number of shifts. Besides, an extra bit per element is needed to distinguish real elements from gaps.

— *Linked*: The order of the elements inside the bucket is set by internal links rather than the implicit left to right order. This requires extra space for the links, but avoids shifting inside the bucket. Thus, this solution is more scalable.

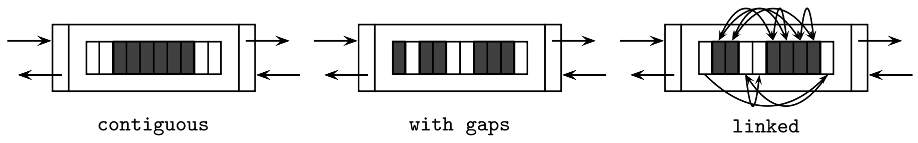


Fig. 3. Bucket arrangement.

These three possibilities are depicted on Figure 3 and just following a simplified definition in C++ is given:

```
struct bucket{
    bucket* next;
    bucket* prev;
    elem[K] a;
    int beg;
    int end;
};
```

```
struct bucket{
    bucket* next;
    bucket* prev;
    elem[K] a;
    bool[K] occupied;
};
```

```
struct bucket{
    bucket* next;
    bucket* prev;
    elem[K] a;
    int[K] next;
    int[K] prev;
};
```

(3) *Bucket reorganization when inserting or deleting elements.* The algorithms involved in the reorganization of buckets preserve the data structure invariant after an insertion or deletion. Our actual invariant will be given latter, in the next section. The main issue here is keeping a good balance between high occupancy, few bucket accesses per operation, and few of elements movements.

(4) *Iterators management.* Our key idea is identifying all the iterators referred to an element with a dynamic node called *relay* that points to it. Thus, there are two levels of indirection: iterators pointing to a relay, and relays pointing to elements. The pointer from a relay to an element is actually a pointer to its bucket and its index in that bucket. In this way, the rest of bucket data can also be accessed. Besides, it keeps count of the number of iterators that are referring to that relay so that it can be destroyed when there are none. A problem remains: when the physical location of an element changes, its relay (if it exists) must be reached and changed accordingly in constant time. We devise two approaches for that:
— *Bucket of pairs*: This solution is depicted in Figure 4 and has the following simplified definition in C++ (arrangement details are obviated here):

```
struct bucket{
    bucket* next;
    bucket* prev;
    pair<elem,relay*>[K] a;
};
```

```
struct relay{
    bucket* b;
    int index;
    int count;
};
```

```
struct iterator{
    relay* r;
};
```

In this obvious solution, for each element, a pointer to its relay is kept. This technique is easy to implement and still uses less space than a traditional doubly linked list because it needs one pointer per element rather than two.
— *2-level*: This solution is depicted in Figure 5 and has the following simplified definition in C++ (arrangement details are obviated here):

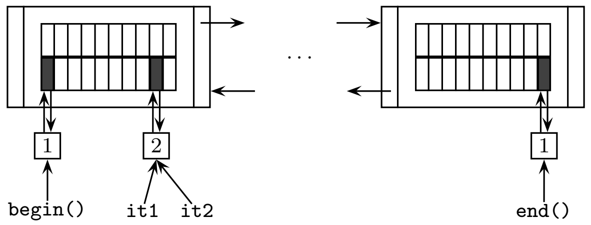


Fig. 4. Bucket of pairs.

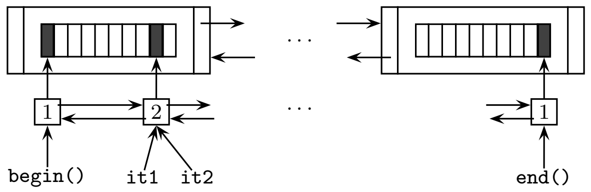


Fig. 5. 2-level list.

```
struct bucket{
    bucket* next;
    bucket* prev;
    elem[K] a;
};
```

```
struct relay{
    relay* next;
    relay* prev;
    bucket* b;
    int index;
    int count;
};
```

```
struct iterator{
    relay* r;
};
```

The key point is taking advantage of the fact that STL list elements can only be accessed through iterators (by library users). Let *it* denote the iterator given as operation parameter, *r* the relay to which *it* points and *elem* the element to which *r* points; our reorganization algorithm guarantees that if the location of an element *other* changes, *other* is at constant (logical) distance of *elem*. Then, to find its relay also in constant time, we keep a doubly linked list of relays in the same relative order that their elements and perform a sequential search starting at *r*. Note that if the relay of *other* does not exist, the search finishes when the first relay pointing to another bucket is found. This solution is more involved to implement but uses less space compared to the previous one, provided there are not many relays.

It is clear that, in both solutions, the locality of iterator accesses decreases with the number of elements with iterators, because the locations of the relays are not related to their elements. However, assuming that there will not be many iterators in a single application, we expect to find the relays in the cache. In any case, our two approaches conform to the standard whatever the number of iterators.

(5) *relay allocation:* Given that iterators are frequently incremented or decremented and typically moving from/to places where there are no other iterators, our implementations avoid (whenever possible) to destroy the relay (to leave an element) and recreate it (to arrive to the next element). Not only that, our implementations also make use of a pool of relay nodes to delay as much as possible the use of the allocator to get or release memory for them.

6. REORGANIZATION OF THE BUCKETS

In this section we present with some detail how we keep the buckets reorganized when insertion and deletion operations are applied to the list. For the sake of simplicity, we assume that the elements in the buckets are arranged using the linked strategy, so that we may ignore intra-bucket movements.

This section is organized as follows: First, we present the notation that we will use. Then, we state the representation invariant of our data structure. Finally, we give the necessary reorganization algorithms to keep with this invariant when insertion and deletions are applied to the list. To do so, we first present some useful operations, then present the algorithm for the general case and finally present the algorithms for various particular cases.

6.1 Notation

In the following, we denote by $b(i)$ the i-th bucket of the list, by $\ell(i)$ the bucket at its left, and by $r(i)$ the bucket at its right, provided that they exist. We also denote by $\ell^j(i)$ the bucket j positions to the left of $b(i)$ and by $r^j(i)$ the bucket j positions to the right of $b(i)$. On the other hand, we denote by $|b(i)|$ the occupancy of bucket $b(i)$, that is, its number of elements divided by its capacity (K). Moreover, we define the occupancy of buckets from i to j as $|b(i \dots j)| = |b(i)| + \cdots + |b(j)|$ and $s_i = |\ell(i)| + |b(i)| + |r(i)|$. From now on, x always denotes the new element to be inserted or deleted and m denotes the number of buckets in the list (there is always at least one bucket).

6.2 Representation invariant

Roughly, our goal is to try to achieve an average occupancy of at least $\frac{2}{3}$ for each bucket. This is a compromise between guaranteeing a very high occupancy for any pattern, which is costly, and very loose restrictions that could turn the data structure useless. Besides, endpoint buckets constraints are relaxed to achieve almost maximum occupancy when lists are built adding elements at the back or the front, which is a common building pattern.

Specifically, the representation invariant of our data structure is given by the following constraints, which are depicted in Figure 6.

— *General constraint:* For all $b(i)$, s.t. $2 < i < m - 1$, we have $s_i \geq 2$.

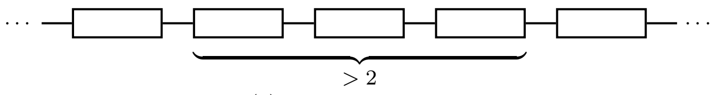
— *Endpoints constraints:* When $m \geq 3$, we have

$$\begin{cases} |b(1)| = 0 \implies |b(2)| > \frac{5}{12}, \\ |b(1)| \neq 0 \implies |b(2)| = 1, \end{cases} \quad \text{and} \quad \begin{cases} |b(m)| = 0 \implies |b(m-1)| > \frac{5}{12}, \\ |b(m)| \neq 0 \implies |b(m-1)| = 1. \end{cases}$$

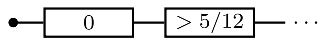
— *Small lists constraints:* We have

$$\begin{cases} m = 4 \implies |b(1 \dots 4)| > \frac{3}{2}. \\ m = 3 \implies |b(1 \dots 3)| > 1. \\ m = 2 \implies |b(1 \dots 2)| > \frac{1}{2}. \end{cases}$$

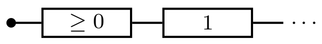
Lists with one single bucket do not have occupancy constraints.



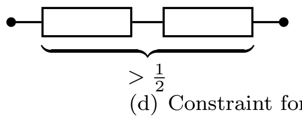
(a) General constraint



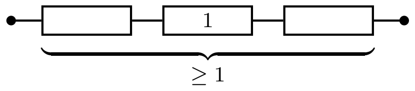
(b) Endpoint constraint, case 1



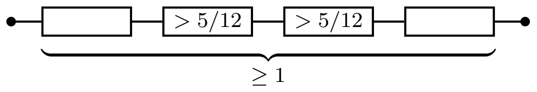
(c) Endpoint constraint, case 2



(d) Constraint for lists with two buckets



(e) Constraint for lists with three buckets (in addition to the endpoint constraints)



(f) Constraint for lists with four buckets (in addition to endpoint constraints).

Fig. 6. Occupancy constraints

6.3 Useful reorganization operations

In order to present our reorganization algorithm, let us define the following basic reorganization operations:

— *Transfer from $b(i)$ to $b(j)$:* One or several elements are moved from bucket $b(i)$ to bucket $b(j)$, which typically corresponds to either $\ell(i)$ or $r(i)$.

— *Split of $b(i)$:* A new bucket is allocated to the left or right of $b(i)$, and the elements are redistributed between $\ell(i)$, $b(i)$, $r(i)$ and the new bucket.

— *Merge of $b(i)$:* The elements from $b(i)$ are redistributed among $\ell(i)$ and $r(i)$, then, $b(i)$ is deallocated.

6.4 Reorganization algorithm for the general case

We consider first the reorganization algorithm for buckets far form the endpoints and on large enough lists. Assume that a deletion or an insertion takes place in bucket $b(i)$, with $2 < i < m-1$. There are three possible cases where reorganization is needed:

(1) *Deletion from $b(i)$, which makes $s_k = 2$ for some $k \in \{\ell(i), b(i), r(i)\}$.* In this case, first x is removed and then $b(k)$ is merged.

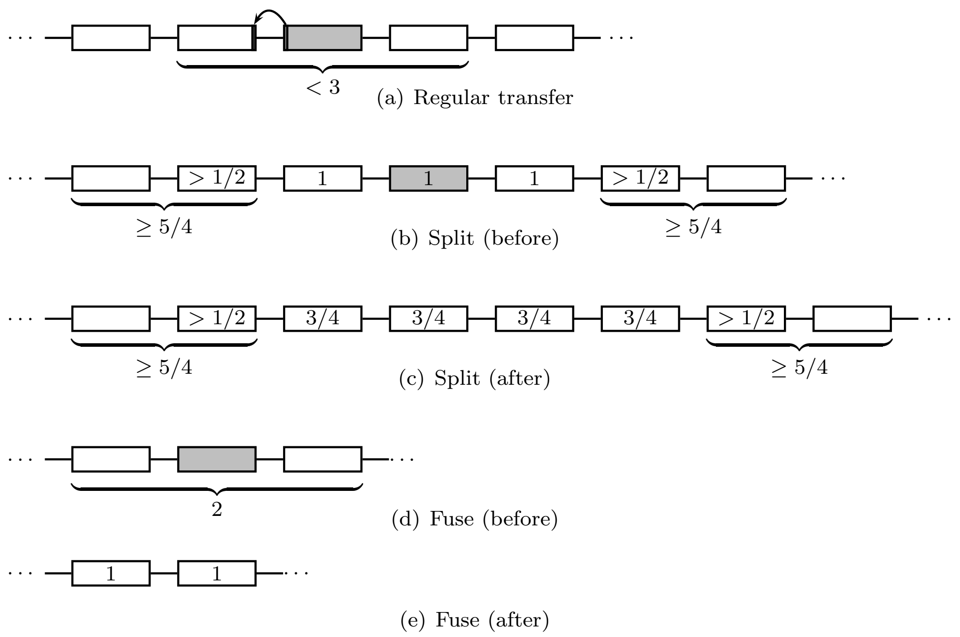


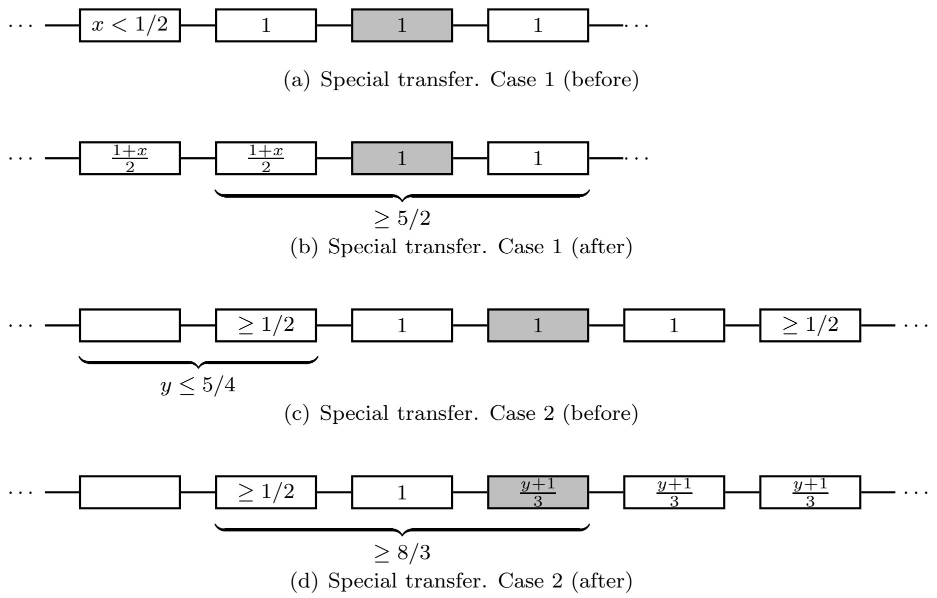
Fig. 7. General reorganization operations.

(2) *Insertion in* $b(i)$ *such that* $|b(i)| = 1$ *and* $s_i < 3$*:*
In this case, an element must be transfered from $b(i)$ to either $\ell(i)$ or $r(i)$ to make room for x. It seems reasonable to choose the emptiest neighbor, but this is not necessary for the forthcoming analysis.

(3) *Insertion in* $b(i)$ *such that* $|b(i)| = 1$ *and* $s_i = 3$ *and* $3 < i < m - 2$*:*
In this case, either a split or a transfer may be performed. The algorithm is the following:

(a) *General case:* first, $b(i)$ is splited, so that $\ell(i)$, $b(i)$, $r(i)$ and the new bucket have capacity of $\frac{3}{4}$ at least. Afterwards, x is put in its place.
(b) *When* $|\ell^2(i)| < \frac{1}{2}$*:* The elements in $\ell^2(i)$ and $\ell(i)$ are equally distributed among these two buckets. In this way, $s_{\ell^3(i)}$ increases, $s_{\ell^2(i)}$ and $s_{\ell(i)}$ remain the same, and s_i decreases (never under $\frac{5}{2}$). Finally, one element from i is moved to $\ell(i)$, thus making room for x, which is put in its place.
(c) *When* $|\ell^3(i)| + |\ell^2(i)| < \frac{5}{4}$*:* The elements in $\ell^3(i)$, $\ell^2(i)$ and $\ell(i)$ are equally distributed among these three buckets. In this way, $s_{\ell^4(i)}$ and $s_{\ell^3(i)}$ increase, $s_{\ell^2(i)}$ remains the same, $s_{\ell(i)}$ decreases (never under $\frac{7}{3}$) and s_i decreases (never under $\frac{8}{3}$). Finally, we move one element from i to $\ell(i)$, thus making room for x, which is put in its place.

The three most general cases (1, 2 and 3a) are shown in Figure 7. The two special transfer cases (3b and 3c) are shown in Figure 8.



(a) Special transfer. Case 1 (before)
(b) Special transfer. Case 1 (after)
(c) Special transfer. Case 2 (before)
(d) Special transfer. Case 2 (after)

Fig. 8. Special transfer operations.

6.5 Reorganization algorithm at the endpoints

We consider now the reorganization algorithm at the endpoints of large enough lists. Here, the rules change to achieve almost perfect occupancy for typical list building patterns at the ends. We wish to note that, if there were no special constraints for the endpoints and the list was built adding elements at the ends, the general reorganization algorithm would lead to an average occupancy of about $\frac{3}{4}$, which is already greater than the lower bound of $\frac{2}{3}$.

The rules are simple and apply symmetrically at the right end. First, we present the transfer rules and second, rules that allocate or deallocate buckets.

6.5.1 *Transfer rules*

(1) *Deletion in* $b(2)$ *such that* $|b(1)| > 0$: First, the element x is removed. Then, the last element from $b(1)$ is transfered to $b(2)$.

(2) *Insertion in* $b(2)$ *such that* $|b(2)| = 1$ *and* $|b(1)| < 1$: The first element from $b(2)$ is transfered to $b(1)$. Then, x is put at its place.

6.5.2 *Allocation and deallocation rules*

(1) *Deletion in* $b(2)$ *such that* $|b(2)| \leq 5/12$: First, the element of $b(2)$ is deleted and $b(1)$ is deallocated. Then, $b(2)$ elements are transfered to $b(3)$ until this is full.

(2) *Insertion in* $b(1)$ *such that* $|b(1)| = 1$: First of all, an empty bucket b is allocated and placed as the new $b(1)$. Then, if x was inserted at the front of the list, we just put x in b. Otherwise, the first element of $b(2)$ is shifted to b to make room for x.

(3) *Insertion in* $b(2)$ *such that* $s_2 = 3$: We perform a split operation with final occupancies (in this order) of 1/2, 1, 3/4 and 3/4. Then, x is put at its place.

(4) *Insertion in* $b(3)$ *such that* $s_3 = 3$: First, we perform a split operation as in the general case. Second, elements from the $b(1)$ are transfered to $b(2)$ until $b(2)$ is full or $b(1)$ is empty. (Note: if the list has only five buckets, this step must be repeated for the right endpoint, because the third bucket from the left is also the third bucket from the right.) Then, x is put at its place.

6.6 Reorganization algorithm for small lists

We consider now lists with four or less buckets ($m \leq 4$). The rules are adapted as follows:

— *Transfer rules in Section 6.5.1*: all apply, except for lists s.t. $m = 2$, in which elements are inserted with no transfers when possible.

— *Allocation of new buckets in lists s.t.* $m \leq 3$: a new bucket is allocated only when there is no room at all. Further, the elements are reorganized as follows: the endpoint buckets are at $\frac{1}{2}$ or more of its capacity; the rest become full.

— *Allocation of new buckets in lists s.t.* $m = 4$: Here, we assume that the left endpoint is the nearest to the insertion point.

—*Insertion in* $b(1)$: the same algorithm as in Section 6.5.2.

—*Insertion in* $b(2)$: If $b(4)$ is not full, one element is transfered from $b(3)$ to $b(4)$, and then, one from $b(2)$ to $b(3)$. If $b(4)$ is full (i.e. all the buckets are full), a split operation must be performed with final list occupancies of $\frac{5}{8}$, 1, $\frac{3}{4}$, 1 and $\frac{5}{8}$.

— *Deallocation of buckets*: A bucket is deallocated when the corresponding minimum average occupancy is reached.

—$m = 3$ *or* $m = 4$: The endpoint bucket b which is empty is deallocated. If $m = 4$, the elements of b neighbor are transfered to the following neighbor until this is full.

—$m = 2$: The elements from the most emptiest bucket are transfered to the other one. Then, it is deallocated.

7. AN UPPER BOUND FOR THE NUMBER OF CREATED AND DESTROYED BUCKETS

In this section, we present and prove two asymptotically optimal properties of our reorganization algorithms related to the number of created/destroyed buckets. In practice, this means that our bucket management strategy is robust whatever the applied sequence of list operations. This goes towards a good (cache) performance.

7.1 Interleaved operations at the same point

THEOREM 7.1. *Let a list conform to the representation invariants given in Section 6. Consider an arbitrary long sequence of interleaved insertions and deletions at the same point. Then, the total number of allocated and deallocated buckets is at most two.*

PROOF. We have the following case analysis:

(1) *The first operation performs no allocation neither deallocation.* There are two cases:

(a) *No element is transfered.* In this subcase, the next operation returns the bucket to its initial state, and so happens again and again, and so, no allocation or deallocation are performed (in fact, neither transfers).
(b) *An element is transfered from bucket b_i to bucket b_j.* If b_i or b_j are endpoint buckets, then this operation is reversed at each step, and so, no allocations or deallocations are performed. Otherwise, this operation must be an insertion (because deletions do not transfer elements except when at the endpoint buckets), and so, the next is a deletion. In this case, a deallocation cannot either occur because the value of s_i is the initial one.

(2) *The first operation is an insertion and allocates a bucket.* Then, the next deletion cannot deallocate a bucket. For small lists, this is due to the fact that allocation and deallocation thresholds do not coincide. For big lists, when the allocated bucket becomes an endpoint, its next bucket is full and therefore, cannot be afterwards deallocated. Finally, in the most general case, a split guarantees that the two central buckets i where the element is actually inserted, have $s_i > \frac{9}{4}$, and so, a right afterwards deallocation cannot either occur.
Therefore, we turn to case (1), for which has been shown that no further allocations or deallocations are performed.

(3) *The first operation is a deletion and deallocates a bucket.* There are two subcases:
(a) *The next operation (insertion) allocates a bucket.* Then, we are in case 2.
(b) *The next operation (insertion) does not allocate a bucket.* Then, we are in case 1.
In any case, we turn to a case for which has been shown that no further allocations or deallocations are performed. Further, for case 2, the total number of allocated and deallocated buckets is exactly two.

□

7.2 Arbitrary sequences of insertions and deletions

THEOREM 7.2. *Let L be an empty list. Consider a sequence of r insertions and/or deletions at arbitrary positions conform to the rules given in Section 6 applied to L. Then, the number of allocated and deallocated buckets is at most $6r/K$.*

PROOF. The amortized cost is shown using the potential method.

Let L_0 be the initial list. For each $i = 1, \ldots, r$, let c_i be the actual cost of the i-th operation, and let L_i be the list resulting after applying the i-th operation to L_{i-1}. Since we are interested in the number of created/destroyed buckets, we have

$$c_i = \begin{cases} 1, & \text{if a split/merging is performed during the } i\text{-th operation;} \\ 0, & \text{otherwise.} \end{cases}$$

Let Φ be a potential function (defined later) that assigns a real number to every list. We define:

—The increment of potential of the i-th operation: $\Delta_i = \Phi(L_i) - \Phi(L_{i-1})$.

—The amortized cost a_i of the i-th operation: $a_i = c_i + \Delta_i$.

—The total actual cost: $C = \sum_{i=1}^{r} c_i$.

—The total amortized cost: $A = \sum_{i=1}^{r} a_i$.

An easy calculation shows that $C = A + \Phi(L_0) - \Phi(L_m)$.

Every list update can be separated conceptually into two steps:

(1) one normal split, or one normal merging, or the equal redistribution of the elements of some non-endpoint buckets, or one specialized operation at a bucket near an end, or just nothing;

(2) one element insertion or one element deletion.

Note that the actual cost c_i comes from the first step. Let us denote by $\Delta_i^{(1)}$ and by $\Delta_i^{(2)}$ the increment of potential due to the first step and to the second step, respectively; that is, $\Delta_i = \Delta_i^{(1)} + \Delta_i^{(2)}$. The rest of this section is devoted to define an adequate potential function, and prove with it

(1) $\Delta_i^{(1)} \le 0$ when $c_i = 0$,

(2) $\Delta_i^{(1)} \le -1$ when $c_i = 1$,

(3) and $\Delta_i^{(2)} \le 6/K$.

Altogether, this will imply $a_i \le 6/K$ for every i, and thus will prove un upper bound of $6r/K$ for C.

The potential function. To define the potential function, it is convenient to use the following definitions:

$$|L| = \sum_{b \in L} |b|.$$

$$f(x) = \begin{cases} 6x - 5, & \text{if } x \ge \frac{5}{6}; \\ 5 - 6x, & \text{if } x \le \frac{5}{6}. \end{cases}$$

$$f_1(x) = \begin{cases} 4x - 2, & \text{if } x \ge \frac{1}{2}; \\ 2 - 4x, & \text{if } x \le \frac{1}{2}, \end{cases}$$

$$f_2(x) = \begin{cases} 4x - 4, & \text{if } x \ge 1; \\ 4 - 4x, & \text{if } x \le 1, \end{cases}$$

$$f_3(x) = \begin{cases} 4x - 6, & \text{if } x \ge \frac{3}{2}; \\ 6 - 4x, & \text{if } x \le \frac{3}{2}. \end{cases}$$

Then, we define the potential of a bucket b as follows:

$$\Phi(b) = \begin{cases} 2|b|, & \text{if } b \text{ is an endpoint}; \\ f(|b|), & \text{if } b \text{ is not an endpoint}. \end{cases}$$

Finally, we define the potential of a list as:

$$\Phi(L) = \begin{cases} f_1(|L|), & \text{if } L \text{ consists on only one bucket}; \\ f_2(|L|), & \text{if } L \text{ consists on two buckets}; \\ f_3(|L|), & \text{if } L \text{ consists on three buckets.} \\ \sum_{b \in L} \Phi(b), & \text{if } L \text{ consists on four or more buckets.} \end{cases}$$

Computation of $\Delta_i^{(1)}$ and $\Delta_i^{(2)}$. From the definitions in the previous sections, it is trivial that $\Delta_i^{(2)} \leq 6/K$. Let us now bound $\Delta_i^{(1)}$ for every case.

Large lists

(1) Normal split: $\Delta_i^{(1)} = 4 \cdot f(\frac{3}{4}) - 3 \cdot f(1) = -1$.

(2) Normal merging: Let b_i, b_j, b_k be the initial buckets. Denote $x = |b_i|$, $y = |b_j|$, $z = |b_k|$ and $\varphi = f(x) + f(y) + f(z)$. It holds that $x + y + z = 2$. Further, suppose w.l.o.g. that $x \leq y \leq z$. Then, $\Delta_i^{(1)} = 2 \cdot f(1) - \varphi = 2 - \varphi$. We devise 3 cases according to the values of x, y and z: (a) $x \leq \frac{2}{6}$, $y \geq \frac{5}{6}$, and $z \geq \frac{5}{6}$, (b) $x < \frac{5}{6}$, $y < \frac{5}{6}$, and $z \geq \frac{5}{6}$ and (c) $x < \frac{5}{6}$, $y < \frac{5}{6}$, and $z < \frac{5}{6}$.
For any of them, the reader can easily check that $\varphi \geq 3$, and so, $\Delta_i^{(1)} \leq -1$.

(3) Equal redistribution of elements when inserting on a bucket i, s.t. $|\ell^2(i)| < \frac{1}{2}$ (case 3b in Section 6). Let $x_2 = |\ell^2(i)|$, then: $\Delta_i^{(1)} = 2 \cdot f(\frac{1+x_2}{2}) - (f(1) + f(x_2)) = -2$.

(4) Equal redistribution of elements when inserting on a bucket i, s.t. $|\ell^3(i)| + |\ell^2(i)| < \frac{5}{4}$ (case 3c in Section 6). Let $x_2 = |\ell^2(i)|$ and $x_3 = |\ell^3(i)|$, then: $\Delta_i^{(1)} = 3 \cdot f(\frac{1+x_2+x_3}{3}) - (f(1) + f(x_2) + f(x_3)) = 8 - 6(x_2 + x_3) - (f(x_2) + f(x_3))$.
We devise two cases according the value of x_2: (a) $x_2 \leq \frac{5}{6}$ and (b) $x_2 \geq \frac{5}{6}$. In both cases, the reader can easily check that $\Delta_i^{(1)} \leq -2$.

(5) Removing the first bucket when the second bucket is at $\frac{5}{12}$ or less: Let b_3 be the third bucket and $x = |b_3|$, then: $\Delta_i^{(1)} = 2(\frac{5}{12} - (1-x)) + f(1) - (2 \cdot 0 + f(\frac{5}{12}) + f(x)) = -\frac{8}{3} + 2x - f(x)$. We devise two cases according to the value of x: (a) $x \geq \frac{5}{6}$ and (b) $x \leq \frac{5}{6}$. In any of them, it can be checked that $\Delta_i^{(1)} \leq -1$.

(6) Splitting at the first bucket: $\Delta_i^{(1)} = 2 \cdot 0 + f(1) - (2 \cdot 1) = -1$.

(7) Splitting at the second bucket: $\Delta_i^{(1)} = 2 \cdot \frac{1}{2} + f(1) + 2 \cdot f(\frac{3}{4}) - (2 \cdot 1 + 2 \cdot f(1)) = -1$.

(8) Splitting at the third bucket: Let b_1 be the first bucket, $x = |b_1|$, $y = \min(x, \frac{1}{4})$, and $z = x - y$, then: $\Delta_i^{(1)} = 2z + f(\frac{3}{4} + y) + 3 \cdot f(\frac{3}{4}) - (2x + 3 \cdot f(1)) = -\frac{3}{2} - 2y + f(\frac{3}{4} + y)$.
There are two cases according to the value of y: (a) $y \geq \frac{1}{12}$ and (b) $y \leq \frac{1}{12}$. In both cases, it can be checked that $\Delta_i^{(1)} \leq -1$.

Small lists

(1) Split 1–2: $\Delta_i^{(1)} = f_2(1) - f_1(1) = -2$.

(2) Merging 2–1: $\Delta_i^{(1)} = f_1(\frac{1}{2}) - f_2(\frac{1}{2}) = -2$.

(3) Split 2–3: $\Delta_i^{(1)} = f_3(2) - f_2(2) = -2$.

(4) Merging 3–2: $\Delta_i^{(1)} = f_2(1) - f_3(1) = -2$.

(5) Split 3–4: $\Delta_i^{(1)} = 2(2 \cdot \frac{1}{2}) + 2 \cdot f(1) - f_3(3) = -2$.

(6) Merging 4–3: Let b_2 and b_3 be the middle buckets of the original 4-bucket list. Let $x = |b_2|$ and $y = |b_3|$, where $y = \frac{3}{2} - x$. Then, $\Delta_i^{(1)} = f_3(\frac{3}{2}) - (2(2 \cdot 0) + f(x) + f(y)) = -(f(x) + f(\frac{3}{2} - x))$. Let us assume that $x \geq y$ without loss of

generality. There are two cases: (a) $x \geq \frac{5}{6}$ and (b) $x \leq \frac{5}{6}$. In both of them, it can be checked that $\Delta_i^{(1)} \leq -1$.

(7) Splitting 4–5 at a non-endpoint when all buckets are full: $\Delta_i^{(1)} = 2(2 \cdot \frac{5}{8}) + 2 \cdot f(1) + f(\frac{3}{4}) - (2(2 \cdot 1) + 2 \cdot f(1)) = -1$.

(8) Extra reorganization after splitting 5–6 at the third bucket: Let b_5 be the fifth bucket, $x = |b_5|$, $y = \min(x, \frac{1}{4})$, and $z = x - y$, then: $\Delta_i^{(1)} = 2z + f(\frac{3}{4} + y) - (2x + f(\frac{3}{4})) = -\frac{1}{2} - 2y + f(\frac{3}{4} + y)$.
There are two cases according to the value of y: (a) $y \geq \frac{1}{12}$ and (b) $y \leq \frac{1}{12}$. In any of them, it can be checked that $\Delta_i^{(1)} \leq 0$.

□

Note that the bound $\Delta_i^{(2)} \leq 6/K$ is not necessarily tight. It is left showing whether it is tight or a tighter bound exists.

8. PERFORMANCE EVALUATION

In this section, we experimentally analyze the performance of our implementations and show their competitiveness in a lot of common settings.

We developed three implementations. Two of them use contiguous bucket arrangement, one of which uses *bucket of pairs* iterator solution and another *bucket of pairs*. The third implementation uses a linked bucket arrangement and the *2-level* iterator solution. All these have been attached to the submission. Notice that in contrast to a flat doubly linked list, our operations deal with several cases and each of them with more instructions. This makes our code 3 or 4 times longer (in code lines).

The results are shown for a Sun workstation with Linux and an AMD Opteron CPU at 2.4 GHz, 1 GB main memory, 64 KB + 64 KB 2-associative L1 cache, 1024 KB 16-associative L2 cache and 64 bytes per cache line. The programs were compiled using the GCC 4.0.1 compiler with optimization flag `-O3`. Comparisons were made against the current STL GCC implementation and LEDA 4.0 (in the latter case the compiler was GCC 2.95 for compatibility reasons).

All the experiments were carried with lists of integers considering several list sizes that fit in main memory. Besides, all the plotted measurements are scaled to list size for a better visualization.

With regard to performance measures, we collected wall-clock times, that were repeated enough times to obtain significative averages (variance was always observed to be very low). Furthermore, to get some insight on the behavior of the cache, we used Pin [Luk et al. 2005], a tool for the dynamic instrumentation of programs. Specifically, we have used a Pin tool that simulates and gives statistics of the cache hierarchy (using typical values of the AMD Opteron).

In the following we present the most significant results. Firstly, we analyze the behavior of lists with no iterators involving basic operations and common access patterns. Then, we consider lists with iterators. Finally, we compare our implementations against LEDA, and consider other hardware environments.

8.1 Basic operations with no iterator load

Insertion at the back and at the front. Given an initially empty list, this experiment compares the time to get a list of n elements by successively applying n calls to either the `push_back` or `push_front` methods.

The results for `push_front` are shown in Figure 9(a); a similar behavior was observed for `push_back`. In these operations, we observe that our three implementations perform significantly better than GCC. This must be due to manage memory more efficiently: firstly, the allocator is called only once for all elements in a bucket and not for every element. Secondly, our implementations ensure that buckets get full or almost full in these operations, and so, less total memory space is allocated.

Traversal. Consider the following experiment: First, build a list; then, create an iterator at its begin and advance it up to its end four times. At each step, add the current element to a counter. We measure the time taken by all traversals.

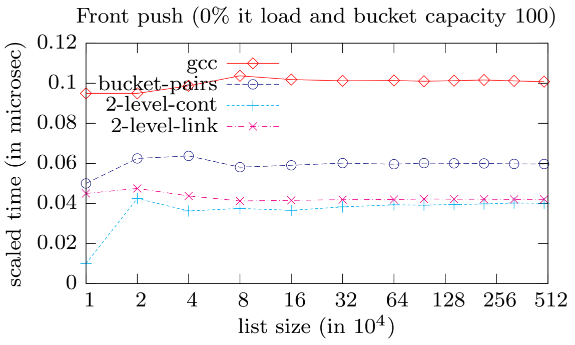
Here, the way to construct the list plays an important role. If we just build the list as in the previous experiment, the traversal times are those summarized in Figure 9(b). These show that performance does not depend on list size and that our 2-level contiguous list implementation is specially efficient even compared to the other 2-level implementation. Our linked bucket implementation is slower than the contiguous implementation because, firstly, its buckets are bigger for the same capacity and so, there are more memory accesses (and misses). Secondly, the increment operation of the linked implementation requires more instructions.

Rather, if we sort this list before doing the traversals, and then measure the time, we obtain the results shown in Figure 9(c). Now, the difference between GCC's implementation and ours becomes very significant and increases with list size (our implementation turns to be more than 5 times faster). Notice also that there is a big slope just beginning at lists with 20000 elements.

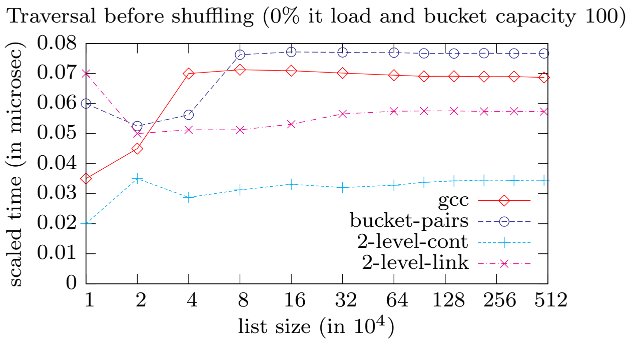
The difference in performance is due to the different physical arrangement of elements in memory (in relation to their logical positions). To prove this claim, we repeated the same experiment using the Pin tool, counting the number of instructions and L1 and L2 cache accesses and misses. Some of these results are given in Figure 12(a). Firstly, these show that indeed our implementations incur in less caches misses (both in L1 and L2). Secondly, the scaled ratio of L1 misses is almost constant because even small lists do not fit in L1. Besides, the big slope in time performance for the GCC implementation coincides with a sudden rise in L2 cache miss ratio, which leads to a state in which almost every access to L2 is a miss. This transition also occurs in our implementations, but much more smoothly. Nevertheless, the L2 access ratio (that is, L1 miss ratio) is much lower because logically close elements are in most cases in the same bucket and so, already in the L1 cache (because bucket capacity is not too big).

Insertion. In this experiment, we deal with insertions at arbitrary points. Firstly, a list is built (using the two abovementioned ways). Then, it is forwardly traversed four times. At each step, with probability $\frac{1}{2}$, an element is inserted before the current. We measure the time of doing the traversal plus the insertions.

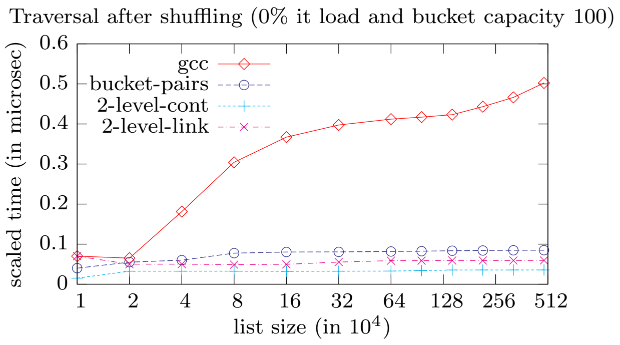
Results are shown in Figures 10(a) and 10(b), whose horizontal axis corresponds



(a) `push_front`

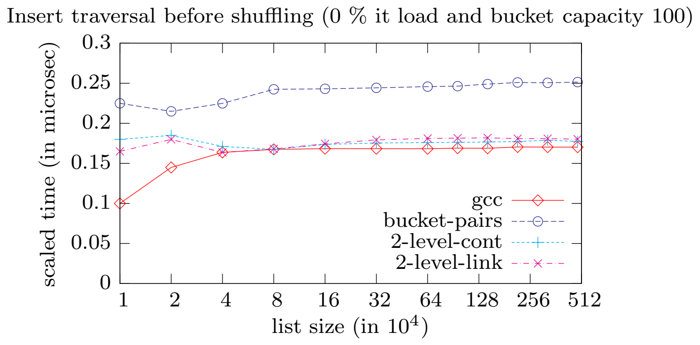


(b) Traversal before shuffling

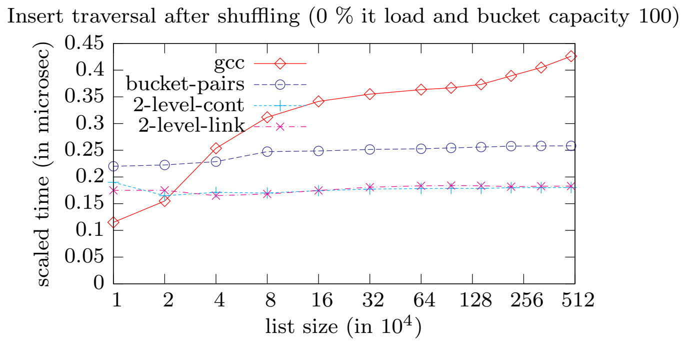


(c) Traversal after shuffling

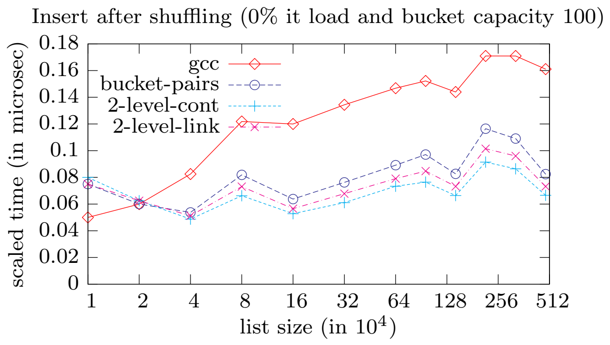
Fig. 9. Performance results for basic operations with no iterator load.



(a) Insertion before shuffling



(b) Insertion after shuffling



(c) Intensive insertion

Fig. 10. Performance results for basic operations with no iterator load.

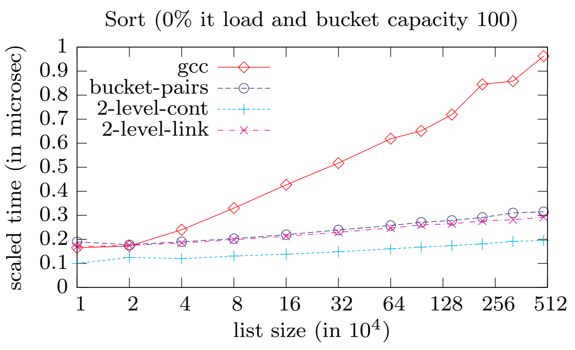


Fig. 11. Performance results for internal sort with no iterator load

to the initial list size.

Analogously to plain traversal, performance depends on the way the list is built. However, as in this case the computation cost is greater, the differences are smoother. In fact, when the list has not been shuffled, the bucket of pairs list performs worse than GCC's. Our two other implementations perform similarly to GCC's though. On the other hand, when the list has been shuffled, GCC's time highly increases, while ours is almost not affected.

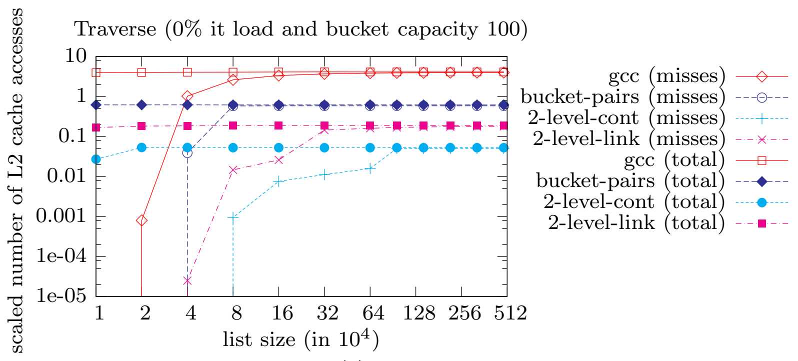
It is interesting to observe that the linked arrangement implementation does not outperform the contiguous ones even though it does not require shifting elements inside the bucket. This must be due to the fact that more memory accesses (and misses) are performed and this is still dominant. This was confirmed performing the same experiment under the Pin environment. As more insertions per element are done, this gain starts to be distinguishable. This is shown in Figure 10(c).

Internal sort. The STL requires an $O(n \log n)$ `sort` method that preserves the iterators on its elements. Our implementations use a customized implementation of merge sort.

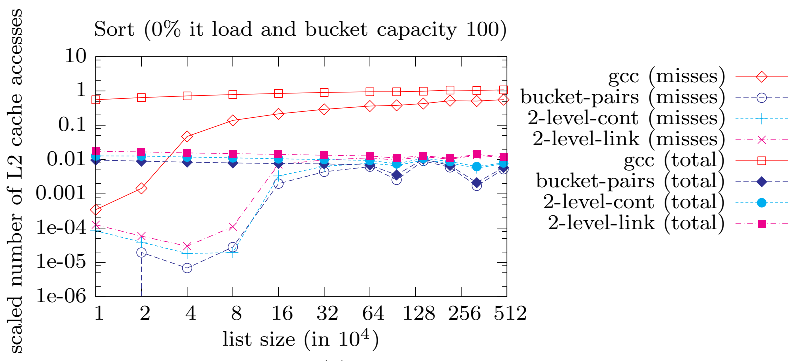
Results of executing the sort method are given in Figure 11. These show that our implementations are between 3 and 4 times faster than GCC. Taking into account that GCC also implements a merge sort, we claim that the significant speedup is due to the locality of data accesses inside the buckets. To confirm this, Figure 12(b) shows the Pin results. Indeed, GCC does about 30 times more cache accesses and misses than our implementations.

Effect of bucket capacity. The previous results were obtained for buckets with capacity of 100 elements. Anyway, this choice did not appear to be critical. Specifically, we repeated the previous tests with lists with other capacities, and observed that once the bucket capacity is not very small (less than 8-12 elements), a wide range of values behaved neatly. Note that a bucket of integers with capacity of 8 elements is already 40-80 bytes long (depending on the implementation and address length) and a typical cache line is 64 or 128 bytes long.

To illustrate the previous claims, we show in Figure 13 insertion results on a shuffled list with initially roughly 5 million elements. These show that for contigu-



(a) Traversal



(b) Internal Sort

Fig. 12. Simulation results on the cache performance (the vertical axis is logarithmic).

ous arrangement implementations, time decreases until a certain point and then starts to increase. In these cases, increasing the bucket size increases the intra-bucket movements which finally results more costly than the achieved locality of accesses. In contrast, the linked arrangement implementation seems to be not affected because no such operations are performed, accesses of a bucket do not interfere between them, and our insert reorganization algorithm takes into account at most three buckets at a time.

If we perform the previous test with several instances at the same time, a smooth rise in time for all implementations can be seen, in particular for big bucket capacities. In fact, it is common dealing with several data structures at the same time. In this case, some interferences within the different objects accesses can occur, which are more probable as the number of instances grows. Therefore, it is advisable to keep a relatively low bucket size.

8.2 Basic operations with iterator load

Now, we repeat the previous experiments on lists that do have iterators on their elements. We use the term *iterator load* to refer to the percentage of elements of a list that have one or more iterator on them. Given how our iterators are

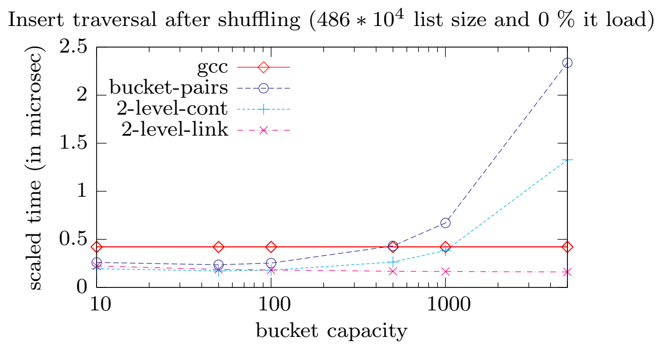


Fig. 13. Experimental results evaluating the effect of bucket capacity: Insertion after shuffling (list size 486000)

implemented, this is what really can affect performance and not the exact number of iterators.

Results are shown for tests in which elements have already been shuffled, iterator loads range from 0% to 100% and a big list size is fixed (about 5 million elements) because then it is crucial to manage data in the cache efficiently.

Traversal. When there are no iterators on the list, our implementations perform a list traversal very fast because the increment operation is simple and elements are accessed with high locality. However, when there are some iterators, it may turn slower because the increment operation depends whether there are other iterators pointing to the element or its successor. In contrast, the increment operation on traditional doubly linked lists is independent of it, and so, performance must be not affected. When the list has not been shuffled, this is exactly the case.

In contrast, when the elements are shuffled, which changes iterators logical order, iterators accesses may score low locality. Results for this case are shown in Figure 14(a), which show indeed that the memory overhead become the most important factor in performance. Nevertheless, the good locality of accesses to the elements themselves makes our implementations more competitive than GCC's up to 80% iterator load even for relatively small lists (about 100000 elements).

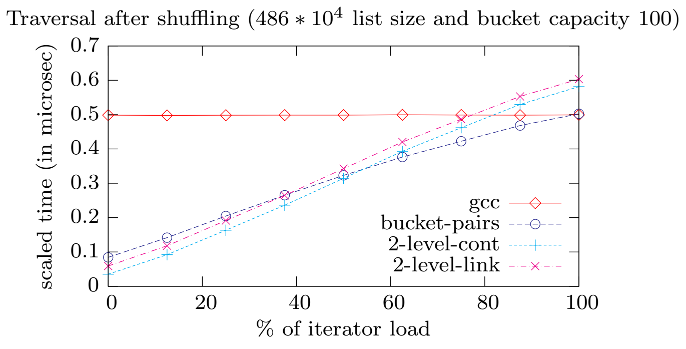
Insertion. When an element is inserted in a bucket with several iterators, some extra operations must be done but are much less than in the traversal case in relative terms. Therefore, performance should be less affected.

Insertion results are shown after the elements have been shuffled in Figure 14(b).

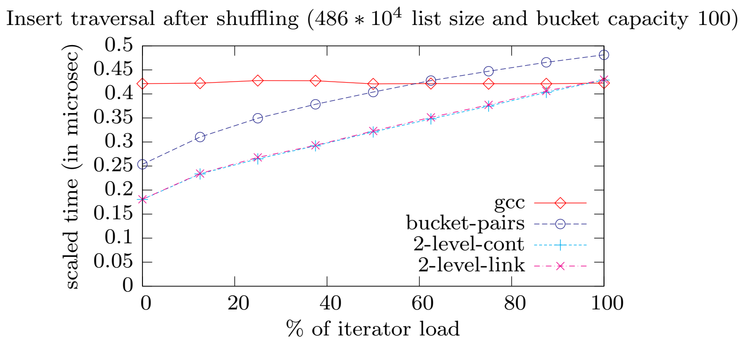
The results are analogous to the traversal test but with smoother slopes, as happened with no iterator load. Specifically, when the list has been shuffled, our implementations are more convenient up to 80% iterator load.

Internal sort. Guaranteeing iterators consistency in our customized merge sort is not straightforward, specially in the case of 2-level approaches that need some (though small) auxiliary arrays. Performance results are shown in Figure 14(c).

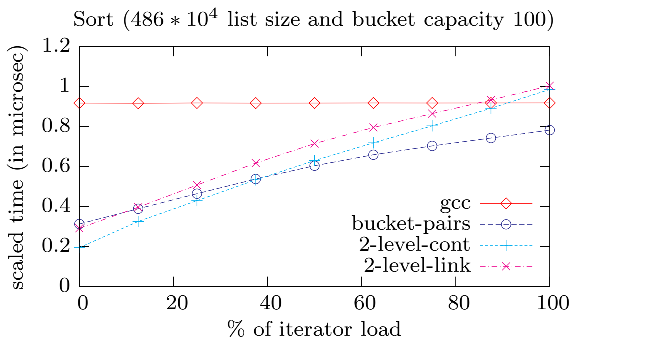
The results indeed show that the 2-level implementations are more sensitive to the iterator load. Anyway, any of our implementation are faster than GCC for



(a) Traversal after shuffling



(b) Insertion after shuffling



(c) Internal sort

Fig. 14. Performance results depending on the iterator load for a list of size $4.86 * 10^6$.

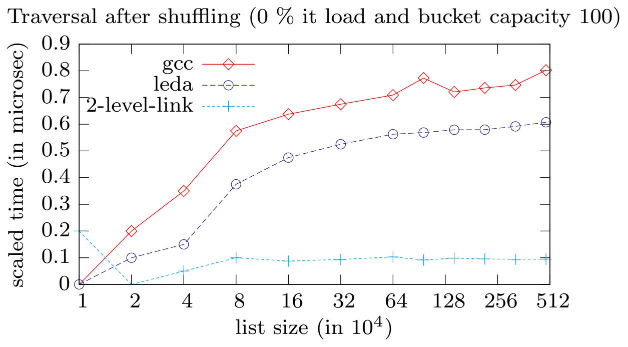


Fig. 15. Performance results for traversal after shuffling against LEDA.

iterators loads lower than 90%.

8.3 Comparison with LEDA

We compare now our lists with the LEDA well-known implementation, which uses a customized memory allocator. Although LEDA does not follow the STL, its interface is very similar and as GCC, it uses classical doubly linked lists.

In Figure 15, we show the results for traversal operation after shuffling. These make evident the limitations in performance of using a doubly linked list compared to our cache conscious approach. LEDA's times are just slightly better than GCC's, but remain worse than our implementations.

We omit the rest of plots with LEDA, because its results are just slightly better than GCC. The only exception is its internal sort (a quicksort) which is very competitive. Nevertheless, it requires linear extra space, does not keep iterators (items in LEDA jargon) and is not faster than ours.

8.4 Other environments

The previous experiments have been run in a AMD Opteron machine. We have verified that the results we claim also hold on other environments. These include an older AMD K6 3D Processor at 450 MHz with a 32 KB + 32 KB L1 cache, 512 KB L2 off-chip (66 MHz) and a Pentium 4 CPU at 3.06 GHz, with a 8KB + 8KB L1 cache and 512 KB L2 cache. On both machines, similar results are obtained in relative terms, and better as newer the machine and the compiler.

9. CONCLUSIONS

In this paper we have presented three cache conscious lists implementations that are compliant with the C++ standard library. Cache conscious lists were studied before but did not cope with library requirements. Indeed, these goals enter in conflict, particularly preserving both constant costs and iterators requirements.

This paper shows that it is possible to combine efficiently and effectively cache consciousness with STL requirements. On the one hand, our reorganization algorithm guarantees a minimum average occupancy of 2/3 and almost maximum occupancy when the list is built adding elements at the endpoints. Besides, we have shown optimal bounds with respect to the number of created and destroyed buckets for some general sequences of operations.

On the other hand, a wide range of experiments has shown that our implementations are useful in many situations. The experiments compare our implementations against doubly linked list implementations such as GCC and LEDA. These show for instance that our lists can offer 5-10 times faster traversals, 3-5 times faster internal sort and even with an (unusual) big load of iterators be still competitive. Besides, in contrast to doubly linked lists, our data structure does not degenerate when the list is shuffled.

Further, the experiments show that the 2-level implementations are specially efficient. In particular, we would recommend using the linked bucket implementation, although its benefits only evince when the modifying operations are really frequent, because it can make more profit of eventually bigger cache lines.

Given that the use of caches is growing in computer architecture (in size and in number) we believe that cache conscious design will be even more important in the future. Therefore, we think that it is time that standard libraries take into account this knowledge. In this sense, this article sets a precedence but there is still a lot of work to do. To begin with, similar techniques could be applied to more complicated data structures. Moreover, current trends indicate that in the near future it will be common to have multi-threaded and multi-core computers. So, we should start thinking how to enhance these features in modern libraries.

REFERENCES

Bender, M., Cole, R., Demaine, E., and Farach-Colton, M. 2002. Scanning and traversing: Maintaining data for traversals in memory hierarchy. In *ESA '02*. 152–164.

Cormen, T. H., Leiserson, C., Rivest, R., and Stein, C. 2001. *Introduction to algorithms*, 2nd ed. The MIT Press, Cambridge.

Demaine, E. 2002. Cache-oblivious algorithms and data structures. In *EEF Summer School on Massive Data Sets*.

Frigo, M., Leiserson, C., Prokop, H., and Ramachandran, S. 1999. Cache-oblivious algorithms. In *FOCS '99*. IEEE Computer Society, 285.

International Standard ISO/IEC 14882. 1998. *Programming languages — C++*, 1st ed. American National Standard Institute.

Josuttis, N. 1999. *The C++ Standard Library : A Tutorial and Reference*. Addison-Wesley.

Lamarca, A. 1996. Caches and algorithms. Ph.D. thesis, University of Washington.

Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI '05*. Chicago, IL.

Mehlhorn, K. and Naher, S. 1999. *LEDA — A platform for combinatorial and geometric computing*. Cambridge University Press.

Sedgewick, R. 1998. *Algorithms in C++*, 3 ed. Addison-Wesley.

Sen, S. and Chatterjee, S. 2000. Towards a theory of cache-efficient algorithms. In *SODA '00*. SIAM, 829–838.

Weiss, M. A. 1998. *Data Structures and Algorithm Analysis in C++*. Addison-Wesley.

Received Month Year; revised Month Year; accepted Month Year