Lists Revisited: Cache Conscious STL Lists

Leonor Frias*, Jordi Petit**, and Salvador Roura***

Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Campus Nord, edifici Omega
08034 Barcelona (Spain).
{lfrias,jpetit,roura}@lsi.upc.edu

Abstract. We present three cache conscious implementations of STL standard compliant lists. Up to now, one could either find simple double 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, we show the competitiveness of our implementations with an extensive experimental analysis. This shows, for instance, 5-10 times faster traversals and 3-5 times faster internal sort.

1 Introduction

The Standard Template Library (STL) is the algorithmic core of the C++ standard library. The STL is made up of containers, iterators and algorithms. Containers consist on basic data structures such as lists, vectors, maps or sets. Iterators are a kind of high-level pointers used to access and traverse the elements in a container. Algorithms are basic operations such as sort, reverse or find. The C++ standard library [6] specifies the functionality of these objects and algorithms, and also their temporal and spatial efficiency, using asymptotical 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 (e.g. [2]). In fact, the design of current widely used STL implementations (including SGI, GCC, VC++, ...) is based on these.

Nevertheless, the performance of some data structures can be improved taking advantage of the underlying memory hierarchy of modern computers. Not in vain, in the last years the algorithmic community has realized that the old unitary cost memory model is turning more 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, mainly studied under the so-called cache aware (see e.g. [8,11]) and cache oblivious models (see e.g. [3,4]).

However, if these data structures are to be part of a standard software library, the design is constrained by its requirements. As far as we know, no work or implementation has taken this into account. Our aim in this paper is to propose standard compliant alternatives that perform better than traditional implementations in most common settings. Specifically, we have analyzed one of the most simple but essential objects in the STL: lists. We have implemented and experimentally

* This author has been supported by grant number 2005FI 00856 of the *Agència de Gestió d'Ajuts Universitaris i de Recerca.*

** This author has been supported by GRAMARS project under grant TIN2004-07925-C03-01.

*** This author has been suppoted by AEDRI-II project under grant MCYT TIC2002-00190.

evaluated three different variants of cache conscious lists supporting fully standard iterator functionality. The diverse set of experiments shows that great speedups can be obtained compared to traditional double linked lists found for instance in the GCC STL implementation [5] and in the LEDA library [10].

The remainder of the paper is organized as follows: In Section 2, we briefly describe STL lists and the behavior of traditional double linked lists with respect to the cache. The observations drawn there motivate the design of three cache conscious implementations for STL lists that we present in Section 3, relating them to previous work. Our implementations are experimentally analyzed in Section 4. Conclusions are given in Section 5.

2 Motivation for cache conscious STL lists

A list in the STL library is a generic sequential container that supports forward and backward traversal using iterators, as well as single insertion and deletion at any iterator position in O(1) time. Additionally, it offers internal sorting, several splice operations, and other utilities.

Iterators behave as high-level pointers. In the case of lists, the Standard states that no operation can invalidate them, that is, they must point to the same element after any operation has been applied (except if the element is deleted). Besides, the number of iterators per list is not bounded. See further documentation on STL lists in [7].

In order to fulfill the requirements of the STL, a classical double linked list together with pointers for iterators suffices. Indeed, this is what all known STL implementations do.

The key property of this and any pointer-based data structure is that the physical position of each element is permanent and independent of its logical position: when an element changes its logical position, it suffices to modify the pointers in the data structure. Consequently, iterators are not affected by these movements. On the other hand, good cache performance is achieved when logically consecutive elements are at physically nearby locations.

Pointer-based data structures, such as linked lists, strongly rely on memory allocators to get and free nodes.These allocators typically answer consecutive memory requests with consecutive addresses of memory (whenever possible). Therefore, if we add elements at the end of a list (and no other allocations are performed at the same time), there is a good chance that logically consecutive elements are also physically consecutive. Consequently, a traversal of this list will incur in a few number of cache misses. However, if elements are inserted at random points or if the list is shuffled, a traversal may incur in a cache miss per access, thus increasing dramatically its total time.

In order to give evidence of the above statement, we have performed the following experiment with the GCC list implementation: First, a list with n random integers is created by pushing them back one by one, and afterwards we measure the time to fully traverse the list. Then, we modify the list, and again we measure the time to fully traverse the resulting list. The modification consists either on sorting (thus randomly shuffling the links between nodes), or on k iterations of the so-called *k-insertion-erase* test: In the i-th iteration of this test ($1 \leq i \leq k$): first, the list is traversed and an element is inserted at each position with probability $1/(3+i)$, then the list is traversed again and each element is erased with probability $1/(4+i)$. Traversal times before modifying the list and after each kind of modification 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). Besides, note that four iterations of the insertion-erase test are enough to register half the worst case time.

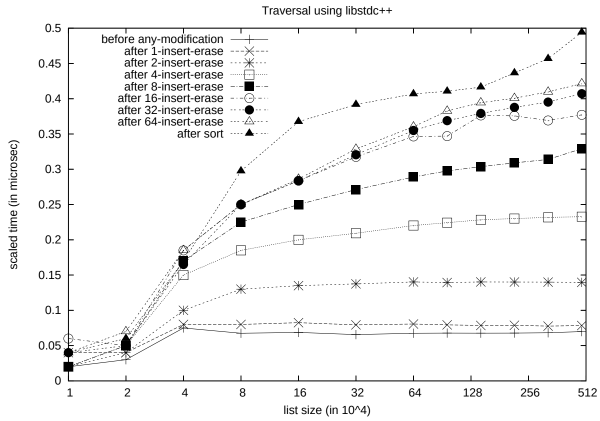


Fig. 1: Time measurements for list traversal before modifying it and after being modified in several ways. The vertical axis is scaled to the list size (that is, time has been divided by the list size before being plotted).

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

3 Design of cache conscious STL lists

In this section we first consider previous work on cache conscious lists. Then, we present the main issues on combining them with STL list requirements. Finally, we present our proposed solutions.

3.1 Previous work

Cache conscious lists have already been analyzed before; see a good summary in [3]. The operations taken into account are traversal (as a whole), insertion and deletion and their cost measured as the number of memory transfers.

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 $\mathrm{O}(n/B)$ amortized traversal cost and constant update cost. The *cache oblivious* solutions are based on the packed memory structure [1], basically an array of $\Theta(n)$ size with uniformly distributed gaps. To guarantee this uniformity updates require $\mathrm{O}((\log^2 n)/B)$, which can be slightly lowered by partitioning the array in smaller arrays. Also, *self-organizing structures* [1] enable to get the same bounds of the cache aware solution, but with amortized cost. There, updates breaking the uniformity are allowed until the list is reorganized when traversed.

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

3.2 Preliminaries

Before proceeding to the actual design, the main problems to be addressed must be identified. In our case, these concern to iterators. Secondly, it may be useful to determine common scenarios in which lists appear to guide the design.

Iterators concerns. In cache conscious data structures, the physical location of an element is strongly related to its logical position. In the case of STL list, this firstly avoids trivially implementing iterators with pointers. Secondly, it enforces that iterators must be reachable to keep them coherent whenever a modification in the list occurs.

The main problem is that there can be an unlimited number of iterators pointing to the same element. Therefore, $\Theta(1)$ modifying operations can be guaranteed only if the number of iterators is arbitrarily restricted, or if iterators pointing to the same element share some data that is updated when a modification occurs.

Hypotheses on common list usages. From our experience as STL programmers, it can be stated that a lot of common list usages are in keeping with the following:

- A particular list instance has typically only a few iterators on it.
- Given that lists are based on sequential access, a lot of traversal operations are expected.
- The elements in the list will often change via insertions, deletions and movements at any position.
- The stored elements are not very big (e.g. integers, doubles, ...).

Note that the last hypothesis, which also appears implicitly or explicitly in general cache conscious data structures literature, can be checked in compile time. In case it did not hold, a traditional implementation could be used instead and this can be neatly achieved with template specialization.

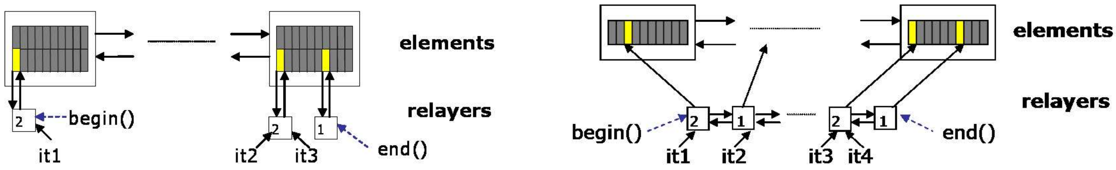
3.3 Our design

Our design combines cache efficient access to data with fully iterator functionality and (constant) worst case costs compliant with the Standard. Furthermore, our approach is specially convenient when the hypotheses on common list usages hold.

The core of our data structures is inspired by the cache aware solution previously mentioned (note that self-organizing strategies are not convenient here because STL-lists are not traversed as a whole but step by step via iterators). Specifically, it is a double linked list of *buckets*. A bucket contains a small array of elements, whose size in elements will be referred as *bucket capacity*, pointers to the next and previous buckets, and extra fields to manage the data in the array. This data structure ensures locality inside the bucket, but logically consecutive buckets are let to be physically far.

To completely define the data structure we must decide a) how to arrange the elements inside a bucket, b) how to reorganize the buckets when inserting or deleting elements, and c) how to manage iterators. Besides, the appropriate bucket capacity must be fixed (this has been studied experimentally).

a) *Arrangement of elements.* We devise three possible ways to arrange the elements inside a bucket:
   - *Contiguous*: The elements are stored contiguously from left to right. In this case, insertions and deletions may require shifting as many elements as the bucket stores.
   - *With gaps*: Elements are still stored from left to right but gaps between elements are allowed. In this way, we expect to reduce the average number of shifts. However, an extra field per element is needed to distinguish real elements from gaps. Additionally, more computation may be needed.
   - *Linked*: The order of the elements inside the bucket is set by internal links instead of the implicit left-to-right order. This requires more extra space for the links, but avoids shifts inside the bucket. Thus, this solution is scalable for large bucket (and cache line) sizes.



(a) Bucket of pairs

(b) 2-level list

Fig. 2: Standard compliant iterators policies.

b) *Reorganization of buckets.* The algorithms involved in the reorganization of buckets preserve the data structure invariant after an element has been inserted or deleted. This includes among others keeping a minimum bucket occupancy load to guarantee the locality of accesses and keeping the arrangement coherency (for example, if contiguous arrangement is used, gaps between the elements cannot be created).
The main issue is keeping a good balance between high occupancy, few bucket accesses per operation, and few elements movements. Besides, it must be guaranteed that no infinite sequence of operations in a fixed position, can create a bucket to just destroy it afterwards.

c) *Iterator management.* Finally, it must be decided how iterators are implemented. Recall from Section 3.2 that this cannot be done trivially with pointers.
Specifically, we have decided to identify all the iterators over an element with a dynamic node (relayer) that points to the element. In this way, we only need to update it when the physical location of its element changes. Of course, each relayer must be found in constant time. We propose two possible solutions (see Figure 2):
   - *Bucket of pairs*: In this solution, for each element, we keep a pointer to its relayer. This is easy to do and still uses less space than a traditional double linked list because it needs two pointers per element.
   - *2-level*: In this solution, we maintain a double linked list of active relayers. In this case, constant time access to the relayers can be guaranteed because STL lists are always accessed through iterators. This solution uses less space compared to the previous one (if there are not much iterators).

   Unfortunately, the locality of iterator accesses decreases with the number of elements with iterators, because relayers can be anywhere in the memory. However, when a list has just a few iterators on it, this is not a big matter because in particular, there is a good chance to find them in cache memory. In any case, our two approaches are standard compliant whatever the number of iterators.

4 Performance evaluation

We developed three implementations. These can be found under http://www.lsi.upc.edu/∼lfrias/lists/lists.zip. Notice that in contrast to a flat double-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).

Specifically, two of our implementations use the contiguous bucket arrangement, one for each of the two standard iterator solutions (*bucket of pairs* and *2-level*). The third implementation uses a linked bucket arrangement and the *2-level* iterator solution.

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

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 [9], 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 of our tests. Firstly, we analyze the behavior of various tests involving basic operations and common access patterns on lists with no iterators. Then, we consider lists with iterators. Finally, we compare our implementations against LEDA, and consider other hardware environments.

4.1 Basic operations with no iterator load

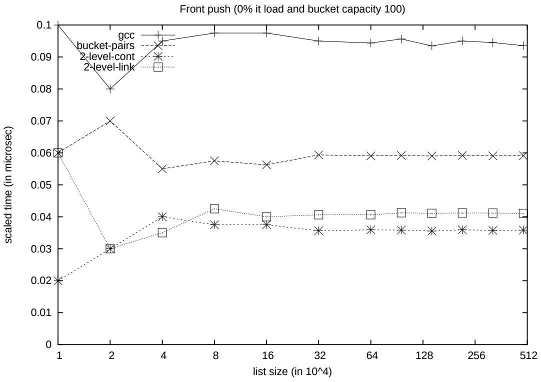
Insertion at the back and at the front. Our first experiment compares the time to construct a list by doing n insertions at its back or at its front. This is accomplished by successively applying n calls to the `push_back` or `push_front` methods on an initially empty list.

The results for `push_front` are shown in Figure 3(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 the buckets get full or almost full in these operations, and so, less total memory space is allocated.

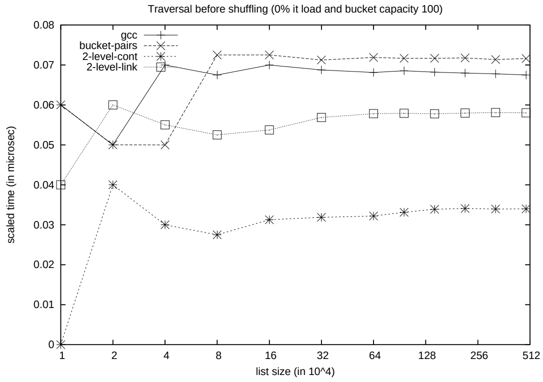
Traversal. Consider the following experiment: First, build a list; then, create an iterator at its begin and advance it up to its end four times. At each step, add the current element to a counter. We measure the time taken by all traversals.

Here, the way to construct the list plays an important role. If we just create a list as in the previous experiment, and then measure the time, we obtain the results given in Figure 3(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 a few more instructions.

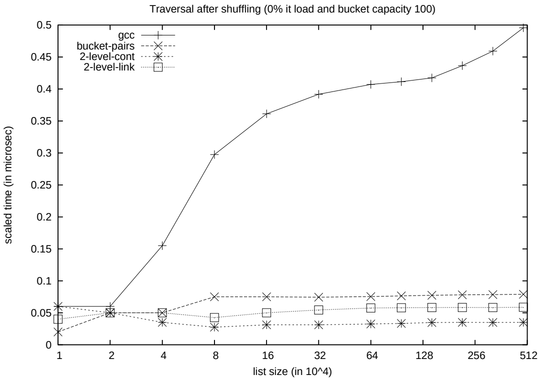
Rather, if we sort this list before doing the traversals, and then measure the time, we obtain the results shown in Figure 3(c). Now, the difference between GCC's implementation and ours becomes



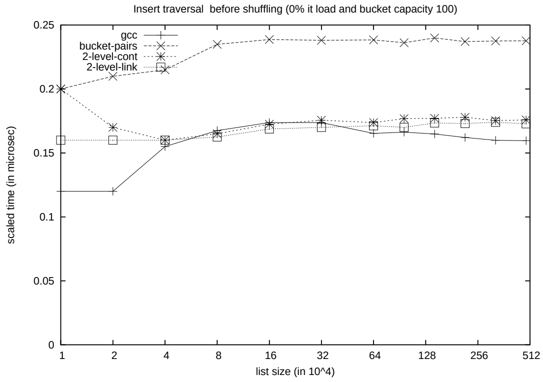
(a) `push_front`



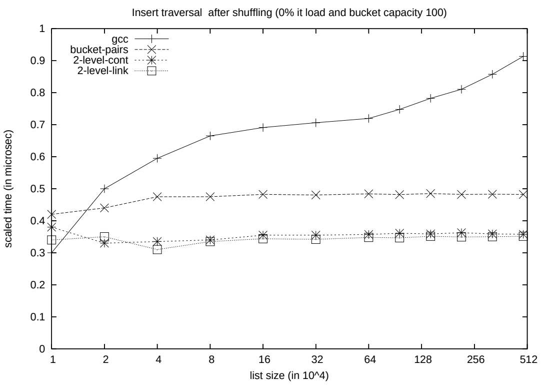
(b) Traversal before shuffling



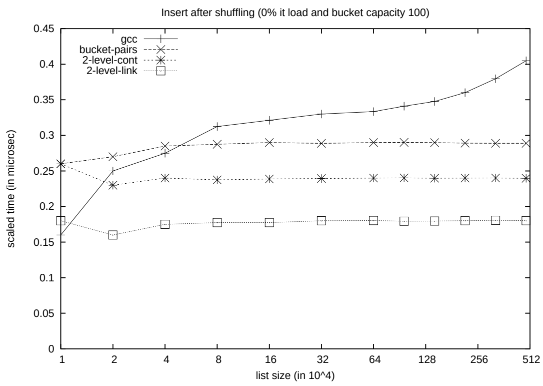
(c) Traversal after shuffling



(d) Insertion before shuffling

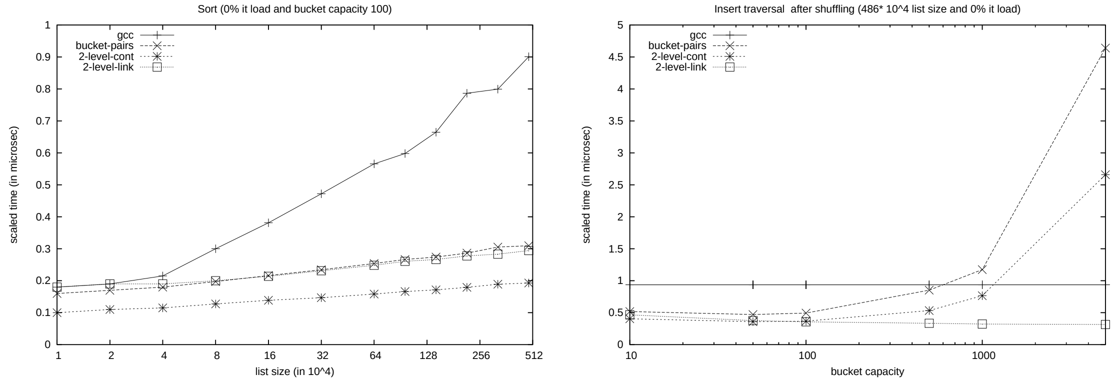


(e) Insertion after shuffling



(f) Intensive insertion

Fig. 3: Experimental results for basic operations with no iterator load.



(a) Internal sort

(b) Effect of bucket capacity for a list with size 486000. Insertion after shuffling

Fig. 4: Experimental results for basic operations with no iterator load.

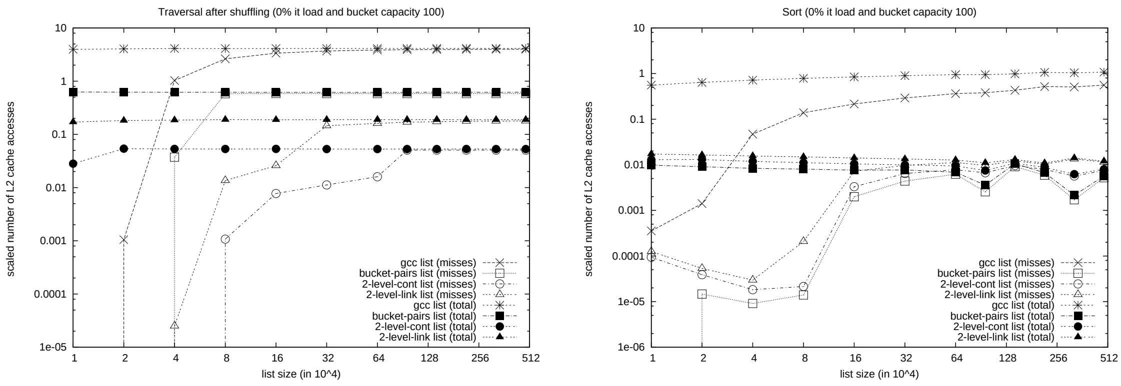
very significant and increases with the list size (our implementation turns to be more than 5 times faster). Notice also that there is a big slope just beginning at lists with 20000 elements.

This difference in performance is caused by the different physical arrangement of elements in memory (in relation to their logical positions). In order to prove this claim, we repeated the same experiment using the Pin tool, which counted the number of instructions and the number of L1 and L2 cache accesses and misses. Some of the results are given in Figure 5(a) (be aware that the vertical axis is logarithmic). 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 order to test insertions at arbitrary points of a list, we designed the following experiment: A list is created (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 one. We measure the time of doing the traversal plus the insertions.

Results are shown in Figures 3(d) and 3(e), whose horizontal axis corresponds to the initial list size. Similar results were obtained with the erase operation.

Analogously to plain traversal, performance depends on the way the list is created. However, as in this case the computation cost is greater, the differences are smoother and our implementations, although still faster, are not so much better. Indeed, 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.



(a) Traversal

(b) Internal Sort

Fig. 5: Simulation results on the cache performance. The vertical axis is logarithmic.

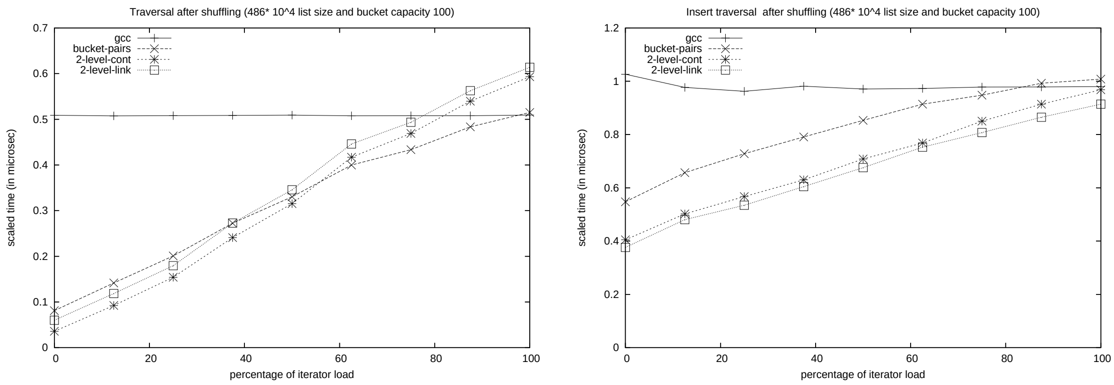
It is interesting to observe that the linked arrangement implementation does not outperform the contiguous ones even though it does not require shifting elements inside the bucket. This must be due to the fact more memory accesses (and misses) are performed and this is still dominant. This was confirmed performing the analogous Pin experiment. Instead, if an intensive insertion test is performed, in which a lot of insertions per element are done and almost no traversal is performed, then this gain is not negligible. This is shown in Figure 3(f).

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 4(a). Our implementations are between 3 and 4 times faster. Taking into account that GCC's method is also a merge sort, we claim that the significant speedup is due to the locality of accesses to elements inside the buckets. To confirm this, Figure 5(b) shows the Pin results. Indeed, GCC does about 30 times more cache accesses and misses than our implementations (recall that the vertical axis is logarithmic).

Effect of bucket capacity. All the previous results were obtained for buckets with capacity of 100 elements. Anyway, this choice was observed to be not critical in our implementations. Specifically, we repeated the previous tests with lists with different capacities, and observed that once the bucket capacity was not very small (we consider small less than 8-10 elements), a wide range of values behaved neatly.

To give an example on how bucket capacity affects performance, we show in Figure 4(b) the insertion operation on lists of about 5 million elements after shuffling. We can observe that for the contiguous arrangement implementations, time decreases until a certain point and then starts to increase. In these cases, increasing the bucket size increases the intrabucket 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.



(a) Traversal after shuffling

(b) Insertion after shuffling

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

If we perform the last 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.

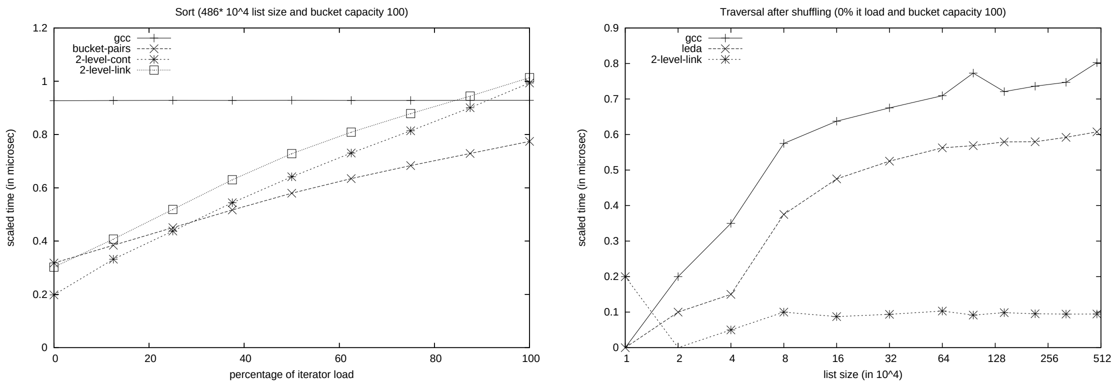
4.2 Basic operations with iterator load

We now repeat the previous experiments on lists that do have iterators on their elements. To do so, we define the iterator load as the percentage of elements of a list that have one or more iterator on them. Results are shown for tests in which elements have already been shuffled, iterator loads range from 0% to 100% and the list size is fixed to about 5 million elements (a big list) because then is crucial making a cache efficient usage of memory.

Traversal. When there are no iterators on the list, our implementations traversal is very fast because the increment operation is simple and elements are accessed with high locality. However, when there are iterators on the list, our implementations may turn slower because the increment operation depends whether there are more iterator pointing to this element or to its successor. In contrast, the increment operation on traditional double linked lists is independent of this, 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, also the logical order of iterators is shuffled, and so, iterators accesses may score low locality. Results for this case are shown in Figure 6(a). These show indeed that the memory overhead become the most important factor in performance. Nevertheless in that case, the good locality of accesses to the elements themselves makes our implementations more competitive than GCC's up to a 80% of iterator load from relatively small list sizes (about 100000 elements).

Insertion. In the case of the insertion operation with several iterators on the bucket, some extra operations must be done but are much less in relative terms than in the increment operation. Therefore, performance should be less affected.



(a) Internal sort (iterators)

(b) Traversal after shuffling (LEDA)

Fig. 7: Experimental results depending on the iterator load for a list of size $4.86 * 10^6$ and LEDA results.

Results are shown after the elements have been shuffled in Figure 6(b).

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

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 7(a).

The results indeed show that the 2-level implementations are more sensitive to the iterator load because more computation is required by each iterator. Anyway, any of our implementation are faster than GCC for iterators loads lower than 90% for big list sizes.

4.3 Comparison with LEDA

In order to compare our lists with some other well-known implementation, we have selected LEDA. Even though LEDA does not follow the STL, its interface is very similar. As GCC, LEDA uses classical double linked lists, but it also uses a customized memory allocator.

In Figure 7(b), we show the results for traversal operation after shuffling. These make evident the limitations in performance of using a double 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.

4.4 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 compiler.

5 Conclusions

In this paper we have presented three specific implementations of cache conscious lists that are compliant with the C++ standard library. Cache conscious lists have been studied before but they did not cope with library requirements. Indeed, these objectives enter in conflict, specially if both constant costs and iterators requirements are to be preserved.

This paper shows that is possible to combine efficiently and effectively cache consciousness with STL requirements. Furthermore, our implementations are useful in many situations, as is shown by our wide range of experiments. The experiments compare our implementations against double 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. Furthermore, in contrast with the classical double linked lists, our data structure does not degenerate when the list is shuffled.

Finally, from the experiments we can also conclude 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. Therefore, another line of research could make profit of these new features in modern libraries.

References

1. M. Bender, R. Cole, E. Demaine, and M. Farach-Colton. Scanning and traversing: Maintaining data for traversals in memory hierarchy. In *ESA '02*, pages 152–164, 2002.
2. T. H. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to algorithms.* The MIT Press, Cambridge, 2 edition, 2001.
3. E. Demaine. Cache-oblivious algorithms and data structures. In *EEF Summer School on Massive Data Sets*, LNCS. 2002.
4. M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS '99*, page 285. IEEE Computer Society, 1999.
5. GNU GCC. GCC C++ Standard Library.
6. International Standard ISO/IEC 14882. *Programming languages — C++.* American National Standard Institute, 1st edition, 1998.
7. N. Josuttis. *The C++ Standard Library : A Tutorial and Reference.* Addison-Wesley, 1999.
8. A. Lamarca. *Caches and algorithms.* PhD thesis, University of Washington, 1996.
9. C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI '05*, Chicago, IL, June 2005.
10. K. Mehlhorn and S. Naher. *LEDA — A platform for combinatorial and geometric computing.* Cambridge University Press, 1999.
11. S. Sen and S. Chatterjee. Towards a theory of cache-efficient algorithms. In *SODA '00*, pages 829–838. SIAM, 2000.