Lists Revisited: Cache Conscious STL lists

Leonor Frias, Jordi Petit, Salvador Roura

Departament de Llenguatges i Sistemes Informàtics.
Universitat Politècnica de Catalunya.

Overview

Goal: Improve STL lists perfomance in most common settings using a cache-conscious data structure.

Previous work: Either

- *double-linked lists* implementations: easily cope with standard requirements
- *theoretical cache-conscious data structures*: do not take into account any of these requirements

Main contribution: merging both approaches.

Main problem: dealing with STL `lists` iterator functionality.

Work done: analysis, design, implementation and comprehensive experimental study.

Index

Standard Template Library (STL)

Core of C++ standard library [International Standard ISO/IEC 14882 1998].

Elements:

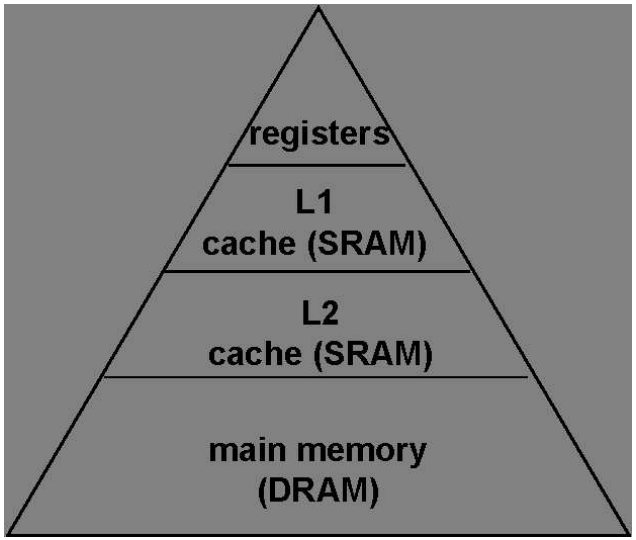
- containers: `list`, `vector`, `map`...
- iterators: high-level pointers
- algorithms: `sort`, `reverse`, `find`...

Implementation: classical literature on algorithms and data structures.

Improve performance

Use memory hierarchy effectively for known / regular access patterns
$\rightarrow$ *cache-conscious* algorithms & data structures
General idea: organize data s.t. logical access pattern
$\approx$ physical memory locations.
Models:

- cache-aware
- cache-oblivious [Frigo et al. 1999]



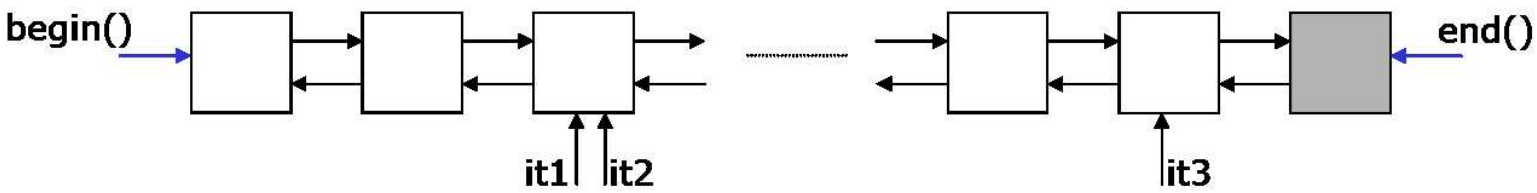
STL lists

Forward and backward traversal container, that supports insertion and deletion in constant time.

STL `list` *iterators properties*:

- arbitrary number
- operations cannot invalidate them

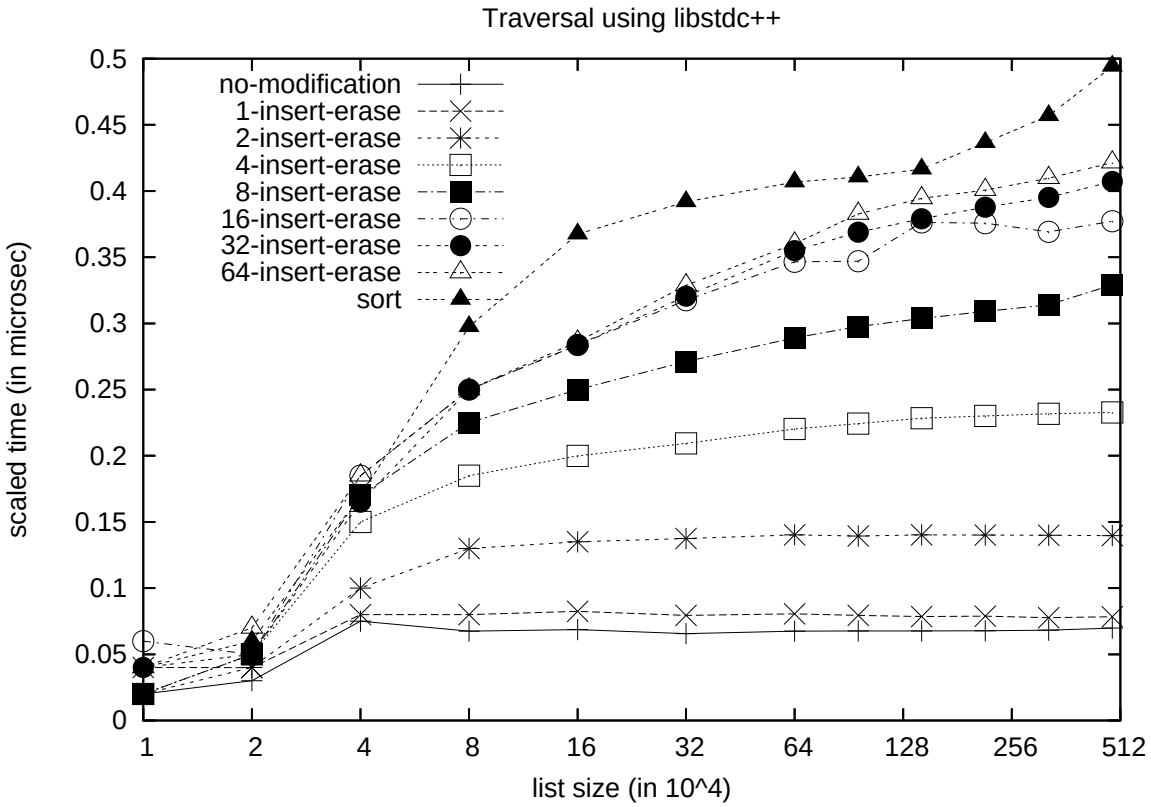
Straightforward implementation:



This is what all known STL implementations do!

Double-linked lists cache performance

Pointer-based data structures *cannot guarantee* good cache performance.



It is worth trying a cache-conscious approach!

Index

Previous work on cache-conscious lists

[Demaine 2002]

Cache-aware: partition of $\Theta(n/B)$ pieces with $(B/2, B)$ elems.

- Traversal: $O(n/B)$ amortized
- Update: constant

Cache-oblivious: uses the packed memory structure,
array of $\Theta(n)$ size with uniformly distributed gaps.

- Traversal: $O(n/B)$ amortized
- Update: $O((\log^2 n)/B)$ (lower by partitioning the array)
  - Amortized constant with *self-organizing structures* (updates may break the uniformity until the list is reorganized when traversed).

Problem

Pointers + cache-conscious data structure:
physical/logical location are not independent.

No trivial pointers $\Rightarrow$ reach iterators whenever a modification occurs.

Main issue: unbounded number of iterators pointing to the same element.

Achieving $\Theta(1)$ *operations:*

- number of iterators arbitrarily restricted
- iterators pointing to the same element share some data

STL `lists` are not traversed as a whole but step by step $\Rightarrow$ NO self-organizing strategies.

Our approach

Efficient *data access* + full *iterator* functionality +
(constant) worst case costs *compliant* with the Standard

Base: cache-aware solution.

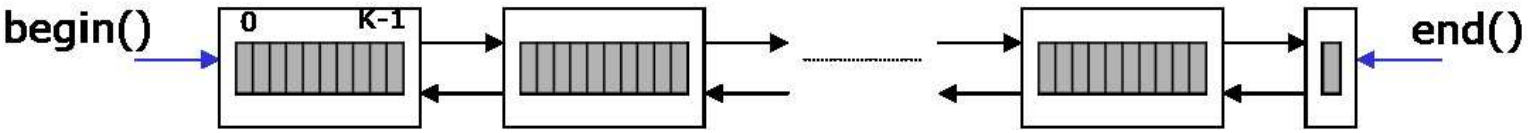
Common list usages:

- Only a few iterators on a list instance
- Many traversals are performed due to sequential access
- Frequent modifications at any position
- Small/Plain old data (POD) types
  (*)Implicit or explicit in general cache-conscious literature

Index

Basic design

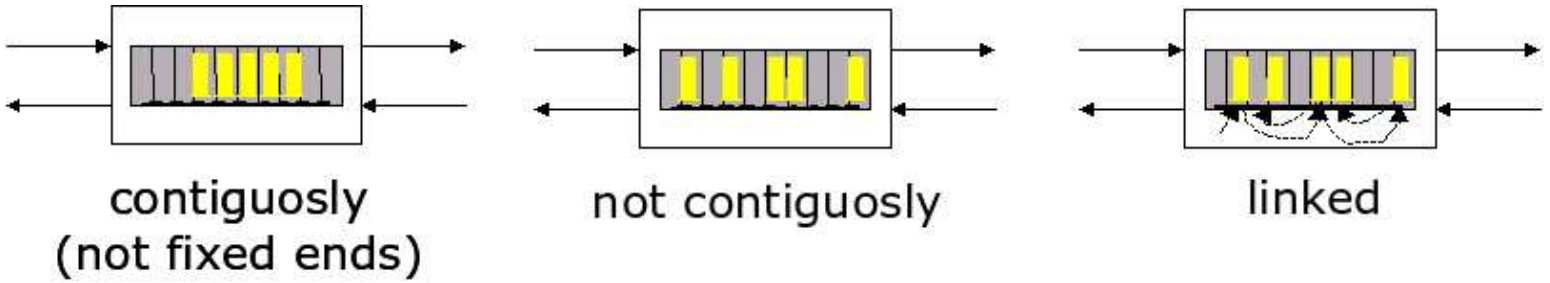
Double-linked list of *buckets*.



What more?

1. how to arrange the elements inside a bucket
2. how to reorganize the buckets on insertion/deletion
3. how to manage iterators
4. bucket capacity? → Experimentally

Arrangement of elements



Reorganization of buckets

Preserve data structure invariant after modification

- minimum bucket occupancy
- arrangement coherency
- ...

Main issue: Keeping balance between:

- high occupancy
- few bucket accessed
- few elements movements

Iterators management

Key idea: all the iterators referred to an element are identified with a dynamic node (*relayer*) that points to it.

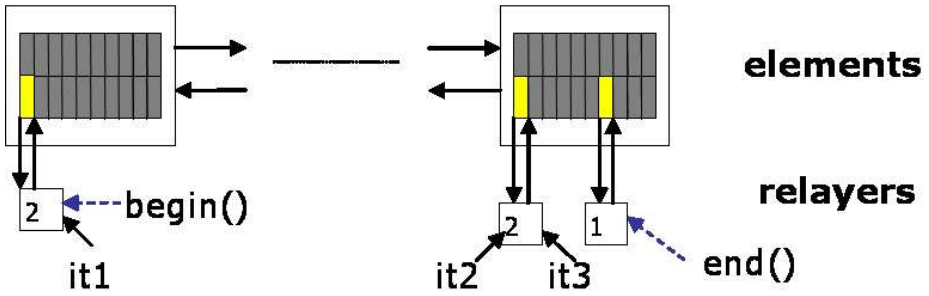


Figure 1: Bucket of pairs

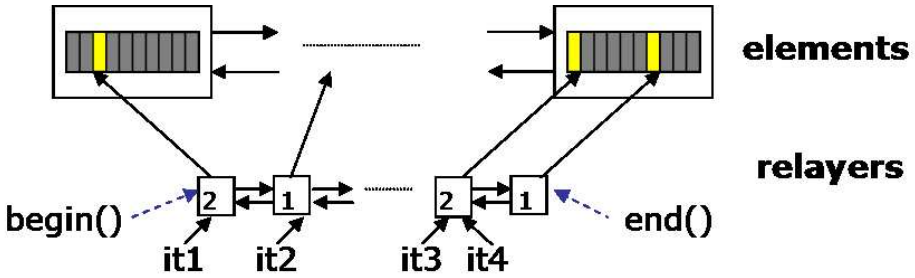


Figure 2: 2-level-list

Index

Set up

Our *three implementations*:

- bucket-pairs
- 2-level-cont
- 2-level-link

against libstdC++ in GCC 4.01.

Basic environment:

- 64-bit Sun Workstation, 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.

Other: Pentium 4, 3.06 GHz hyperthreading CPU, 900 Mb of main memory and 512 Kb L2 cache.

Which experiments

Performance measures:

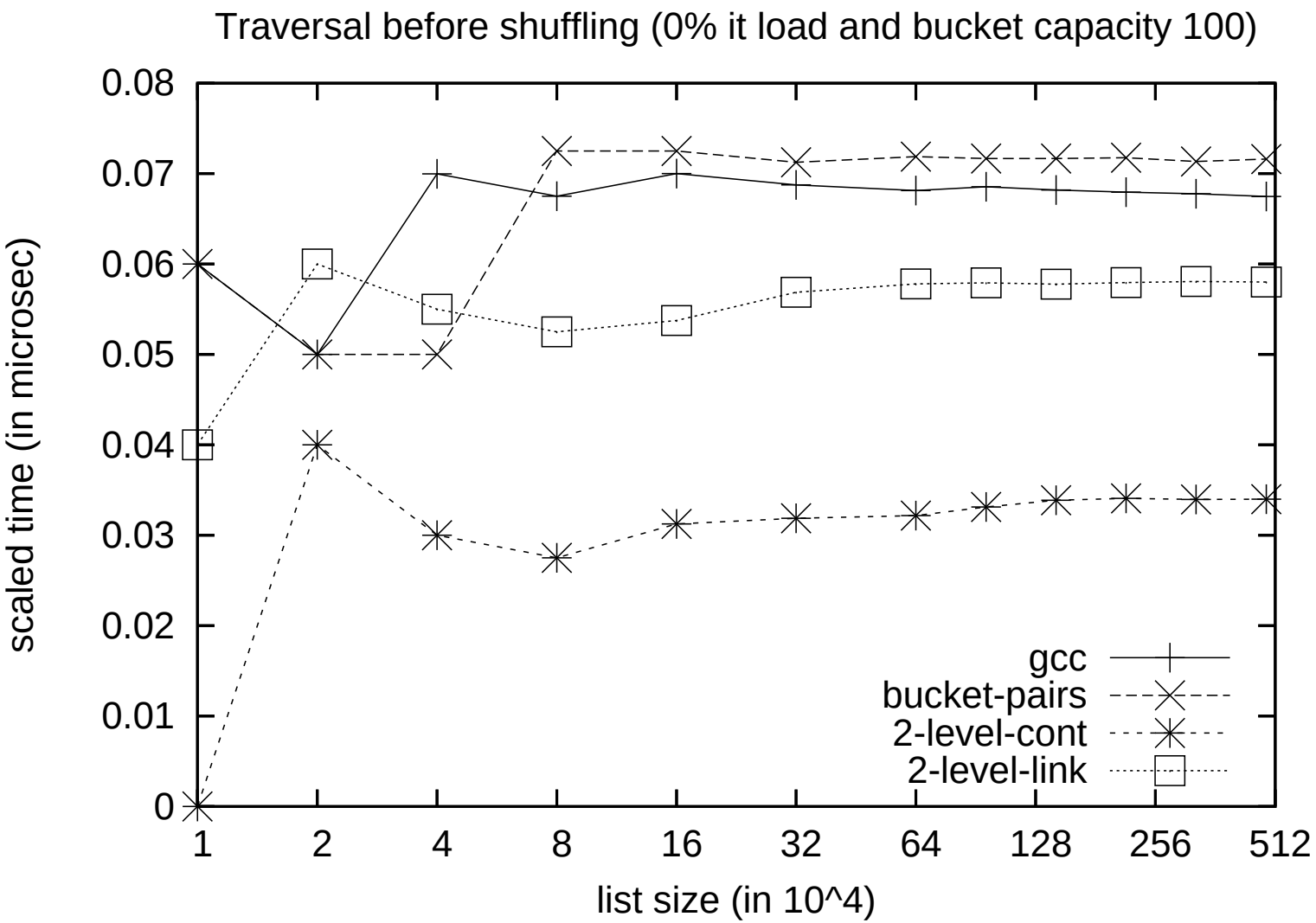
- wall-clock times
- cache performance data: Pin [Luk et al. 2005]

Types of experiments:

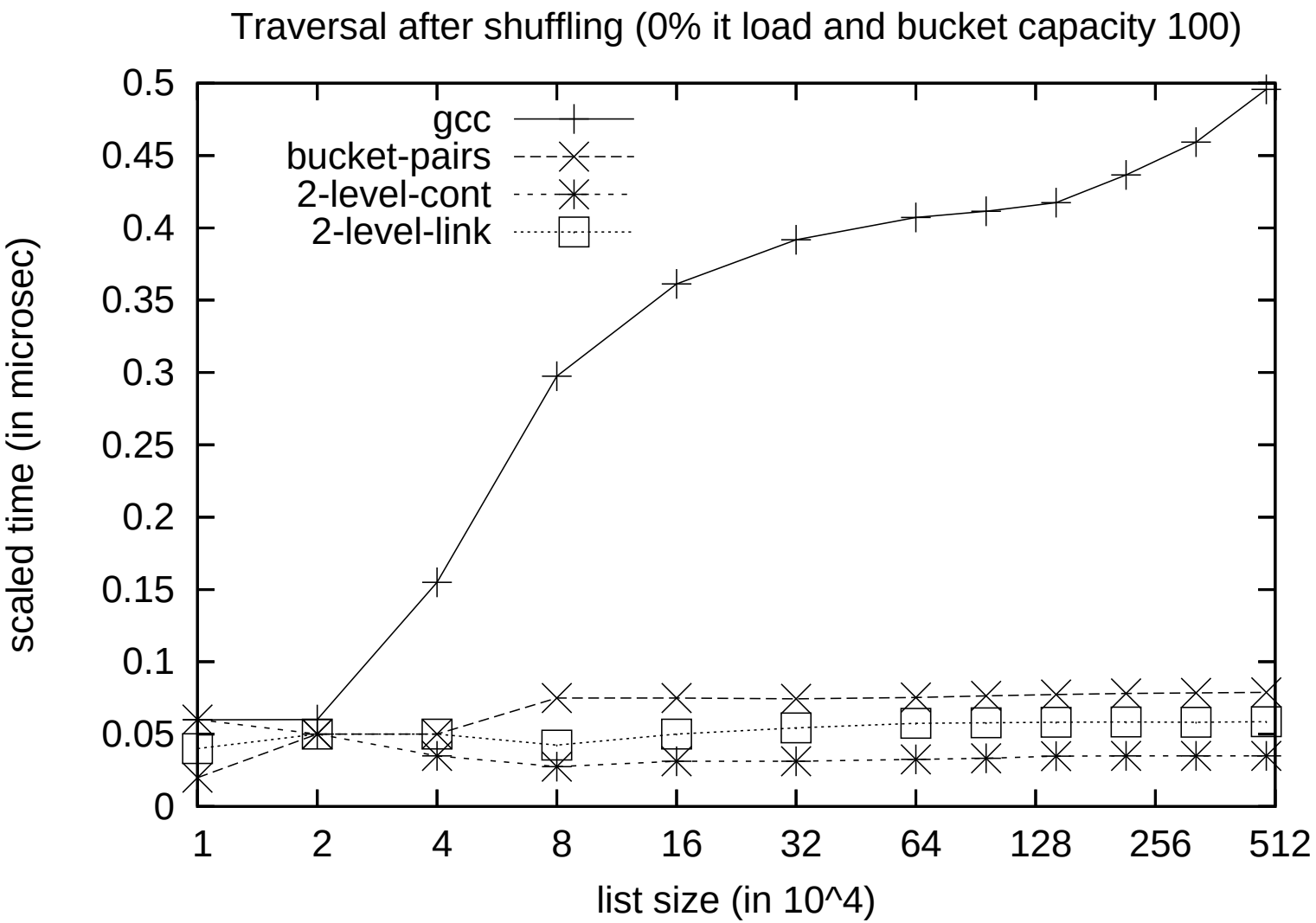
- lists with *no* iterators
- lists with iterators
- lists with several bucket capacities
- LEDA

Lists before and after elements reorganization (by sorting).

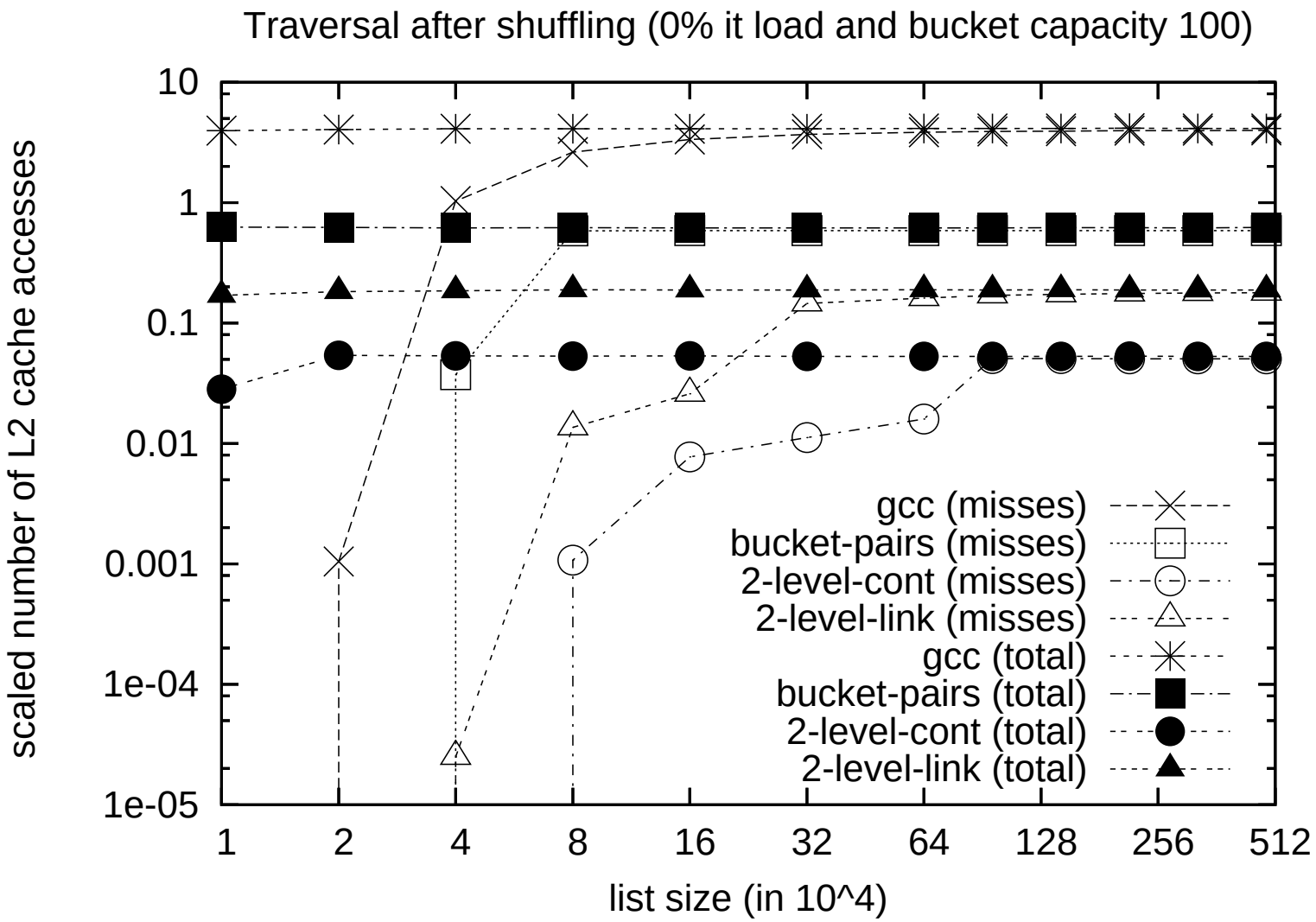
Traversal before



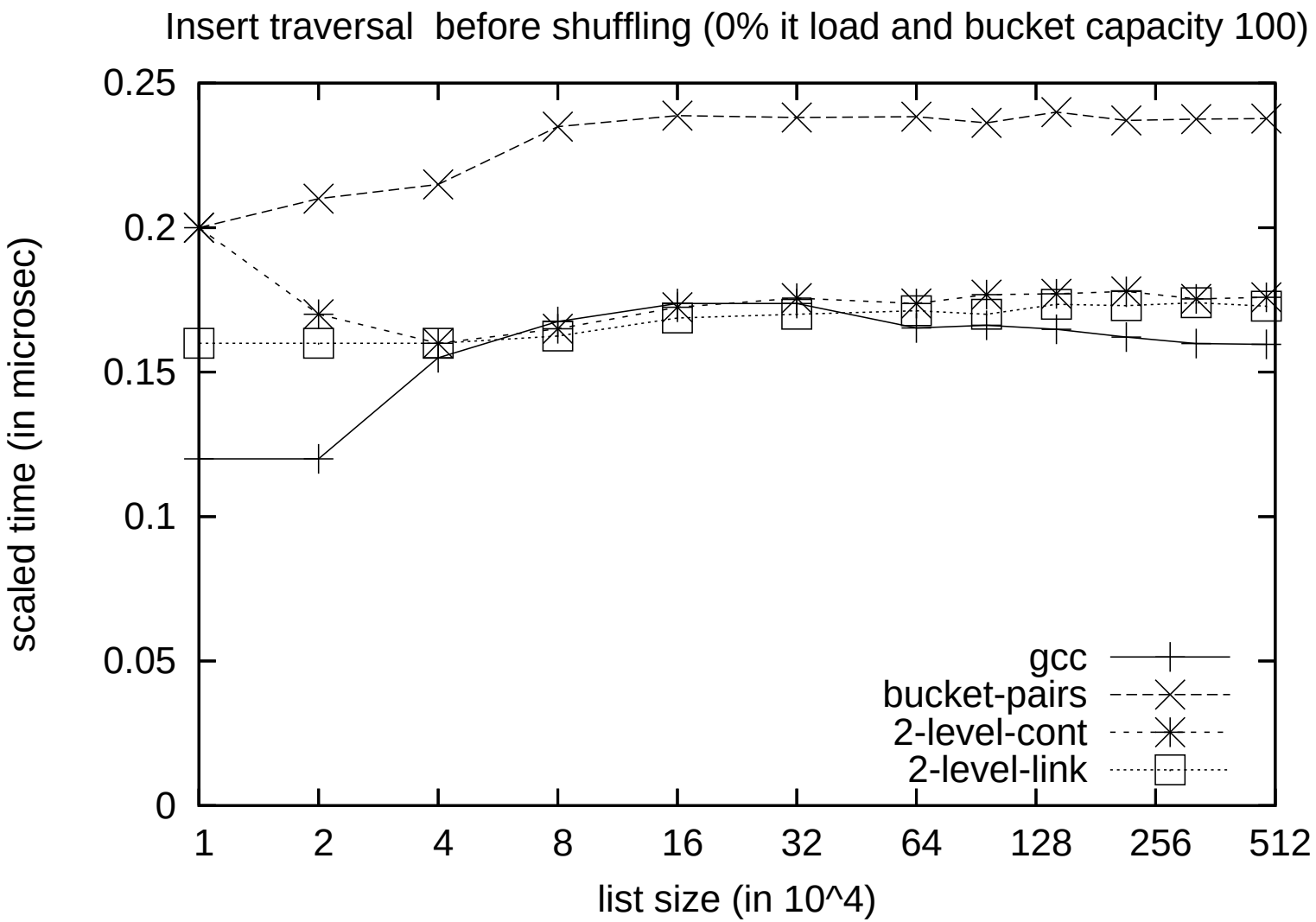
Traversal after



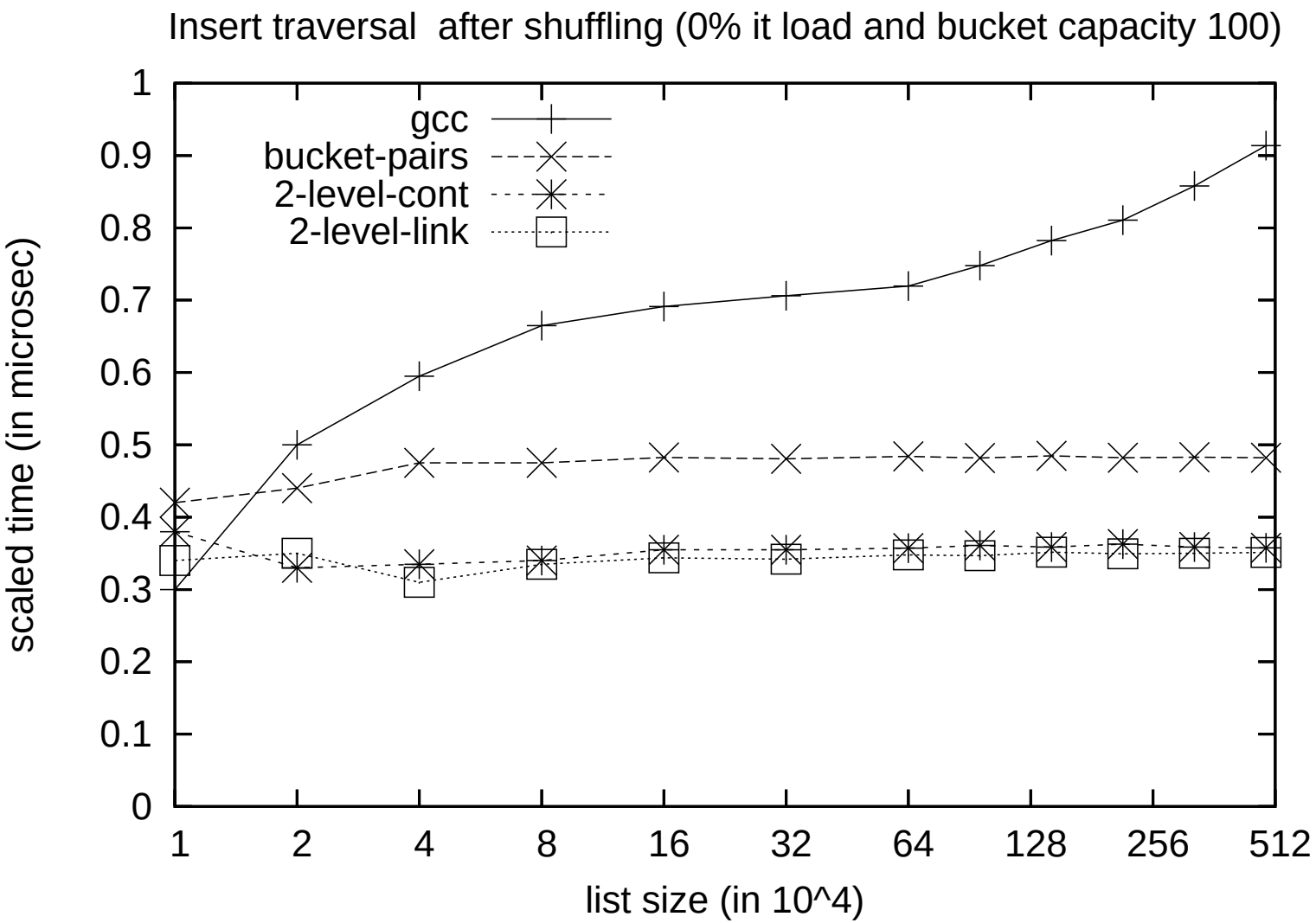
Pin Traversal after



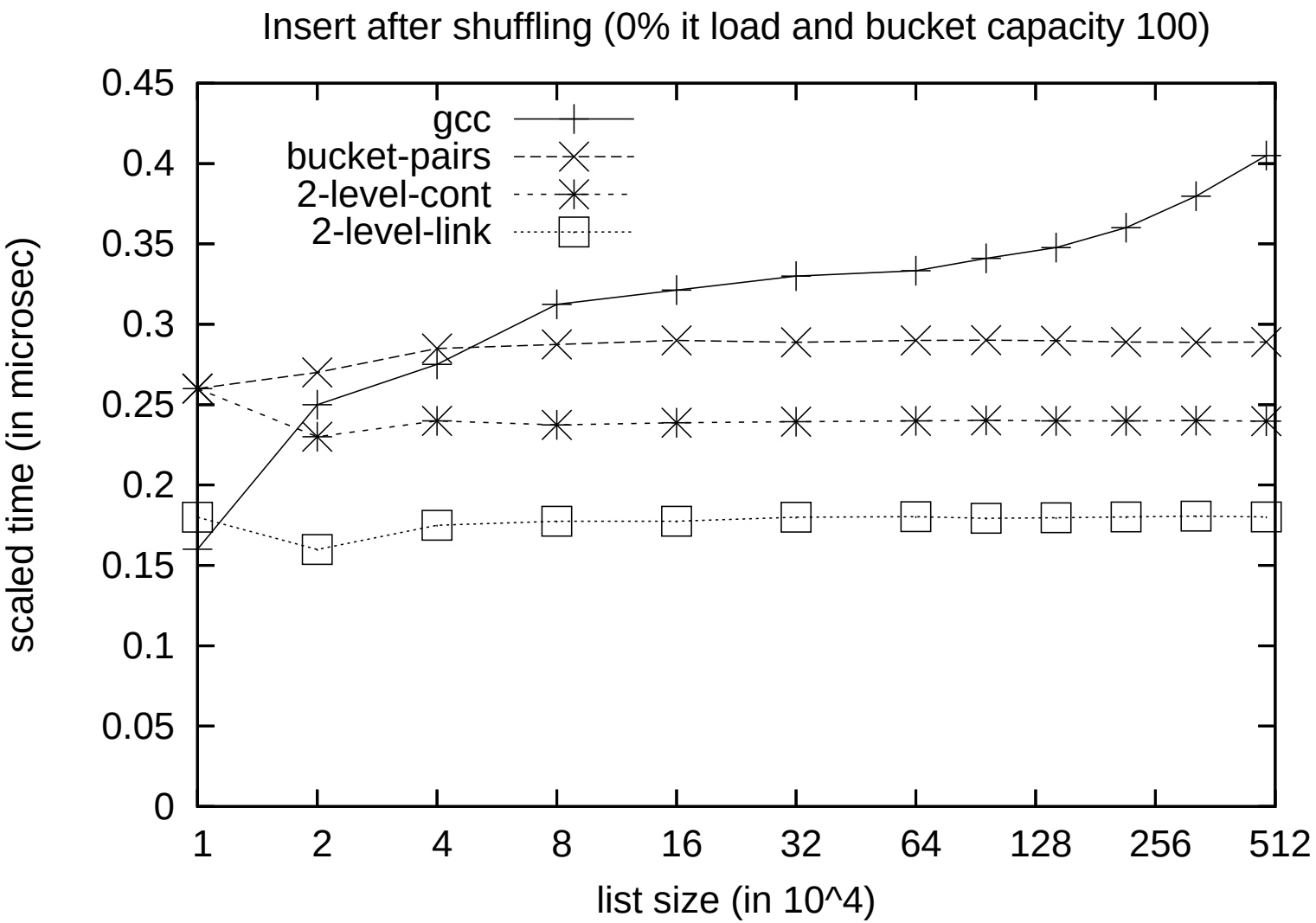
Insert before



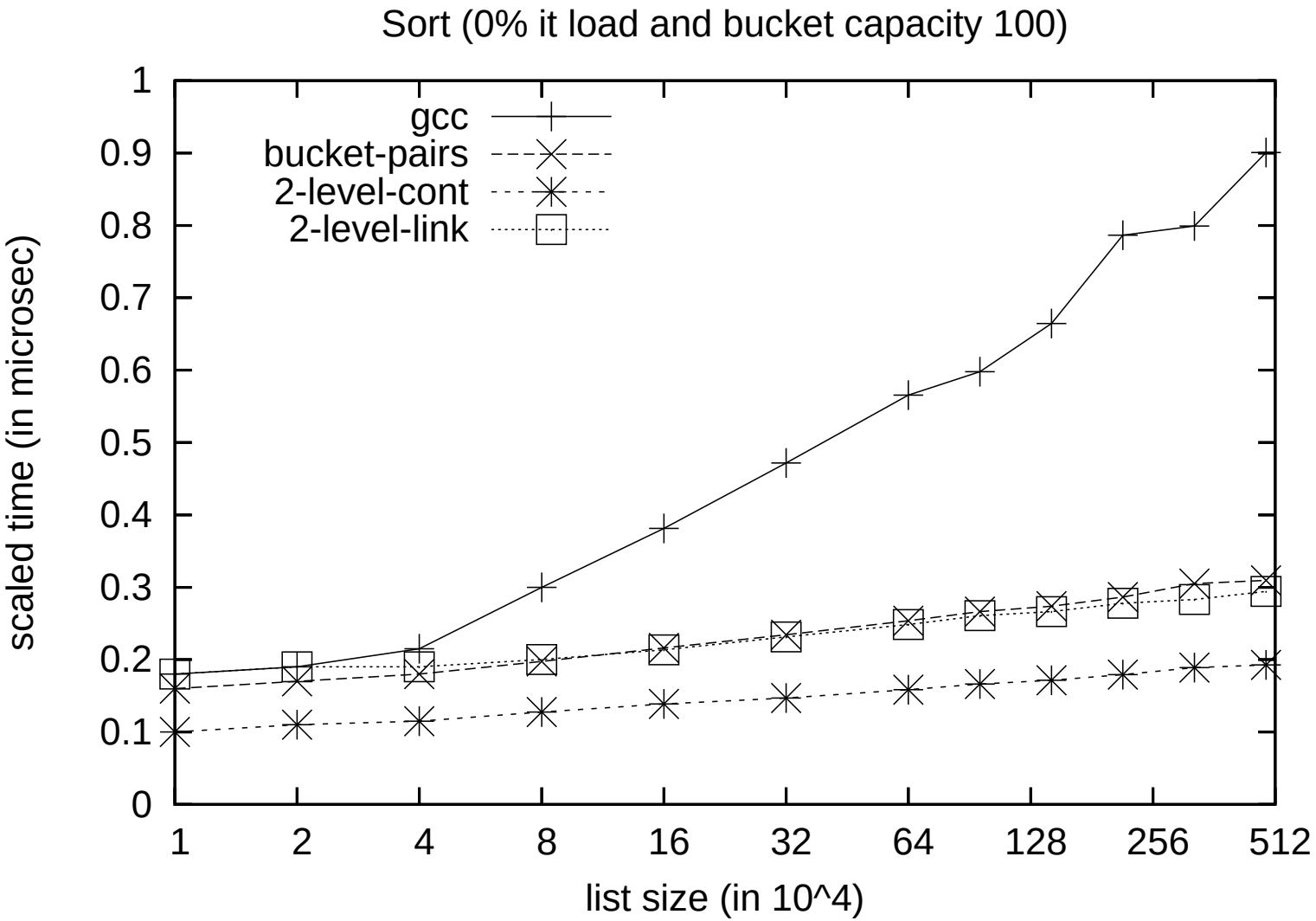
Insert after



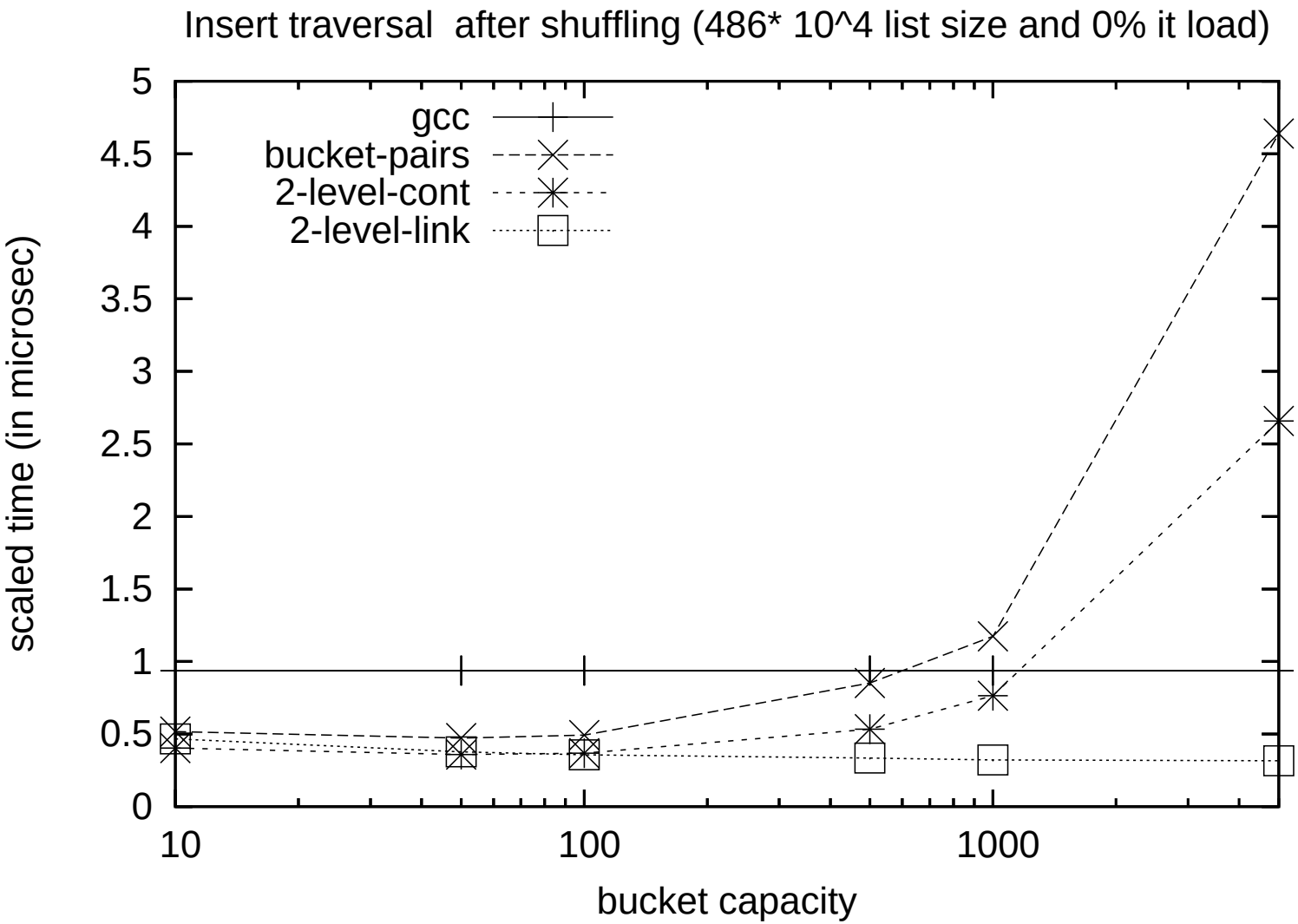
Intensive insertion



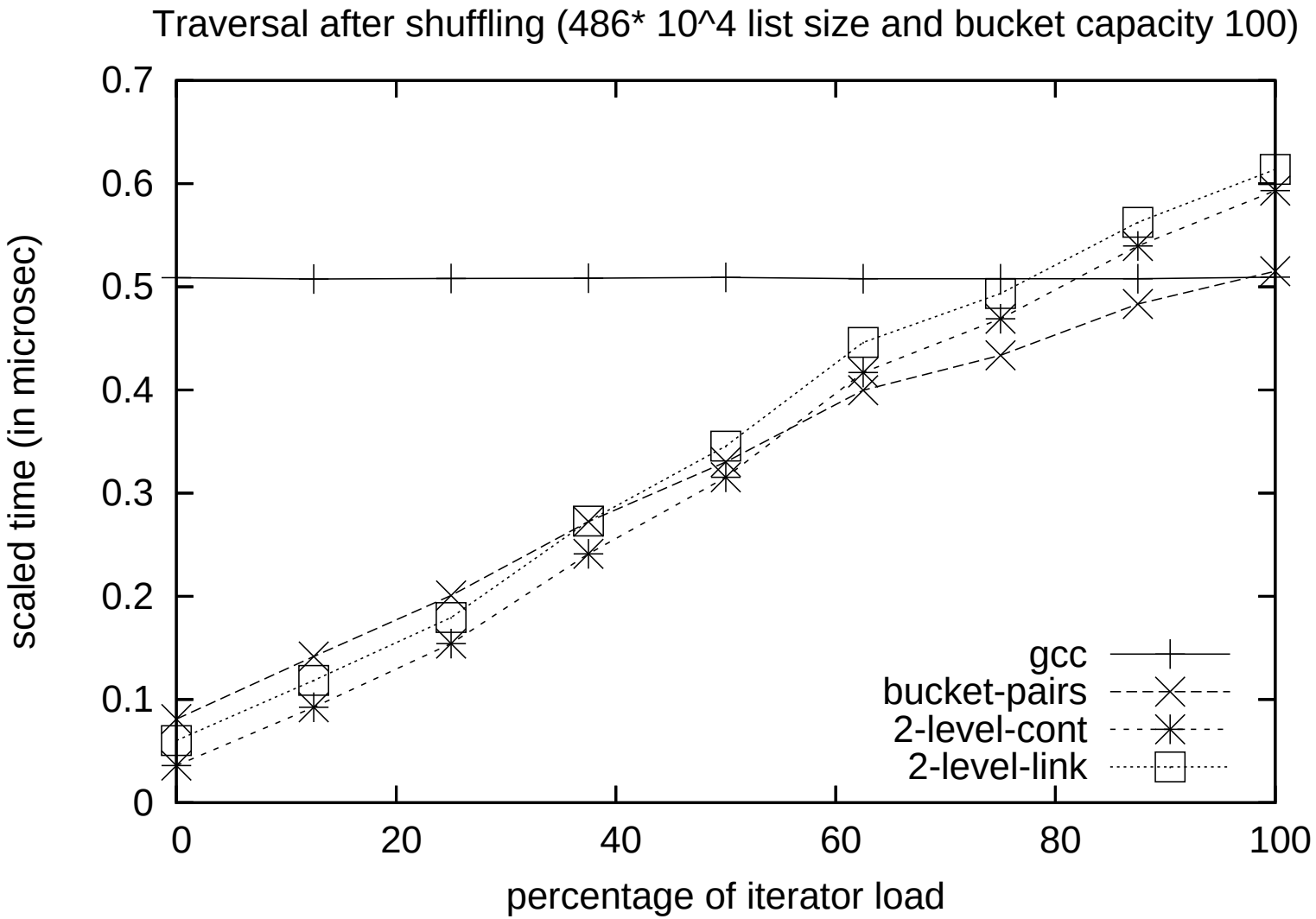
Internal sort



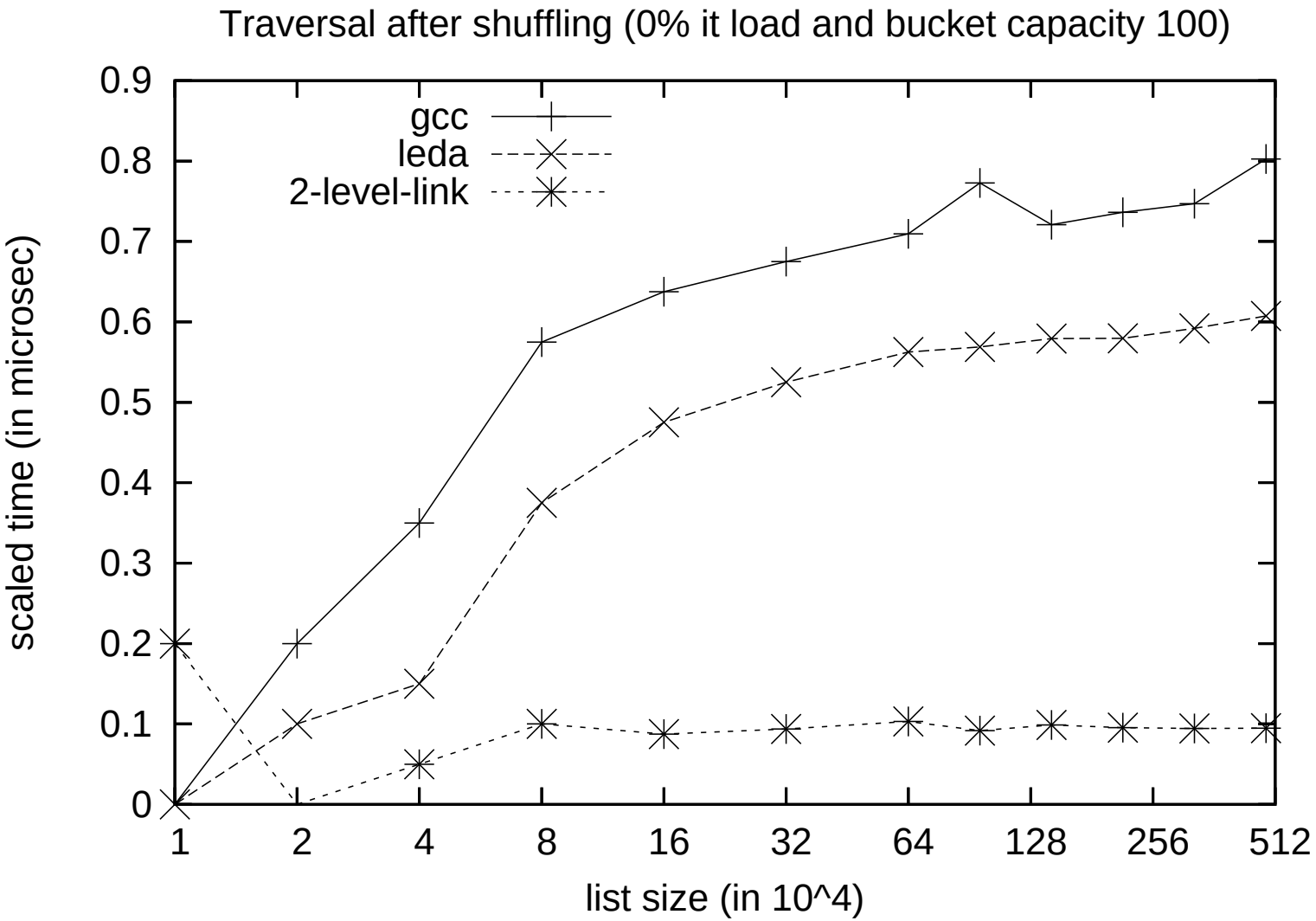
Effect of bucket capacity



Iterators


Traversal after shuffling (486* 10^4 list size and bucket capacity 100)
scaled time (in microsec)
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
20
40
60
80
100
percentage of iterator load
gcc
bucket-pairs
2-level-cont
2-level-link

LEDA


Traversal after shuffling (0% it load and bucket capacity 100)
gcc
leda
2-level-link
scaled time (in microsec)
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
1
2
4
8
16
32
64
128
256
512
list size (in 10^4)

Index

Conclusions (1)

Pioneering to show the importance of porting existing *theory* and *practice* on cache-conscious data structures to standard libraries, as the STL.

Provided *three* standard compliant cache-conscious lists implementations. This is not straightforward, although based on simple existing data structures.

- Kept with *standard requirements*, in particular with *iterators*. We have provided two standard compliant iterators designs.
- The algorithms involved must be designed carefully to keep up some properties.

Conclusions (2)

Provided a *comprehensive experimental* study.

Our implementations are *prefferable* in many (common) situations to classical double-linked list implementations, such as GCC (or LEDA).

Specifically,

- 5-10 times faster traversals
- 3-5 times faster internal sort
- still competitive with (unusual) big load of iterators
- bucket capacity is not a critical parameter

Between *our implementations*:

- 2-level linked implementation
- linked bucket implementation

What next?

My *webpage*: `www.lsi.upc.edu/~lfrias`

Extended article: reorganization algorithm analysed in detail.

- Using amortized analysis, we show that the number of created/destroyed buckets is assymptotically optimal.

Thank you

Questions?

References

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

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

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

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