Cache-Oblivious Priority Queue and Graph Algorithm Applications

by:

Lars Arge Michael A. Bender Erik D. Demaine
Bryan Holland-Minkley J. Ian Munro

presented by :

Ouafae Lachhab

Goal:

The main purpose of the article is:

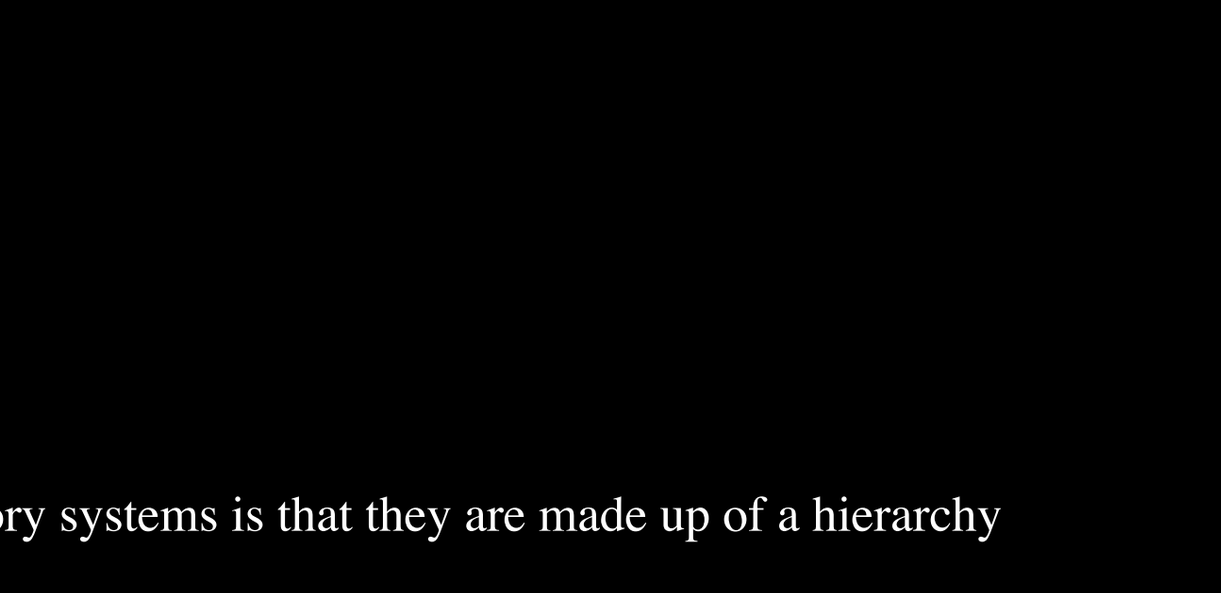
- Developing an optimal cache-oblivious priority queue data structure.
- This data structure should support the usual insertion, deletion and delete min operations in a multilevel memory hierarchy.
- Priority Queues are a critical component in many of the best known cache-aware graph algorithms. this cache oblivious data structure should match in performance the cache-aware algorithm.

Parameters of the cache are unknown contrary to cache aware algorithms.

Introduction:

One of the essential features of modern memory systems is that they are made up of a hierarchy of several levels of cache, main memory and disk.

In order to amortize the large access time of memory levels far away from the processor, memory systems often transfer data between memory levels in large blocks. Thus it is becoming increasingly important to obtain high data locality in memory access patterns.



Problem:

The standard approach to obtaining good locality is to design algorithms parameterized by several aspects of the memory hierarchy, such as the size of each memory level, and the block size of memory transfers between levels.

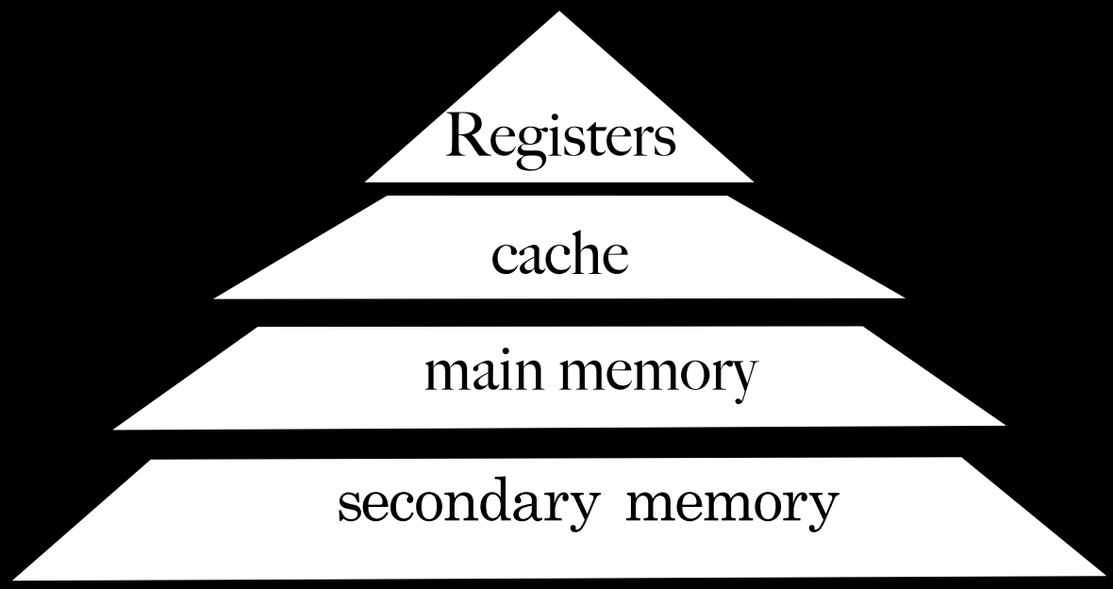
Unfortunately, this parameterization often leads to complex algorithms that are tuned to particular architectures. As a result, these algorithms are inflexible and not portable.

… and The Solution

Recently, the research has aimed at developing memory-hierarchy-sensitive algorithms that avoid any memory-specific parameterization. It has been shown that such cache-oblivious algorithms work optimally on all levels of multilevel memory hierarchy (and not only on a two-level hierarchy, as commonly used for simplicity).

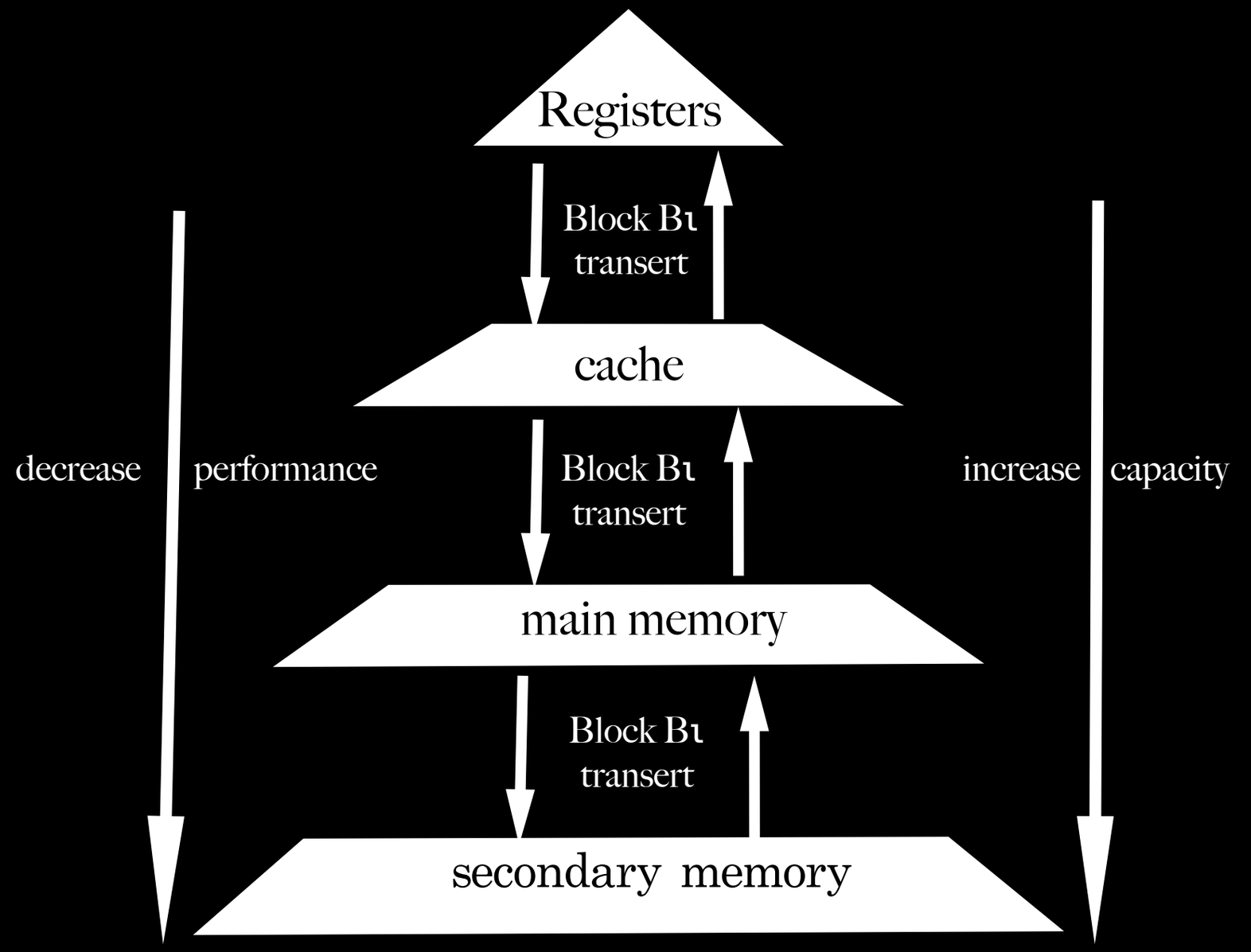
Memory Hierarchy

A typical hierarchy consists of a number of memory levels, with memory level l having size M_l and being composed of M_l/B_l blocks of size B_l. In any memory transfer from level l to $l-1$, an entire block is moved atomically. Each memory level also has an associated *replacement strategy*, which is used to decide what block to remove in order to make room for a new block being brought into that level of the hierarchy.



Efficiency

Efficiency of an algorithm is measured in terms of the number of block transfers the algorithm performs between two consecutive levels in the memory hierarchy (they are called *memory transfers*). An algorithm has complete control over placement of blocks in main memory and on disk.



Assumptions and Generalization

- $M > B^2$ (the *tall cache* assumption).

- When an algorithm accesses an element that is not stored in memory, the relevant block is automatically fetched with a memory transfer. *Optimal* paging strategy - If the memory is full, the *ideal* block in memory is elected for replacement based on the future characteristics of the algorithm.

- Since an analysis of an algorithm in the two-level model (which was in common use for simulating a multilevel memory hierarchy) holds for any block and memory size, it holds for *any* level of the memory hierarchy. As a consequence, if the algorithm is optimal in the two-level model, it is optimal on *all* levels of the memory hierarchy.

background

A priority queue maintains a set of elements each with a priority (or key) under the operations *insert*, *delete* and *deletemin*, where a deletemin operation finds and deletes the minimum key element in the queue. Common implementations for a priority queue are *heap* and *B-tree*, both support all operations in $O(\log_B N)$ memory transfers.

Bounds:

- Sorting N elements requires $\Theta((N/B)\log_{(M/B)}(N/B))$ memory transfers.
- In a B-tree, insert/deletemin takes $O(\log_B N)$ memory transfers, therefore sorting N elements takes $O(N\log_B N)$ memory transfers, which is a factor of $(B\log_B N)/(\log_{(M/B)}(N/B))$ from optimal.

- **From now on, we will use $\mathbf{sort}(N)$ to denote $(N/B)\log_{(M/B)}(N/B)$.**

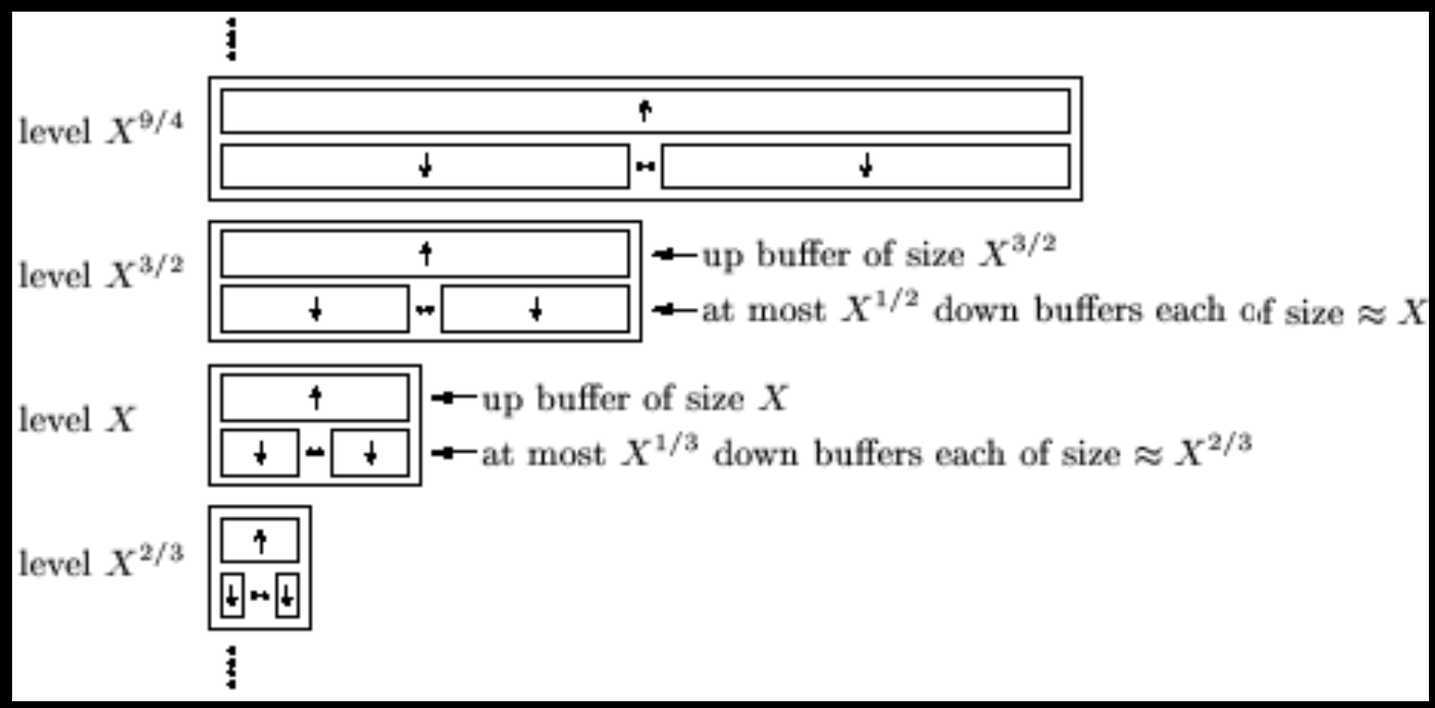
Priority Queue(The structure)

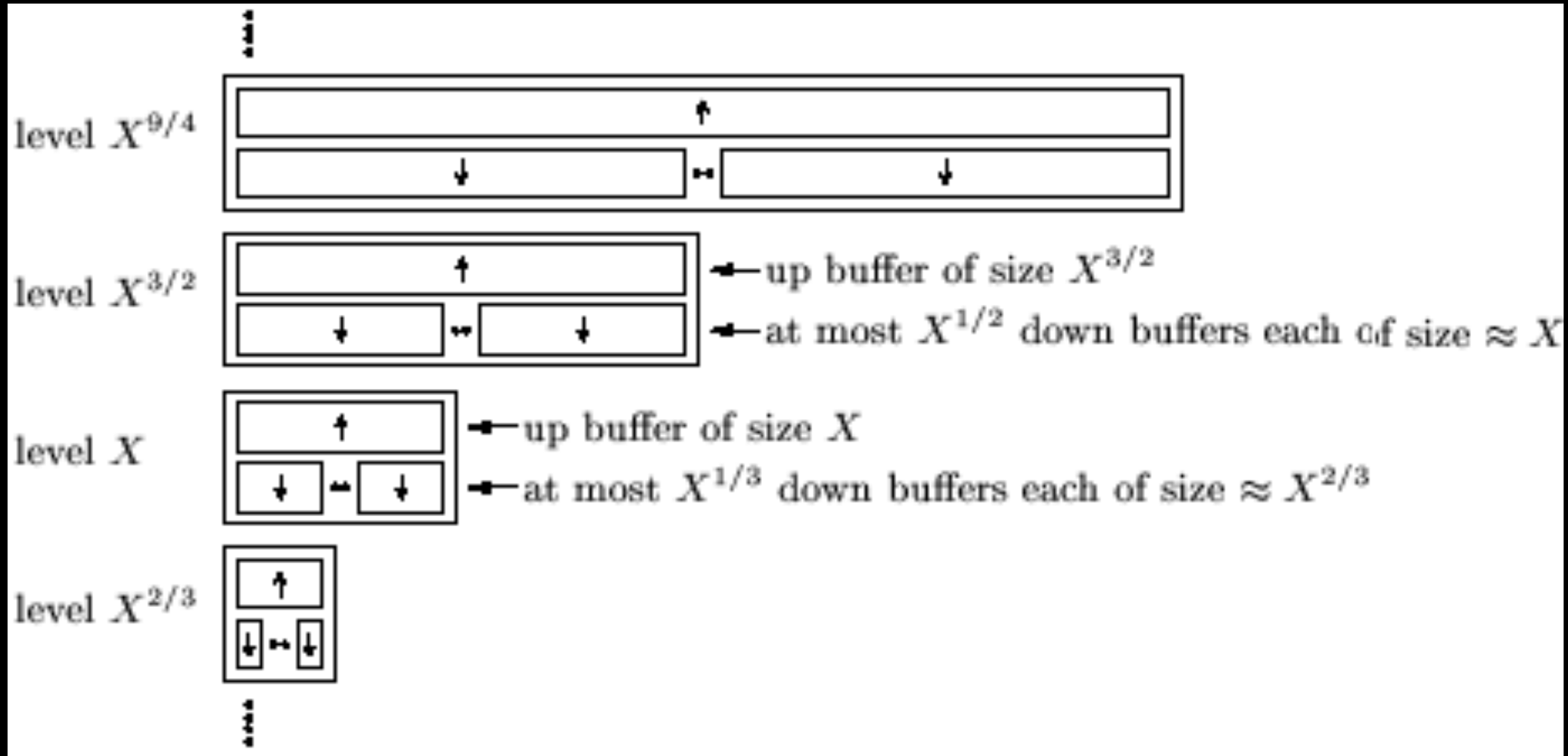
The priority queue data structure consists of $\Theta(\log \log N)$ levels whose sizes vary from N to a constant size c. The size of a level corresponds asymptotically to the number elements that can be stored within it.

For example, the i'th level from above has size $N^{2/3^{i-1}}$

The levels from largest to smallest are $N, N^{2/3}, N^{4/9}, \ldots, X^{9/4}, X^{3/2}, X, X^{2/3}, X^{4/9}, \ldots, c^{9/4}, c^{3/}, c$.

Smaller levels store elements with smaller keys or elements that were more recently inserted. The minimum key element and the most recently inserted element are always in the smallest (lowest) level c. Both insertions and deletions are initially performed on the smallest level and may propagate up through the levels.





Elements are stored in a level in a number of *buffers*, which are also used to transfer elements between levels. Level X consists of one *up buffer* u^X that can store up to X elements, and at most $X^{1/3}$ *down buffers*, each containing between $\frac{1}{2}X^{2/3}$ and $2X^{2/3}$ elements. Thus the maximum capacity of level X is $3X$. The size of a down buffer at one level matches the size (up to a constant factor) of the up buffer one level down.

The Invariants

Invariant 1:
At level X, elements are stored among the down buffers, that is, elements is d_i^x have smaller keys than elements in d_{i+1}^x, but the elements within d_i^x are unordered.

The element with largest key in each down buffer is called a *pivot element*, and is used to mark the boundaries between the ranges of the keys of elements in down buffers.

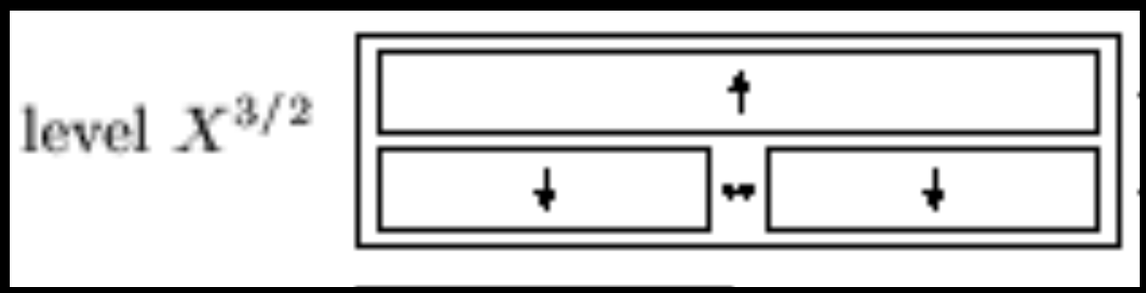
Invariant 2:
At level X, the elements in the down buffers have smaller keys than the elements in the up buffer.

Invariant 3:
The elements in the down buffers at level X have smaller keys than the elements in the down buffers at the next higher level $X^{3/2}$.

The invariants ensure that the keys of the elements in the down buffers get larger as we go from smaller to larger levels in the structure. At one level, keys of elements in the up buffer are larger than the keys in the down buffers. The keys of elements in an up buffer are unordered relative to the keys of the elements in the down buffer one level up.

Transferring Elements (in general)

Up buffers store elements that are “on their way up” – they have yet to be resolved as belonging to a particular down buffer in the next level.
Down buffers store elements that are “on their way down” – they have yet to be resolved as belonging to a particular down buffer in the next level down. The element with overall smallest key is in the first down buffer at level c.



Priority Queue - Layout

The priority queue is stored in a linear array. The levels are stored consecutively from smallest to largest. Each level occupies 3 times its size, starting with the up buffer (occupying one time its size) followed by the down buffers (occupying two times its size).

The total size of the array is $O(N)$.

Operations on Priority Queues

We use two general operations – *push* and *pull*. Push inserts X elements into level $X^{3/2}$, and pull removes the X elements with smallest keys from level $X^{3/2}$, returning them in sorted order.

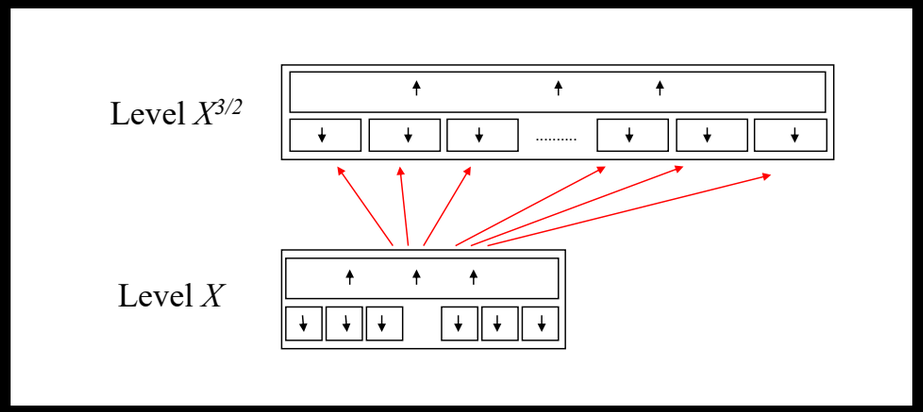
Deletemin (insert) corresponds to pulling (pushing) an element from (to) the smallest level.

Whenever an up buffer in level X overflows, we push the X elements in the buffer one level up, and whenever the down buffers in level X become too empty, we pull X elements from one level up.

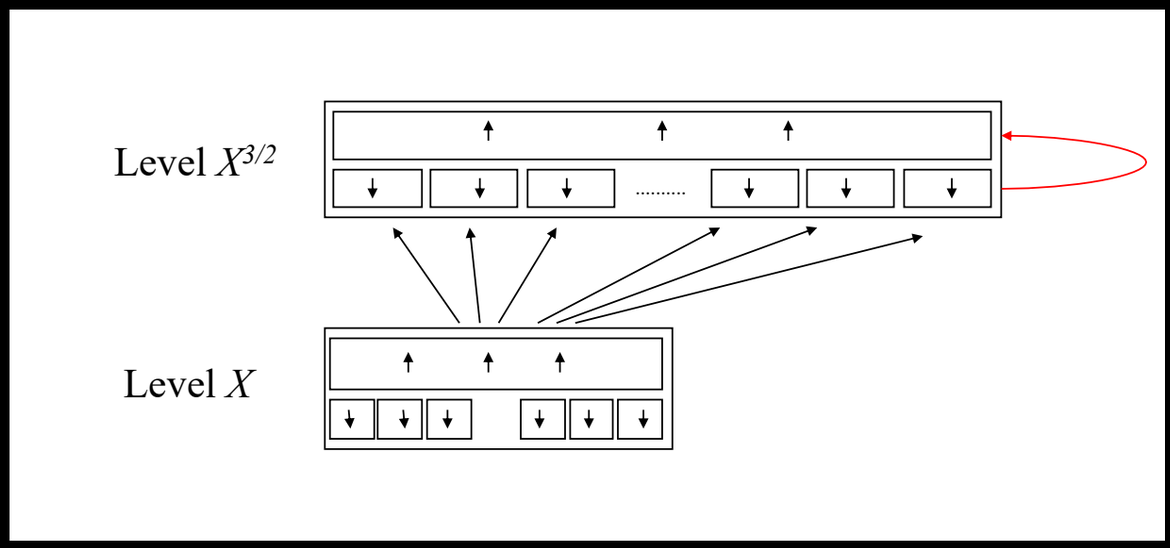
Both the *push* and the *pull* operations maintain the three invariants.

Push operation:

To insert X elements into level $X^{3/2}$, we first sort them using $O(1+(X/B)\log_{(M/B)}(X/B))$ memory transfers and $O(X\log_2 X)$ time. Next we distribute them into the $X^{1/2}$ down buffers of level $X^{3/2}$. We append an element to the end of the current down buffer, and advance to the next down buffer as soon as we encounter an element with larger key than the key of the pivot of the current down buffer.



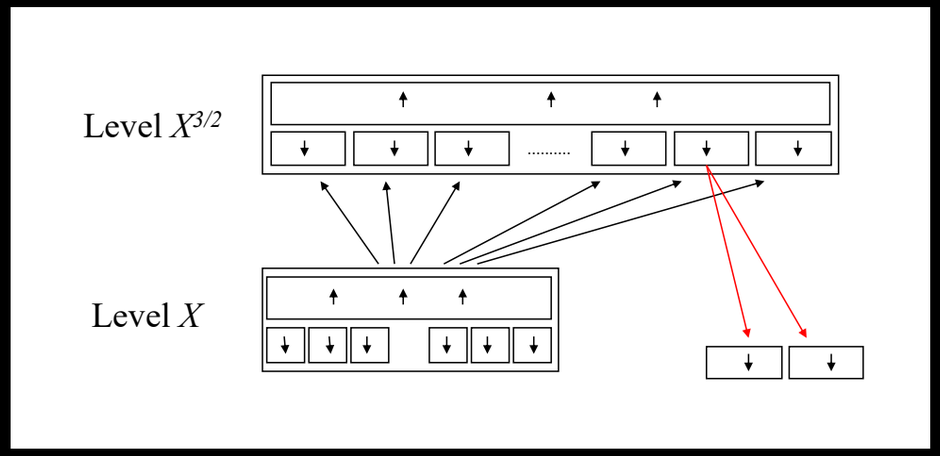
Elements with keys larger than the keys of the pivot of the last down buffer are inserted in the up buffer.



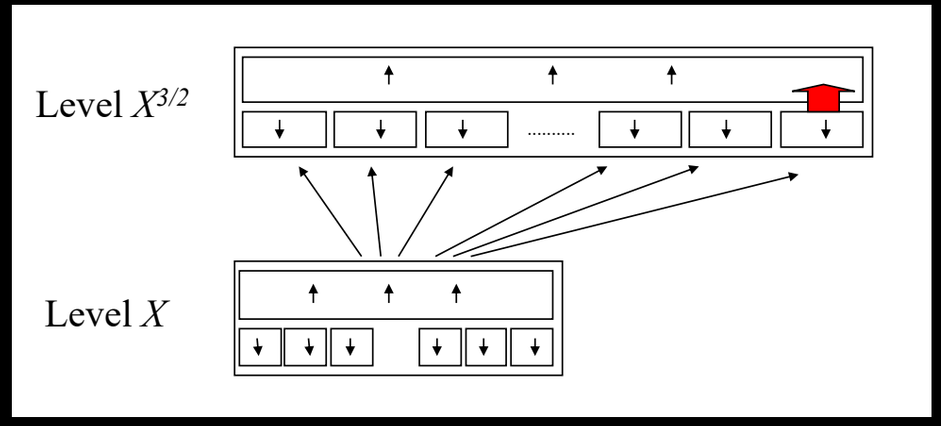
Scanning through the elements takes $O(1+X/B)$ memory transfers and $O(X)$ time. We perform one memory transfer for each of the $X^{1/2}$ down buffers (in the worst case), so the total cost of distributing the elements is $O(X/B+X^{1/2})$ memory transfers and $O(X+X^{1/2}) = O(X)$ time.

Push - The Elements Distribution

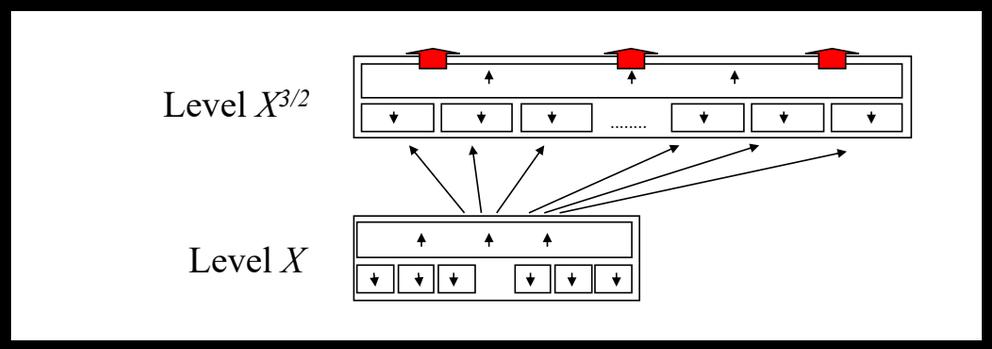
If a down buffer runs full (contains $2X$ elements), we split the buffer into two down buffers each containing X elements, by finding the median of the elements in $O(1+X/B)$ memory transfers and $O(X)$ time and partitioning them in a simple scan in $O(X)$ memory transfers. We assume that the down buffer can be stored out of order, so we just have to update the linked list of buffers.



If the level has already the maximum of $X^{1/2}$ down buffers before the split, we remove the last down buffer by inserting its elements into the up buffer.

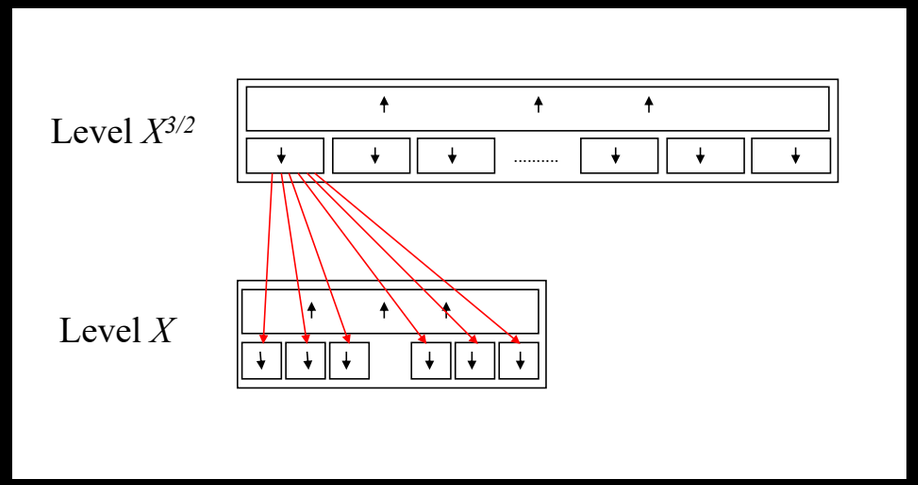


If the up buffer runs full (contains more than $X^{3/2}$ elements), then all of these elements are pushed into the next level up.

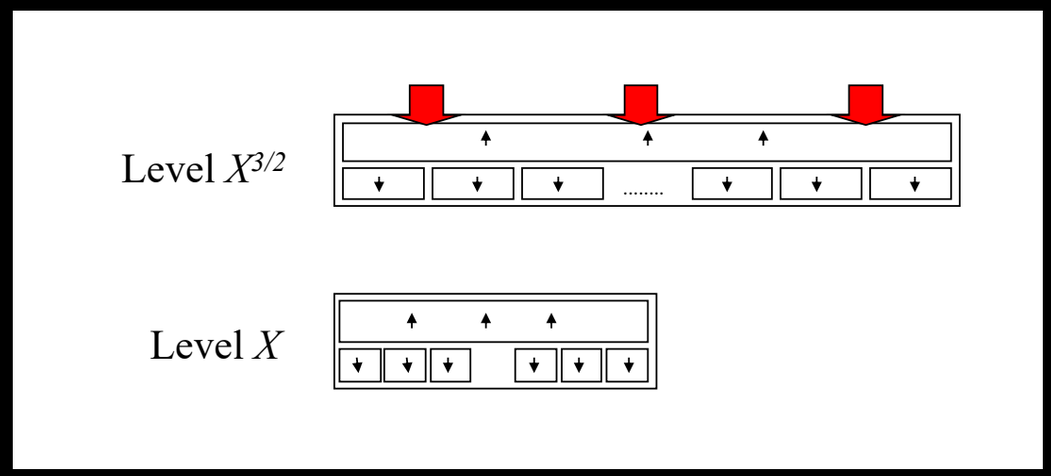


Pull - Demonstration

To delete X elements from level $X^{3/2}$, we first assume that the down buffers at level $X^{3/2}$ contain at least $3/2*X$ elements each, so the first three down buffers contain between $3/2*X$ and $6X$ elements. We find and remove the X smallest elements by sorting them using $O(1+(X/B)\log_{(M/B)}(X/B))$ memory transfers and $O(X\log_2 X)$ time.

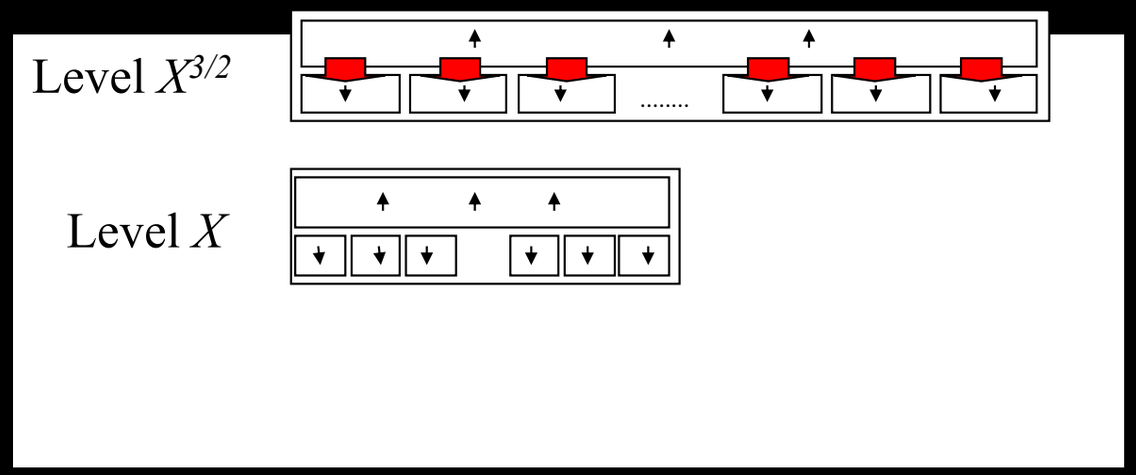


If the down buffers contain less than $3/2*X$ elements, we first pull the $X^{3/2}$ elements with smallest keys from one level up.



Pull(Demonstration)

Because these elements do not necessarily have smaller keys than the U elements in the up buffer, we first sort the up buffer, and then we insert the U elements with the largest keys to the up buffer and distribute the remaining elements in the down buffers.



Afterwards, we can find the X elements with smallest keys as described earlier and remove them. The total cost of a pull of X elements from level $X^{3/2}$ down to level X is $O(1+(X/B)\log_{(M/B)}(X/B))$ memory transfers and $O(X\log_2 X)$ time amortized, not counting the cost of any recursive push operations, while maintaining invariants 1-3.

Pull/push(Total_Cost)

lemma : following the previous results, push or pull of X elements to level X use $O(X^{1/2}+(X/B)\log_{(M/B)}(X/B))$ memory transfers. We can reduce this cost to $O((X/B)\log_{(M/B)}(X/B))$ by examining the cost for differently sized levels.

The delete-min is done by performing a pull on the smallest level. This may require recursive pushes (pulls) on higher levels. To maintain that the structure uses $\Theta(N)$ space, and has $\Theta(\log \log N)$ levels,

List Ranking – Main Idea

In this problem we have a linked list with V nodes, stored as an unordered sequence, each containing the position of the next node in the list (an edge). Each edge has a weight and the goal is to find for each node v the sum of the weights of edges from v to the end of the list.

For example, if all the weights are 1, then the goal is to determine the number of edges from v to the end of the list (called the *rank* of v).

List Ranking - Explanation

The list ranking algorithm in details:

Step 1:
Finding an independent set. This step is done cache-obliviously in O(sort(V)) memory transfers.

Step 2:
Taking out the independent set nodes by contracting edges incident to these nodes. We first identify a node u with a successor v in the independent set, by sorting the nodes by their successor position.
We also identify the successor w of the independent set node v.

Next we create in a simple scan a new list where the two edges (u,v) and (v,w) have been replaced with an edge (u,w). The new edge has weight equals to the sum of the two old edges.
We remove the independent set nodes and store the remaining nodes in a new list. We also create a list that indicates the old position of every node. Finally, we “compress” the remaining nodes.

List Ranking - Explanation

Step 3:

Ranking the remaining list.

Step 4:

Repeating steps 1-3 recursively

Step 5:

Finally, the independent set nodes are reintegrated into the list, while computing their ranks.

how to find the independent set cache-obliviously. The algorithm is based on 3-coloring – every node is colored with one of three colors such that adjacent nodes have different colors. The independent set (of size at least $V/3$) then consists of the nodes with the most popular color…

Computing 3-Coloring

An edge (*v*,*w*) is called a *forward* edge if *v* appears before *w* in the sequence of nodes, otherwise it is called a *backward* edge. First the list is split into two sets – one consists of forward lists and one consists of backward lists. Every node is included in at least one set, when a node which is the head or the tail of a list may be included in both of the sets.

The nodes in the forward lists are colored red or blue alternatively (the heads are red), and the nodes in the backward lists are colored green or blue alternatively (the heads are green).

This way every node is colored in one color, except for the heads / tails, which have two colors. If the heads / tails of the lists were colored red and green or red and blue, they are colored red only. If they were colored blue and green, they are colored green only.

3-Coloring and Priority Queue

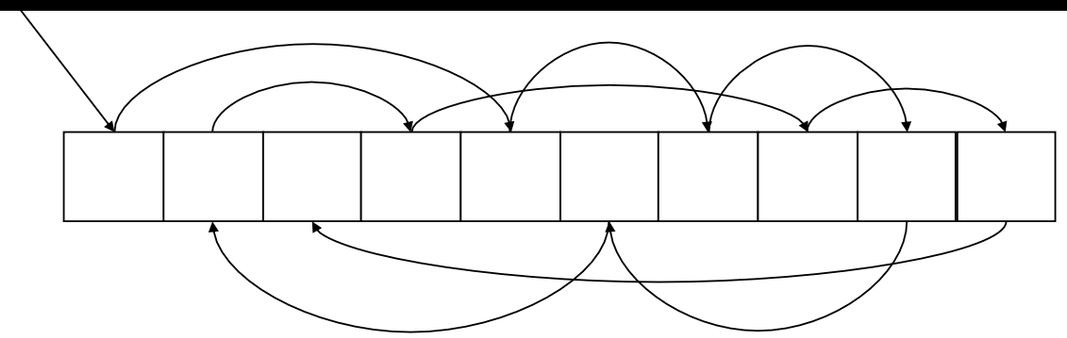
In a single scan the head nodes are identified, and for each such node *v* a red element is inserted in a cache-oblivious priority-queue with key equal to the position of *v* in the unordered list. Then the minimal key element *e* is repeatedly extracted from the queue.

The successors of *v* are inserted in the queue. The inserted elements is colored the opposite color of *e*…*etc*

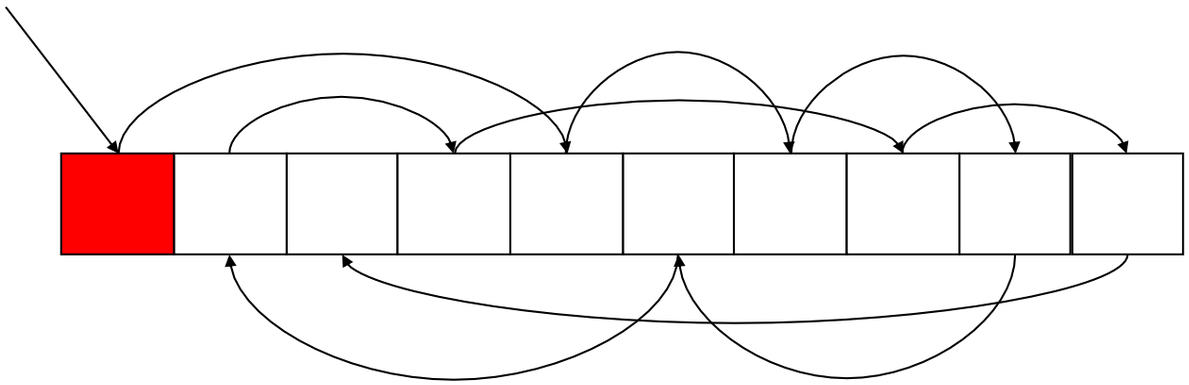
After finishing scanning and coloring all the forward lists, the backward lists are scanned and colored the same way.

3-Coloring – Demonstration

Starting the coloring:

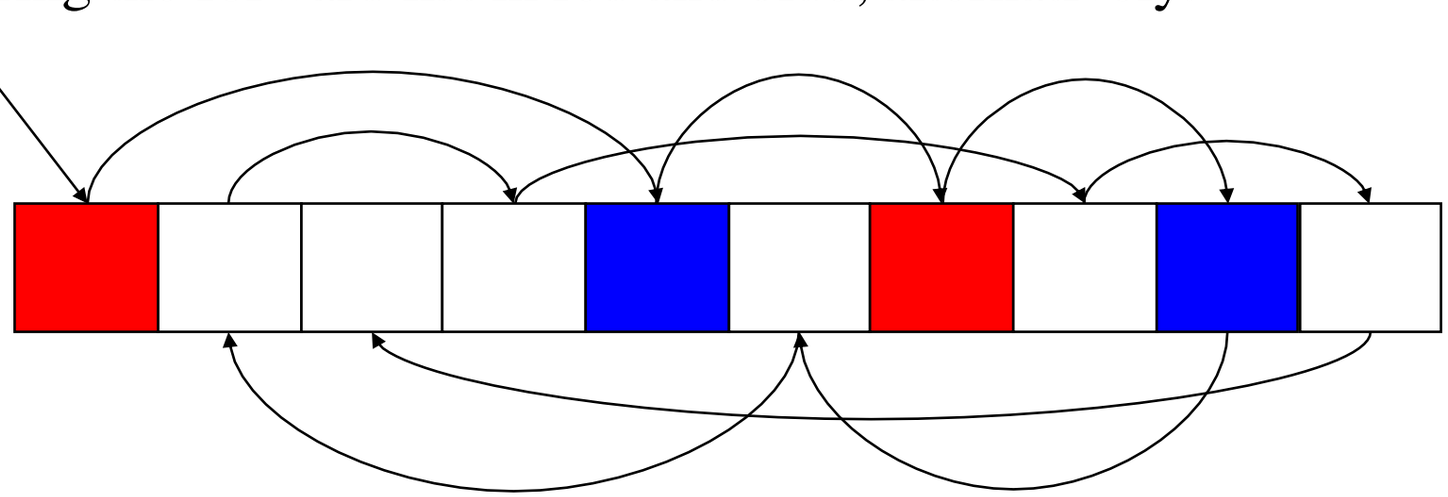


The head of a forward list is colored red.

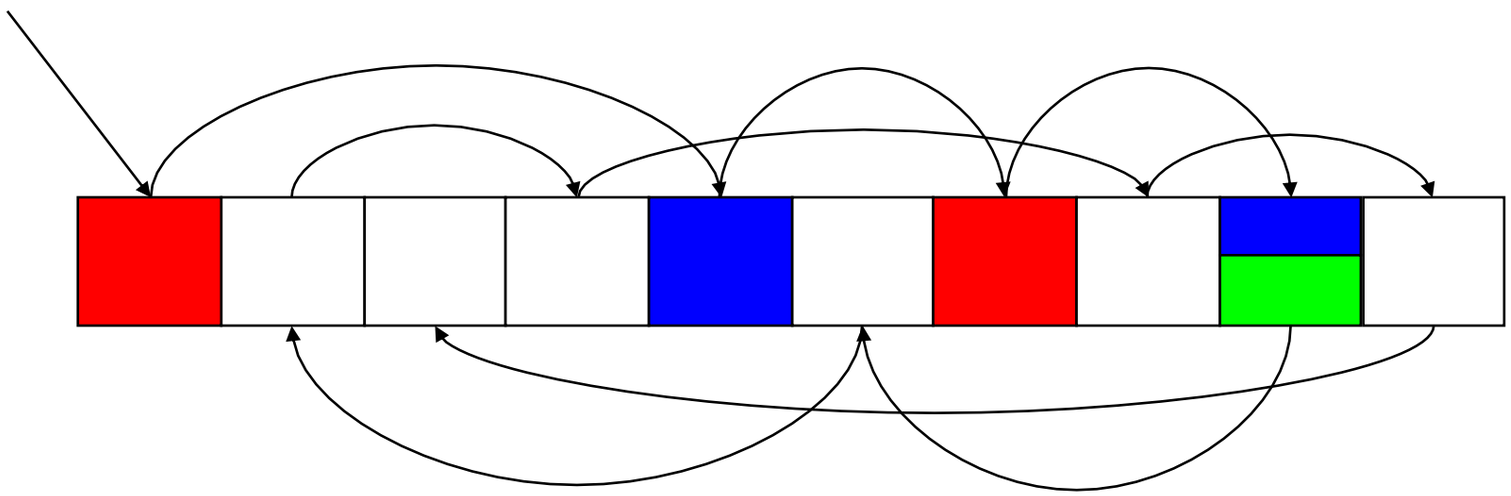


3-Coloring – Demonstration

Coloring the forward list in red and blue, alternatively.



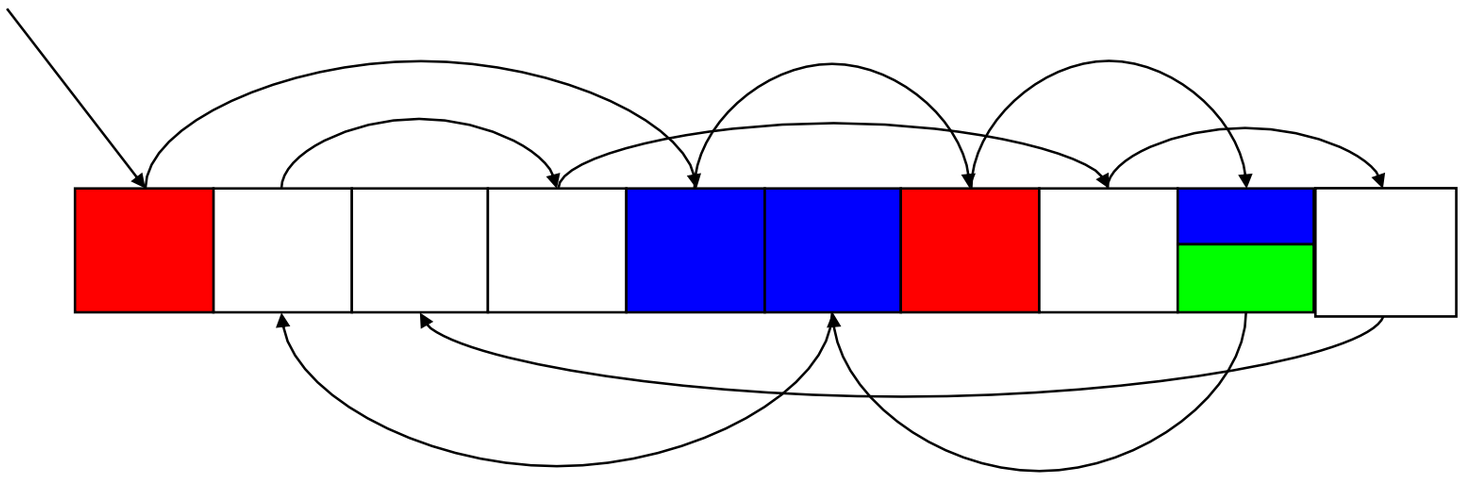
The head of a backward list is colored green.



3-Coloring – Demonstration

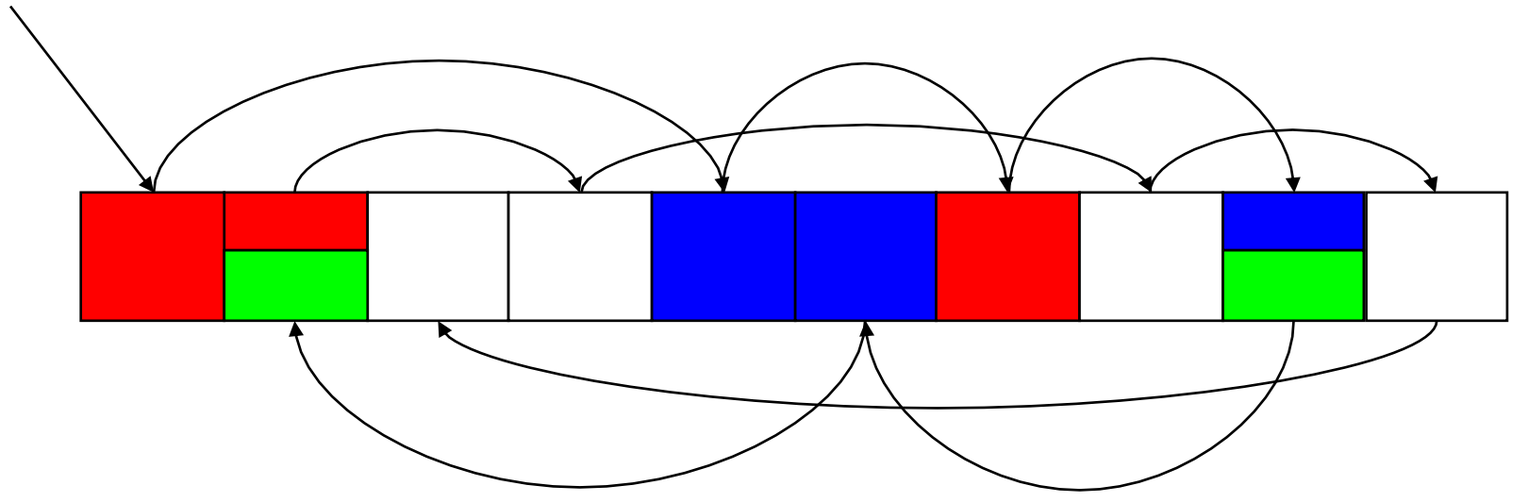
Coloring the backward list in green and blue, alternatively.

Coloring the backward list in green and blue, alternatively.

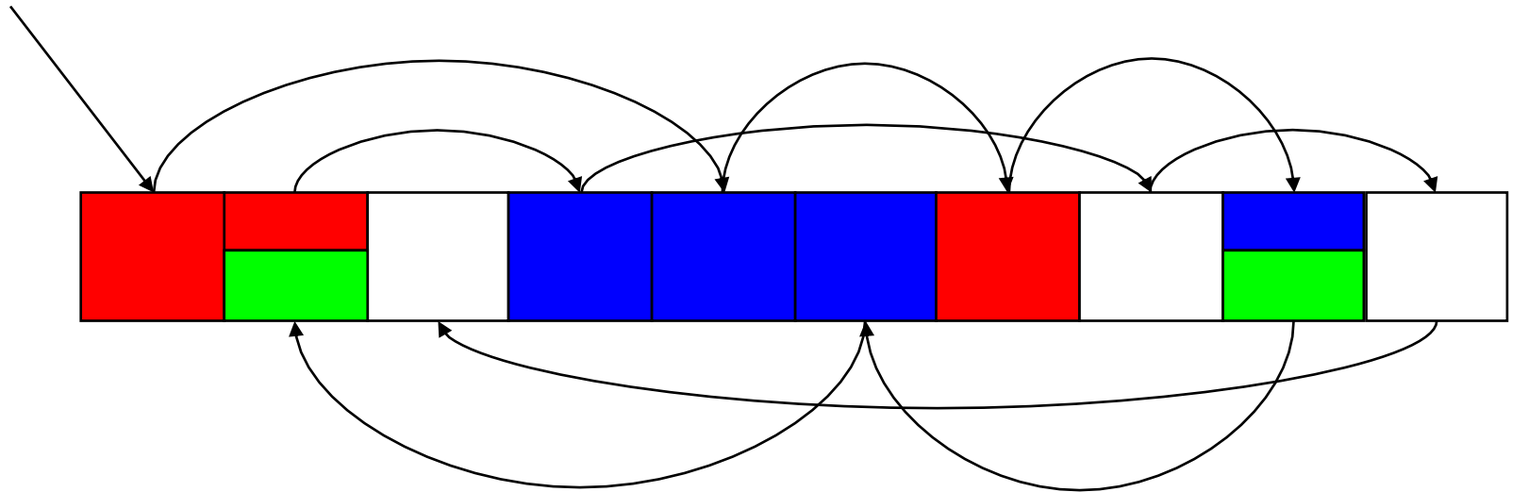


3-Coloring – Demonstration

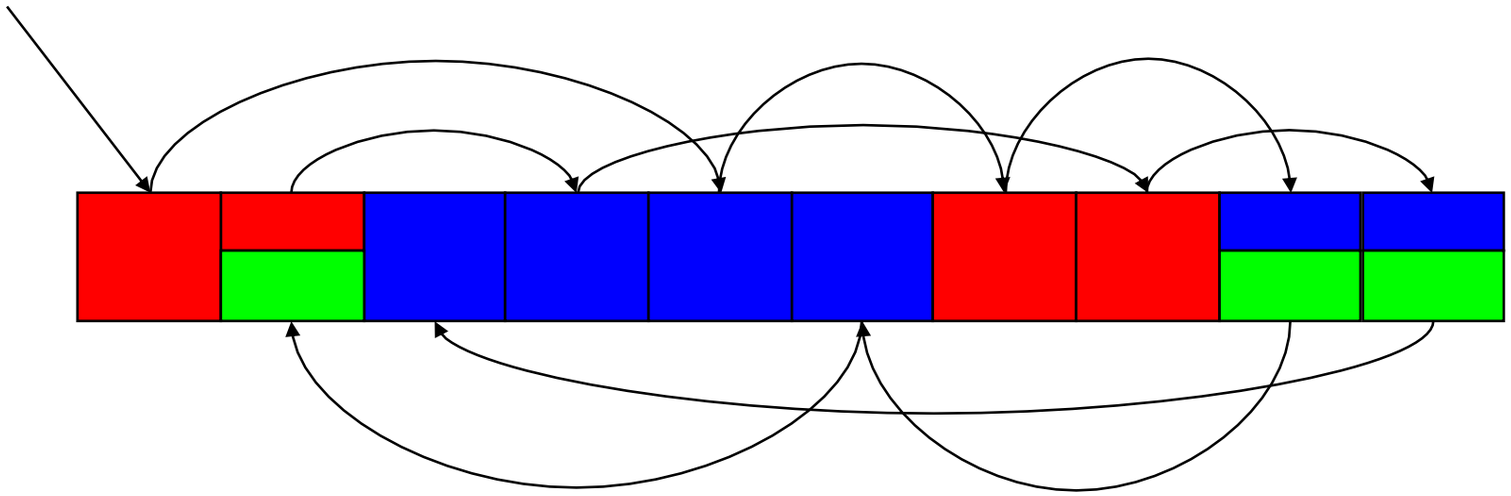
The head of a forward list is colored red.



Coloring the forward list in red and blue, alternatively.



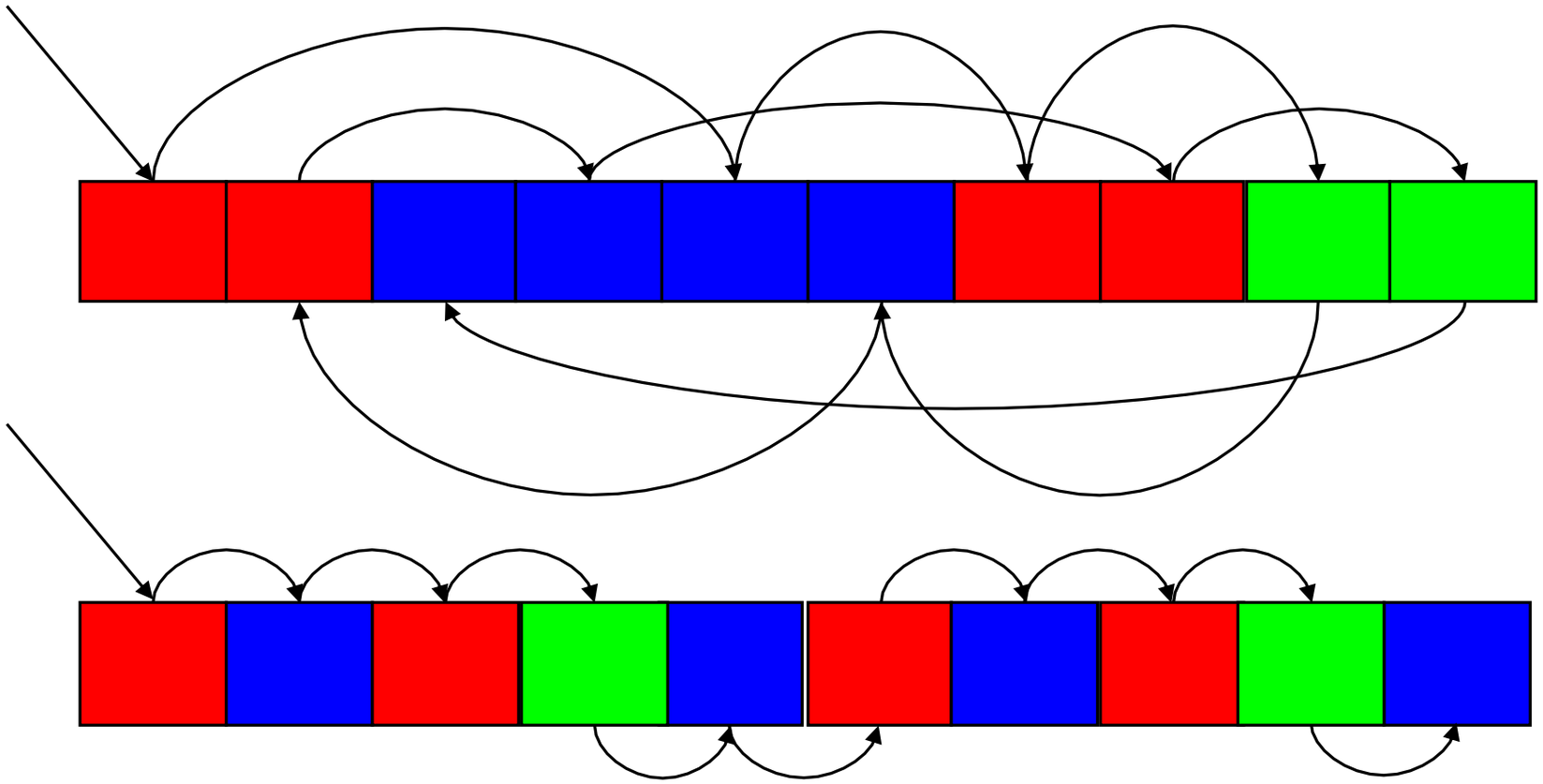
3-Coloring – Demonstration



If the head / tail of a list was colored red and green or red and blue, it is colored red only. If it was colored blue and green, it is colored green only.

3-Coloring – Demonstration

Now the whole list is 3-colored, and so the next two lists are identically colored.



List Ranking - Complexity

Finding the independent set can be done in O(sort(*V*)) memory transfers. The rest of the steps can be done in O(sort(*V*)) as well. Therefore, the list ranking problem on a *V* node list can be solved cache-obliviously in O(sort(*V*)) memory transfers.

results recap

Problem	Our cache-oblivious result	Previous best cache-aware result
Priority queue	$O(\frac{1}{B}\log_{M/B}\frac{N}{B})$	$O(\frac{1}{B}\log_{M/B}\frac{N}{B})$
List ranking	$O(\text{sort}(V))$	$O(\text{sort}(V))$
Tree algorithms	$O(\text{sort}(V))$	$O(\text{sort}(V))$
Directed BFS and DFS	$O((V+E/B)\log_2 V+\text{sort}(E))$	$O((V+E/B)\log_2 V+\text{sort}(E))$ $O(V+\frac{EV}{BM})$
Undirected BFS	$O(V+\text{sort}(E))$	$O(V+\text{sort}(E))$ $O(\sqrt{\frac{V\cdot E}{B}}+\text{sort}(E))$
Minimal spanning forest	$O(\text{sort}(E)\cdot\log_2\log_2 V)$ $O(V+\text{sort}(E))$	$O(\text{sort}(E)\cdot\log_2\log_2\frac{VB}{E})$ $O(v+\text{sort}(E))$

Introduction: Semantic Locality

Developing an optimal cache-oblivious data structures to get the best of memory hierarchy

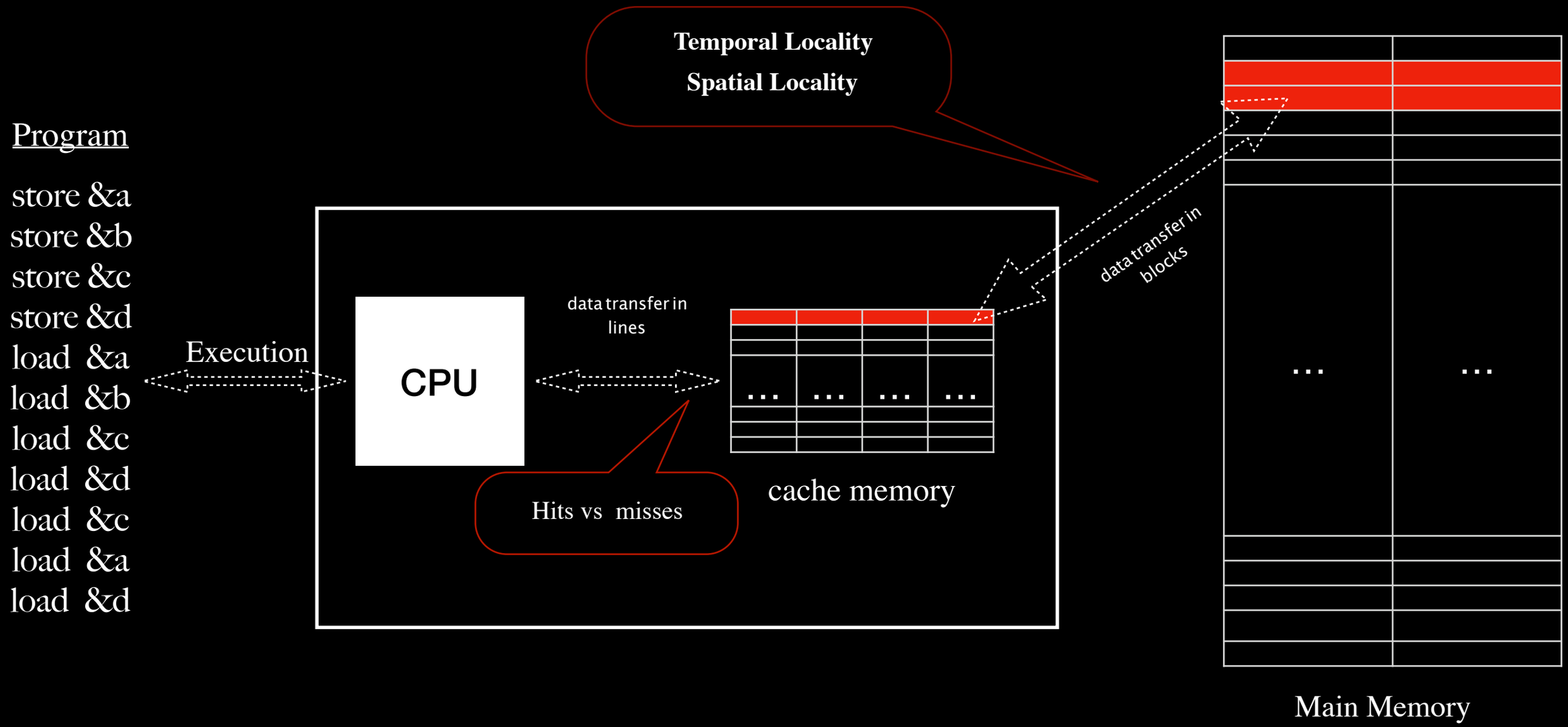
Example: Cache-Oblivious Priority Queue and Graph Algorithm Applications

vs

manipulating the hierarchy characteristics to get the best performance out of programs

Example: Semantic Locality

Introduction: Semantic Locality

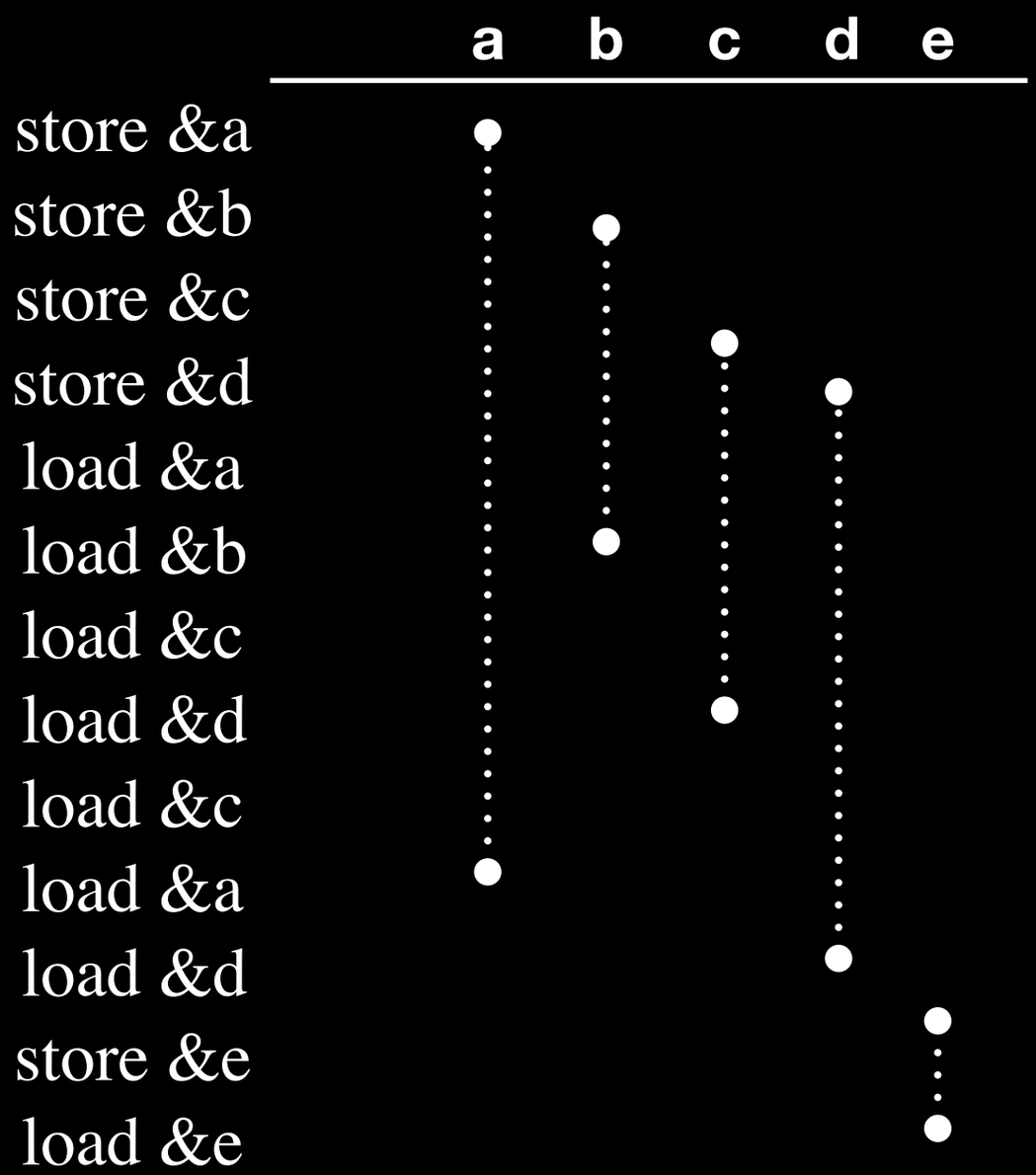


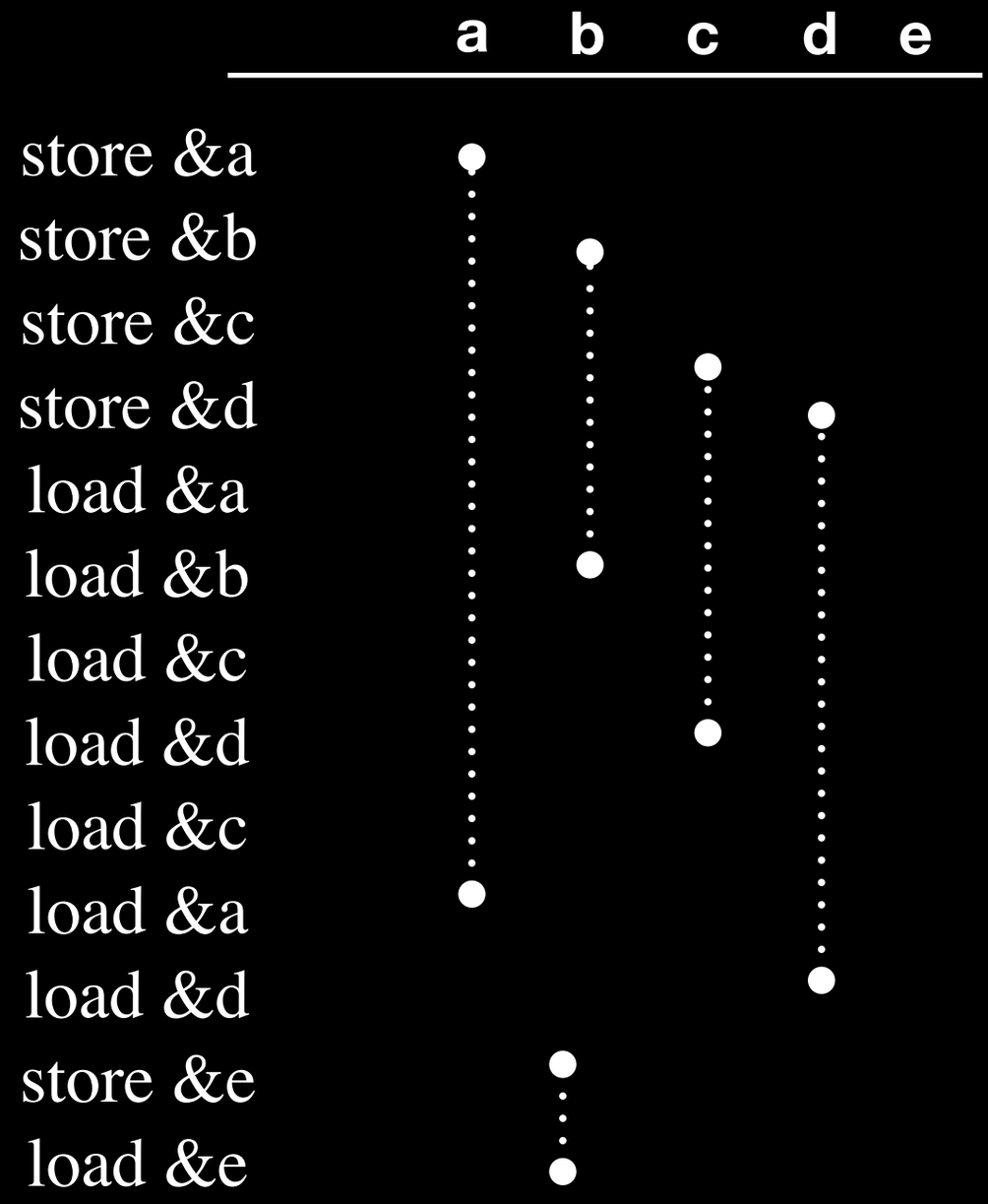
Constraints:

- Memory cache speed vs main memory speed.
- Cache capacity is limited (LRU strategy).

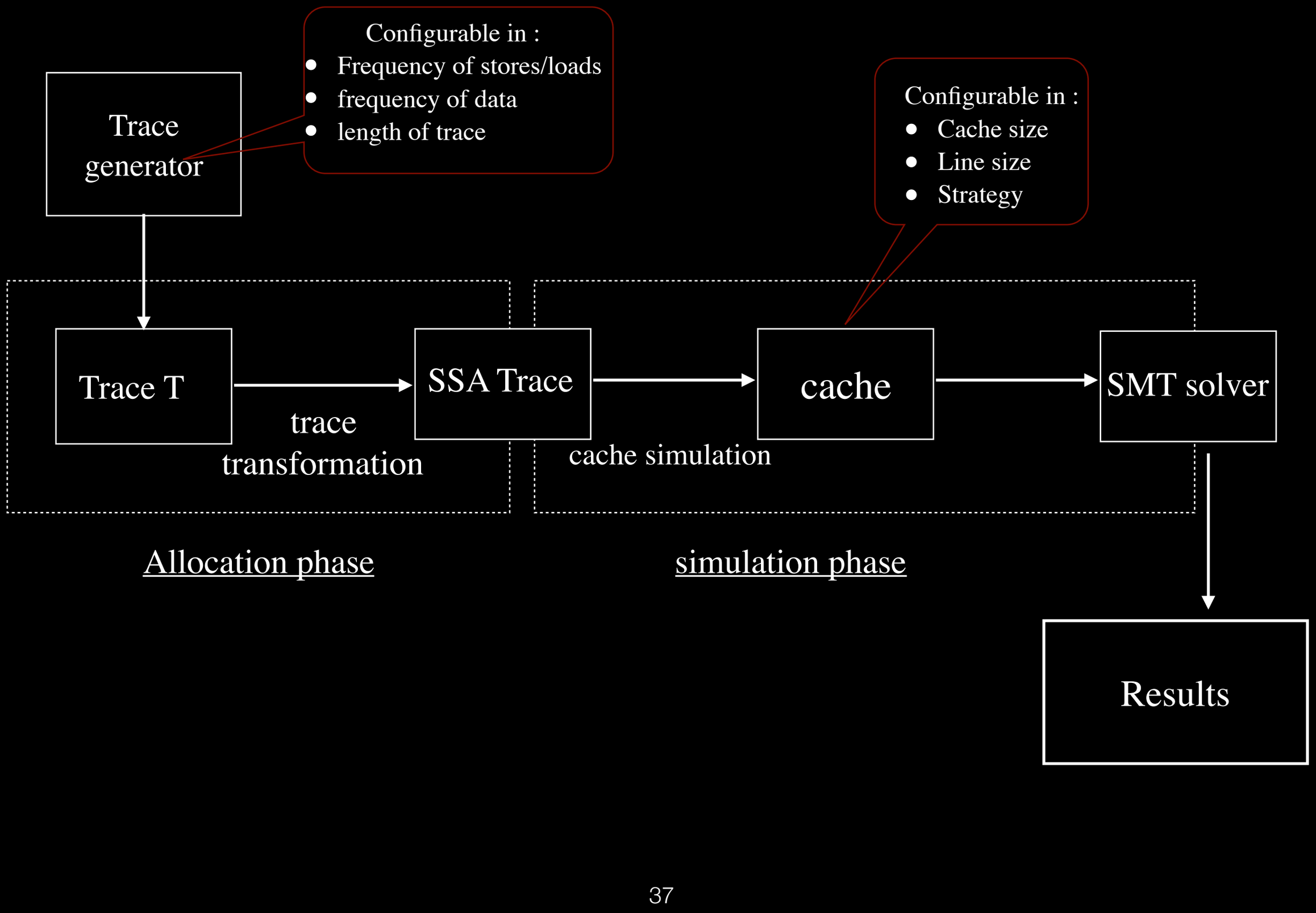
Goal: Semantic Locality

How can we influence execution speed just by changing the addresses used by a program?
Can recycling addresses improve execution time?





Semantic Locality: Implementation



Questions?