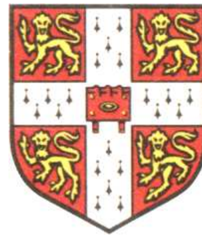
Cache Optimisation of Dynamic Recursive Data Structures



Mark Adcock
University of Cambridge
cmsa2@cam.ac.uk

MMW06: 10 November 2006

Introduction

- Performance of a large Recursive Data Structure (RDS) depends on good cache behaviour

- Eg:
  - Lookup in binary tree can be $2\times$ slower if layout poor
  - Depth First Search in binary tree can be $10\times$ slower!

Static RDSs

- Use good data layout at allocation time

- Choose 'good' data layout using:
  - Programmer's knowledge of RDS
  - Profiling
  - Compile-time analysis

Dynamic RDSs

- Lay out well at allocation time, as for static RDS
- Original data layout may become inefficient when RDS structure changes
- Therefore, RDS performance may degrade

Approach #1: Cache-Aware RDSs

Programmer uses an RDS that is designed for good cache behaviour.

Eg instead of using vanilla binary search tree, use the more complicated tree described in Brodal et al *('Cache-Oblivious Search Trees via Binary Trees of Small Height', '02)*

Advantage: Most effective solution, since cache behaviour considered from the start

Disadvantages:

- Contradicts 'Code first, optimise later' principle
- Cache-Aware RDSs are hard to design
- Every structural change to RDS must be careful with data layout

Approach #2: Locality-Improving Garbage Collectors

Programmer ignores cache behaviour, and relies on a good runtime environment to improve data layout.

Eg Chen et al's copying GC *('Profile-guided proactive garbage collection for locality optimization', '06)*

- Online profiling identifies hot objects, and access patterns
- Moves related hot nodes into the same block to improve locality

Advantages:

- Programmer can ignore cache behaviour
- Some of the work can be combined with normal GC work

Disadvantages:

- Poor quality of layout for programs that don't fit in with the hot/cold object classification scheme
- Sampling overhead
- Inefficient for programs that don't require a GC
- memory-hungry [possibly fixable?]
- interrupts program work [possibly fixable?]

Approach #3: Explicit Data Movement

Programmer

- writes program using any RDS
- later, inserts code which move objects at runtime to improve data layout

Requires programmer knowledge of a good layout for RDS

This knowledge could come from a 'cookbook' of optimisation, or profiling.

Eg, Linked list: Programmer observes that laying out nodes in access order is efficient, and so inserts this code:

Every N operations: walk list, moving nodes into new block of memory. Then free old block

Advantages:

- Fairly simple to implement
- Do not need to understand structural changes to RDS
- All we need to know is a good layout for RDS

Disadvantages:

- Have to ‘tune’ N
- Unnecessarily copies parts of the list that already have good layout
- Interrupts program work
- May use twice as much memory because of fragmentation

We present an Explicit Data Movement optimisation for

trees accessed using lookup

with

low overhead
low extra memory usage
low program interruption

Example C Program

Build complete binary search tree, using pre-order. Each node looks like:

Key	Value	Child Ptr 1	Child Ptr 2

The main loop consists of repeated alternating random lookups, deletions and insertions

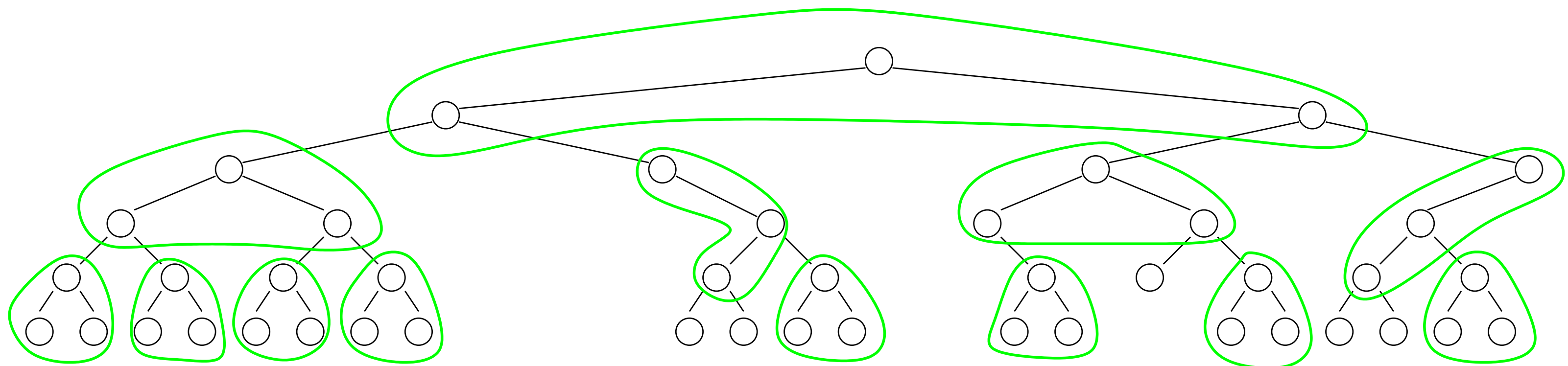
Applying Explicit Data Movement

1. Pick good layout
2. Insert Code

Applying Explicit Data Movement

1. Pick good layout ← Fill cache lines using Breadth First Search

---



(Usually called 'clustering')

Applying Explicit Data Movement

1. Pick good layout $\leftarrow$ Fill cache lines using Breadth First Search

---

Formally:

1. Perform Breadth-First Search to fill cache line, then deal with subtrees recursively
2. If < 3 nodes found, place nodes wherever there is space in memory

This layout uniquely specifies the location of each 'cluster'

Applying Explicit Data Movement

1. Pick good layout ← Fill cache lines using Breadth First Search
2. Insert Code

We insert code into (read-only) lookup function that:

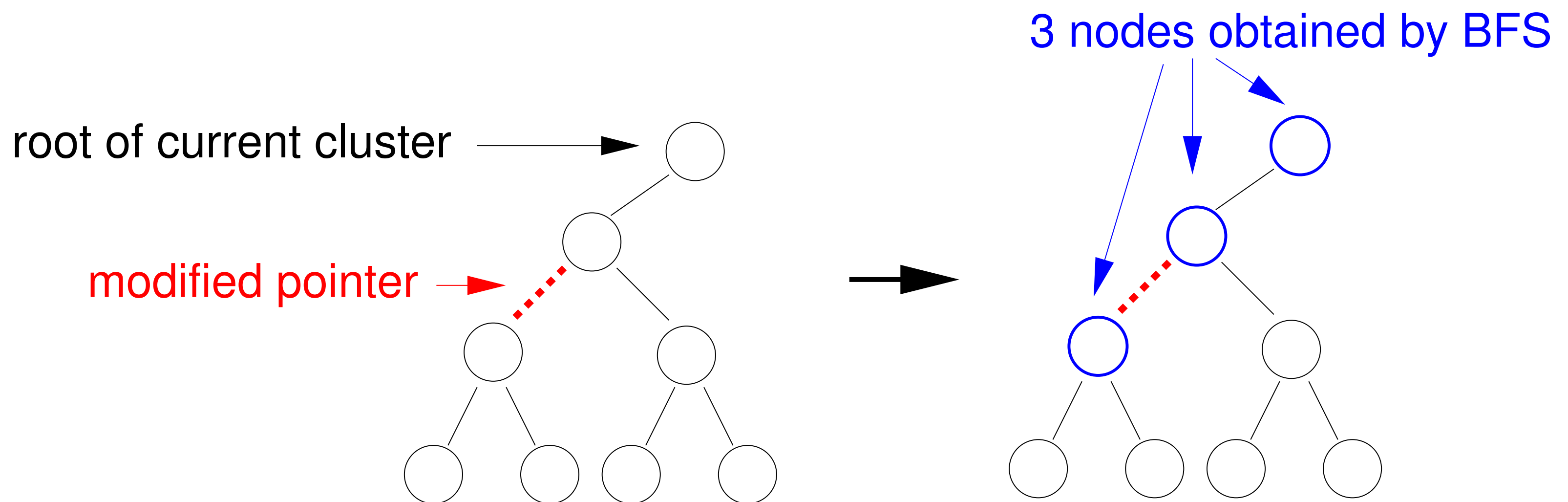
1. Detects changes to the structure of the RDS
2. Inspects data layout if structure changes
3. Moves nodes if layout is poor

Detecting Structural Changes

- Use low bit of each child pointer as 'modified' bit
- Use compiler to transform the whole program to:
  - set the bit on pointer write
  - mask the bit on pointer read
- Inserted code must clear the bit once it has dealt with the structural change
- Checking the bit in the lookup loop has a low overhead ($\approx 2\%$)

Inspecting data layout

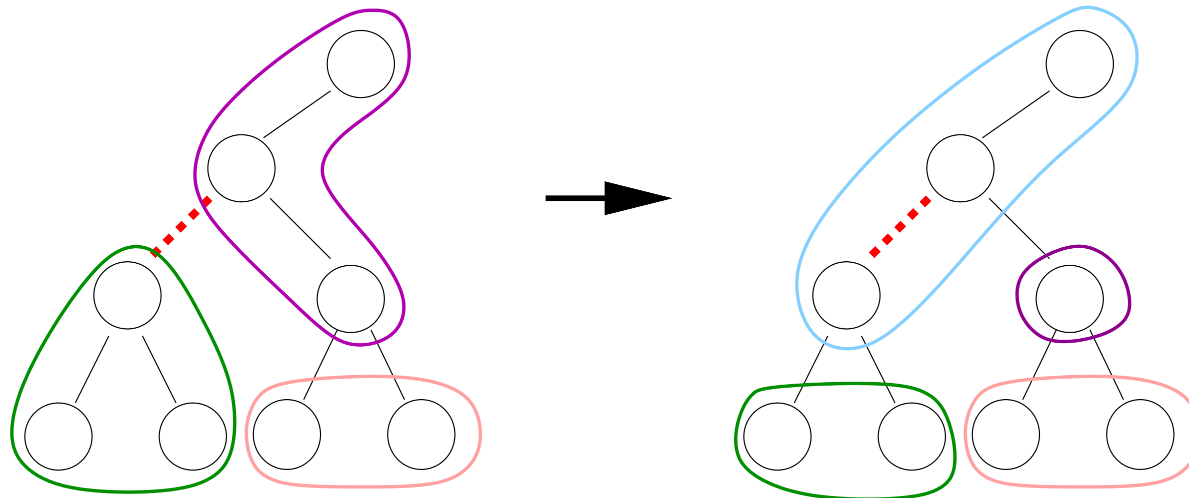
When encounter a set modified bit, breadth first search 3 nodes from the root of the current cluster.



We know where the root of the current cluster is, because the position of clusters is unique for each tree

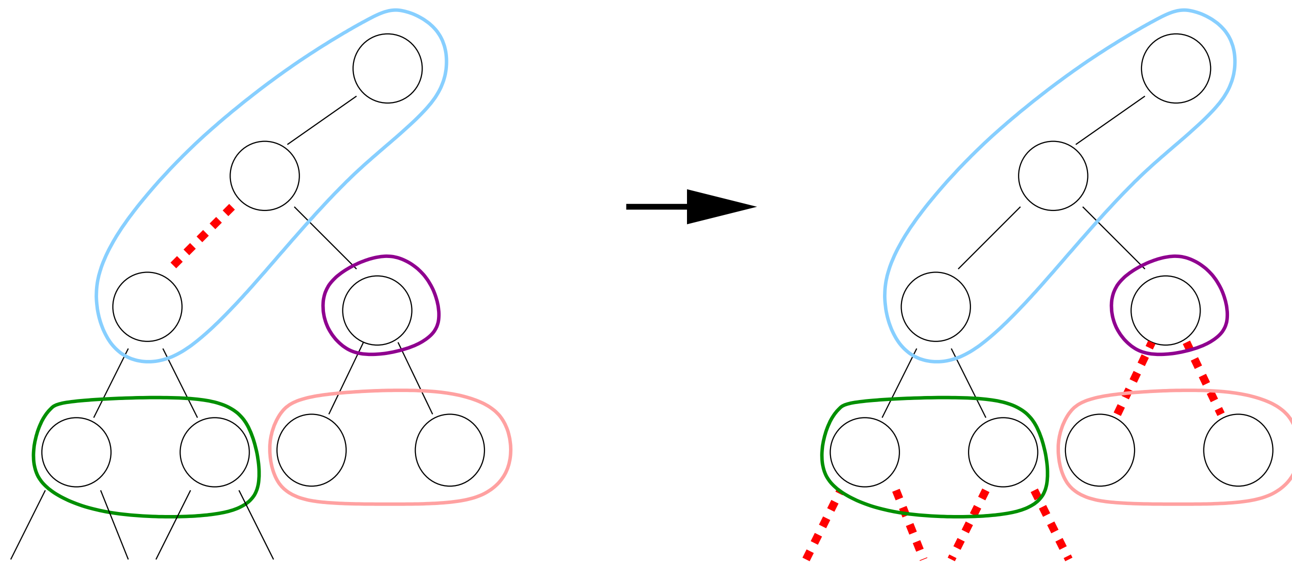
Moving Nodes - 1

- If the 3 nodes are in same line, layout is ok
- Otherwise, move all 3 nodes into empty line:



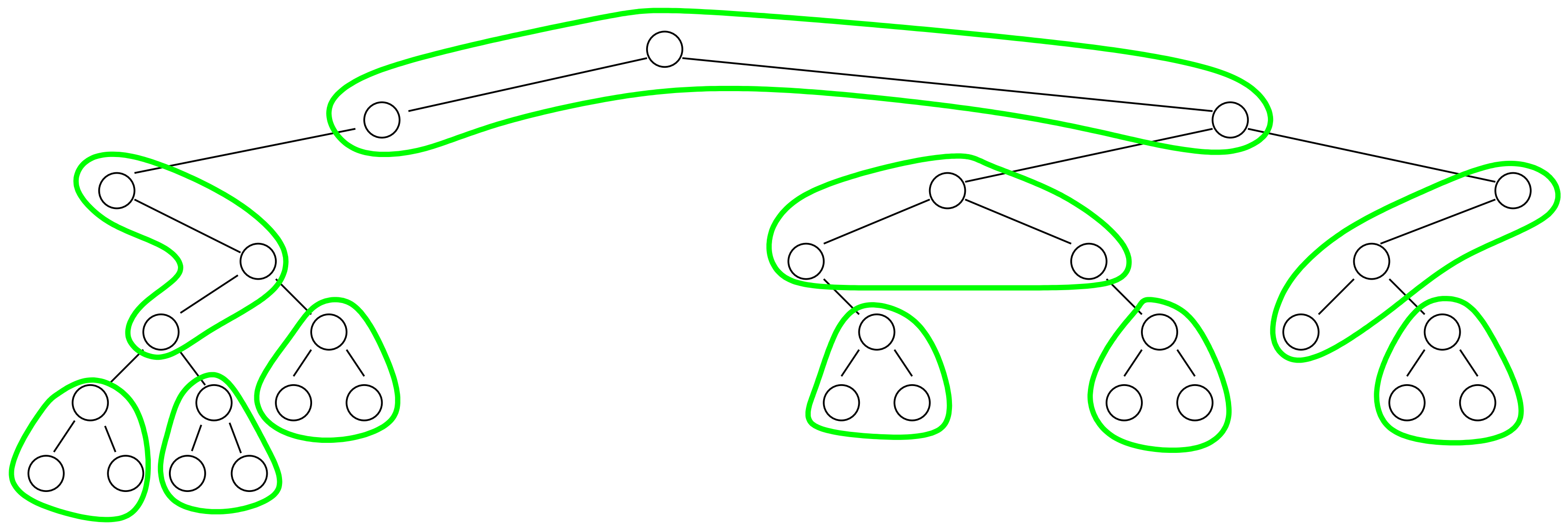
Moving Nodes - 2

Finally: propagate the bits *beyond* the boundary of the cluster:



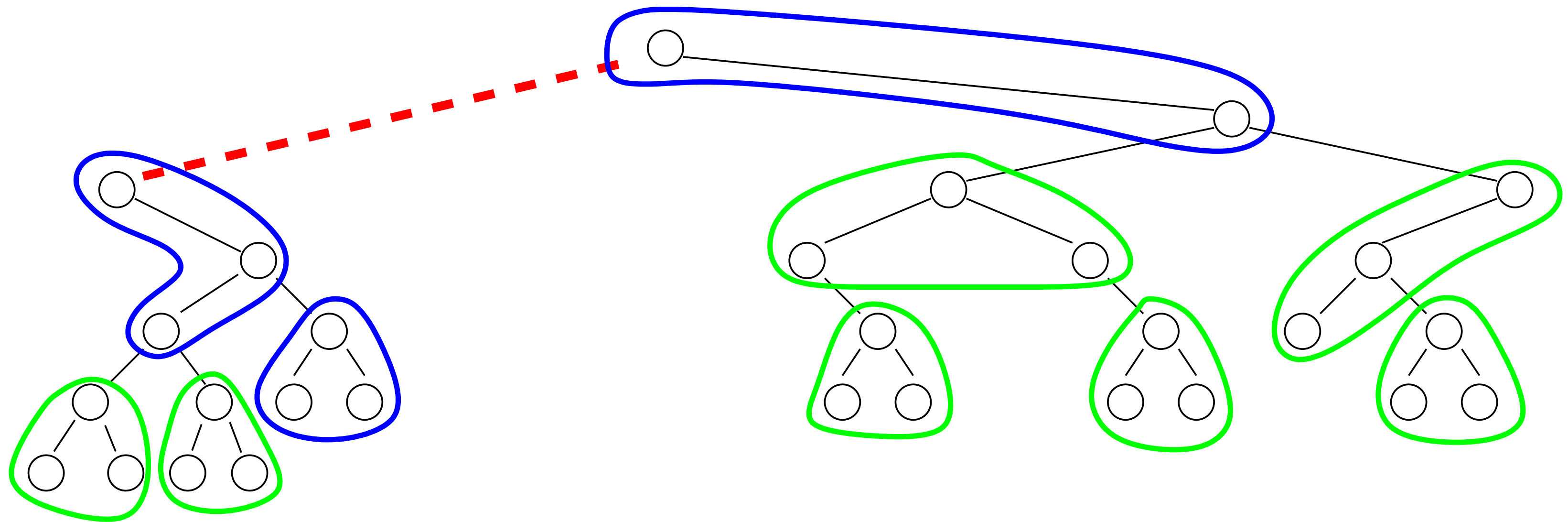
Subtrees dealt with when propagated bits are encountered

Cluster - Example



Now demonstrate the effect of deleting a node from a correctly-clustered tree. We will show the data movement during a typical lookup operation.

Cluster - Example



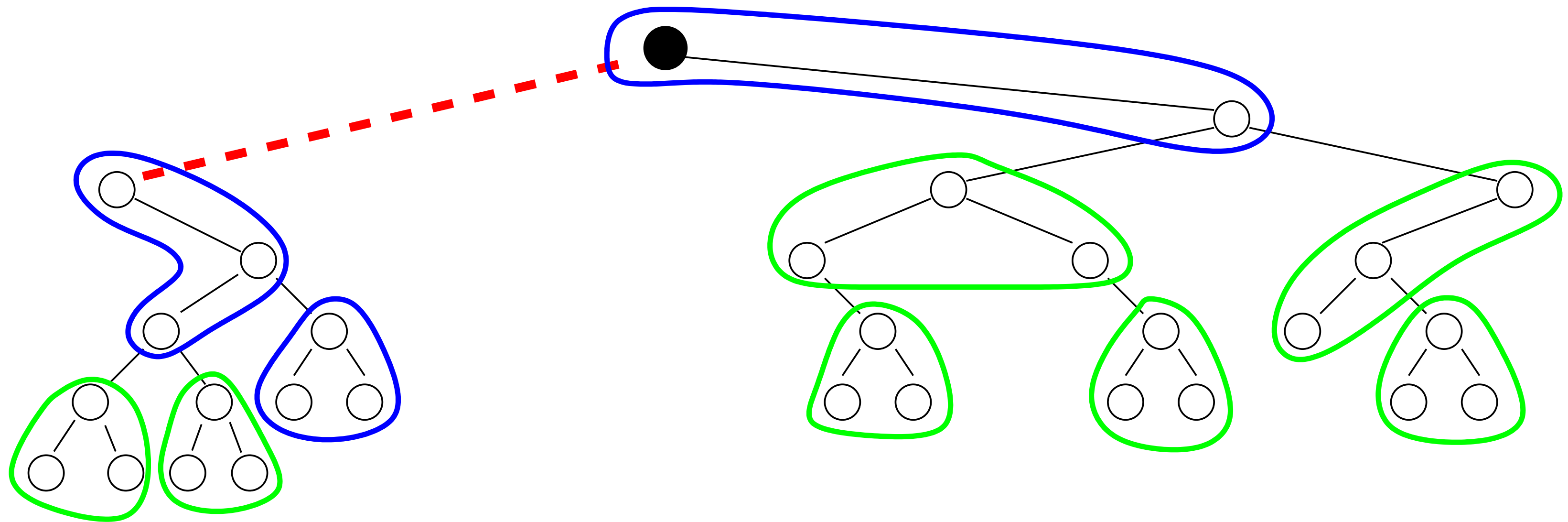
black nodes: path of lookup operation

red pointers: are 'modified'

green clusters: correct clusters

blue clusters: incorrect clusters

Cluster - Example



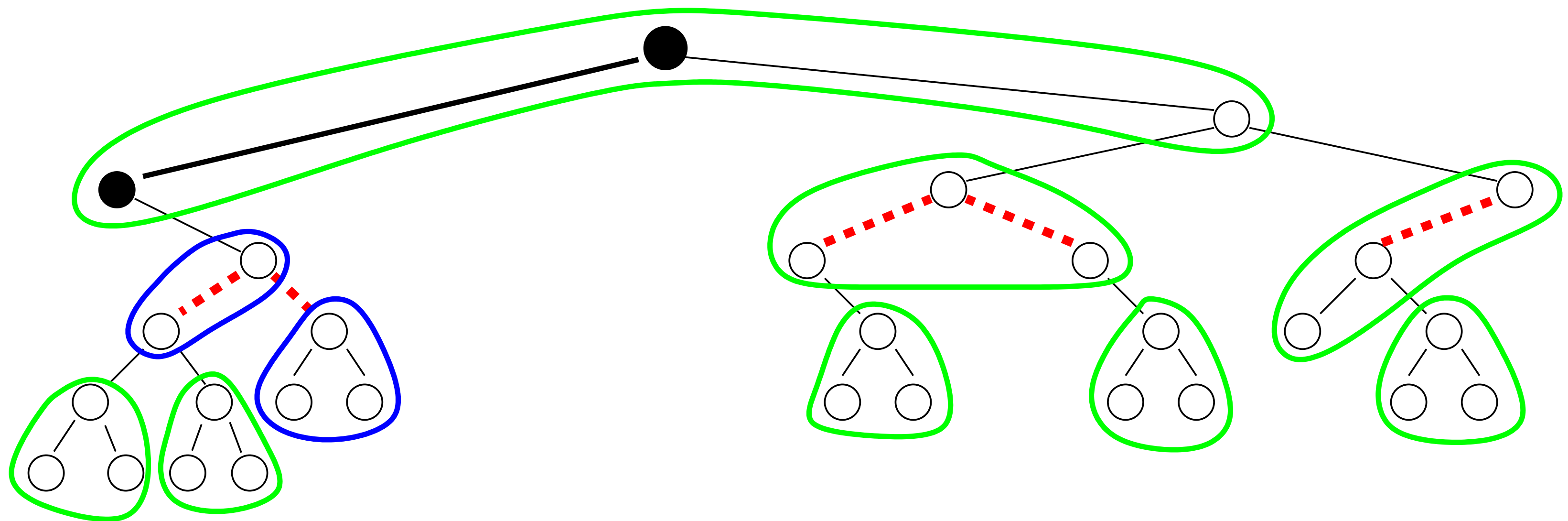
black nodes: path of lookup operation

red pointers: are 'modified'

green clusters: correct clusters

blue clusters: incorrect clusters

Cluster - Example



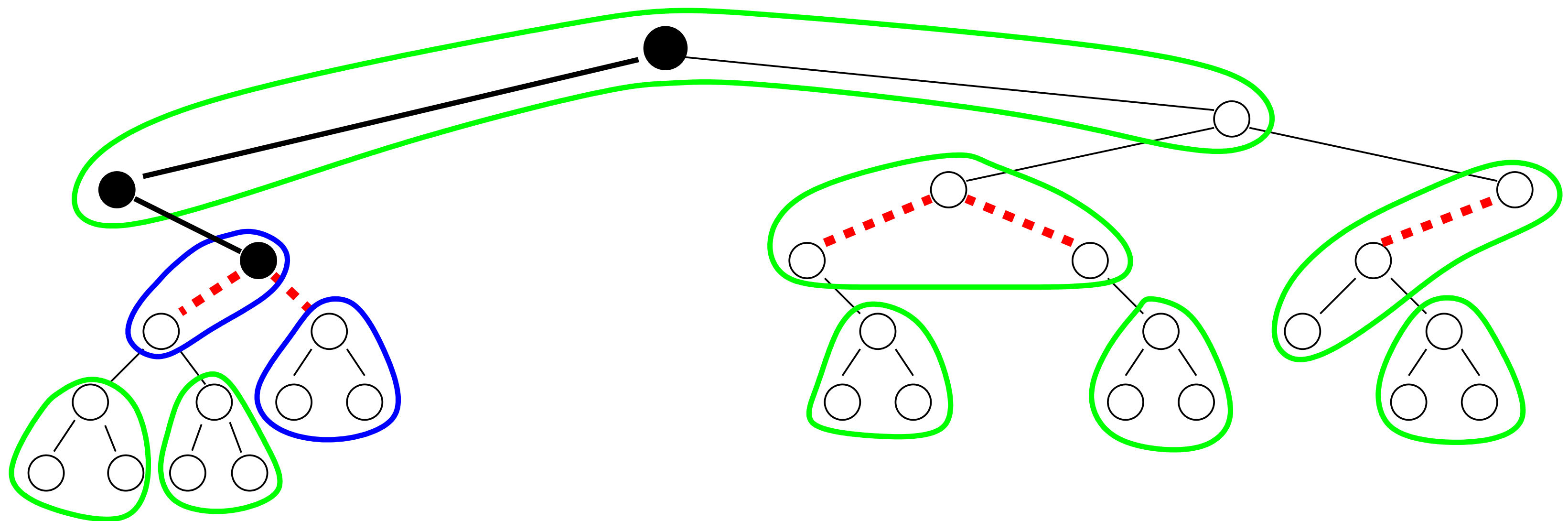
black nodes: path of lookup operation

red pointers: are ‘modified’

green clusters: correct clusters

blue clusters: incorrect clusters

Cluster - Example



black nodes: path of lookup operation

red pointers: are ‘modified’

green clusters: correct clusters

blue clusters: incorrect clusters

Cluster - Example

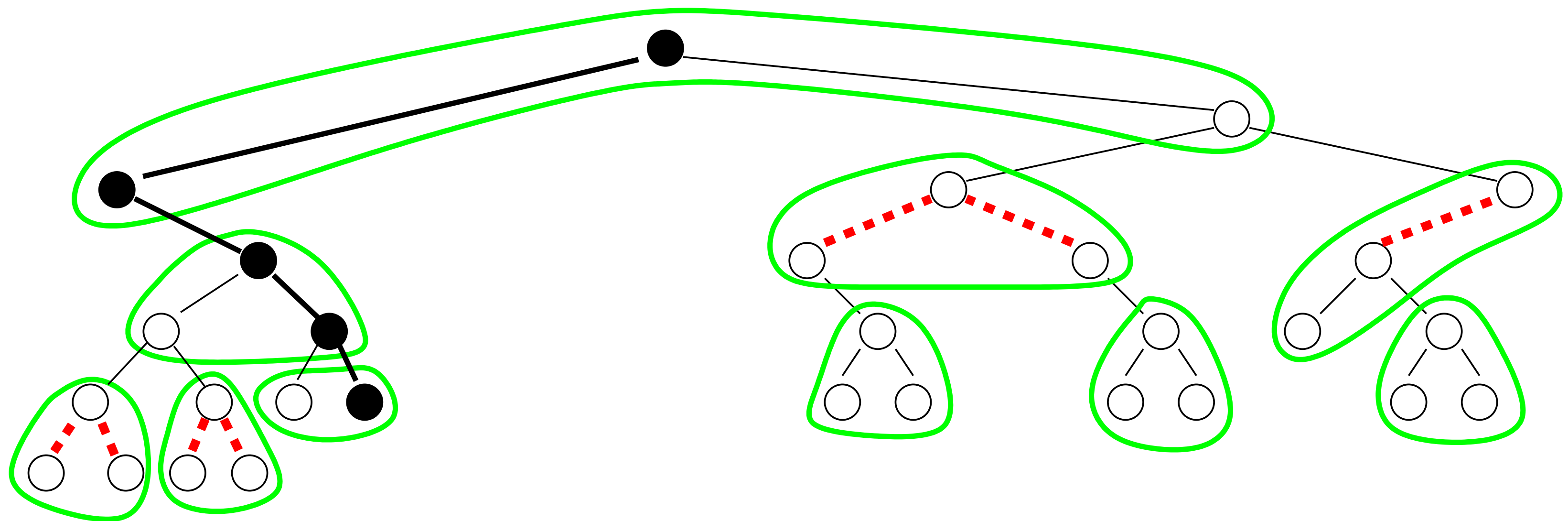
black nodes: path of lookup operation

red pointers: are 'modified'

green clusters: correct clusters

blue clusters: incorrect clusters

Cluster - Example



black nodes: path of lookup operation

green clusters: correct clusters

red pointers: are 'modified'

blue clusters: incorrect clusters

- Simple method: Clusters not repaired, always rebuilt
- Works because of the distribution of structural changes
- As long as there's a low expected amount of reclustering, the nature of the structural changes doesn't matter
- It is this that determines the applicability of the optimisation

- Node movement safe because programmer has guaranteed there are no pointers into the tree

- A simple custom allocator is used allow efficient node movement and allocation of free lines

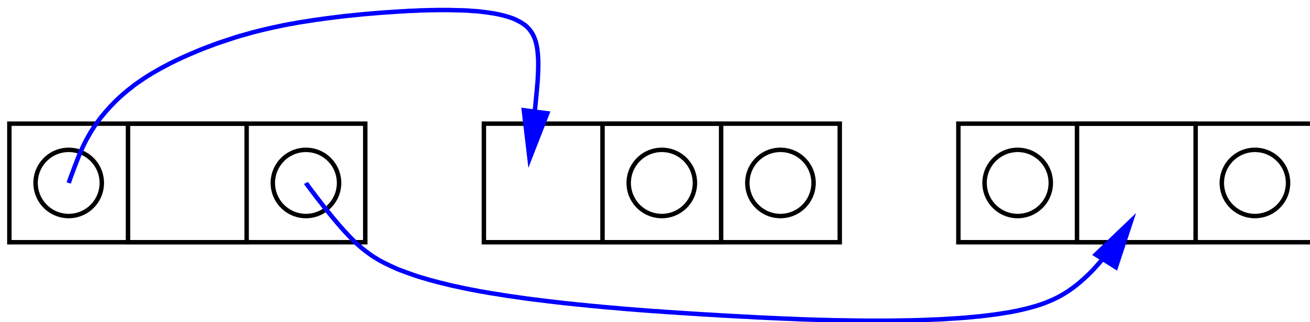
Bounding Memory Usage - 1

- Bound number of lines in use
- Fragmentation may cause a lack of empty lines, stopping clustering
- compact nodes to produce empty lines

Bounding Memory Usage - 2

What we'd like to do:

Move the nodes in a partially-filled line to somewhere else, to empty the line



Bounding Memory Usage - 3

Why we can't do this:

- We cannot move a node until its parent has been found
- Nodes do not have parent pointers
- Thus cannot empty an arbitrary line in one step

Bounding Memory Usage - 4

How to detect parents?

- Could walk the RDS
- No need, however, because eventually lookup will visit every node

Bounding Memory Usage - 5

How we compact nodes:

Use a buffer of nodes from partially-filled lines with known parents.

> When a node n in a partially-filled line is found during lookup: If buffer contains enough nodes to fill the line, move them into the the line (line is filled). Otherwise, add node n to buffer

Bounding Memory Usage - Example

a	b	c

d		

	e	f

g	h	

	i	

Buffer

Bounding Memory Usage - Example

a	b	c

d		

	e	f

g	h	

	i	

e

Buffer

Bounding Memory Usage - Example

a	b	c

d		

	e	f

g	h	

	i	

e h

Buffer

Bounding Memory Usage - Example

a	b	c		d	e	h				f		g					i	

Buffer

Bounding Memory Usage - Example

a	b	c

d	e	h

		f

g		

	i	

f

Buffer

Bounding Memory Usage - Example

a	b	c

d	e	h

		f

g		

	i	

f g

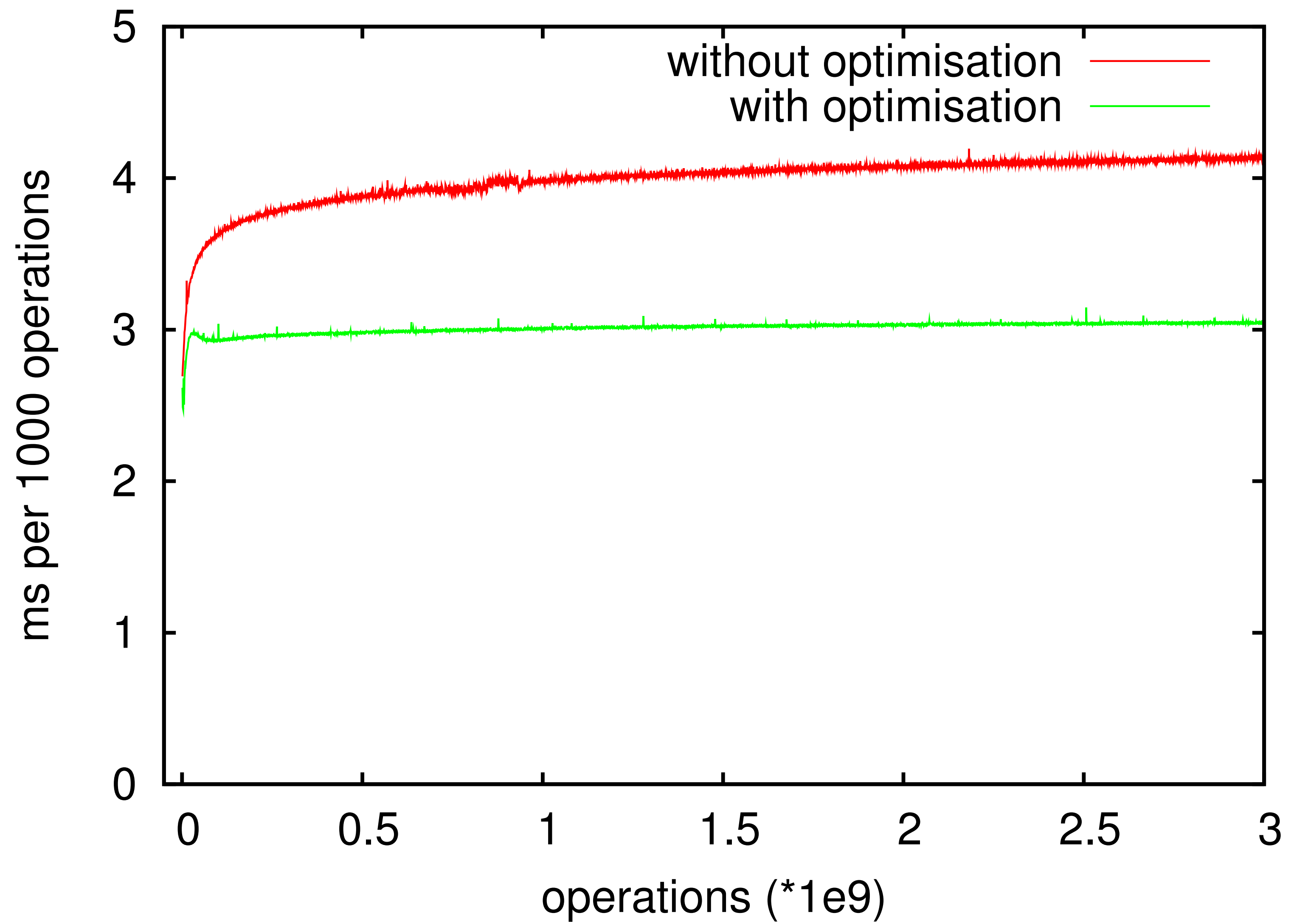
Buffer

Bounding Memory Usage - Example

a b c | d e h | | | f i g

Buffer

- Allocate large complete tree using pre-order
- Mainloop performs repeated random lookups, deletions and insertions
- Insert code into program to time each 1000 iterations
- Both balance and layout of tree will get worse, increasing execution times


without optimisation
with optimisation
ms per 1000 operations
operations (*1e9)
0
1
2
3
4
5
0
0.5
1
1.5
2
2.5
3

- Lookup/Delete/Insert time decreased by 25%

- Memory Usage: Best performance from only 5% extra memory
  - Less: slower due to slower rate of production of empty lines
  - More: *slightly* slower due to low node density

- Low Overhead: Only 8% of time is spent executing inserted code

Interruption - 1

To discuss interruption, must consider:

- The scale in which we are interested: Eg 1e6 operations
- The worst interruption we are prepared to accept: Eg 10% increase

And try to impose:

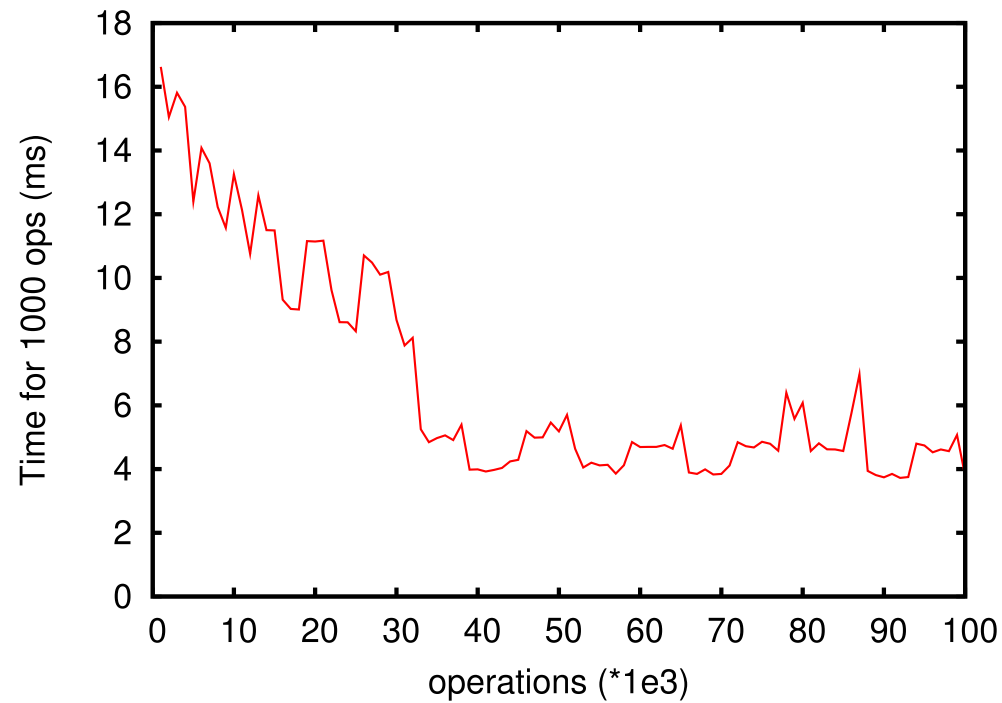
$$time(\text{1e6 ops with optimisation}) < time(\text{1e6 ops without optimisation}) \times 1.1$$

Interruption - 2

- Interruption usually low, even at the scale of 1000 operations ($\approx 1/100th$ second)
- This is because node movement work is very finely interleaved with normal program work
- The worst case arises when layout has been completely destroyed between lookups

Interruption - 3

Time for 1000 operations:

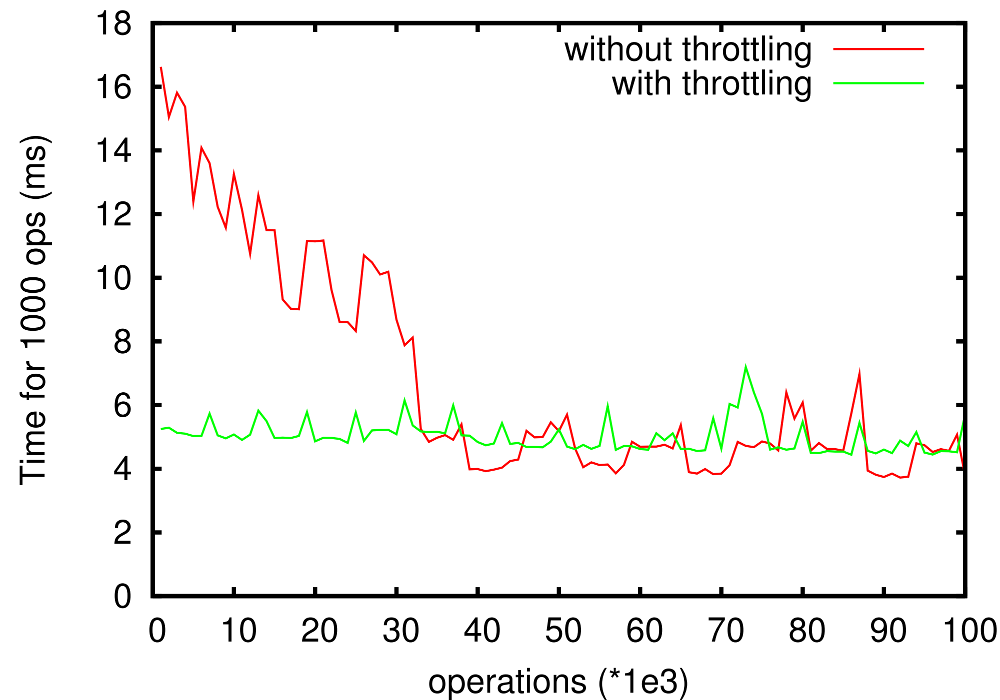


Interruption - 4

Impose 10% bound in execution time by throttling number of extra line fetches caused by inserted code

- Periodically sample number of lines fetched during normal program work, n
- Throttle node movement so only αn lines are fetched
- Lower $\alpha \Rightarrow$ lower interruption

Interruption - 5



Note: Throttling does not effect the 25% decrease in execution time!

Conclusion

- Inserting code is an effective way of improving cache behaviour after program development

- Applicability:

  - This optimisation is for branching traversals of trees with favourable distribution of structural changes

  - Some simple modifications to deal with pages instead of lines yielded good improvement for depth first search