Technical Report

UCAM-CL-TR-757
ISSN 1476-2986

Number 757

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Improving cache performance by runtime data movement

Mark Adcock

July 2009

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted June 2009 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Christ's College.

Some figures in this document are best viewed in colour. If you received a black-and-white copy, please consult the online version if necessary.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Improving cache performance by runtime data movement

Mark Adcock

Abstract

The performance of a recursive data structure (RDS) increasingly depends on good data cache behaviour, which may be improved by software/hardware prefetching or by ensuring that the RDS has a good data layout. The latter is harder but more effective, and requires solving two separate problems: firstly ensuring that new RDS nodes are allocated in a good location in memory, and secondly preventing a degradation in layout when the RDS changes shape due to pointer updates.

The first problem has been studied in detail, but only two major classes of solutions to the second exist. Layout degradation may be side-stepped by using a 'cache-aware' RDS, one designed to have inherently good cache behaviour (e.g. using a B-Tree in place of a binary search tree), but such structures are difficult to devise and implement. A more automatic solution in some languages is to use a 'layout-improving' garbage collector, which attempt to improve heap data layout during collection using online profiling of data access patterns. This may carry large performance, memory and latency overheads.

In this thesis we investigate the insertion of code into a program which attempts to move RDS nodes at runtime to prevent or reduce layout degradation. Such code affects only the performance of a program not its semantics. The body of this thesis is a thorough and systematic evaluation of three different forms of data movement. The first method adapts existing work on static RDS data layout, performing ad-hoc single node movements at a program's pointer-update sites, which is simple to apply and effective in practice, but the performance gain may be hard to predict. The second method performs infrequent movement of larger groups of nodes, borrowing techniques from garbage collection but also embedding data movement in existing traversals of the RDS; the benefit of performing additional data movement to compact the heap is also demonstrated. The third method restores a pre-chosen layout after each RDS pointer update, which is a complex but effective technique, and may be viewed both as an optimisation and as a way of synthesising new cache-aware RDSs.

Concentrating on both maximising performance while minimising latency and extra memory usage, two fundamental RDSs are used for the investigation, representative of two common data access patterns (linear and branching). The methods of this thesis compare favourably to upper bounds on performance and to the canonical cache-aware solutions. This thesis shows the value of runtime data movement, and as well as producing optimisation useful in their own right may be used to guide the design of future cache-aware RDSs and layout-improving garbage collectors.

Acknowledgements

I would like to thank my supervisor Martin Richards and advisor Alan Mycroft, and others at the Computer Lab for their guidance and advice during the preparation of this thesis. Many thanks are due also to my parents, my brother and Louise.

Contents

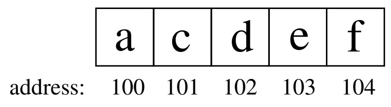
Chapter 1

Introduction

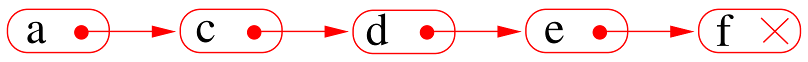
This chapter introduces the problem addressed by this thesis. Current approaches to the problem are briefly discussed, and a gap in knowledge is identified. Finally, the content of the thesis is outlined, its main contributions listed, and a chapter plan is given.

1.1 Recursive data structures

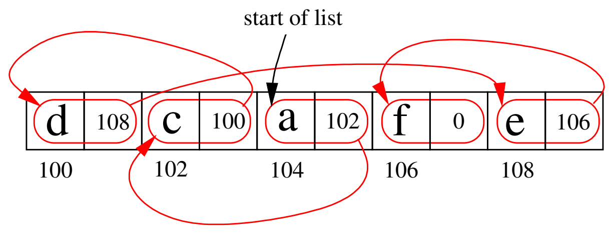
Computer memory consists of billions of same-sized cells into which data can be placed. Each cell is referred to by number (its *'address'*), and the cells can be accessed in any order. Structured data can be created using the address of each cell. For example, a program may wish to store the letters a, c, d, e, f in order. This can be done by storing the i^{th} number in the $x + i^{th}$ cell (an *'array'*), as shown below for $x = 100$:



Inserting the letter b in the correct place requires moving c, d, e and f one cell to the right; as the array grows, the amount of work required for insertion also grows. An alternative approach is to use a *'recursive data structure'* (or *'RDS'*), which is a method of representing unbounded amounts of data using small *'nodes'* that are linked together. Each node contains a small, fixed amount of data and some number of *'pointers'* to other nodes:



The nodes are stored in memory in any order. The pointer is achieved by storing the address of the next cell:



This is known as a *'linked list'*, each node consisting of a single data field and a single pointer field. Inserting b into the sequence now only requires creating a new node anywhere in memory, setting its data field to b and its pointer field to point to c, and then setting a's pointer to point to the new node.

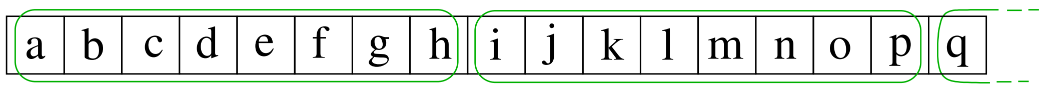
In the array, the position of the data in memory affects the meaning of the structure, but in the linked list the position of individual nodes in memory (the *'layout'*) is unimportant. However, on most architectures the layout affects the time it takes for the processor to fetch nodes from memory. If the layout is poor, scanning along the list will be considerably slower than in the array, despite insertion being quicker. Thus in terms of performance neither recursive data structures nor more rigidly shaped structures (such as arrays) are superior. Furthermore, the linked list uses twice as much memory to store the same data. In practice, RDSs are very widely used due to their flexibility and ease of implementation.

1.2 Data layout

Transferring data from memory to the processor takes time, and to reduce the time requires more expensive hardware. It is not feasible to build the memory out of the fastest memory available, and a good compromise is to insert smaller *'cache'* memories between the main memory and the processor. Typically several caches are used, forming a *'memory hierarchy'*, with the smallest, fastest caches at the top, nearest the processor.

To reduce overhead, the hardware does not transfer memory cell-by-cell, but in small groups. When the processor attempts to access a cell, the hardware looks for its group, starting in the smallest, fastest cache at the top of the memory hierarchy, moving down to the larger, slower caches until the main memory is reached[1]. When the group is found, it is copied to the top of the hierarchy. Eventually, the space a group occupies in a cache will be taken over by another group that has been accessed more recently – the original group is *'evicted'*. Performance can be improved by reducing the number of transfers, which can be achieved by increasing the number of uses of groups of cells before their eviction.

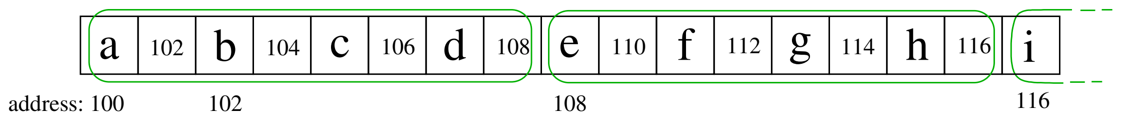
On most architectures the groups are identically- and fixed-sized *'blocks'* of neighbouring cells. An array structure has particularly good *'cache performance'*, because all the cells in a block are used before it can possibly be evicted: (the green lines encircle cells in the same block)



Here only one block transfer is needed for every eight letters, and this remains true whenever extra data is inserted into or deleted from the array.

The linked list's cache performance depends on the layout. In the worst case, where nodes are scattered throughout memory, each block is only used once before it is evicted. In the best case, where nodes are close in memory, four nodes fit into a block, and so only one block transfer is needed for every four letters:

[1]If *'virtual memory'* is being used, the cell may not be found in the main memory, and is instead located somewhere on disk.



Block sizes are typically larger than in this example, and a poor layout for a linked list is up to 40 times slower on normal desktop hardware[2]. Even if the linked list is created with a good layout it may quickly degrade to a poor layout. This *'layout degradation'* is a potential problem for all recursive data structures.

This thesis presents techniques to prevent layout degradation of RDSs by adding code to a program that moves nodes in memory as the RDS is updated (*'runtime data movement'*). The aim of the data movement code is to improve the performance of a program, without otherwise altering its behaviour.

1.3 This thesis

There are two distinct past approaches to preventing the layout degradation of RDSs.

Firstly, instead of using a simple RDS, a more complex *'cache-aware'* RDS may be used. These RDSs are designed from the outset with good cache performance in mind, and so (in principle) will yield better performance than any attempt to shore up an existing RDS using runtime data movement. However, cache-aware RDSs are considerably more complex to implement than traditional RDSs, and also require the programmer to consider cache behaviour from the outset rather than optimising the program once it is written. One aim of this thesis is to see how close to the performance of a cache-aware RDS one can get using a combination of a simple RDS and runtime data movement.

Secondly, a *'garbage collector'* (or *'GC'*) may be adapted to maintain a good layout. In overview, some programming languages rely on the programmer inserting instructions to free memory once the program is done with it, but others do not. In such languages *'garbage'*, memory that the program can no longer reach because no pointer points to it, is discovered and freed by the garbage collector – another program running at the same time as the main program. The garbage collector works by exhaustively exploring all the memory the main program can reach, and then freeing anything which isn't reachable. Often, objects are moved around in memory during this process, and this movement can be co-opted to improve the data layout of an RDS. Such a collector is called a *'layout-improving garbage collector'* or *'LIGC'*.

The most common approach to layout-improving garbage collection is to work out a new layout for the program's data by observing how the data is used as the program runs, rather than requiring explicit instruction from the programmer. The advantage of an LIGC is thus that it attempts to maintain a good data layout without extra effort from the programmer. This automation comes at a cost – in general, garbage collection can incur large time and memory overheads, reducing the effect of any improvement in layout, and may pause the normal work of a program for long periods, which is is undesirable for some applications.

Thus, there is a gap in knowledge. Cache-aware data structures are hard to implement, and LIGCs, which perform runtime data movement, may have large time and memory overheads and may pause normal program work. In this thesis we investigate other forms

[2]Pentium 4. See §3.1.

of runtime data movement, which may also be viewed as ways to synthesising new cache-aware data structures.

1.3.1 Aim of the thesis

The aim of this thesis is to investigate the opposite end of the runtime data movement spectrum to an LIGC. The runtime data movement of an LIGC is achieved within an existing framework, and thus must require little or (more usually) no programmer involvement. This thesis takes the opposite approach, focusing on discovering the best forms of runtime data movement for a particular RDS, without providing tools to apply these optimisations to a program automatically (although this thesis does discuss how such tools may be created). Furthermore, the optimisations in this thesis attempt to improve performance without using large amounts of extra memory and without significant pausing of normal program work.

The body of this thesis is a clear and systematic development of three families of optimisations, using as a starting point existing methods for improving the static (initial) layout of an RDS. As well as producing optimisations useful in their own right, this thesis is intended to guide the design of future cache-aware RDSs and fully-automatic solutions such as LIGCs, and to explore the limits of runtime data movement.

1.3.2 Achieving the aim

To allow the evaluation a large number of different techniques, this thesis makes several compromises:

1. **Benchmarks:** Two simple *microbenchmarks* are used, based on linked lists and binary trees. These are fundamental structures in their own right, but should also be thought of as representative of two major classes of data access pattern (linear and branching). Past work on automatic runtime data movement has focused on benchmark suites of far larger programs. By concentrating on these common and easily-understood structures, the thesis provides greater insight than if larger benchmarks were used. Indeed, applying the techniques of this thesis produces rich behaviour.

2. **Hand-application:** The aim of this thesis is to identify useful forms of runtime data movement, rather than to investigate how to apply them automatically. Thus, all data movement code is applied to a program by hand. Together with the use of smaller benchmarks, permitting hand-application of optimisation code allows many different techniques to be investigated.

3. **Tuning:** All parameters of an optimisation are tuned by re-running the benchmark many times, rather than requiring the optimisation to select its own parameters as the program runs. This ensures that the efficacy of an optimisation is maximised, and makes optimisation code simpler, allowing more techniques to be evaluated.

4. **Updating parent pointers:** All the techniques in this thesis assume the existence of a mechanism to move nodes quickly and safely – in other words, at some selected program points where a node may be moved, the applier of the optimisation has arranged for all pointers to a node (the node's *'parent pointers'*) to be updated if it is moved.

The benchmarks in this thesis avoid this issue by using structures where nodes have a single parent within the RDS, and by moving nodes at program points where any other pointers to nodes are known. In general, updating all pointers to a node may be achieved in many ways, including an exhaustive search as performed by a GC, at the cost of higher overhead and memory usage.

1.3.3 Contributions

This thesis makes the following contributions:

1. **The scale of the problem:** The severity and speed of layout degradation of fundamental RDSs is demonstrated.

2. **Exploration of runtime data movement:** This thesis addresses a number of different decisions that must be made when implementing a runtime data movement optimisation:

    1. The form of data movement: Several forms of data movement are compared. Firstly, data movement may be performed to correct the degradation caused by each individual pointer update, or it may be performed less frequently and en-masse. Secondly, data movement may be performed with the aim of producing an exact, well-defined layout, or to produce only an approximate layout whose quality is not easy to predict. It is demonstrated that all four combination of these binary choices are useful in different situations. It is also demonstrated that the correct level of the memory hierarchy to focus an optimisation at depends on both the RDS and the optimisation.

    2. Compaction: This thesis investigates techniques that perform data movement aimed not directly at improving layout but at creating empty contiguous regions of memory that can be used to improve layout in the future, and finds this *'compaction'* essential in some situations when an optimisation is not allowed a large amount of extra memory to work with.

    3. Complexity: Some techniques in this thesis involve a large number of changes to a program (e.g. perfect data movement – see below), but others do not (e.g. bulk data movement). In general, the more complex an optimisation, the harder it will be to apply automatically. This thesis investigates how complexity affects the efficacy of an optimisation.

    4. Memory usage and latency: This thesis investigates how both constraining memory and reducing the latency of small units of normal program work affect the design and performance of runtime data movement optimisations.

3. **Three families of techniques:** Three families of techniques are explored. The first and second produce simple and effective optimisations, while the third produces rather complex and often more effective optimisations aimed primarily at demonstrating some of the limits of runtime data movement. These optimisations compare favourably with, and sometimes outperform, the canonical cache-aware RDSs.

1. Reallocation: Memory allocators designed to improve the static layout of an RDS are adapted for dynamic use, producing simple ad-hoc optimisations that perform well and have naturally low memory usage and do not cause significant latency. In particular it is shown that an allocator must be chosen more carefully when intended for dynamic use.
2. Bulk data movement: From the starting point of the simplest data movement optimisation – periodically re-laying out the entire RDS – the latency of normal program work, memory usage and overhead are reduced systematically, using a mixture of novel techniques and techniques borrowed from garbage collection. It is shown that using compaction and embedding data movement in existing traversals of the RDS sometimes greatly improves performance.
3. Perfect data movement: Code is inserted into each pointer update site to perfectly restore a pre-chosen layout, producing complex but effective and stable optimisations which often out-perform the previous two techniques. The trade-off between layout quality and overhead is investigated. For binary trees use of this technique synthesises a new cache-aware RDS that outperforms (in practice) the canonical cache-aware structure.

1.3.4 Chapter plan

The chapters of this thesis are as follows:

1. **Introduction**
2. **Improving RDS cache performance:** Related work, and the approach taken by this thesis.
3. **Benchmarks, and methodology:** The benchmarks used in this thesis, and their unoptimised static and dynamic cache behaviour, and their cache-aware equivalents. Also, experimental methodology.
4. **Reallocation:** Ad-hoc pointer-update-time data movement, based on pre-existing memory allocators for static RDS layout.
5. **Bulk data movement:** En-masse re-laying out of parts of the RDS, borrowing techniques from GCs and novel techniques, including embedding data movement code in existing program loops and compaction to produce emptier blocks.
6. **Perfect data movement:** Restoring a chosen layout after each pointer update operation; a source of optimisation and of new cache-aware structures, and a demonstration of the limitations of runtime data movement.
7. **Evaluation:** An evaluation of each technique in isolation, together, and in comparison with the canonical cache-aware structures. Also, future work.
8. **Conclusion**

Some derivations from chapter 6 can be found in an appendix, followed by the bibliography and a list of terms.

Chapter 2

Improving RDS cache performance

In this chapter, we discuss the behaviour of the memory hierarchy of a modern desktop PC, and how in general to improve its performance. We then discuss various techniques from the literature which may be used to improve RDS memory hierarchy performance. Finally, we discuss the approach taken in this thesis.

2.1 The memory hierarchy

The memory hierarchy of a typical desktop PC has several levels. The top of the hierarchy, the processor, has only a few hundred bytes of local storage. The main memory (RAM) is typically $\sim$1GB. Between the processor and the main memory lie several smaller cache memories, usually at least two: an L1 cache of a few tens of kB in size (closest to the processor), and an L2 cache of size $\sim$1MB. As the hierarchy is descended, the memories get larger and slower – if RAM is exhausted, the disk ($\sim$1TB) can be used for extra storage, but is significantly slower and is not considered further in this thesis.

There are many different considerations when designing a cache, but the most important are captured by the 'ABC' model: Associativity, Block size and Capacity. Capacity C is the total size of the cache as given above, in bytes. The block size B, measured in bytes, is the unit of transfer between the cache and the level below in the hierarchy. For the L1 and L2 caches, the blocks are known as *'lines'*, around 32–256 bytes.

The cache is organised into C/BA distinct *'cache sets'*. The number of blocks of size B within each set is given by the associativity A. Every possible block that may be fetched from lower in the hierarchy can only be placed in one of the sets – the mapping is based on a simple transformation of the block's address (its location in the RAM). When the level above requests a particular block, the cache inspects the relevant set, returning the block if it is found (a *'hit'*), and fetches the block from the level below if it is not (a *'miss'*). The time taken for a miss, the *'miss penalty'*, is larger for lower levels of the hierarchy.

When a miss occurs some other block is likely to be evicted from the new block's cache set. The *'replacement policy'* governs how this choice is made. The most commonly used replacement policy is to evict (some approximation to) the *'least recently used'* block.

2.1.1 The Translation-Lookaside Buffer

Most architectures employ two distinct methods of referring to a particular cell in the RAM. The actual location of the cell – the address supplied to the hardware that makes up the RAM – is the *'physical address'*. The processor works instead in terms of *'virtual addresses'*. This allows data to be moved to disk when the RAM becomes full, and reduces fragmentation[1]. The bottom k bits of a physical address are the same as the virtual address, and so one may consider virtual addressing as a rearrangement of physical addressing using blocks of size 2^k, known as *'pages'*. The translation of virtual to physical addresses is contained in a *'page table'*, itself stored in RAM. To reduce the number of times translation entries must be fetched from RAM, a small number are stored in a cache memory known as the *'translation lookaside buffer'*, or *'TLB'*. The TLB does not store the content of a page (this will be stored on a line-by-line basis in the L1/L2 caches), merely a small amount of information used to locate the page in the RAM. With this distinction made clear, we may observe that in terms of performance the memory hierarchy behaves as if it has a cache with block size 2^k. Typically TLBs have just one set ($C = BA$), and the block size is 4096 ($k = 12$), much larger than the line size (32-256 bytes).

2.1.2 The hardware prefetcher

Some desktop PCs, including the one we use for experiments in this thesis, contain a *'hardware prefetcher'*. This is a small component which observes the data that is being accessed by the processor, and attempts to identify linear sequences ($a, a + b, a + 2b, a + 3b, \ldots$). With this information, it speculatively starts to bring data from the RAM into the L2 cache before the processor asks for it, which means that misses in the L2 cache may be avoided. Thus, if a program can arrange its data in order, it can avoid many L2 misses. The hardware prefetcher works only within a page – it will not attempt to fetch data across a page boundary because of the cost of virtual to physical address translation.

2.1.3 Improving memory hierarchy performance

Let $miss(C)$ denote the number of cache misses a program experiences in a cache C. The misses may be characterised by reference to two other caches [5, 32, 70]. C_∞ is an infinite cache, one that retains every block that has been accessed. C_{FA} is the optimal cache of the same size as C: a fully-associative cache that uses an *'omniscient'* replacement policy. When a block must be evicted from a set, it evicts a block that will not be accessed again by the program, or failing that, it evicts the block whose next access is the furthest in the future. Note that both C_∞ and C_{FA} are simply analytical devices constructed to model the best-case behaviour of a real cache.

Misses are split into three categories, as follows:

1. **Conflict**, given by $miss(C) - miss(C_{\text{FA}})$. Misses caused by the division of the cache into sets and the replacement policy within sets, rather than using the optimal finite cache.

[1]Fragmentation occurs when a program requires a large contiguous chunk of memory, but no such contiguous amount of space exists in RAM, even though the *total* amount of free space in the RAM is larger than size of the chunk. The problem is not that the RAM is too small, but that it has been mismanaged.

2. **Capacity,** given by $miss(C_{\text{FA}}) - miss(C_\infty)$. Misses caused caused by the finite size of the cache.

3. **Compulsory**, given by $miss(C_\infty)$. Misses caused because data has never been accessed before.

Compulsory misses are irrelevant for any program that runs long enough to experience layout degradation, at least when the space created by freed nodes is reused for new ones (to do otherwise is pathological).

Conflict misses are less severe on desktop hardware than in the past because L1/L2 cache associativities are now higher[2], and recursive data structures tend not to exhibit access patterns that cause conflict misses (contrast this with interleaved array accesses, which can suffer from severe conflict misses). Although past work on heap data layout has considered conflict misses (e.g. Calder et al. [11] in 1998, see §2.2.2.1), the trend in more recent work is to focus on capacity misses only (e.g. the garbage collectors in §2.2.2.2).

Thus in this thesis, as with most recent work, we focus entirely on capacity misses. Capacity misses may be split [70] into *'fundamental misses'* (ones that are unavoidable even with an optimal data layout), and *'layout misses'* (ones that are caused by the layout actually in use). We aim to reduce layout misses by changing the layout to maximise the usage of each block before it is evicted – in practice, this is achieved by storing within a block nodes that are accessed close in time. Note that changing the layout may also affect conflict misses, but we expect capacity misses to dominate.

2.2 Related work

In this section, we summarise past approaches to improving the cache performance of heap data, including prefetching, correct allocation, and runtime data movement using garbage collectors. Finally we discuss cache-aware RDSs.

2.2.1 Prefetching

Modern desktop processors have a prefetch instruction. Like a load instruction, a prefetch instruction specifies an address, and when executed causes the memory hierarchy to move data from lower down the hierarchy to the top. Unlike a load instruction, the data that is moved is not available to the program; the purpose of the instruction is purely to start the movement of data up the memory hierarchy so when the program tries to use it (using a load instruction), the miss penalty is reduced or avoided. The insertion of explicit prefetch instructions into a program by a programmer or compiler is known as *'software prefetching'*, distinct from *'hardware prefetching'*, as performed for example by the memory hierarchy's hardware prefetcher. The prefetch instruction affects not the output of a program, only its performance. Prefetching can be used for any data structure, and has been useful in the past for RDSs. Achieving performance improvement using prefetching is a matter of ensuring that exactly the right addresses are prefetched and that they are prefetched early enough to completely remove the miss.

[2]See the next chapter for the dimensions of the caches on the test machine.

Deciding which addresses to prefetch can be done in many different ways, such as greedily prefetching all pointer fields of an object when it is encountered [10, 45], imposing particular layouts on the RDS which allow address arithmetic to be used [45], adding explicit prefetch pointer field(s) to objects [10, 40, 45, 59], and software or hardware runtime methods to detect patterns in addresses and perform prefetching (including the linear address prefetch units in modern desktop processors) [3, 16, 36, 47, 53, 58, 67, 75, 76]. In all these schemes, the emphasis is on prefetching what is needed and no more, because unnecessary prefetches will significantly increase the required memory bandwidth, evict useful data from the cache, incur an instruction overhead, and increase node size unnecessarily when prefetch field(s) are in use.

2.2.1.1 Linear access patterns

Prefetching is most successful for linear sequences of accesses – that is, ones where the same pattern of addresses occur in sequence over and over (mutating slightly as pointers are updated in the RDS, perhaps), instead of branching traversals such as binary tree lookup. In 1996 during the first work on RDS prefetching, Luk and Mowry [45] observed that trying to impose a linear layout on linear data structures such as linked lists would allow software prefetching by address arithmetic (on today's hardware, this would be achieved by the processor's hardware prefetcher). Another approach taken by Luk and Mowry, and later by others [10, 59], was to add an extra *'prefetch pointer'* field to each node, referring to another node some distance ahead in the traversal. During traversal the node referred to by the prefetch pointer is prefetched. Provided the destination of the prefetch pointer is far enough away in the traversal, the cache miss penalty may be completely avoided.

A different approach to the software prefetching of linear structures is *'stride'* prefetching, and various other methods which attempt to detect access patterns dynamically. Stride prefetching is the insertion of code near a particular load instruction that detects linear sequences of addresses ($a, a + b, a + 2b, a + 3b, \ldots$, where b is the stride) and prefetches ahead of the current access. Stoutchinin et al. [67] modified a compiler to identify pointer-chasing loops and conservatively determine whether there is available bandwidth to perform prefetching. Wu et al. and others [47, 75, 76] use profiling to guide the insertion of code into suitable loops, which allows less conservatism, and report that their scheme outperforms the Itanium's hardware prefetch unit by about a third. Inagaki et al. [36] inserted stride prefetch instructions using a just-in-time compiler, detecting strides by partially executing the method that is being compiled, using its actual arguments. Similar to the scheme of Inagaki et al. is the scheme of Chilimbi et al. [16], which partially executes the method looking for repeated *sequences* of addresses (instead a repeated address difference), and then inserts prefetch instructions.

2.2.1.2 Branching access patterns

Branching traversals are significantly harder to tackle with prefetching. There is a fundamental bandwidth issue involved – the traversal may simply be too unpredictable (e.g. high branch factor in a tree) to know what to prefetch. One ad-hoc method for all traversal types is *'greedy'* prefetching, which is to prefetch all pointers of a node when it is encountered [10, 45], which is sometimes effective, but often does not issue prefetch instructions early enough, or imposes excessive bandwidth demands on the machine.

Karlsson et al. [40] tackled the prefetching of k-ary trees by adding an array of k^D prefetch pointers to each node, where D is the prefetch distance (how much further down the tree the prefetching is compared to the main program). In principle, the prefetch distance is proportional to the miss penalty divided by the time the program spends working on each node. For example, a prefetch distance of three in a binary tree requires the addition of eight (2^3) prefetch pointers to each node. Karlsson et al. found some improvement for binary trees, but as the branching factor of the tree or the value of miss penalty divided by work per node increases the bandwidth demands and the increased size of the nodes prevent prefetching from being effective.

2.2.1.3 Other schemes

Software prefetch instructions do not have to be inserted directly into a program. For a statically scheduled VLIW processor, Rabbah et al. [53] identify high-miss-rate loads, and then use a compiler to schedule code that performs speculative execution of load dependence chains. This code typically runs far ahead of the main program's access stream and therefore has a prefetching effect. The authors propose the addition of an *'informing load instruction'* to the processor, which would allow such speculation code some degree of feedback from the memory hierarchy, in particular to abort itself if it begins to cause too many misses (i.e. begins to consume too much bandwidth). Thus the statically scheduled prefetching code can respond to unpredictable dynamic memory events.

Roth et al. [58] and Annavaram et al. [3] have previously proposed similar, but entirely hardware-based, methods to perform such prefetching.

2.2.1.4 Summary

Prefetching is a useful technique when the access sequence is predictable (either linear or repeating), but may run into bandwidth problems in other situations, particularly for branching traversals. Furthermore, it has been observed that the performance hit of incorrect prefetching can be as large as the performance gain from correct prefetching. We note that the high bandwidth needed by prefetching can be reduced by placing related nodes in the same cache line, and it is past work on improving RDS layout that we summarise next.

2.2.2 Layout

In this section we examine two previous approaches to improving RDS layout. The first approach is to ensure that an object is allocated in a good location, which may also prevent layout degradation if it is caused by allocating new objects rather than changing the linkage of old ones. The second approach is to use a layout-improving garbage collector to perform runtime data movement, preventing layout degradation.

Most of the solutions described below attempt to solve the problem of heap object layout in general, rather than specific recursive data structures as we tackle in this thesis. We observe that improving data layout in general requires the use of heuristics, as observed by Petrank et al.: *'Suppose one is given a sequence of memory accesses and one has to place the data in the memory so as to minimize the number of cache misses for this sequence. [We] show that if $P \neq NP$, then one cannot efficiently approximate the optimal solution even up to a very liberal approximation ratio.'* [52]

2.2.2.1 Allocation

Attempts to achieve a good layout at allocation time vary a great deal in complexity. Feng et al. [22] show that a general memory allocator can improve layout with some simple changes, without using any information about the object being allocated other than its size. Their allocator improves layout simply by using fine-grained size classes, removing headers of small objects to increase cache line utilisation, and through the use of regions (see below). Seidl and Zorn in a series of papers [62, 63, 64, 65] develop low-overhead methods for identifying heap objects at runtime, which allows them to guide their allocation into four classes based on object lifetime and usage frequency. Frequently used objects are packed together, reducing working set size, and the division of long-lived objects from short-lived objects helps to reduce fragmentation. The identification of objects can be done quite effectively by calculating a hash by XORing the last few stack return addresses.

A region (a.k.a. pool) is a contiguous part of the heap that stores objects of only one type (or perhaps of only one size). Allocation and deallocation are quick and fragmentation is reduced. Regions can be used to pack objects from the same RDS instance together in memory, improving cache performance. The definition of regions appears to be inconsistent – some authors allow region creation, node creation, node deletion and region deletion, while others disallow (individual) node deletion. We will use the first definition. The transformation to use regions can be performed by hand [27, 26], or by static analysis [44, 68]. Berger et al. [8] have demonstrated that, of the many custom memory management strategies used by programmers, regions are one of the few that outperform a state-of-the-art general-purpose allocator (the Lea allocator).

The schemes above attempt to categorise objects, either based on their size, the RDS they belong to, or their lifetime and usage frequency. The greater the resolution used in the categorisation of objects, the larger the potential improvements. Truong et al. [70] showed by simulation that if each object instance is dealt with individually, even quite simple heuristics can significantly improve data layout. Calder et al. [11] used profile information to guide the allocation of heap objects by identifying them in a similar way to Seidl and Zorn, but used finer-grained allocation classes than the four they considered.

Finally, Chilimbi et al. [15] investigated an allocator called ccmalloc which takes as argument the address of another *'hint'* object in the heap. The allocator attempts to allocate the new object in the same line or page as the hint, thus improving layout if a suitable hint can be identified (in their work, by hand). Their results indicate that choosing a hint is usually simple, and greatly improves the layout of an RDS. In Chapter 4, we discuss the use of similar allocators to maintain RDS data layout.

2.2.2.2 Using garbage collection to maintain layout

As with the body of work concerned with allocation, attempts to use garbage collectors to maintain good RDS layout vary in complexity. The simplest method is to vary the order in which the heap is traversed, and hence the layout of nodes produced by a copying collector. The traditional order to traverse the heap in is *'breadth-first search'* (or *'BFS'*) order, which can be done without using extra storage to store the queue, as noted by Cheney et al. [13]. White et al. [72] first suggested that a GC could be used to improve the mutator's data layout, observing that a *'depth-first search'* (or *'DFS'*) traversal order would have better layout than BFS – placing parents and children close in memory – but that BFS is easier to achieve without increasing memory usage. Moon et al. [50] modified the traversal order of Cheney et al. to be closer to a depth-first traversal, and Wilson et al. [74] provided further improvements by hierarchically decomposing the traversal – filling pages by BFS traversal, then traversing pages in BFS order. Shortly thereafter, Lam et al. [43] observed that the optimal grouping depends heavily on the structure – there is no one grouping that performs well universally.

Next we consider schemes that take a more detailed approach to RDS layout, usually using some form of profiling to discover how the program uses heap data (i.e. without any programmer involvement).

Per-object-instance profiling: Here we consider schemes that attempt to discover how individual object instances are accessed. The first work of this type was the incremental GC of Courts et al. [21], which used custom hardware that had the property of moving objects to to-space in their access order, thus often improving layout (1988). The first software-based scheme was not seen for ten years, the stop-the-world collector of Chilimbi et al. for the language Cecil [17]. Online profiling detects hot objects (frequently accessed objects) and constructs an *'object affinity'* graph which is used to impose a good data layout while garbage collection occurs. The authors note that although some improvement is seen, there is much room for improvement in reducing the overhead of profiling and improving the layout produced.

Chen et al. [12] take a more aggressive approach to maintaining data layout, triggering object movement using their collector when global miss rates begin to rise, rather than just when collection for garbage is needed. Their profiling and method of choosing a data layout is like Chilimbi's in that individual object instances are considered. They found that bursty profiling was necessary to reduce overhead, incurring a 60% increase in execution time for between 2 and 20ms, with shorter bursts producing poorer profile information and hence poorer layout. Latency is tackled at a finer scales in this thesis, as discussed in §3.3.4.

Per-object-type profiling: The collectors of Chen et al. and Chilimbi et al. perform profiling per-object-instance, but other collectors perform it per-object-type. The first layout-improving collector for Java was the collector of Shuf et al. [66], who combined two approaches. Firstly, they identify which object types are *'prolific'* (those with a large number of instances) and then using a graph with nodes as object types and edges as references, identify clusters of prolific types. For a prolific type α, denote the other prolific types in its cluster as α's *'co-prolific'* types. During allocation of an object a of type α, an attempt is made to colocate a with some instance of one of α's co-prolific types that a has a pointer to. During traversal the collector attempts to keep pages of from-space together in to-space, the effect being that as garbage is removed, layout quality is either preserved or increased. Performance is improved for Java SPEC [30] and Olden [57] suites by up to 14% (6% on average) in the Jikes RVM with a copying two-space collector. The collector of Huang et al. [35] is similar to the collector of Shuf et al.

Semi-automatic solutions: Novark et al. [51] suggested that a programmer could provide the collector with a function that is invoked when a particular RDS is collected, which traverses the RDS in the correct order to impose a good layout. This is likely to impose a better layout than profiling methods, and may be a good compromise between programmer effort and program performance.

The cost of garbage collection: Historically, garbage collection is seen as rather expensive. We refer to a 2005 study by Hertz et al. [31], who use profiling to work out when an object in a garbage-collected program becomes garbage. The program is then run by simulation with added explicit free operations. The authors note that this approach is more favourable to garbage collection than the opposite approach (removing frees from a non-garbage collected language and then applying a GC). They find that with five times as much memory, GC is often better than explicit freeing, but with three times as much memory, a GC is 17% slower, and with twice as much memory, it is 70% slower. They use a stop-the-world collector, which will have lower overhead than an incremental GC.

A *layout-improving* GC may be viewed (e.g. by Huang et al. [35]) as an improvement over explicit deallocation because it can prevent layout degradation by runtime data movement. However, such schemes may use large amounts of memory – increasing the heap size increases the time between collections, reducing overhead to manageable levels. Huang et al. do not give results for heaps smaller than 1.8 times the amount of storage required by the program, whereas the techniques of this thesis often perform well using a heap of relative size 1.25.

Summary: Heap data layout can be improved using a garbage collector with the help of profiling to discover how either particular object instances, or particular object types, are used by the program. A less common approach is allow the programmer to specify traversal orders for identified data structures. In both situations the traditional time and memory overheads of garbage collection must be considered.

The aim of this thesis is to demonstrate where the upper limits of well-tuned runtime data movement optimisation are, using simple benchmarks, not addressing in detail how to apply such optimisation automatically. This allows these overheads to be reduced in several ways.

2.2.3 Comparing prefetching and layout

There have been very few studies comparing prefetching to layout optimisations. Hallberg et al. [28] compare Luk and Mowry's greedy prefetching [45] and Chilimbi et al.'s ccmalloc [15] using the Olden suite [57]. They conclude that unless line sizes are very small, ccmalloc is much more effective than prefetching. Badaway et al. [4] draw similar conclusions for Luk and Mowry's prefetch pointer scheme for linear data structures [45] using the Olden Health benchmark.

The consensus appears to be that for RDSs that do not change shape as the program runs, it is more effective to try to impose a good initial layout than use prefetching. Indeed, statically, prefetching has a number of implementation difficulties: It is hard to cope with high branching factors or obtain sufficient prefetch distance, it incurs significant runtime overhead and the cost of incorrect prefetches is high. Dynamically, prefetching still suffers from these problems, but becomes more attractive because maintaining prefetch pointers may be far simpler than moving objects at runtime to preserve data layout.

Prefetching may be combined with layout optimisation, and arguably this makes more sense than using either alone: layout optimisation minimises the number of block accesses (the required memory bandwidth), then prefetching minimises the miss rate. Realising this is another matter; neither Hallberg et al. nor Badaway et al. found any benefit in combining the two.

2.2.4 Cache-aware RDSs

Algorithms and RDSs are usually evaluated within the the *'RAM model'*, which assumes that all data accesses have the same cost. This conclusion does not hold in the presence of data caches. A cache-aware RDS is one that is optimal in the *'Input-Output model'* (or *'I/O-model'*) [2]. The I/O-model assumes two levels of memory – a fast memory of finite size M, and a slow memory of infinite size, with data transfered in blocks of known size B. Computation is performed only on data in the fast memory. The complexity of an algorithm is measured in block transfers. All block transfers are explicit – in this sense the model more accurately models virtual memory (explicit transfer of pages between main memory and disk), rather than the memory hierarchy (multiple levels of cache, and automatic transfer of blocks, affected by the geometry and replacement strategy). Nevertheless, it has been found that the I/O model *'adequately models the situation where the memory transfers between two levels of the memory hierarchy dominate the running time, which is often the case when the size of the data exceeds the size of main memory'* [49].

We now discuss cache-aware equivalents of the two RDSs we consider in this thesis – linked lists and binary trees.

2.2.4.1 Trees

The principle behind a cache-aware search tree is to use nodes as wide as a cache line, which both maximises line utilisation and reduces tree depth. The canonical solution is to use a B-tree, which supports optimal insertion, delete and lookup in the I/O model when nodes of size $\Theta(B)$ are used, and is also considered very good in practice [18, 56]. Furthermore, it is observed [18, 55] that using a node size larger than the cache line size can improve performance, due to reduced tree depth, at the cost of extra memory (because B-trees in general contain some empty space in nodes).

2.2.4.2 Lists

As with trees, the principle behind cache-aware lists is to store more than one key per node, and make nodes the same size as cache lines. By allowing the number of keys in a node to vary between some minimum value b and some maximum B, keys can be kept in order, improving cache behaviour, without significant update costs. On an insertion, if a node x overflows, $k \geq 0$ empty nodes are allocated, and the key from x and some set Y of neighbouring nodes are redistributed amongst the $1+k+|Y|$ nodes, with k and Y chosen so that no node overflows. Similarly, on delete, node(s) may be freed. The simplest algorithm [6] based on this method is to set $B = 2b$. If overflow occurs on insertion ($B + 1$ keys within a node), the extra key is either moved to the next node, or the node is split into two nodes of $B/2 = b$ and $B/2 + 1 = b + 1$ keys. Deletion likewise involves either the movement of a key or the merging of two adjacent nodes. In the I/O model, this provides traversals in an optimal $O(\text{TraversalLength}/B)$ block transfers, and update cost of $O(1)$ block transfers. The amount of memory used is no worse than a linked list implementation – the minimum memory density has been halved, but this is cancelled out by the removal of internal pointers. Updates are more expensive if b is increased, but cache line utilization increases, potentially increasing layout quality, and thus the implementation may be tuned. Experimental results from structures with $b \in [1, B - 1]$ are given by Rubin et al. [60], and results from a C++ STL-compatible structure by Frias et al. [25].

2.2.4.3 Discussion

Cache-aware structures are designed from the outset with cache performance in mind, and thus by construction will outperform any other approach. However, they have a number of disadvantages. As demonstrated above, they are harder to implement than vanilla structures. Indeed, the cache-aware equivalent of a particular 'hand-rolled' data structure required by a program may not exist. Furthermore, the precise requirements of a data structure may not be fully known until the program has been written, and so attempting to write the program using a cache-aware structure from the outset may be too restrictive (a case of 'premature optimisation'). Thus, the approach of this thesis (augmenting vanilla data structures with data movement code) may be preferred in practice because the programmer does not have to expend significant effort until it is certain that optimisation is required, and the final form of the RDS is known.

2.2.5 Changing RDS node definitions

Several researchers have presented optimisations that focus on the arrangement and shape of individual nodes of a vanilla RDS, without changing the fields that are stored within each node. The focus of these optimisations is either to reorder fields (i.e. change the object definition) so related nodes are more likely to be in the same cache block, which can be done by hand [14, 69, 71] or by online profiling and dynamically recompiling methods when object definitions change [41]. Nodes may also be split into a hot (frequently-used) part and one or more cold parts, either by keeping the cold part(s) at a large constant offset, which has the effect of interleaving nodes [24, 69, 71], or by linking the cold part to the hot part with a pointer [14].

The effect of these optimisations is to magnify the effects of prefetching and methods of improved RDS layout for situations where RDS nodes have cold fields (for example, object headers, or large value fields). The benchmarks we use in this thesis do not have cold fields in nodes, so we do not use any of these techniques, but such techniques are vital when the aim of an optimisation is to pack as many nodes into a cache line as possible. Thus these methods may be useful to increase the performance of the techniques in this thesis when applied to real programs.

2.3 This thesis

In this thesis we use runtime data movement to improve the cache behaviour of RDSs. The focus is on maximising performance, while at the same time using only modest amounts of extra memory and not significantly increasing the latency of small units of normal program work. The conditions on latency and memory make optimisation difficult (exactly how difficult will be made apparent in §7), and thus we are concerned only with investigating what sort of optimisations are useful, rather than working out the details of how to apply them easily. This is in contrast to past work on runtime data movement, for example Garbage Collection [12, 17, 35, 66], where the concern is (typically) how much layout degradation can be prevented without any programmer involvement within an existing framework. Although the aim of our work is to investigate what forms of runtime data movement are effective, and where the limits of these technique lie, we will briefly discuss later how such techniques could be applied more easily using, for example, feedback-directed optimisation.

To apply an optimisation from this thesis, the applier (programmer, compiler or some combination) must:

1. **Select**: Identify suitable optimisations based on the traversal type and update operations, and the memory allowance and whether large latencies of small units of normal program working are allowed.

2. **Apply**: Insert code, and enable safe data movement by providing some mechanism to update all pointers to a node at a few program points

3. **Tune**: Explore the parameter space of the chosen optimisation, to maximise performance given constraints on memory and latency, etc.

We discuss these three tasks in more detail below. Recall that within this thesis we focus on microbenchmarks, based on fundamental RDSs, rather than using standard benchmark suites such as SPEC [30] or Olden [57]. Below we will explain why we believe this is a useful approach.

2.3.1 Selecting optimisations

The selection of an optimisation is guided firstly by the shape and usage of the structure, secondly by the way the structure changes shape as the program runs, and thirdly by how much extra memory can be used and whether large latencies are allowed.

Firstly, although data layouts based purely upon the connectivity/linkage of RDS nodes will yield good performance (refer to the past work on GCs in §2.2.2) more effective data layouts can be achieved by identifying how the structure is used – the shape and frequency of each traversal loop. This could be achieved in various ways, from exploiting the programmer's knowledge of the program through online or offline profiling and possibly even static analysis. Different optimisations require different levels of knowledge of the usage of the RDS. For example, some of the 'bulk data movement' optimisations of Chapter 5 require only knowledge of a good layout for the structure, whereas others in the same chapter require the applier to embed data movement code in an existing traversal loop, which requires the applier to know the behaviour of the loop.

Secondly, understanding how the structure changes shape may be necessary to apply an optimisation. For example, the optimisations of Chapters 4 and 6 (reallocation and perfect data movement) attempt to correct layout degradation when a pointer update occurs. For reallocation this can be made more effective if the applier can decode several pointer updates into a higher-level operation such as substituting one node for another within the RDS. For perfect data movement, much more precise knowledge of how the structure changes shape is needed.

Finally, the amount of latency allowed and extra memory usage an optimisation can incur will guide the choice of optimisations. For example, in Chapter 5 the simplest optimisation we discuss simply periodically moves the entire RDS to a different block of memory, restoring layout to some known quality – this requires twice as much memory and causes large latency, but is very effective.

2.3.2 Applying optimisations

The most significant task for the applier of the optimisation is to, for a small set of program points, allow a node to be moved in memory by explicitly identifying all its parents, or providing some mechanism to update them, which could involve some form of exhaustive search and use of forwarding entries as in a GC, but most efficiently will not. This will be discussed further in §2.3.2.1.

There are two other tasks the applier must carry out to apply the optimisation. All optimisation in this thesis use a specially managed *'RDS heap'*, which supports certain operations not found in a general purpose memory manager, and is very efficient because all nodes are the same size. The program must be transformed so all nodes of the RDS being optimised are allocated and freed through the memory manager of the RDS heap. Past work suggests that this may be done automatically [44].

The applier must also actually insert the data movement code into the program. This may take the form of applying code to sites in the program where pointers in the RDS are updated (closely coupled to the program), or simply ensuring that the optimisation's code is invoked regularly (loosely coupled to the program). In between these extremes, the applier may embed data movement code within one or more of a program's existing traversal loops.

2.3.2.1 Updating parents on data movement

Vital to the approach we take in this thesis is the ability to move objects very often to maintain a good data layout. For example, for the linked list benchmark we discuss in the next chapter, best performance is obtained when ten percent of the time the RDS is being traversed, nodes are being moved (or expressed another way, every tenth time a node is visited, it is moved).

To move a node requires updating all the pointers that point to it (its parent pointers), which can be done in general in a number of different ways:

1. **Update pointers immediately:** The advantage of moving a node immediately, is that, in the absence of any other effects, its space is freed up and can be reused (which means not only that less memory is required, but also that the optimisation may be able to achieve a better layout). In this thesis, we rely on applier knowledge to update

all pointers to a node immediately when it is moved. In general this will not be possible. There are a variety of ways to make this automatic – static analysis may be performed, but is difficult and expensive, or the semantics of the RDS or programming language may be changed to make the locating of parents easier, but programmers are likely to be unwilling to have such restrictions. Dynamic solutions are a possibility too: using the observation that the number of pointers to heap objects is small[3], a dynamically-sized parent array (e.g. linked list) may be added to each node. Although the time and memory overhead of maintaining such a structure is large, the benefits of easier parent updating on node movement may outweigh them. Furthermore, because the optimisations in this thesis move nodes only at well-defined program points, the structure does not have to be kept up-to-date at all times.

2. **Update pointers eventually:** This is the strategy used by garbage collectors – a node is replaced with a forwarding entry, and a flag set to indicate that the node is a forwarding entry[4]. Pointers are updated by write or read barrier and exhaustive heap walk, after which all forwarding entries can be removed. This method is sensible if garbage collection is already being performed, but as stated above increases the memory overhead and may make it harder to achieve a good layout.

   Forwarding entries are a safe but rather heavyweight solution. Many objects have only a single parent[3], and so space that could be reclaimed immediately after an object's movement is being unnecessarily occupied by a forwarding entry until the next heap walk (recall that when moving nodes for the purpose of improving data layout it is necessary to understand how nodes are linked by pointers - thus it is expected that when a node is moved at least one parent is known).

   Furthermore, the overhead of removing forwarding entries gets worse if the heap is large compared to the RDS: the RDS must be re-laid out frequently, but this cannot be done without large overhead due to the cost of walking the entire heap. To cause significant loss in performance due to poor data layout an RDS need only be few times larger than the L2 cache ($\sim$ 1MB), whereas the program's heap could be hundreds of MB.

3. **Separate inter- and intra-RDS pointers:** Pointers to RDS nodes from other nodes in the same RDS are potentially simpler to keep track of than those from the rest of the heap. Thus, a scheme where all accesses to nodes from outside the RDS heap go through a forwarding table may allow the applier to exploit their knowledge of how the RDS functions, while not assuming anything about what pointers the program retains to RDS nodes. Lattner et al. demonstrate a static analysis able to transform a program in this way [44]. In the extreme case the RDS may be fully segregated from the rest of the program and accessed through a small number of identified root nodes.

There are thus many possible methods to update parents when a node moves, and best performance will be obtained (in terms of time, and probably also memory and latency)

[3]For a set of Java benchmarks, including the SPECjvm98 suite, the Java-Olden suite, and a number of other applications including a web server, Hirzel and Hind state that fewer than 40% of heap objects have more than one parent [34].

[4]This can be done by using, for example, the unused low bits of pointers (which are often necessarily aligned to 4 bytes), or by special bits in hardware such as in Luk and Mowry's scheme [46].

when this can be achieved with the minimum of effort – which involves both the instruction and data costs to locate the parents and then the cost of actually updating them. The latter is often cheap, based upon the measured number of references to objects [34], and the properties of fundamental RDSs.

For the optimisations in this thesis, identifying parents need not be done in full generality. Nodes are moved at only a few program points, either embedded within the original program's traversals of the RDS, or within the optimisation's code. In both cases, the applier may exploit the fact that a well-defined traversal has been used to reach the node (rather than, say, walking the heap in address order, which would visit nodes out of context). If the applier has a choice of where to allow node movement, they may reduce overhead and implementation difficulty by, for example, moving nodes during read-only operations of the RDS.

In this thesis we avoid the issue of updating multiple parents by using RDSs with only one parent per node from within the RDS, and by keeping track of any stack references.

2.3.3 Tuning optimisations

Most of the optimisations of this thesis have parameters that can be *'tuned'* to give the best performance. These parameters may be numerical, binary, or chosen from a small set of distinct choices. The aim of this thesis to demonstrate the best performance that can be obtained from the optimisations, and thus we tune all parameters optimally, by hand. In general, tuning may be achieved automatically, either statically or dynamically.

'Static tuning' is any method that sets the parameters of the optimisation before execution time. This could be done by feedback-directed compilation (i.e. repeatedly compile and run and measure execution time): in some areas, such as compilation for embedded devices, where the hardware is precisely understood and small improvements in efficiency have large consequences for unit cost, expending a lot of effort optimising a program is commonplace. Moreover, methods exist to reduce the search space (e.g. Chow et al. demonstrate a method for multiple binary parameters [19], and Bodin et al. a method for multiple numerical parameters [9]).

'Dynamic tuning' is any attempt to adjust parameters online, to adapt to changes in architecture of the machine and changes in how the RDS behaves. Online profiling methods have been combined successfully with data layout optimisations on several occasions [12, 17, 41], but clearly carry some overhead.

2.3.4 Choice of benchmarks

In this section, we discuss why we do not use the traditional large benchmark suites, and discuss how the techniques in the thesis may generalise to programs that use graphs instead of trees and lists.

2.3.4.1 Large benchmark suites - SPEC and Olden

Past work on RDS layout and prefetching has used either SPEC2000 [30] or Olden [57] and thus we need to justify this thesis's use of microbenchmarks instead of these larger suites.

Firstly, using smaller benchmarks makes sense practically. This thesis is not restricted by the necessity to apply optimisations automatically. To hand-apply optimisations to SPEC/Olden would take a lot of time, limiting the number of optimisations that could be evaluated.

Secondly, there is some doubt that SPEC2000 and Olden contain RDSs whose layouts degrade, or whose layout degradation causes a loss in performance. The study of Sair et al. [61] concludes that *'only a few applications [of SPEC2000] place more than modest demands on the memory system'*. More seriously, the work of Raman et al. [54] demonstrates that most RDSs in Olden and SPEC2000 are stable, that is, are used in a read-only manner and therefore do not suffer from degrading data layout[5]. Clearly, data structures are not just used for read-only accesses in real programs. Furthermore, as noted by the study of Agaram et al. [1], it appears that many recursive data structures (or irregular pointer-bases structures) within SPEC2000 are emulated using arrays, which may make data movement difficult.

The Olden suite consists of fairly small programs, and is commonly used to investigate static data layout for recursive data structures. The SPEC2000 suite is also often used, but consists of far larger programs and so hand-applying optimisations is not feasible without significant knowledge of the data structures involved. Given a general program, working out the data structures in use is a difficult problem, and still the subject of ongoing research[6]. The previous paragraph highlighted the deficiencies of both suites for the investigation and optimisation of RDS dynamic data layout. A suite of suitable programs of similar size to the benchmarks in Olden ('dynamic Olden') could be constructed, using the simple linked list and binary search tree benchmarks of this thesis as a starting point.

Finally, and most importantly, we believe there is an inherent benefit to studying fundamental RDSs instead of more complicated benchmarks: they are well understood, indeed sufficiently simple that their cache behaviour can be obtained analytically [23, 42, 77], and cache-aware versions exists, giving us some performance upper bounds to compare optimisations against. Trees and lists lie at the heart of more complicated data structures and algorithms, and furthermore model two fundamental traversal types (linear and branching). Despite the simple behaviour of lists and trees, rich behaviour arises when they are optimised using even simple techniques, as the rest of this thesis demonstrates, but because the benchmarks are simple we have a better chance of explaining why an optimisation does or does not work. This thesis aims to demonstrate what forms of optimisation are suitable, rather than providing automatic ways of applying them; we therefore believe that simpler benchmarks are far more instructive than using the traditional suites.

2.3.4.2 Graph benchmarks

In this thesis we use benchmarks based around lists and trees. In this section we discuss graph benchmarks.

[5]Their aim was precisely opposite to ours – to find stable structures so they could impose a better initial layout, especially to rearrange linked lists linearly in memory.

[6]For example, the conclusion of a 2007 paper on shape analysis states: *'In general, real-world systems programs contain much more complex data structures than those usually found in papers on shape analysis, and handling the full range of these structures efficiently and precisely presents a significant challenge.'* [7]

It is important to distinguish between physical graphs (the objects in the program and how they are connected by pointers) and logical graphs (what the objects and their pointer-linkage represent). For example, a logical graph can be represented by a single boolean matrix object. Another representation of a logical graph, allowing unlimited fanout per logical node, is to represent nodes in the graph by an object of type A which has a pointer to some auxiliary structure composed of objects of type B. Each B object contains the graph node's pointer(s) to other graph nodes (physically, pointers to other objects of type A). Most simply, the B structure is a linked list, but tree representations are also possible, and in general we would expect the B structures not to be a physical graph. If the number of children is large, the layout of the B structures will dominate, and thus the problem of layout/runtime data movement is actually the problem of layout/runtime data movement for the many B structures, which are not physical graphs (cf. MList, the multiple linked list benchmark, see §3.2.2). Only when the B structures are small does it become relevant to consider the layout/runtime data movement of the physical pointer graph composed of A and B objects.

Thus, noting that effective data layout/runtime data movement in a program with a logical graph does not necessarily involve dealing with physical graphs, we now discuss data layout/runtime data movement for physical graphs. There are several important issues:

1. **Object size**: Object size determines how many objects may be located in the same line or page; the larger the objects, the less important layout quality is.

2. **Number of parents per object**: The number of parents an object has determines in part how much overhead is incurred by moving a node (because all parents must be updated). As discussed previously, if the number of parents is usually one, and objects with more than one parent are kept track of, the old location of an object can usually be freed immediately, allowing the space to be reused, which may improve the data layout achieved by an optimisation.

3. **Number of child pointers per object, and the distribution of their use**: The number of child pointers per object[7], and the distribution, determines the nature of the traversal and the layout that should be aimed for. If only a small subset of children are usually visited (a small 'effective fanout'), then the traversal is quite similar to a branching traversal in a tree of low fanout or a linear traversal in a list (with the exception that loops may exist, since the structure is a physical graph[8]), and a good layout is therefore to colocate a node with its small number of frequently visited children. Thus, much of the techniques in this thesis are applicable, given some mechanism to update parents. As the effective fanout rises, it becomes harder to colocate an object with all its children - data layout optimisation is not possible, and so runtime data movement is not possible either.

In conclusion, for physical graphs where data layout makes a large contribution to performance, the techniques of this thesis are applicable, provided some mechanism to update

[7]In general we would expect the average number of child pointers per object and the average number of parents per object to be the same.

[8]We do not expect loops to have a significant impact on data layout. If a loop is short, the revisited nodes are likely to still be in the cache, and so the loop is irrelevant from the point of view of data layout. If the loop is long, a node is only revisited infrequently, and so the loop is again irrelevant.

multiple parents exists. We have been unable to find for real-world programs any statistics on the number of parents per object, number of children and distribution of their use. However, the significant amount of work on static data layout optimisations for general heap data (see §2.2.2.1) suggests that such physical graphs are common.

2.4 Summary

In this chapter we described in more detail the memory hierarchy of a modern desktop PC, and how its performance may be improved in general. We then discussed different specific approaches from the literature, including software prefetching, use of garbage collectors to improve layout, and cache-aware RDSs. We noted that software prefetching is often less effective than the two layout-based methods.

The first of these methods is to use a layout-improving garbage collector. This typically requires no programmer involvement, but may not produce a very effective layout, and may have high latency and memory overheads.

The second method is to use a cache-aware RDS, or some other structure designed from the outset for good cache performance. In principle this will be the most effective solution, but the implementation effort of such a structure is considerable, and indeed a cache-aware equivalent of a particular structure required by a program may not exist. Furthermore, developing a program using data structures with good cache performance may be too restrictive if the requirements of the structure are not known from the outset.

In this thesis we investigate the movement of data at runtime (as in a GC), but prioritise performance, low memory and low latency, rather than the ease of applying the optimisation. The aim is to investigate the the limitations of runtime data movement, guiding the design of future layout-improving GCs and cache-aware RDSs, in addition to producing optimisations useful in their own right.

To apply an optimisation from this thesis, three tasks must be carried out. Firstly, the correct optimisation must be selected based on the shape and usage of an RDS, and how much latency and extra memory is allowed. Secondly, the optimisation must be applied to the program, the most difficult aspect of this task being identifying all pointers to a node at the few program points where nodes are moved. This task is not difficult for the benchmarks used in this thesis, and we outline a number of ways it may be achieved in general, the important point being that some efficient method must exist if runtime data movement is to be used to prevent layout degradation. Finally, the parameters of an optimisation must be tuned.

Two simple microbenchmarks based on linked lists and binary trees are used, representative of two major classes of data access pattern. This allows the thesis to generate greater insight than if larger benchmark suites (SPEC, Olden, etc.) were used. Furthermore, since optimisations are applied by hand within this thesis, the small size of the benchmark allows a larger number of optimisations to be evaluated.

In the benchmarks of this thesis, performance is determined solely by a single data structure, which undergoes uniformly-distributed insertions and deletions. All objects are the same size, allowing the memory manager to be more efficient, and nodes have only a single parent pointer, which allows immediate and low-overhead node movement. Thus, as with all program optimisation techniques demonstrated using microbenchmarks, the results in thesis will provide an upper bound on performance, rather than the typical case.

However, noting firstly the size of the performance loss caused by layout degradation in the microbenchmarks, and secondly that the techniques of this thesis often rectify a large proportion of this loss, it is clear that both the problem and the solutions presented in this thesis are of significant relevance to real-world programs.

Chapter 3

Benchmarks and methodology

This chapter details the test machine and the benchmarks used, discusses experimental methodology and evaluates the static and dynamic cache behaviour of the unoptimised benchmarks. These results guide the design of the optimisations described in the next three chapters. The performance of some relevant cache-aware RDSs are also investigated.

3.1 Test machine

The test machine is a 3.2GHz Intel Pentium 4 with the following memory hierarchy[1]:

cache	size	associativity	block size	miss penalty
L1	16 kB	8-way	128 Bytes[2]	20 cycles
L2	1 MB	”	”	300 cycles
TLB	64 entries	full	4 kB pages	60 cycles

The machine has 1GB of RAM, and a hardware prefetch unit, which is able to detect concurrent independent linear sequences of addresses, and attempts to stay 256 bytes ahead of the currently accessed location [33]. It does not fetch across page boundaries [38].

The L1 cache is virtually-addressed, but the L2 cache is physically-addressed. Thus, if a miss occurs in the L2 cache, a virtual to physical address translation is required to fetch the line from RAM, which may cause a further miss in the TLB. Typically, the effect of the L1 cache is unimportant or sufficiently similar to the L2 cache that performance can be thought of in terms of a single ‘line’ cache (L2), and a single ‘page’ cache (TLB).

3.2 Benchmarks

In this thesis we consider C benchmarks based on two different problems: The dictionary problem (storing key-value pairs, and supporting insert, delete and lookup, implemented using a binary search tree), and insertions and deletions in a set of linked lists. Both of these are fundamental data structures, and we may also think of them as representative of two common classes of access pattern – branching and linear traversals.

[1]Obtained by the `cpuid(2)` instruction [37], and Manegold’s Cache Calibrator tool [48].

[2]The L1/L2 cache actually has a 64 Byte line size, but with two sectors. In other words, the unit of transfer is a 64 Byte block, but the adjacent 64 Byte block is immediately prefetched [33]. In practice the machine behaves as if the line size were 128 Bytes.

3.2.1 The Dict benchmark: The dictionary problem

Given an initially empty set S, the dictionary problem is to *'execute on-line any sequence of operations of the form S.membership(s), S.insert(s) and S.delete(s), where each s is an object...[This can be implemented using] arrays, linked lists, hash tables, binary search trees, AVL-trees, B-trees, 2-3 trees, weighted balanced trees or balanced binary search trees (i.e. 2-3-4 trees, symmetric B-trees, half-balanced trees or red-black trees)'* [49]. We will use the binary search tree solution to this problem as the benchmark, and the B-tree implementation as the benchmark's cache-aware equivalent.

In the Dict benchmark, the tree is built out of nodes containing four 32-bit fields: two child pointers, and a key and a value. The tree is built by inserting n random keys from the range $[0, 2^{29} - 1]$, allowing duplicates. This phase of the benchmark is not timed. During the timed part of the benchmark, repeated random key lookups are performed, using keys that are known to be in the tree. For each lookup, with probability $s \in [0, 1]$, the node is deleted and a new randomly chosen key inserted. This combination of lookup and optional delete/insert is termed an *'operation'*. The output of the program is the time taken for operations $[1, k]$, $[k + 1, 2k]$, $[2k + 1, 3k]$, and so on. We discuss why we do not time single operations in §3.3.3.

Key lookup is only performed for keys that are in the tree, which is achieved by maintaining an array duplicating the keys currently in the tree. To select a random key, a random element of the array is read. When a key is deleted from the tree and a new one inserted, the array is updated by overwriting the old key with the new key. Notice that because we time groups of operations, rather than a single operation, the accessing and updating of this array is necessarily included in the timing figures. Note also that accessing the array will pollute the data cache. Both of these factors increase the time measured for the group of k operations, but the effect is unlikely to be significant, and appears likely to cause the performance of an optimisation to be underestimated, not overestimated[3].

The **Delete** function is performed in the standard way [20]. If the deletee x has zero or one children, it is simply cut from the tree. If x has two children then it is replaced with (choosing randomly), either the rightmost node of the left subtree of x, or the leftmost node of the right subtree of x. The replacement of node x by another node y can be done in two ways, depending on whether the reusing of the memory previously occupied by node x (x's cell) is allowed or not:

- moveFields: We assume that the code that implements the RDS, and any optimisation subsequently applied to it, can reuse x's cell. In that situation, to cause the minimum amount of layout degradation, it is most efficient to copy the key and value of node y into node x. Node y is cut from the tree and the address of its cell returned by the delete function.
- moveNode: We assume that the delete function must return node x, and therefore the RDS code or an optimisation cannot reuse its cell. In this situation, node y is

[3]When the tree layout is poor (optimisations not in use), each operation fetches a line per node, and so the array access accounts for a small proportion of the time (1 access in about 26 in practice). When tree layout is good (optimisations in use), fewer lines are fetched, and so the array accesses account for a larger proportion of the time (1 access in 10 if layout is optimal). Thus optimisation performance is slightly underestimated.

substituted for node x – node x is removed from the tree, and pointers are updated so node y takes its place. The address of node x's cell is returned by the delete function.

The **Insert** function is performed by attaching a new leaf node to the tree, and unless an optimisation is applied always reuses the cell returned by the deletion.

We consider both MOVENODE and MOVEFIELDS variants because they are both plausible approaches, and have different layout degradation behaviour, as will be demonstrated in §3.4.1. This serves to illustrate that seemingly insignificant low-level changes in a program can have serious implications when it comes to layout degradation and hence unoptimised performance. Secondly, we aim to demonstrate that runtime data movement works well irrespective of which implementation is used; this is important if the optimisations are applied after the program has been written. Finally, we also want to be able to give advice to the programmer about which of the implementations might yield the best performance if they are writing the program with the application of runtime data movement in mind.

Our 'core' set of parameters for the benchmark will be $n = 1e6, s = 1$: the tree contains $1e6$ nodes and a delete/insert is performed with probability 1 after every lookup. The RDS occupies 16MB, which is sixteen times larger than the L2 cache.

3.2.2 The MLIST benchmark: Multiple linked-lists

The second benchmark used in this thesis performs random node insertions and deletions in a set of singly-linked lists. Linearly-traversed RDSs, of which the linked list is the simplest, are even more common than branching traversals such as the DICT benchmark.

List nodes occupy eight bytes, consisting of a 32-bit pointer to the next node and a 32-bit data field. The data field is not used for the benchmark but is maintained properly when a node is moved in memory. List nodes from different lists share the same heap. The head and number of nodes in each list are stored in arrays, allowing random access to any list. These arrays can exceed the size of the cache, depending on the number of lists, but combined always total less than $1/8^{th}$ of the space used by the list nodes, so will not have a significant affect on execution time.

When the benchmark starts, v empty lists are created, and each of the n list nodes are inserted to a randomly chosen list. During the timed period of the benchmark alternating deletions and insertions are performed. Each insertion is performed to a uniformly randomly chosen node in a uniformly randomly chosen list. Deletions are performed likewise. Choosing a uniformly random node from a list is possible because the length of each list is stored in the list header. The inserted node reuses the previous deleted node's cell (unless the cell is freed by a runtime data movement optimisation).

We assume that every node before the place in the list where the insertion or deletion takes place must be accessed. In other words, however the benchmark is optimised, pointer chasing *must* be used to reach the correct place in the list. This is especially relevant when using a cache-aware RDSs (§3.5.2) or perfect data movement (Chapter 6).

We will use a constant number of nodes $n = 2^{22}$, occupying 32MB, and use v in $\{2^7, 2^{11}, 2^{15}, 2^{19}\}$, giving average list lengths of $\{32768, 2048, 128, 8\}$ nodes. This is equivalent to 64 pages, 4 pages, 8 lines (0.25 pages), and 0.5 lines. This covers four distinct situations: (i) lists are very large compared to the page size, (ii) lists occupy only a few pages, (iii) lists fit within a page, and (iv) lists fit within a line. It is arguable that the longer linked lists are unlikely to arise in well-written programs, but we consider list of this length firstly because we expect them to exhibit different behaviour to the shorter lists, and secondly because it is harder to argue that long near-linear or even linear *traversals* will not arise in more complex data structures.

3.3 Methodology

Here we discuss several points of methodology.

3.3.1 Initial layout

Our aim is to demonstrate how useful a runtime data movement optimisation is at maintaining a good layout. Therefore, the 'unoptimised' version of a benchmark will use a very good initial layout.

3.3.2 Length of experiment

We expect both benchmarks to have a degrading data layout, and hope that they will eventually reach some terminal layout quality much worse than the initial layout, making the overhead of trying to prevent layout degradation worthwhile. The wall-clock time to reach this equilibrium will depend on the rate at which updates to the structure occur, and the sort of updates. For example, for DICT, we will show later that the MOVENODE variant has far higher rate of layout degradation than the MOVEFIELDS variant. For MLIST, we observe that although the number of insertions and deletions needed to reach the terminal layout may well be the same irrespective of list length, the two orders of magnitude difference between the shortest and longest list length would require a similar difference in wall clock time to reach terminal layout.

We will give graphs of layout degradation in §3.4.1, which will demonstrate the impracticality of running all experiments to terminal layout. Thus for most experiments we will use a length corresponding to around ten minutes for the unoptimised benchmarks. When the long term behaviour is of interest, we will run experiments until terminal layout is reached (note that the time for the optimised benchmark to reach equilibrium is not the same as the unoptimised benchmark).

Stopping some benchmarks short of terminal layout may be considered unrealistic (because some benchmarks have reached terminal layout, and others have not), but the same is true of the alternative (running some benchmarks for over a hundred times longer). We choose the first option because it increases the practicality of performing experiments while actually making the problem of runtime data movement harder (specifically, there is less scope to improve program performance because less layout degradation has occurred).

The experiment lengths used are as follows: For DICT, $1e8$ operations. For MLIST, since the time for an operation is proportional to the length of the list, we use between $1e6$ and $50e6$ operations depending on the average list length (determined by v).

3.3.3 Measuring times

Unlike some work on measuring native execution times, our benchmarks are *self-timing*. They build the structure, then perform a sequence of N operations. The time for each subsequence of k operations is measured using the C `gettimeofday` function (which measures 'wall clock' time to a resolution of ≤ 2 microseconds on the target machine[4]), and the list of N/k times is outputted. If k is too small, systematic timing errors occurs.

Linux was used on all test computers. Because we measure wall clock time, not process time, measurements are susceptible to additive noise from other processes on the system (i.e. the benchmark process gets significantly less than 100% of CPU usage, increasing the time to perform k operations).

For most experiments, we use a large k. It has been our experience that provided a system is otherwise unloaded, the time for large k is repeatable, moreover two identical machines usually perform very similarly. Most experiments are run more than once with slightly different parameters or on different machines, so in the event of, say, another process starting on the machine and taking up large amounts of CPU usage, any measurement errors would be very obvious, and the result checked by rerunning the program.

To measure latency, we must use a lower k. This may cause 'spikes' to appear on otherwise smooth timing graphs. If a benchmark is run twice (seeding random number generators identically, etc), these spikes occur in different places. Thus, the spikes correspond to another process running at high priority for a short period of time, and these can be dealt with by averaging several runs.

3.3.4 Reporting improvement and latency

To report the performance of an optimisation, we compare the time for the last 10% ($N/10$) operations in the optimised and unoptimised programs. For DICT we will report as percentage reduction in execution time compared to the unoptimised benchmark, or just 'reduction', indicated with %. For MLIST, because the improvements are far larger, it is more sensible to use the value of unoptimised time / optimised time, or 'speedup', indicated by $\times$.

Latency is measured by splitting this sequence of $N/10$ operations into smaller equally-sized subsequences, with the size chosen so the time in the unoptimised program is around $1ms$. We do this for the unoptimised and optimised program, and observe the improvement for each subsequence. We treat latency in a binary fashion (either high or low): If the improvement is *significantly* negative for any subsequence, we consider the optimisation to have high latency. If the improvement is positive or only slightly negative for all sequences, the optimisation has low latency.

This seems rather simplistic and subjective, but in this thesis most optimisations fall unambiguously into one category or the other. More quantitative definitions of latency are certainly possible. We now turn to memory usage, which we consider in more detail, and quantitatively.

[4]Measured by finding the smallest nonzero difference between the results of two calls to `gettimeofday`.

3.3.5 Reporting memory usage

In terms of memory usage, the optimisations of this thesis fall into two groups. The first have bounded memory usage, chosen by the applier. The bound is given by the figure $m \geq 1$, which is the number of cells allocated to the optimisation divided by the number of nodes in the RDS. Note that optimisations often need more cells than nodes if they wish to find contiguous blocks of empty space in which to colocate related nodes.

The second group of optimisations do not have bounded memory usage, and we *report* the memory usage by the figure $m \geq 1$, calculated in the same way. This is technically a function of time, $m(t)$, but for the benchmarks in this thesis is often a constant after an initial 'warm-up' period. For these optimisations, we derive a constant m_{worst}, which is the largest value that $m(t)$ could take $\forall t$ and for all possible sequences of outputs of `rand()` – the 'worst-case' memory usage.

Managing the memory that the RDS inhabits incurs a space overhead. In the worst case, a header of size 16 bytes is stored within each 128 byte line, an overhead of $128/112 \approx 1.14$. We therefore use a figure M, given by $128/112 \times m$ in this example, to express the total memory in use. M_{worst} is defined in a similar way.

From the applier's point of view, M is more relevant than m, but we will usually talk in terms of m because it expresses more clearly how much extra space an optimisation is allowed or is using. We consider a range of different values of m. We consider $m = 1.1$ to be 'low' memory usage, and $m = 2.0$ to be 'high' memory usage, translating to $M \in [1.13, 1.25]$ and $M \in [2.06, 2.28]$, respectively (depending on memory manager overhead).

3.3.6 Random number generation

Random numbers are generated using the C `rand()` function, reseeded with the same constant each time a benchmark runs. This makes all experiments repeatable, and makes it possible to verify the correctness of the optimised form of a benchmark (by calculating checksums, etc). Experiments suggest that the calls to `rand()` do not account for a significant amount of execution time, nor are more uniformly distributed numbers required, so using the standard `rand()` is a sensible choice.

3.3.7 Code transformation

Applying data movement to a program may require transforming parts of it so the parent of a node is kept track of, so it can be updated when the node is moved. This may require rewriting parts of the program to work in terms of `node**` rather than `node*`, which incurs an overhead.

3.4 Cache behaviour of benchmarks

In this section we will show how quickly layout degrades if no effort is made to maintain it, and then we will evaluate in more detail different layouts, and hence obtain upper bounds on the performance improvement a runtime data movement optimisation could give.

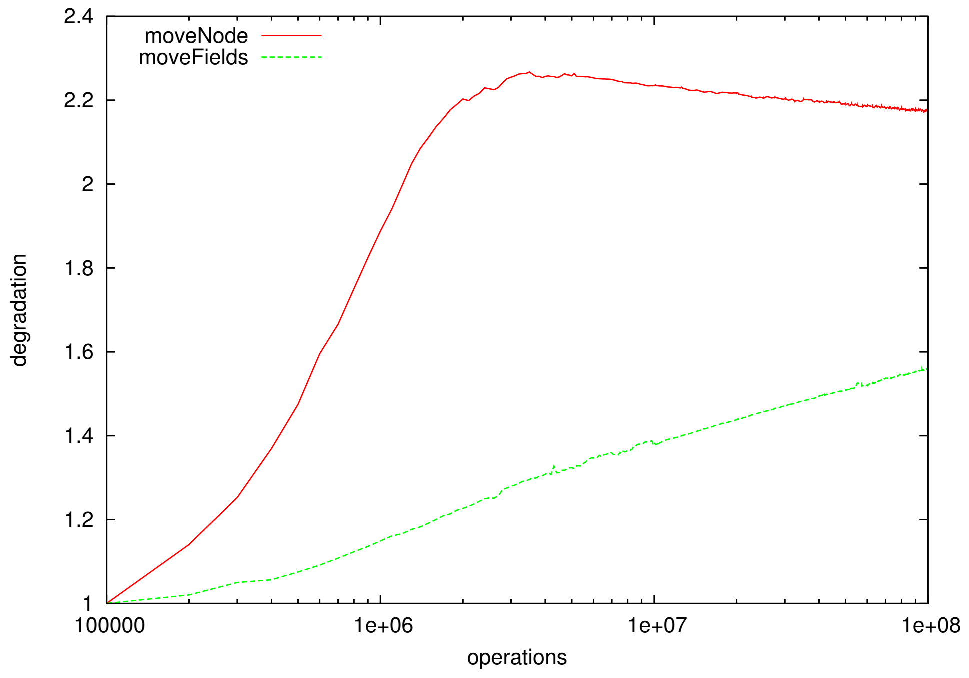


Figure 3.1: Degradation in performance of the unoptimised DICT benchmark. Degradation after operation k = time for the k'th operation / time for the 1st operation.

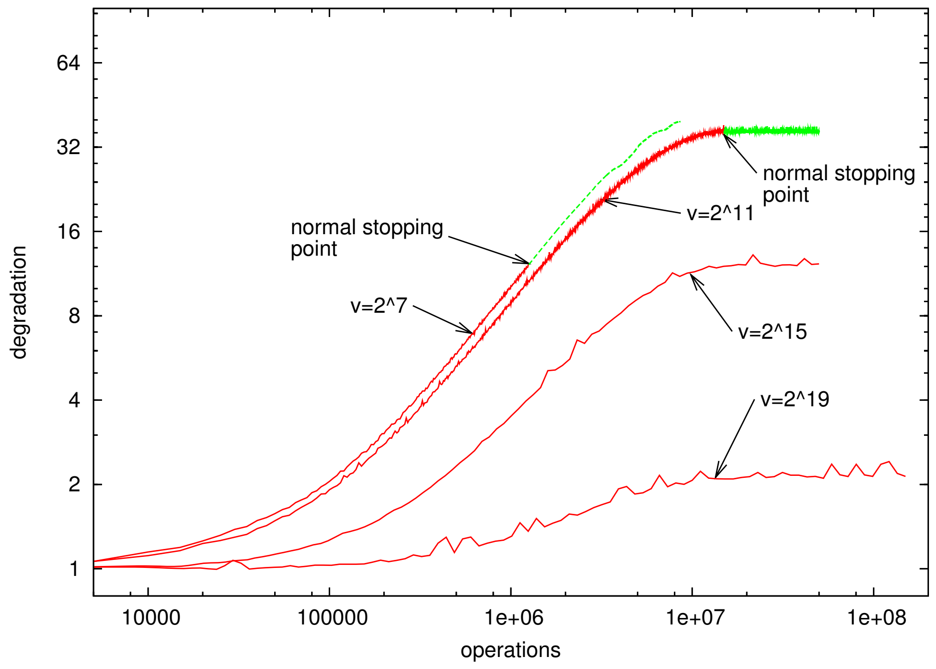


Figure 3.2: Degradation in performance of the unoptimised MLIST benchmark. Degradation after operation k = time for the k'th operation / time for the 1st operation. For $v = 2^7$ and $v = 2^{11}$, extended data is given, showing degradation after the normal stopping point of the experiment.

3.4.1 Layout degradation

The RDSs are created using the best available layout, and then the benchmarks are run for the normal number of operations – $1e8$ for Dict, and for MList the number of operations depends on v, chosen to keep the experiments to around 10 minutes. Results are in Figs. 3.1 and 3.2.

For MList, $v = 2^7$ does not reach equilibrium, even when run for ten times longer than the experiment length we will use in this thesis. $v = 2^{11}$, $v = 2^{15}$ and $v = 2^{19}$ do reach an equilibrium, a layout about a factor of 1.2 faster than arranging nodes randomly in memory.

For Dict, moveNode reaches a terminal layout quality. Inspection of the cache statistics confirms that the peak of the execution time graph corresponds to the worst possible layout (one miss per node). Beyond this point, layout remains as bad as possible and the decrease in execution times is caused because the average number of nodes visited per lookup decreases. In other words, the balance of the tree increases slightly as the benchmark runs, as shown in Fig. 3.3.

moveFields does not reach a terminal value after $1e8$ operations, and as can be seen in Fig. 3.4, it is unfeasible to run this benchmark until a terminal value is reached.

3.4.2 RDS layouts in more detail

Here we investigate in more detail the performance of different layouts, which will be useful when designing runtime data movement optimisations. We will not just restrict our investigation to optimal layouts, because a data movement optimisation must always balance the overhead of maintaining a given layout with the benefit it gives – an optimisation that maintains a reasonable layout with low overhead may perform better than one that maintains an optimal layout with high overhead. We will pay particular attention to whether it is worthwhile concentrating on L1/L2 (i.e. line) or TLB (i.e. page) miss rates.

Benchmarks are run for their normal time (e.g. $1e8$ operations for Dict), then laid out using the desired layout. The average time for an operation under the desired layout is then measured.

3.4.2.1 Dict

We consider four different layouts. **worst** has an L1/L2 miss and a TLB miss for almost every node accessed, and is obtained by randomly arranging nodes in the heap. **goodPage** has low TLB misses, and almost the highest number of L1/L2 misses possible *given the TLB constraint*. This layout is achieved by filling pages with nodes using breadth-first search, and then recursively dealing with any subtrees. Within each page, nodes are arranged randomly (maximising L1/L2 misses, discounting any pathological layouts that may exist). **goodLine** is the L1/L2 analogue – lines are filled by BFS, and then the lines are randomly arranged in the heap – producing low L1/L2 misses and almost the highest TLB misses possible given given the L1/L2 constraint (again discounting pathological layouts). **goodBoth** has low L1/L2 and TLB misses, and is achieved by filling lines with nodes using BFS, and then filling pages with lines. This 'hierarchical' method of constructing layouts has been used before, e.g. by Wilson et al. within a GC [74].

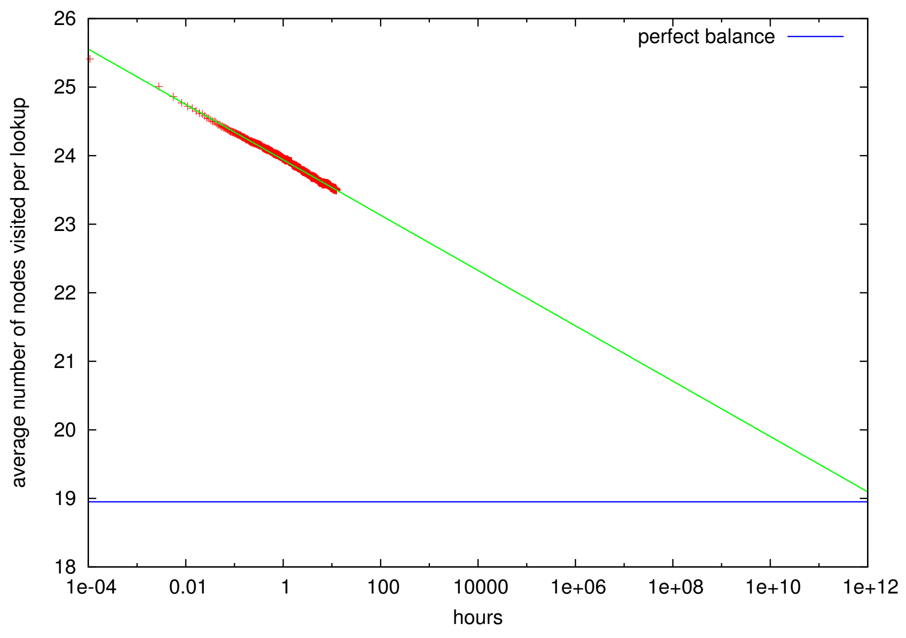


Figure 3.3: How the tree balance of the DICT benchmark improves as random insertions and deletions are performed, including the extrapolated time to reach perfect balance.

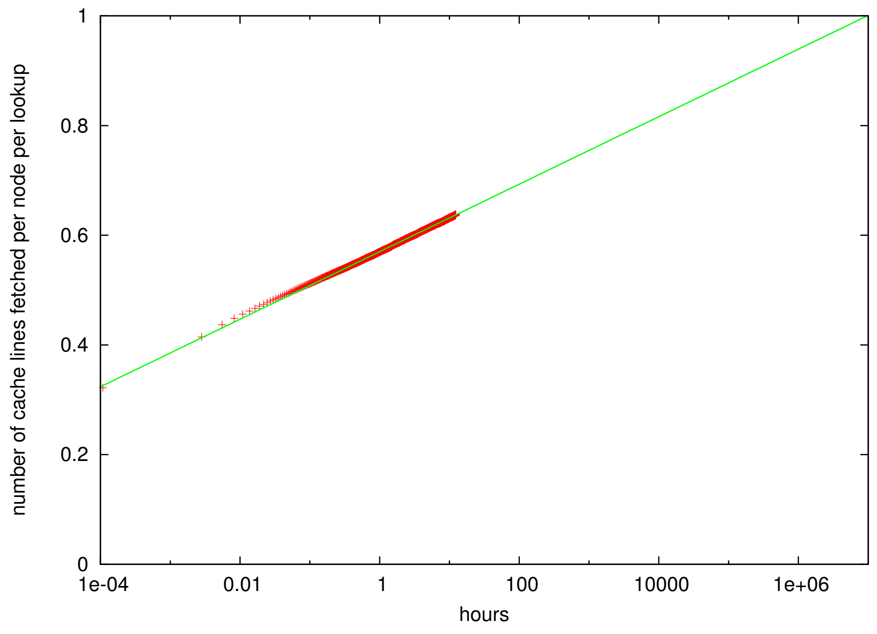


Figure 3.4: Extrapolated number of hours required to run DICT-MOVEFIELDS to terminal layout. The y axis shows layout quality, measured in the number of cache lines fetched per node per lookup, with 1 corresponding to the worst possible layout, and $1/3$ to the best possible layout.

	new blocks seen per node		
layout	lines	pages	red'n
worst	1.00	1.00	0%
goodPage	0.92	0.16	30
goodLine	0.33	0.33	56
optimal	0.33	0.16	60

Figure 3.5: How data layout determines the time to perform an operation, for DICT-MOVENODE (DICT-MOVEFIELDS is similar). The number of new blocks (lines or pages) accessed per node fetch, on average, is given in columns two and three: 0.33 lines is optimal, and for page, 0.16 is optimal. See §3.4.2.1 for description of the layouts.

Results are given in Fig. 3.5, expressed as percentage reduction compared to the time obtained using **worst**. Notice that the **goodPage** layout has near-worst L1/L2 performance. This means that an optimisation that focuses only on TLB performance may in the worst case have a high number of L1/L2 misses. Doing so produces only a 30% reduction in execution time. By contrast, an optimisation that focussed on producing good L1/L2 performance will achieve quite good TLB performance, essentially for free, and achieve a reduction of 56% (**goodLine**). An optimsation that focusses on optimal TLB as well as optimal L1/L2, will only see an additional 4% reduction (**goodBoth**).

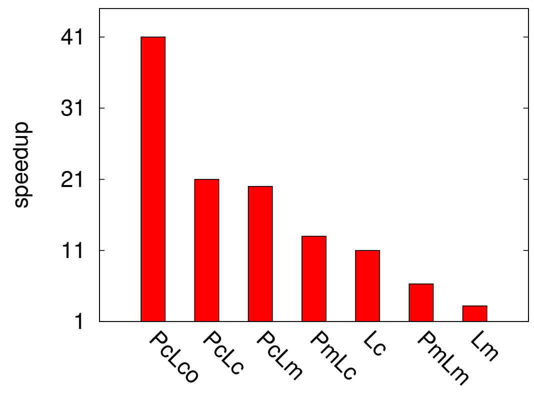
Thus, when designing runtime data movement optimisations for DICT, the most sensible approach is to focus on improving line layout alone, rather than considering pages as well. It is likely that the extra overhead involved in moving around a whole page's worth of nodes would prevent the extra 4% from being realised. A purely page-focused approach is only appropriate when improving L1/L2 performance is hard – for example when nodes are larger, line sizes are smaller or there are fragmentation problems with packing nodes into lines.

3.4.2.2 MLIST

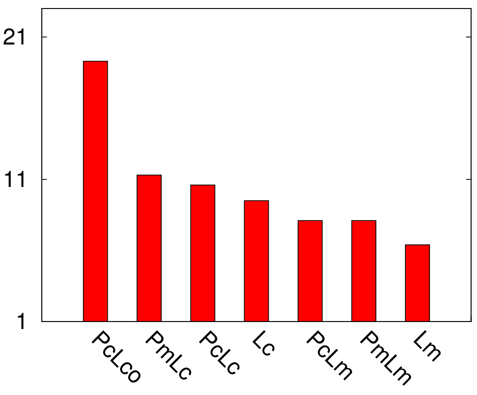
For MLIST, we express layouts using a number of constraints on the line and page layout of the lists. Nodes are uniformly randomly placed in the heap in a way that does not violate the constraints. Each constraint is either a page constraint (**Px**) or a line constraint (**Lx**), where **x** is one of the following:

1. Minimisation (**m**): Within each list, the minimal number of blocks are used; if there are n nodes in the list, then only $\lceil n/nodesPerBlock \rceil$ distinct blocks are seen when traversing it end to end (*nodesPerBlock* is the maximum number of nodes that fit into a block).

2. Clustering (**c**): Numbering nodes by their position in the list, nodes $1 \ldots \text{nodesPerBlock}$ occupy the same block, and $\text{nodesPerBlock} + 1 \ldots 2 \times \text{nodesPerBlock}$ occupy the same block, etc. Note that no nodes from other lists share any blocks.



(a) $v = 2^7$



(b) $v = 2^{15}$

Figure 3.6: The performance of MList for different layouts, for two different list lengths, expressed as speedup compared to the unconstrained layout, -.

3. Clustering + Ordering (**co**): We only use **Lco**. Nodes are clustered as above, and then lines are clustered within pages so that lines 1 . . . linesPerPage occupy the same page and are in order within it, and linesPerPage + 1 . . . 2 × linesPerPage occupy the same page and are in order within it, etc. Note that no nodes from other lists share any pages.

Notice that **Lco** ⇒ **Lc** ⇒ **Lm**, and **Pc** ⇒ **Pm**, and so at most one of the constraints on a block can be used at once. Note also that **Pm** ⇒ **Lm** (but not vice-versa), and that **Lco** ⇒ **Pc**. This gives eight valid combinations of constraints (writing, e.g. **Pc** ∧ **Lm** as **PcLm**, and **-** for the unconstrained layout):

PcLco, **PcLc**, **PcLm**, **PmLc**, **PmLm**, **Lc**, **Lm** and **-**

The motivation for each constraint is as follows. If we ignore the effects of the hardware prefetcher, use of **c** will minimise the miss rate for the block. Using **co** linearises the line address stream within each page, which will hopefully allow the hardware prefetcher to lower miss rates even further. We do not consider the linear ordering of the page address stream (‘**Pco**’), because the hardware prefetcher does not prefetch across page boundaries, so within a L1+L2+TLB+Hardware Prefetcher intuition of the memory hierarchy, using **Pco** should not make a difference. Inevitably, in reality, it does, sometimes improving and sometimes degrading layout, but it is not clear exactly why, and the reasons are probably fairly architecture specific.

The action of the **m** constraint is harder to predict; depending on the list length and the geometry of the cache, it may produce as good a layout as **c** or as poor a layout as **-**. We consider it because it is a far less strict constraint than **c**, and so may be easier to maintain at runtime.

Fig. 3.6 (a) shows the behaviour of the different layouts when $v = 2^7$ (an average of 2^{15} nodes or 64 pages per list), expressed as speedup compared to the unconstrained layout. We may obtain an extremely large speedup of 41× when **PcLco** is used. At least half of this is due to the hardware prefetcher, because **PcLc** only obtains 21×, and **PcLm** obtains 20×. Thus we see that when clustering nodes into pages, there is no point hierarchically clustering nodes into lines as well, unless the lines are in order (**PcLco** ≫

PcLc $\simeq$ **PcLm**). Clustering nodes into lines (**Lc**) yields about half the speedup (11×), and a bit more if the number of pages used can be minimised (**PmLc** – 13×). Even page minimisation (**Pm**) and line minimisation (**Lm**) provide some improvement (6.3× and 3.2×, respectively).

When $v = 2^{15}$, each list is smaller than a page, and the distinction between layouts is less severe, as shown in Fig. 3.6(b). Simple line clustering (**Lc**) yields a speedup of 9.5×, and up to 11.3× if page clustering or minimisation is used (**PcLc**, **PmLc**). Simple line minimisation (**Lm**) yields 6.4×, and 8.1× if page clustering or minimisation is used (**PcLm**, **PmLm**). Clustering nodes into lines and ordering them within pages yields 19.3×, but wastes a lot of memory (**PcLco**), because lists cannot share pages. If lists are forced to share pages, a layout that cannot be described using the constraints, memory is much more reasonable but a speedup of only 13.8× is obtained. The discrepancy is possibly because the hardware prefetcher does not prefetch across page boundaries.

In conclusion, for MList there is a range of layouts, of varying efficacy and difficulty to achieve. The smaller the average list length, the smaller the distinction between different layouts. Greatest speedup is obtained by clustering lines in order within pages (thus exploiting the hardware prefetcher), followed by other forms of page clustering, followed by line clustering. Minimisation can also be useful by itself or combined with clustering.

3.4.3 Upper bounds

If we knew the optimal layout for each benchmark, we could use it to produce a quite convincing upper bound on the performance improvement a data movement optimisation could yield. The benchmark would be run for its normal length (e.g. $1e8$ operations for Dict), then the structure would be re-laid out using the optimal layout, then the time for a small number of operations would be measured. Layout degradation is sufficiently slow that the layout stays close enough to optimal while the operations are being timed. This gives an upper bound because no optimisation can produce a superoptimal layout, nor could it run with lower overhead.

The optimal layout for these benchmarks isn't known, but it seems unlikely that a runtime data movement optimisation would achieve a better layout than the static ones we have used here. Using the best layouts from this chapter, we obtain the bounds given in Fig. 3.7.

The MList figures needs a little explanation, because we would not expect the intermediate values of v have the highest upper bound. If all benchmarks were run until they reached terminal layout, the upper bounds would indeed increase monotonically as v decreased. $v = 2^{19}$, $v = 2^{15}$ and $v = 2^{11}$ reach equilibrium, but $v = 2^{7}$ doesn't (because operations take longer due to increased list length), and so the figure of $v = 2^{7}$ is artificially low because not enough degradation has taken place.

3.5 Cache-aware solutions

As well as comparing the optimisations in this thesis against the rough upper bounds we produced in the previous section, we will compare with cache-aware RDSs.

DICT-MOVENODE $s = 1$	55%
DICT-MOVENODE $s = 0.1$	57
DICT-MOVENODE $s = 0.01$	48
DICT-MOVEFIELDS $s = 1$	36
DICT-MOVEFIELDS $s = 0.1$	23
DICT-MOVEFIELDS $s = 0.01$	8
MLIST $v = 2^7$	11.7×
MLIST $v = 2^{11}$	36.1
MLIST $v = 2^{15}$	11.3
MLIST $v = 2^{19}$	2.14

Figure 3.7: Rough upper bounds on the improvement a runtime data movement optimisation could give. See §3.4.3 for a description of how the bounds were produced.

3.5.1 DICT

We will compare against a B-Tree, the canonical cache-aware tree structure for the dictionary problem. Our implementation is as described in Cormen, Leiserson and Rivest [20]. Each node stores the number of keys k ($\leq max$), k keys, k values, $k+1$ pointers and a field indicating if the node is a leaf or not. Note the key-value pairs are stored within the tree, not just at leaves. In the spirit of the optimisations in this thesis, we try several different orders of fields within nodes and the maximum degree max to find the best solution for the test hardware. We also optionally 'pad' nodes until their size is a multiple of the cache line size. The internal layout of the keys, values and child pointers are important when nodes are larger than the line size.

An important part of the insert, delete and lookup operations is finding which subtree to descend next (or checking to see if a key is within a node). This can be done either by linear scan (linear) or binary chop (logarithmic). In practice the former is faster for the node sizes that we use.

The typical effect of varying node size is shown in Fig. 3.8. The traditional approach of using nodes the same size as lines works well, but it is slightly better to use nodes two lines wide. The best times obtained are given in Fig. 3.9. Also shown in this figure are times expressed as percentage reductions from the unoptimised DICT-MOVENODE and DICT-MOVEFIELDS variants (the reduction that is obtained if these benchmarks were changed to use a B-Tree implementation), and the maximum reduction figures of Fig. 3.7, as calculated in §3.4.3. The figures are for worst-case memory (when all nodes in the B-Tree have the minimum number of keys), $M_{worst} = 1.7$. M_{worst} can be decreased to around 1.5, but this will half the reduction obtained for MOVENODE and make the the MOVEFIELDS reduction negative.

We observe that sufficiently low-overhead data movement may be able to equal or even beat the cache-aware solution, particularly for MOVEFIELDS.

3.5.2 MLIST

Recall from §2.2.4.2 that cache-aware linked list structures store several keys in one block, allowing the number of keys to vary between min and max. We evaluate the simplest

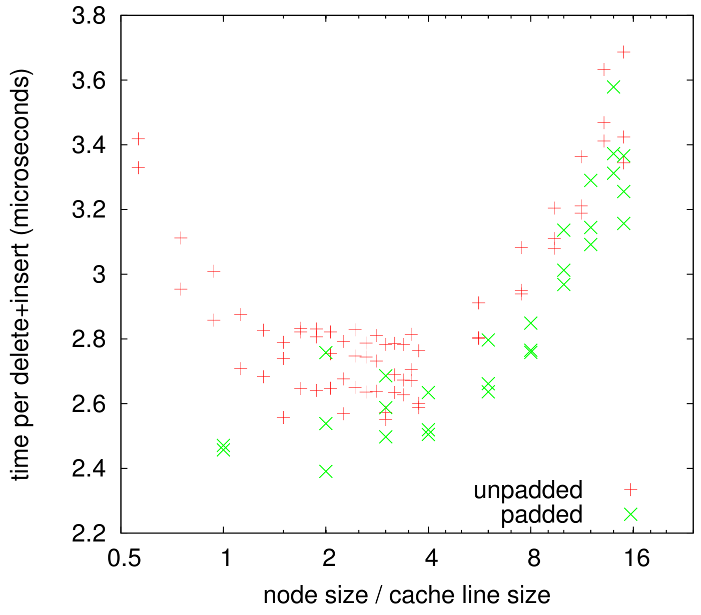


Figure 3.8: How node size determines B-Tree performance. Data obtained by varying the order of keys, values and pointers within nodes, and either padding nodes to a whole number of cache lines or not.

s	time	MOVENODE		MOVEFIELDS	
	(μs/op)	red'n	max red'n	red'n	max red'n
0.01	0.93	42%	48%	0%	8%
0.1	1.02	53	57	16	23
1	2.39	39	55	9	36

Figure 3.9: The performance of the best B-Tree (cache-aware) implementation of the DICT benchmark, compared against the rough upper bound on a runtime data movement optimisation's performance. The "red'n" columns are the performance of the cache-aware implementation, expressed as reduction in execution time from the unoptimised benchmark, the "max red'n" is the upper bound as given in Fig. 3.7. See §3.5.1.

method, which uses $min = max/2$ (the 'half-full' structure of [6]), and a more complicated method allowing any $min < max$ as described by Rubin et al. [60], the *'VCL'* structure (which is similar in spirit to the reimplementation of C++ STL Lists of Frias et al. [25]).

We investigate the performance of the VCL structure and the half-full structure by altering the benchmark to use them and then running the benchmark for the usual time. We vary the block size (i.e. the value of max) and for VCL we also vary min. The values of min used are $\{max/4, max/2, 3max/4,$ <three evenly-spaced intermediate values>$, max-1\}$. We use $min = max - 1$ because it has the minimal worst-case memory requirement possible for the choice of max.

Results are given in Fig. 3.10, only the best of the two structures is shown. We express results as speedups compared to the unoptimised form of the benchmark, plotting against worst-case memory $M_{worst} \leq 3$, calculated as described in §6.3.3. Also shown is the maximum speedup possible by runtime data movement, as shown in Fig. 3.7.

Almost all the time the VCL structure is better than the half-full structure, which is expected because the former allows better balancing of node density (memory usage) and

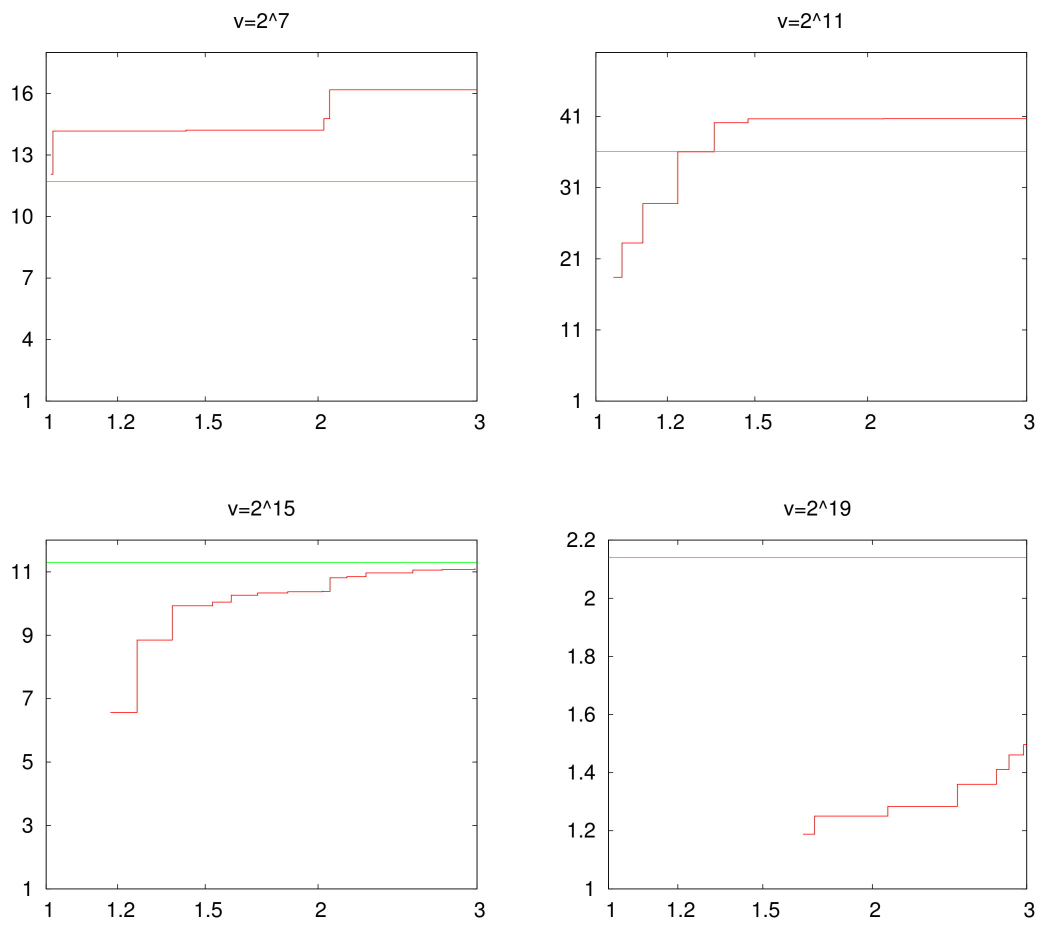


Figure 3.10: The performance of the best cache-aware solutions for the MList benchmark (see §3.5.2). Figures are given as speedups compared to the unoptimised form of the benchmarks, plotted against *worst-case* memory usage M_{worst}. For values of M_{worst} where no data is given, there is no cache-aware solution. The green horizontal line is the maximum possible speedup achievable by runtime data movement, as given in Fig. 3.7.

overhead. The results show that for longer lists less memory is required to obtain reasonable performance, and increasing M_{worst} results in steadily better performance. The cache-aware structures exceed the performance of the best possible runtime data movement optimisations when $v = 2^7$, but for other values of v we conclude that sufficiently clever data movement could be more efficient than the cache-aware solutions. Even where the cache-aware solutions exceed the runtime data movement upper bound, they do so by at most a factor of 1.4, so runtime data movement may still be useful.

3.6 Summary

In this chapter we have described the test machine, the benchmarks, the latter's layout and layout degradation properties on the former, produced upper bounds on the performance of runtime data movement, and related this to the performance of the cache-aware RDS equivalents of the benchmarks.

3.6.1 Benchmarks

We use two benchmarks, lookup/insertion/deletion in binary search tree (Dict) and insertion/deletion in multiple linked lists (MList). We may consider these as representative of two important classes of traversal pattern – branching and linear. This thesis differs from previous work on static RDS layout and automated runtime data movement schemes like GCs in that the benchmarks are simpler than traditional suites like SPEC or Olden, but as discussed in §2.3.4 this makes it possible to investigate a greater number of optimisations and understand their performance.

3.6.2 Experiments

The test machine is relatively modern desktop hardware, with an effective line size of 128 Bytes (8–16 RDS nodes) and a standard 4k page size. A very good (near optimal) data layout is used for the initial layout of the RDS for both benchmarks. For most experiments, the benchmark is run for a fixed number of operations (chosen so that the unoptimised form takes around ten minutes), rather than waiting for layout to reach a terminal quality. When the terminal behaviour of an experiment is of interest, experiments will be run until terminal layout is reached. The result reported is the time for a small group of operations at the end of the execution of the benchmark.

3.6.3 Layout and degradation

We investigated the layout and layout degradation of both benchmarks. Good performance for the binary search tree in the Dict benchmark requires only a good line layout, but the linked lists of the MList benchmark require good page and good line layout, with the relative importance depending on the average list length. For MList we investigated the performance of approximations to the optimal layout **PcLco**, which we described hierarchically as **PxLx**, where **x** is **c** (clustered), **co** (clustered ordered) or **m** (minimised). We demonstrate that layout degradation causes significant reduction in performance after only a few minutes of execution, and for some benchmarks performance will continue to degrade beyond this time.

3.6.4 Upper bounds on runtime data movement

Upper bounds on the performance of runtime data movement were produced by running the benchmarks for their normal length, then re-laying out the RDS using a very good (effectively optimal) layout, and reporting the difference between the good layout and the unoptimised layout. This simulates the performance of a data movement optimisation that achieves optimal layout with zero overhead; no runtime data movement optimisation can exceed this.

3.6.5 Cache-aware alternatives

The behaviour of the canonical cache-aware structures for both benchmarks were investigated – B-Trees for Dict (we varied the node size to find the best solution), and two solutions based upon relaxing node density for MList, where allowing higher values of worst-case memory increases performance (we varied block size and minimum node density where appropriate). The cache-aware structures are sometimes beaten by the data movement upper bound, which suggests that it may be possible to produce optimisation that are more effective than the traditional cache-aware structures.

Chapter 4

Reallocation

In this chapter we review the use of coallocators to obtain a good initial layout for an RDS. We then enumerate a much larger class of coallocators and demonstrate how they can be adapted to maintain layout by runtime data movement.

4.1 Introduction

A *'coallocator'* is a memory allocation function that attempts to allocate a new object (the *'allocatee'*) in the same memory block as some other object specified by the caller (the *'hint'*). Any call to `malloc` (at an *'allocation site'*) can be replaced with a call to a coallocator without changing the semantics of the program. However, the intended use of a coallocator is to allocate objects so those objects that are accessed close in time share a line or page, reducing cache miss rates. The job of the applier is to identify a suitable hint object at the time when the allocatee is being allocated, and pass a pointer to it to the coallocator.

In practice, a good choice of hint is usually one of the objects that takes a pointer to the allocatee in the immediate vicinity of the allocation site. Failing that, another choice of hint is an object that the allocatee points to. A simple example can be seen in Fig. 4.1. The applier has made the assumption that `b` is likely to be accessed through `a`, and thus attempting to locate these objects in the same block may remove a cache miss. If the applier's assumption is wrong and some other object takes a pointer to `b` and `b` is always accessed through this new pointer, then memory hierarchy performance is unlikely to improve.

4.1.1 An example coallocator

Chilimbi et al. [15] present a coallocator called ccmalloc which comes in three variants. All three variants try to allocate in the same line as the hint. If that fails, because the hint's line is full, then the three variants proceed as follows:

1. ccmalloc-closest: in the hint's page, use the closest line to the hint with enough space
2. ccmalloc-firstFit: in the hint's page, use a line with enough space using a first-fit policy

```
struct B {...};
struct A {... struct B *b; ...};
...
struct A *a;
a=(struct A*) malloc(sizeof(struct A));
...
a->b = (struct B*) malloc(sizeof(struct B));
```

```
struct B {...};
struct A {... struct B *b; ...};
...
struct A *a;
a=(struct A*) malloc(sizeof(struct A));
...
a->b = (struct B*) coalloc((void*)a,
                           sizeof(struct B));
```

Figure 4.1: Typical use of a coallocator at allocation time, before and after application. The function `coalloc` takes two parameters, the first is the hint object, the second is the size of the new object being allocated.

3. ccmalloc-newBlock: in the hint's page, use an empty line

These 'second stages' can all fail: the first two if there is no single line with enough space, the third if there is no empty line. The action taken when the second stage fails (the 'third stage') was not specified by Chilimbi et al. and we will later show that the choice of the third stage is very important when coallocators are used dynamically.

It was found that closest and firstFit performed similarly, and newBlock usually performed much better, and only rarely a little worse. The intuition behind this is that allocating a node x in an empty line if the hint's line is full will enable future calls to ccmalloc to succeed when x becomes the hint. In a typical RDS, a large proportion of nodes have some descendants, and thus any node is quite likely to be used as a hint at some point in the future.

Chilimbi et al.'s *'experience with ccmalloc indicates that a programmer unfamiliar with an application can select a suitable [hint] by local examination of code surrounding the allocation statement and obtain good results'*, and so it appears that often the obvious choice of hint is the correct one. Later in this chapter we will consider the use of multiple hints.

4.1.2 Runtime data movement

Maintaining a good data layout for a structure is not just a matter of correct allocation, because layout degradation may be caused by the movement of existing nodes within the structure. To prevent degradation in these situations, a node must be moved in memory. This can be easily done by allocating a new object, copying the fields into it, then freeing the old object. Some mechanism must exist for updating all pointers to the object, as discussed in Chapter 2. We call this procedure *'reallocation'*. To use this technique, the applier must inspect the program for promising *'reallocation sites'* – program points where it is possible and useful to move an object in memory to improve data layout. These points are distinct from allocation sites, where a call to `malloc` already exists. Under this definition, a node may be reallocated at any program point. In this thesis, we restrict ourselves to those reallocation sites that are the result of pointer updates. More specifically, when a pointer in an object a is updated to point to an object b, a reallocation of either object a or b is attempted, using suitable hint object(s) – the most natural choice being the other object. A simple example is given in Fig. 4.2. Note that in this example,

```
struct B {...};
struct A {... struct B *b; ...};
...
struct A *a; struct B *b;
a=(struct A*)malloc(sizeof(struct A));
b=(struct B*)malloc(sizeof(struct B));
...
a->b = b;
```

```
struct B {...};
struct A {... struct B *b; ...};
...
struct A *a; struct B *b;
a=(struct A*)malloc(sizeof(struct A));
b=(struct B*)malloc(sizeof(struct B));
...
struct B * temp = (struct B *)
   coalloc((void*)a,sizeof(struct B));
memcpy(temp,b,sizeof(struct B));
<update all other pointers to b>
free(b);
a->b = temp;
```

Figure 4.2: Typical use of a coallocator to perform runtime data movement, before and after application. The function `coalloc` takes two parameters, the first is the hint object, the second is the size of the new object being allocated.

the coallocator takes a parameter giving the size of the allocatee, but in the rest of this thesis, since all objects are same size, this parameter is omitted.

We cannot say easily which end of a pointer should be reallocated when a pointer update occurs, nor which hints to use. The job of the applier is to use their higher-level understanding (i.e. a compiler's static analysis or a programmer's understanding of the program) to resolve a sequence of pointer updates into sensible choices of reallocatees and hint objects. For example, in the Dict-moveNode benchmark, during deletion, when the deletee x is substituted by some other node node y, the applier might resolve the three pointer updates necessary to perform the substitution (parent+two children) into a single reallocation site involving node y, with its new parent and new children as sensible choices for hints.

4.2 Implementation

All the nodes of the RDS live in a separate area of memory, called the *'RDS heap'*. The RDS heap is created when the first node of the RDS is created and destroyed when the last node is freed, and supports node allocation and freeing during its lifetime (it is a region a.k.a. pool [26, 27, 44, 68]). The heap is controlled by a memory manager, which provides various functions to the program, such as allocation and freeing, and to inspect the space in blocks, etc. The size of the RDS heap is chosen by the programmer, expressed as a factor m larger than the number of nodes in the RDS (recall the definition of m from §3.3.5).

Reallocation (movement of an object when a pointer update occurs at a reallocation site) is achieved by inserting code into the program, which typically invokes various memory manager functions, and may eventually result in the allocation of a new node. If allocation is successful, the code copies the fields of the node and updates pointers, then frees the old node by invoking a memory manager function.

Furthermore, when reallocation is applied, all allocation sites (allocation of RDS nodes using `malloc`) must be altered to allocate in the RDS heap using the memory manager's functions.

We refer to the optimisation of a program by inserting data movement code into reallocation sites and replacing calls to `malloc` at allocation sites simply as 'reallocation'. We do this with the understanding that in a small subset of programs all layout degradation may be caused by allocation not pointer update and so no reallocation sites exist.

Note that the behaviour at the two types of sites is very different. At an allocation site, a node is simply allocated. At a reallocation site, an *attempt* is made to allocate a new node x, and if successful, the fields of the reallocatee are moved into it, and then all parents of the reallocatee must be updated (See §2.3.2.1). With some mechanism to update parents in place, we can mostly ignore the difference between the two types of sites, apart from for ensuring that allocation always succeeds at allocation sites.

In the benchmarks of this thesis, there are no allocation sites within the main loop of the program because the node deleted from the structures is reused during insertions, but in general allocation sites will be common.

4.2.1 The RDS heap

Because the heap stores only homogeneous nodes, it can be decomposed into nested blocks, which contain an optional header and several smaller blocks. The smallest block is *cell* which either holds a node or not. A *line* contains several cells and an optional header, by default corresponding to a L2 cache line – 128 bytes on the test machine. A *page* contains several lines and an optional header, and by default corresponds to the 4096 byte machine page. The RDS heap itself is composed of thousands of pages, and an optional header.

More flexible heap arrangements are possible, but the advantage of this arrangement is that the memory manager is far simpler and efficient. Also, information relevant to a block is found within it, which will reduce cache misses. For RDSs built out of nodes of several different sizes, recall that node size may be altered by padding or splitting into hot and cold parts using one of the methods of §2.2.5.

4.2.2 Applying reallocation

This section discuss the application of reallocation with reference to the symbols in the grammar in Fig. 4.3.

site, strategy, siteStrategy, siteStrategies: Applying reallocation to some part of the program is achieved by applying a *'strategy'* to a site. A site is a labelled place in the program, and is either an allocation site or a reallocation site. Each site has an associated node, an *'allocatee'* or *'reallocatee'*. For example for DICT there is a site labelled insert, which is a reallocation site (because the node from the previous deletion is reused), and the associated node is the one being inserted. The aim of the strategy is always to (re)allocate the (re)allocatee.

stage: A strategy is a list of one or more *'stages'*. Some types of stage will always allocate, but others will sometimes fail (for example, a stage may attempt to allocate in a particular block, which will fail if the block is full). The stages are thus tried in turn until the (re)allocatee has been successfully (re)allocated.

coalloc, coallocHint, coallocGlobal: Each stage is split into two halves, the coallocator and its input. The coallocator describes how to (re)allocate the (re)allocatee, and may either take the address of a single hint object, or a boolean. For example, allocation in the

Terminal symbols are in san-serif, non-terminals in normal font. (A|B) is used for either A or B. (C)* is used to mean zero or more repetitions of the symbols C.

siteStrategies		$\rightarrow$	siteStrategy ('$\wedge$' siteStrategy)*
siteStrategy		$\rightarrow$	site ':=' strategy
strategy		$\rightarrow$	'(' stage (',' stage)* ')'
stage		$\rightarrow$	(coallocHint ':' search | coallocGlobal ':' bool)
coalloc		$\rightarrow$	(null | coallocHint | coallocGlobal)
coallocGlobal		$\rightarrow$	find_{CL} find_{LP} find_{PH}
		$\rightarrow$	find_{CP} find_{PH}
		$\rightarrow$	find_{CL} find_{LH}
coallocHint		$\rightarrow$	find_{CL} find_{LP} page
		$\rightarrow$	find_{CP} page
		$\rightarrow$	find_{CL} line
coallocPattern		$\rightarrow$	'$\langle$' (coalloc ((','|';') coalloc)* '$\rangle$'
searchPattern		$\rightarrow$	'$\langle$' (search ((','|';') search)* '$\rangle$'
bool		$\rightarrow$	(true | false)
find_{XY}		$\rightarrow$	(lowest_{XY} | cyclic_{XY} | first_{XY} | newblock_{XY} | emptiest_{XY})
DICT:	site	$\rightarrow$	(insert | deleteCut | deleteMove)
	search	$\rightarrow$	(parent_k | child_k) $(k \geq 0)$
MLIST:	site	$\rightarrow$	(insert | delete)
	search	$\rightarrow$	(first | last_k | next_k) $(k \geq 0)$

Figure 4.3: The grammar used in this chapter.

same line as the hint may be attempted, or allocation in an empty line (with and without a hint address, respectively). The form of the coallocators is discussed in the next section.

search: For coallocators that take a hint address, the second half of the stage is the *'search type'*. This describes where the stage searches for a hint object. For example, a simple search type for DICT's insert site is just to use the parent of the inserted node as the hint. The form of the searches depends on the benchmark, and is discussed in a later section.

bool: For coallocators that take a boolean, the second half of the stage is a boolean, enabling or disabling the coallocator.

Note that in general a stage cannot be decomposed into some code that performs the search, then some code that performs the coallocation – for example in the MLIST benchmark when the search type is 'continue along the list' and the coallocator is 'allocate in same line', the code that is inserted is a loop (the search) which traverses the list, the body of which inspects the cache line of each node to see if there is space to allocate (the coallocator). Moreover, it may often be more efficient to not code each stage separately: for example, the first stage might be to continue along the list for some distance looking for a non-full line, and the second stage the same but for a non-full page, which can be implemented more efficiently as one search, rather than two.

In summary, applying reallocation requires selecting a *'strategy'* for each *'site'*. Each strategy is composed of a number of *'stages'*. There are two types of stages. The first type of stage is composed of a coallocator and a *'search type'*, expressing coallocation in the same block as some hint object. The search type defines *where* to look for the hint, and the coallocator defines both *what sort of hint* to look for (e.g. one occupying a line with some space in it), and *what to do* once the hint is found (e.g. allocate in the line). The other type of stage is composed of a coallocator and a boolean, expressing allocation without use of a hint, where the boolean enables or disables the allocation. Code is inserted into the program to execute the stages in order until allocation or reallocation succeeds.

4.2.3 Coallocators

In this thesis, a coallocator is a function that takes either one cell address (the hint) or a boolean, and yields one cell address, which it has allocated[1]. Coallocators are a sequence of *'memory manager operations'*, each of which take either an input address (either a line or page address) or a boolean, and yield one address (either a cell, line or page address). If the result is a cell address, the operation has allocated it. We write the sequence of operations right to left as if function application – i.e. AB means B is applied first.

There are two *'block-mapping operations'* line and page, which both take a cell address as an input and output the cell's line or page address. The other operations are of the form find_{XY}, where X is either C(ell), L(ine) or P(age), and Y is either L, P or H(eap). If Y is Heap, the input to the operation must be a boolean, otherwise the input must be the address of a Y (i.e. a line or page address). If any find operation fails, the coallocator fails. In the grammar, the right hand side of the coallocHint and coallocGlobal rules gives all possible combinations of memory manager operations.

Example: For DICT-MOVEFIELDS there are two sites insert and deleteCut, and the reallocation applied might be of the form:

insert := (find_{CL} line : search, find_{CL} find_{LH} : true)

$\wedge$ deleteCut := (find_{CL} find_{LH} : true)

The terminals in the grammar are shown in san-serif, non-terminals are in normal font. The statement above says that the reallocation at the insert site attempts to allocate in some manner in the same line as a hint object found by some search, failing that some line in the heap is found and allocation occurs within it. deleteCut is similar.

We use five different simple find_{XY} operations, based around extending ccmalloc for dynamic use (in the grammar, the right hand side of the find_{XY} rules). These five operations satisfy two basic requirements. Firstly, the need to obtain *any* non-full X within a Y, as quickly as possible, without caring about which X is obtained. Secondly, the need to obtain a new (a.k.a. empty) X within a Y (as in ccmalloc-newBlock's finding of an empty line in the hint's page). We will extend this to consider what happens if we use the emptiest block instead of an empty block (more properly, a block of maximal space). This is appropriate because we wish to strictly bound memory usage. We will find that coallocators made of even such simple operations will yield both good improvements to execution times and interesting behaviour.

[1]Note that in Figs. 4.1 and 4.2, coallocators took an additional parameter giving the size of the allocatee, but this is not required for DICT and MLIST since all nodes are the same size.

Finding any X in a Y:

1. lowest_{XY}: returns the lowest (in memory) nonfull X in the Y. Since fullness is a binary property, this may be done quickly using operations on bitfields.

2. cyclic_{XY}: as above, but for each Y in the RDS heap the last position looked is remembered, and the next search is started from that point, cycling around to the bottom of the Y when the top is reached. Sometimes this offers an improvement over lowest, because less scanning work is done overall – this is a heuristic that depends on a favourable distribution of allocation and frees.

3. first_{XY}: Every Y has doubly-linked list implementation of a stack associated with it, storing all its nonfull X. When an X becomes full it is cut from the list, likewise when an X becomes nonfull it is pushed onto the stack.

These operations behave differently depending on the choice of Y. For example, lowest_{CL}, cyclic_{CL} and first_{CL} will produce equally good layout, because the memory hierarchy doesn't deal with scales smaller than a cache line. When considering lowest_{CP} etc, a different layout will be produced because the cells found by the three methods may be in different lines. Note also that the three have very different overheads depending on the choice of Y – For example within a line first, lowest and cyclic are all constant-time procedures (lowest and cyclic are done by table lookup). Within a heap, however, lowest and cyclic are linear in the size of the heap (but given a non-pathological distribution of allocation and frees one expects the latter to be quicker), but first is still constant-time.

Finding an empty or emptiest X in a Y:

1. $\mathsf{newblock}_{XY}$: Every Y has a stack, as with first_{XY}, which stores all empty blocks (instead of all nonfull blocks).

2. $\mathsf{emptiest}_{XY}$: Every Y has an X-listset associated with it. An X-listset is a set of list whose elements are Xs. The purpose of the listset is to partition all the X within the Y into sets based upon the number of cells in the X. The simplest partition is to locate all the X with i occupied cells in them on the i^{th} list. Each set is implemented as a doubly linked list. To find the emptiest X within a Y, the lists of the Y are searched in increasing numerical order until a nonempty list is found. Each allocation or free in an X requires moving the X from one listset to another (respectively, to a higher- or lower-numbered list).

   An alternative is to partition so that, for example, blocks with $[0, k-1]$ full cells go in the first list, blocks with $[k, 2k-1]$ go in the second list, etc, with a special list reserved for blocks that are full. The advantage of this partitioning is that the update and search cost of the listset is reduced by a factor of k, with a corresponding loss of resolution. For pages, experiments show that it is most efficient to use the second scheme, and it appears that the value of k does not need to be tuned very precisely. For lines, the first partitioning is necessary, otherwise too much resolution is lost and the ability of an optimisation to find the best quality lines is reduced. Unless otherwise stated, for the rest of this thesis we use the most efficient listset partitioning.

Note that by scanning lists in the reverse order (ignoring the list of full blocks), we may find an X of *minimal* nonzero space, denoted fullest$_{XY}$, which we use in the next chapter (but not this one).

Example revisited: Again for DICT-MOVEFIELDS, the reallocation applied might be of the form:
insert := (lowest$_{CL}$ line: search, lowest$_{CL}$ emptiest$_{LH}$: true)
$\wedge$ deleteCut := (lowest$_{CL}$ emptiest$_{LH}$: true)
The statement above says that the reallocation at the insert site attempts to allocate in the hint's line using the lowest operation, and failing that attempts to allocate in the same way in a line of maximal space in the heap.

4.2.4 The memory manager

Coallocators are composed of distinct operations, each of which must be performed by the memory manager in some manner. At the very simplest level, the memory manager could do this simply by knowing which cells were in use in the heap, and by maintaining whichever lists are required (if first, newblock, emptiest or fullest are used). However to increase performance the memory manager can optionally maintain other data structures which may (or may not) speed up certain operations. Overall improvement only occurs if the cost of maintaining the structures is not too high. For example, to evaluate emptiest$_{PH}$, we must maintain a set of lists. When an allocation or free occurs, it may be necessary to move a page from one list to another. This requires knowing how many cells are in the page. This can be calculated by counting how many cells in the page are in use, but this is inefficient – it is more efficient to maintain a counter of full cells for each page. Formally, evaluating emptiest$_{PH}$ requires maintaining a set of lists, which requires evaluating the value of num_{CP}. This can be done from scratch (by counting) or by memoization (by explicitly storing a count of occupied cells in every page). Similarly, evaluating cyclic$_{XY}$ and lowest$_{XY}$ requires evaluating full_X. This could be done from scratch by counting the number of occupied cells in the X, or by memoizing this information by maintaining a bitfield containing values of full_X.

The data structures maintained by the memory manager (*'components'*) are as follows.

1. Bitfields: It is natural to memoize full_X as one large bitfield – ie the i^{th} bit in the bitfield is 1 if the i^{th} X in the heap is full – but for reasons of locality, we may split the bitfield into chunks. For example, it may be better for each line to have a bitfield stored within it saying which of its cells are full, rather than having one large bitfield stored in the heap header. Enabling the component `bitfield`$_{XY}$ adds a bitfield to each Y's header, where the i^{th} bit of `bitfield`$_{XY}(y)$ is set if the i^{th} X of Y y is full. Note `bitfield`$_{XY}$ stores identical information to `bitfield`$_{XZ}$, just split up into different-sized chunks.

2. Counts: Enabling the `bitcount`$_{XY}$ component adds a 32-bit word to each Y containing a count of the number of full X in the Y. It can be thought of as a memoization of num_{XY}, and is also a count of the total number of set bits in `bitfield`$_{XY}$, if it is enabled.

parent$_k$	($0 \leq k < \infty$) During each descent of the tree, a circular buffer of size k is used. Each time there is a block transition, the block address is pushed to the buffer. When the stage executes, each block in the buffer is inspected (starting with the most recent), until a non-full block is found (or all blocks are full). The descent is either due to the insert operation, or due to the lookup operation followed by any further traversing within the delete operation.
child$_k$	($0 \leq k < \infty$) The children, then grandchildren, up to a depth of k, are searched. Within a generation, nodes are tried in random order. Note that this search type is for deleteCut and deleteMove only, because the (re)allocatee in insert does not have any descendants.

Figure 4.4: Search types used for DICT.

3. listsets: Enabling the `listset`$_{XY,type}$ component adds a listset of type *type* to each Y. The listset type expresses how all the X within each Y are to be partitioned into lists, encompassing free lists (storing only non-full blocks on a single list, used for first), empty block lists (storing only empty blocks on a single list, used for newblock) and partitioning the X into finer size classes (used for emptiest and fullest). Listsets are implemented by adding one or more list root objects to each Y's header, and then adding a list node object to each X. List node objects and root objects occupy 8 Bytes and contain forward and backward pointers. The list must be doubly linked so an X can be cut out of its list in $O(1)$. Adding list node objects is done differently depending on the X. If X is a line or page, then the node object is put in the X's header. If X is cell, then the node object lives within the cell. Since only free listsets are relevant when X is cell, the cell is always empty and so this is possible. Note that this places a restriction on the the number of `listset`$_{C*,*}$ components that can be enabled, since a cell may not have enough space to store all the list node objects.

To implement a particular coallocator, each operation must be 'bound' to the component that implements it. For example lowest$_{CL}$ could be bound to any three of `bitfield`$_{C*}$. Once this has been done, the C code that implements the memory manager – i.e. methods to carry out each operation and to update components after allocation or freeing – is generated automatically and then compiled with `gcc -O3`. It was found that this process generated code as efficient as an early handwritten memory manager.

4.3 Sites and searches

This section explains the (re)allocation sites and search types used for each benchmark.

4.3.1 DICT

When a node is inserted there is a reallocation site. This is site insert.

Recall that to delete a node x we inspect the number of children it has. If it has one, the delete algorithm just cuts the node out of the tree. In this situation, we may move either the node above the new edge, or the node below it, or both. We consider this as two separate sites deleteOneCutAbove, deleteOneCutBelow.

In the two child case, the delete algorithm (randomly) either, replaces x with the rightmost node of the left subtree of x, or replaces x with the leftmost node of the right subtree of x. Let y be the node that replaces x. The replacement can be done either by moving the fields of y into x and then cutting y from the tree (moveFields), or by cutting y out of the tree and then moving node y to x's position in the tree (substituting node y for node x, moveNode). In both situations, we can deal with the cutting of y as above (yielding two more sites, deleteTwoCutAbove, deleteTwoCutBelow). For moveNode, when the substitution of y for x occurs, we can reallocate the node y, which is site deleteMove.

To make a more thorough exploration of reallocation possible, we will perform reallocation at the deleteOneCutBelow and deleteTwoCutBelow, but not deleteOneCutAbove or deleteTwoCutAbove. In other words when a cut occurs ($x \to y \to z$ becomes $x \to z$), the node z is reallocated, rather than x. This is slightly easy from an implementation and efficiency point of view because the parent of z is known, whereas the parent of x would have to be kept track of explicitly, incurring an overhead.

Furthermore, we will use the same strategy at deleteOneCutBelow and deleteTwoCut-Below, denoting the merged site deleteCut. Thus, when applying reallocation to Dict, three strategies must be chosen:

1. insert=*strategy*: reallocating an inserted node
2. deleteCut=*strategy*: reallocating the node below a cut, during delete
3. deleteMove=*strategy*: reallocating the moved node, during delete (for moveNode only)

The search types used for the insert, deleteCut and deleteMove sites are summarised in Fig. 4.4. Observe that parent_k stores the address of the last k blocks that caused a block transition, and then inspects until an empty block is found, moving further up the tree away from the (re)allocatee. There are several reasons why we have adopted this simple heuristic. The best solution from the point of view of layout would be to inspect every block during traversal, and use the last nonfull one. This incurs too much overhead, due to inspecting block headers. Thus we store a buffer of k block address and inspect them for nonfullness only if the parent_k search is invoked. Ideally, one would like to use the last k distinct blocks, but this cannot be done efficiently. For pathological sequences of blocks, the heuristic will be quite ineffective, however, the tendency of the reallocation methods we try in this chapter is to produce clusters of nodes within blocks, and so if reallocation is working well, this block transition heuristic will be quite close to using the last k distinct blocks.

parent_k does not access any additional nodes apart from those used in the traversal. In particular, if the search is to find a *line*, no cache misses will occur because determining if the line has space only requires reading the line header, which is in stored within the line. Similarly if the search is to find an empty *page*, no TLB misses will occur (although L1/L2 misses may occur). Use of $\mathsf{child}_{\geq 1}$ will involve additional node accesses and further misses because it searches in the tree below the (re)allocatee.

lastNonFull	During the traversal to the place to perform the insertion/deletion, the block of each node seen is inspected to see if there is space. The last such block is used.
firstNonFull	As above, except the first such block is used.
last_k	$(0 \leq k < \infty)$ During traversal, a circular buffer of size k is used. Each time there is a block transition, the block address is pushed to the buffer. When the stage executes, each block in the buffer is inspected (starting with the most recent), until a non-full block is found (or all blocks are full).
next_k	$(0 \leq k < \infty)$ The list is traversed until a non-full block is found, or k nodes are seen.
next_∞	The list is traversed until a non-full block is found, or the list ends.

Figure 4.5: Search types used for MList.

Finally, note that to reduce the number of possible different strategies, these search types do not allow us to interleave searching below and above the reallocatee. Furthermore, in this thesis we will always use parent before child. This decision was based on experiment and can be justified intuitively: moving X to its parent P's block removes a miss whenever X is accessed (say with some probability q), whereas moving X to one of its children's blocks removes a miss with probability between $\sim q/2$ (if the two children are in different blocks and are equally likely to be accessed), and $\sim q$ (if the two children are in the same block – recall that the traversal may terminate at X).

4.3.2 MList

When a node is inserted there is a opportunity to reallocate the node. This is site insert. As with Dict, when a node y is cut out of the structure during the delete operation ($x \to y \to z$ becomes $x \to z$), we will reallocate its child z. This is site delete.

The search methods used for the insert and delete sites are summarised in Fig. 4.5. Note that lastNonFull finds the block closest to the (re)allocatee, whereas firstNonFull finds a block that may be much further away. For long lists, the second strategy will produce a better layout because the block is unlikely to be evicted before the (re)allocatee is reached. However, whereas lastNonFull must inspect the header of every block seen, firstNonFull only inspects until a nonfull block is found and thus incurs lower overhead. As with Dict's parent search strategy, this has different implications for miss rates depending on whether the blocks are lines or pages.

Note also that the last_k method uses a block transition heuristic like Dict's parent_k search.

At first sight, the next_∞ method may seem very inefficient. In the worst case, the rest of the list must be traversed, which would potentially double execution times. The actual overhead of this searching method depends on the distribution of nodes in blocks, and we will later assess its overhead experimentally.

Finally, note that we will perform all 'before' searches (lastNonFull, firstNonFull, last_k) before the 'after' searches (next_k, next_∞). This is done to reduce the number of different strategies. Unlike for DICT, interleaving before and after would actually be sensible: At least in terms of layout, blocks before and blocks after the (re)allocatee are of approximately equal worth because the traversal does not branch.

4.4 Frequently-used coallocators

Here we will assign labels to frequently used coallocators. For example, we use L as a shorthand for lowest$_{CL}$ line, the coallocator that attempts to allocate in the same line as the hint object using the lowest$_{CL}$ operation. The other methods of allocation in the same line (using first$_{CL}$ and cyclic$_{CL}$) are slower, and have identical layout properties, because the location of nodes within lines is irrelevant to the memory hierarchy[2].

N.B. In this section alone, we use regular expression-style notation (..|..|..) *and* [..|..|..] *to combine fragments of labels to express sets of labels. Thus A(M|N)[Z] is shorthand for the set of labels {AM, AN, AMZ, ANZ}.*

The second stage of ccmalloc-newBlock is to use a new block in the hint's page (NLIP – new line in page). We will also consider using the emptiest block in the hint's page (ELIP). The second stage of ccmalloc-firstFit is to use a first-fit allocator to find a line with enough space and then allocate in it (FLIP – first line in page). Since our allocators are based around only one node size, we can actually allocate more efficiently than this, either by using a free list of cells per page, or by using lowest(c,p) or cyclic(c,p). It appears that the last of these is quickest, and we will denote it P.

The second stage of ccmalloc-closest is to use the closest line to the hint with enough space, which is not relevant when all nodes are same size: from the point of view of layout, two nodes are either in the same line, or different line but same page, or in a different page. Thus we will not consider dynamic analogues of ccmalloc-closest, because its behaviour for same-sized nodes has been covered by the other two ccmalloc variants.

The action of ccmalloc when coallocation in the same page fails was not discussed. We will consider a number of third stages. We will use a first-fit cell allocator (F), two line allocators – first-fit (FL), and emptiest line (EL) – and their page equivalents (FP, EP). We do not use newblock line or page ('NL', 'NP') because our aim is to bound memory usage, and these coallocators may fail, which would require a fourth strategy.

We also consider third stages that hierarchically find a page, then a line, then a cell. Finding a cell within a line is always done by lowest(c,l) because it is most efficient and the choice of cell has no effect on layout. This gives nine options – first/newblock/emptiest line in first/newblock/emptiest page ((F|N|E)LI(F|N|E)P). Of these, (F|N|E)LINP, NLIEP and NLIFP, can all fail if there is no empty page. Thus there are four hierarchic coallocators, and five single block ones.

Some of these choices may seem a little inefficient in isolation, for example, there is no apparent value in terms of layout in using FLIFP, but it may be more efficient to allocate using memory manager components that are already enabled instead of enabling extra components. One final point is that each block pair can have only one listset associated with it – for example, we do not allow the memory manager to simultaneously support emptiest(p,h) and first(p,h), and so one cannot combine EP with (E|F)LIFP. Of the 100

[2]This is true using an L1 + L2 + TLB + hardware prefetcher model of the memory hierarchy's behaviour, but probably not in practice.

$\mathbb{SL}$		L	lowest$_{CL}$ line
$\mathbb{SP}$	$\mathbb{SP}_{\bar{L}}$	P	cyclic$_{CP}$ page
		FLIP	lowest$_{CL}$ first$_{LP}$ page
	$\mathbb{SP}_{L}$	ELIP	lowest$_{CL}$ emptiest$_{LP}$ page
		NLIP	lowest$_{CL}$ newblock$_{LP}$ page
$\mathbb{G}$	$\mathbb{G}_{\bar{L}\bar{P}}$	F	first$_{CH}$
		FL	lowest$_{CL}$ first$_{LH}$
		FP	cyclic$_{CP}$ first$_{PH}$
		FLIFP	lowest$_{CL}$ first$_{LP}$ first$_{PH}$
	$\mathbb{G}_{L\bar{P}}$	EL	lowest$_{CL}$ emptiest$_{LH}$
		ELIFP	lowest$_{CL}$ emptiest$_{LP}$ first$_{PH}$
	$\mathbb{G}_{\bar{L}P}$	FLIEP	lowest$_{CL}$ first$_{LP}$ emptiest$_{PH}$
		EP	cyclic$_{CP}$ emptiest$_{PH}$
	$\mathbb{G}_{LP}$	ELIEP	lowest$_{CL}$ emptiest$_{LP}$ emptiest$_{PH}$

Figure 4.6: Frequently-used coallocators and their labels, and sets of coallocators, see Fig. 4.3 for summary of all notation. L=line, P=page, I=in, F=first, N=new (i.e. empty block), E=emptiest (i.e. a block of maximal space).

combinations[3] of the above stages, this condition removes sixteen.

All coallocators and their labels are in Fig. 4.6. We also identify some sets of coallocators. The sets $\mathbb{SL}$ ('Same Line') and $\mathbb{SP}$ ('Same Page') contain all coallocators we use to allocate in the same line or page as the hint respectively. The set $\mathbb{G}$ contains all the global coallocators – ones that don't use a hint address. The set $\mathbb{SP}$ has two subsets: $\mathbb{SP}_L$, which contains coallocators which select a location within the hint's page by first selecting a line in the hint's page based upon the number of nodes in it (beyond the fact that allocation in a full line is not possible). This is either a new (empty) line (NLIP) or the emptiest line (ELIP). $\mathbb{SP}_{\bar{L}}$ contains all other members of $\mathbb{SP}$ (P and FLIP) – notice that FLIP falls into this set because it does not select any particular line, just the first one that is possible to allocate in. Similarly, $\mathbb{G}$ is divided into four sets $\mathbb{G}_{\bar{L}\bar{P}}, \mathbb{G}_{L\bar{P}}, \mathbb{G}_{\bar{L}P}, \mathbb{G}_{LP}$ depending upon whether lines or pages are selected based on the number of nodes in them or not.

4.4.1 Layout properties

In this section we discuss how the different coallocators can be used to improve data layout.

L directly improves line and page layout, because the reallocatee is in the same page and line as the hint.

P, FLIP directly improve page layout, because the reallocatee is in the same page as the hint, but not necessarily in the same line.

[3]$(1+1)*(4+1)*(9+1) = 100$. Recall that each stage may also be null.

ELIP, NLIP directly improves page layout by locating the reallocatee and the hint in the same page, and line layout is indirectly improved, because using the emptiest line available means that coallocation in the same line as the reallocatee is more likely to be possible in the future.

Another way to view these methods, is that L is the most optimistic (we try to improve L1/L2 and TLB performance), P and FLIP are the most pessimistic (we give up on L1/L2 performance, just try to improve TLB performance) and ELIP and NLIP is somewhere in between (we improve TLB performance, but only indirectly attempt to improve L1/L2 performance).

EL indirectly improves line performance, by increasing the probability that a future L will succeed.

EP, FLIEP indirectly improves page performance, and hence also perhaps line performance, since an empty page is more likely to have emptier lines than an arbitrary page.

ELIEP is a compromise, aiming to improve both line and page layout. The page selected will be as empty as selected by EP, but the line will in general be less empty than EL.

F is the quickest non-failing allocator.

Finally, FL, FP, FLIFP, ELIFP have no particular layout benefit, and are included for symmetry's sake.

4.5 Patterns

Refer to the grammar in Fig. 4.3.

It is often useful to decompose a strategy (a tuple of coallocator-search pairs), into separate coallocator and search *'patterns'*. We will do this to concisely describe experiments where we wish to vary the coallocation and the searching independently.

For example, the strategy

$$(\mathsf{lowest_{CL}line} : \mathsf{parent_1},\quad \mathsf{lowest_{CL}line} : \mathsf{child_1},\quad \mathsf{lowest_{CL}emptiest_{LH}} : \mathsf{true})$$

may be written as a composition of a coallocator pattern and a search pattern using the $\times$ operation as follows

$$\langle \mathsf{lowest_{CL}line};\ \ \mathsf{lowest_{CL}emptiest_{LH}}\rangle \quad\times\quad \langle \mathsf{parent_1},\ \mathsf{child_1};\ \ \mathsf{true}\rangle.$$

Observe that the use of patterns allows a particular coallocator (e.g. $\mathsf{lowest_{CL}line}$) to be repeated for several searches, or alternately a particular search to repeated for several coallocators.

Formally, the $\times$ operation takes a coallocator pattern and a search pattern and yields a strategy. Patterns have some degree ≥ 1. A coallocator or search pattern of degree d is of the form $\langle \mathsf{a}^1_1, \ldots, \mathsf{a}^1_{\mathsf{n}_1}; \ldots; \mathsf{a}^d_1, \ldots, \mathsf{a}^d_{\mathsf{n}_d}\rangle$ – d lists where list i is of degree n_i. The $\times$ operator acts on degree-1 patterns as follows (to produce a strategy):

$$\langle \mathsf{c}_1, \ldots, \mathsf{c}_n\rangle \times \langle \mathsf{s}_1, \ldots, \mathsf{s}_m\rangle \;\equiv\; (\mathsf{c}_1 : \mathsf{s}_1, \ldots, \mathsf{c}_1 : \mathsf{s}_m, \ldots, \mathsf{c}_n : \mathsf{s}_1, \ldots, \mathsf{c}_n : \mathsf{s}_m)$$

where $\langle \mathsf{c}_1, \ldots, \mathsf{c}_n\rangle$ is a coallocator pattern and $\langle \mathsf{s}_1, \ldots, \mathsf{s}_m\rangle$ is search pattern.

ccmalloc-newBlock-vanilla	$\ll \{\mathsf{L}\} \; ; \; \mathbb{SP}_L \; ; \; \mathbb{G}_{\bar{L}\bar{P}} \gg$
ccmalloc-newBlock-extended	$\ll \{\mathsf{L}\} \; ; \; \mathbb{SP}_L \; ; \; \mathbb{G} \gg$
ccmalloc-firstFit-vanilla	$\ll \{\mathsf{L}\} \; ; \; \mathbb{SP}_{\bar{L}} \; ; \; \mathbb{G}_{\bar{L}\bar{P}} \gg$
ccmalloc-firstFit-extended	$\ll \{\mathsf{L}\} \; ; \; \mathbb{SP}_{\bar{L}} \; ; \; \mathbb{G} \gg$

Figure 4.7: Definition of the four sets of coallocator patterns that are dynamic forms of ccmalloc. Note the division into those that use a final coallocator that selects blocks based upon the number of nodes in them ('extended'), and those that do not, apart from not using a full block ('vanilla').

The $\times$ operator acts on degree-d patterns as follows:

$$\langle \mathsf{c}^1_1, \ldots, \mathsf{c}^1_{\mathsf{n}_1}; \ldots; \mathsf{c}^\mathsf{d}_1, \ldots, \mathsf{c}^\mathsf{d}_{\mathsf{n}_\mathsf{d}} \rangle \times \langle \mathsf{s}^1_1, \ldots, \mathsf{s}^1_{\mathsf{m}_1}; \ldots; \mathsf{s}^\mathsf{d}_1, \ldots, \mathsf{s}^\mathsf{d}_{\mathsf{m}_\mathsf{d}} \rangle$$

$$\equiv \langle \mathsf{c}^1_1, \ldots, \mathsf{c}^1_{\mathsf{n}_1} \rangle \times \langle \mathsf{s}^1_1, \ldots, \mathsf{s}^1_{\mathsf{m}_1} \rangle @ \ldots @ \langle \mathsf{c}^\mathsf{d}_1, \ldots, \mathsf{c}^\mathsf{d}_{\mathsf{n}_\mathsf{d}} \rangle \times \langle \mathsf{s}^\mathsf{d}_1, \ldots, \mathsf{s}^\mathsf{d}_{\mathsf{m}_\mathsf{d}} \rangle$$

where the c^i_j are coallocs and the s^i_j are searches, and @ is strategy concatenation (i.e. $(x_1, \ldots, x_q)@(y_1, \ldots, y_r) \equiv (x_1, \ldots, x_q, y_1, \ldots, y_r)$), and $\times$ binds more tightly than @.

In other words, each coallocator or search pattern is a ;-separated list of ,-separated sublists. When combined with the $\times$ operator, the i^{th} sublist from the two patterns are combined, and then the results are concatenated.

4.6 Dynamic forms of ccmalloc

We now define properly how ccmalloc is extended for dynamic use. In more detail, we define four sets of coallocator patterns, ccmalloc-newBlock-vanilla, ccmalloc-firstFit-vanilla, ccmalloc-newBlock-extended and ccmalloc-firstFit-extended, which can be found in Fig. 4.7, with the definition of the $\ll . \gg$ operation as follows: (where A^i_j are sets of coallocators, and the result is a set of coallocator patterns)

$$\begin{aligned}&\ll A^1_1, \ldots, A^1_n; \ldots; A^m_1, \ldots, A^m_n \gg \\ &\equiv \{\langle a^1_1, \ldots, a^1_n; \ldots; a^m_1, \ldots, a^m_n \rangle : \forall a^1_1 \in A^1_1, \ldots, \forall a^m_n \in A^m_n\}\end{aligned}$$

Members of the vanilla sets use only final coallocators whose aim is to allocate quickly irrespective of the effect on layout ($\mathbb{G}_{\bar{L}\bar{P}}$), not considering the number of nodes when selecting a block (apart from not using full blocks), which is more in the spirit of the static form of ccmalloc. The extended sets include all of $\mathbb{G}$.

4.7 Summary

In this chapter we reviewed the use of coallocators to improve the initial layout of an RDS. We constructed a larger family of coallocators than existed previously in the literature, including dynamic forms of ccmalloc, and explain the effect they may have on data layout. We described 'reallocation', the use of coallocators to maintain RDS data layout, and discussed how to apply it to the DICT and MLIST benchmarks. We will evaluate reallocation in Chapter 7.

Chapter 5

Bulk data movement

5.1 Introduction

The aim of reallocation is to prevent layout degradation by performing individual node movements when a pointer update occurs. The ability of these movements to prevent layout degradation depends on the availability of space in the hint's block, and thus the quality of the layout produced is not easy to predict. In this section we consider a different form of data movement, known as *'bulk data movement'*. A typical bulk data movement optimisation makes infrequent movement of large numbers of nodes, not necessarily when pointer updates occur, repairing the layout of part of the RDS, typically by moving all the nodes to a new part of the heap. Like reallocation, the quality of data layout produced by this form of movement is sometimes not easy to predict, but in practice more stable behaviour is often observed than with reallocation.

In this chapter we present bulk data movement optimisations that move data in one of three ways. Firstly, an infrequent stop-the-world movement of the entire RDS may be used. Secondly, more frequent smaller movements may be used, to reduce latency. Thirdly, data movement code may be embedded into existing traversals, to reduce overhead and latency. In all three methods, the applier has only to ensure that the data movement occurs frequently enough to maintain a good layout. This is in contrast to reallocation, where better performance is likely to be obtained by (at least) investigating the insertion of data movement code into all pointer update sites. In particular, we expect reallocation to be far more sensitive to the precise form of the pointer updates occurring in the RDS, since it must reverse the layout degradation caused by each update, whereas a bulk data movement optimisation needs only to know a good layout for (parts of) the RDS. Thus bulk data movement and reallocation should be thought of as two complementary approaches, either of which may be more suitable depending on whether a larger number of simple changes to a program is preferable to a small number of more complex changes.

We now discuss a very simple bulk data movement optimisation.

5.1.1 Simple bulk data movement

The simplest method of preventing serious layout degradation of an RDS is to occasionally move the entire structure into a new block of memory (*space*), using some good layout that the applier has chosen before runtime. Even this simple method has several subtleties, which we discuss now.

The shape of the RDS changes at runtime and so the role of the applier is not to provide a layout consisting of rules of the form 'place the third node along in the fourth level of the tree in some location' (a map from a node's position in the tree to location in memory). Instead, they must provide an algorithm that takes as input the structure, and outputs a mapping of nodes to their new location in memory. Concretely, their role is most likely to be to provide a function that walks the structure visiting each node (possibly multiple times, but most efficiently just once), moving nodes as it goes, instead of explicitly computing a map of the whole structure.

The nodes of the structure are being moved into a new, empty space rather than being moved around within the current space, and so the new layout of the structure depends only on its shape, and not on its old layout. This simplifies the optimisation, and allows full control over the layout quality, but doubles memory usage. Later in this chapter we will present optimisations using less memory where the previous layout does affect the new layout.

We observe that a *good* layout may yield a more effective bulk data movement optimisation than an *optimal* layout. Our experience with static layouts for Dict and MList from Chapter 3 suggests that often there will exist a layout that performs almost as well as the optimal but is far easier to produce. For dynamic use, the best balance between layout quality and overhead must be found. The role of the applier may therefore be to explore several different layouts

We now address the issue of when the function should be invoked. In general, layout degradation occurs gradually as a program runs, because a single pointer can only have a small impact on the layout of a structure. Invoking the function the programmer has provided will dramatically improve the layout of the structure, and incur a large overhead. Thus layout quality follows a sawtooth pattern (not necessarily with equally-sized teeth), and execution time is a sawtooth except with a large pause when the function is invoked (see Fig. 5.1 – note that the vertical axis is logarithmic). Thus, if the function is invoked too frequently, the average execution time for a large number of operations is too high because of data movement costs, and if it is invoked too infrequently, too much layout degradation occurs, also increasing execution time.

For a constant rate of layout degradation, as exists in the benchmarks of this thesis, execution time is minimised by invoking the function periodically with some static period N. The optimal period depends on the modification rate of the RDS, overhead and many other factors, and so we will find it by experiment (tuning) rather than theoretically.

5.1.2 Structure of this chapter

The rest of this chapter is as follows. In §5.2, we discuss the implementation of the simple stop-the-world two-space optimisation for both benchmarks. In the remaining sections, we discuss several techniques to reduce memory usage, latency and overhead:

1. **Approximation:** We present a method to reduce memory usage, at the cost of layout quality. (§5.3)

2. **Incrementalisation:** We demonstrate that techniques previously used to make incremental garbage collectors can be relatively easily applied to reduce the latency of a

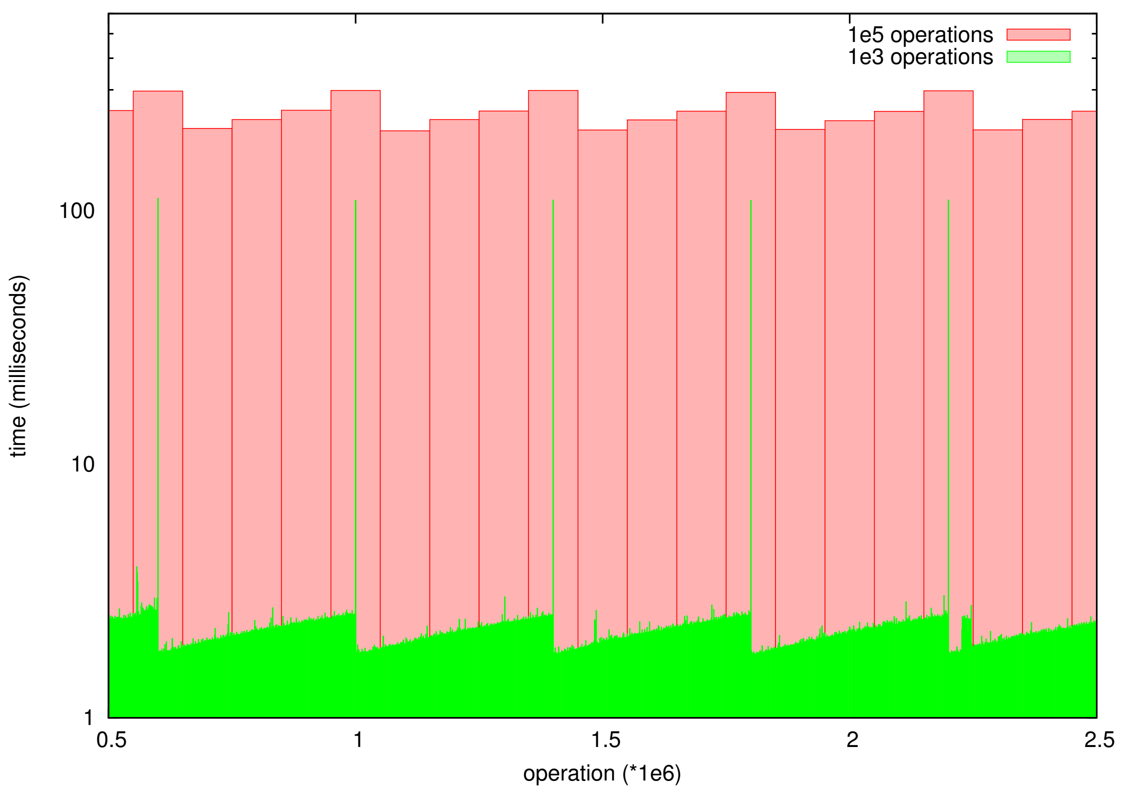


Figure 5.1: Histogram showing the behaviour of the periodic2space optimisation. Observe that data movement pauses are very visible at the $1e3$-operation scale (approx. $1ms$), but not at the $1e5$ operation scale (approx. $100ms$).

bulk data movement optimisation. (§5.4)

3. **Embedding:** We demonstrate how inserting optimisation code into existing traversal loops of a program can be used to reduce latency and may in principle reduce the overhead. (§5.5)

4. **Compaction:** We describe a lightweight method of performing additional data movement to create emptier lines or pages, which are later used to directly improve the layout of part of the structure. (§5.6)

5.2 Using two semi-spaces

The implementation of the simple stop-the-world two-space bulk data movement optimisation is as follows. Twice as much memory is allocated (in other words, $m = 2$). Every N operations, the entire RDS is traversed, and all nodes moved into 'to-space'. The 'from-space' is now empty because the applier has guaranteed that all nodes allocated in from-space are in the RDS (every node is reachable from the roots of the structure). Movement of nodes is possible because the programmer has provided some method for updating all parent pointers, as discussed in §2.3.2.1.

For DICT, the optimisation performs Breadth-First Search (BFS) to collect a line's worth of nodes (a 'cluster'). These are moved ('clustered') into an empty line. The procedure is then repeated for all children of the cluster. The BFS may fail to find enough nodes to fill a line (i.e. at the bottom of the tree), but the nodes must still be moved to to-space, and the exact details of how this is done don't seem to effect layout quality very much. This is layout **goodLine** from §3.4.2.1. We will denote this optimisation periodic2space.

We also investigate a variant of this optimisation, periodic2space-nested, that minimises TLB misses as well as L1/L2 misses. Lines are filled using BFS as before, but then pages are filled by performing breadth-first search over lines (this is layout **goodBoth** from §3.4.2.1). We found that this optimisation performs worse than periodic2space in practice, for two reasons: firstly, periodic2space-nested incurs greater overhead, and secondly because periodic2space produces quite good TLB performance simply because of the order in which the tree is traversed[1]. If a simple stop-the-world optimisation cannot obtain improved performance from lower TLB misses, it is unlikely that the more complex optimisations in the rest of the chapter can (and the implementation would be far more difficult), and thus for the rest of this chapter we shall focus solely on lines when optimising DICT.

As we discussed above, the periodic optimisations pause normal program work for a significant time during the traversal. This may be unacceptable for some applications. See for example the performance of periodic2space in Fig. 5.1 (note the vertical axis is logarithmic). Groups of $1e3$ operations may experience a very significant latency, but groups of $1e5$ do not.

For MLIST, the periodic2space optimisation traverses each list in turn, moving nodes to a pointer which increments through to-space – list are arranged end-to-end in memory, without any gaps. This has the advantage that to-space is denser, and we need not keep track of which cells are in use in to-space, simply remember the value of the pointer. The disadvantage is that a list may incur one more L1/L2 or TLB miss than necessary, because no attempt is made to align lists to line or page boundaries – however, since sixteen nodes fit into one line, this is unlikely to have a significant effect on execution times.

One significant advantage that the periodic2space optimisation has for both benchmarks over the more complicated optimisations we will discuss later in the chapter is that there is very little allocator overhead. During the traversal, the memory manger does not need to keep track of which cells are occupied in from-space or to-space, because normal program work is halted. Once the traversal is finished, all memory manager data structures must be rebuilt, but this is a simple task because to-space is divided into an entirely full contiguous region and an entirely empty contiguous region.

5.3 Using one space

In general, the optimisation of the previous section (periodic2space) requires two contiguous semi-spaces large enough to hold the RDS, giving a memory usage of $m = 2$. In this section, we discuss how we can produce a 'one-space' optimisation (periodic1space) that

[1]**goodLine** achieves ~ 6 pages per node lookup, compared to the **goodBoth**'s ~ 3, where a lookup visits ~ 25 nodes.

periodic1space

```
traverse(n):
   let k = space in emptiest line
     let S = BFS(n,k)
     if (|S| == k)
        let L = emptiest(line,heap)
        move nodes in S into line L
     else
        handleSmallSubTree(S)
     for all children m of set S:
        traverse(m)
```

The lookup macro

```
lookupStep(n):
   if (!n) return 0;
   if (n->key == searchKey) return n;
   if (searchKey < n->key ) n=n->c[0];
   else n=n->c[1];
```

embedded1space+throttle=p%

```
Lookup(n):
   with probability 1-p:
      while (1)
         n=lookupStep(n)
   else
      while (1)
         let k = space in emptiest line
         if (k<=1)
            n=lookupStep(n)
         else
            let S = BFS(n,k)
            if (not all of S in same line)
               if (|S| == k)
                  let L = emptiest(line,heap)
                  move nodes in S into line L
               else
                  handleSmallSubTree(S)
            while (n ∈ S)
               n=lookupStep(n)
```

embedded1space+throttle=p%+compact +thresh=K

```
Lookup(n):
   with probability 1-p:
      while (1)
         n=lookupStep(n)
   else
      while (1)
         let k = space in emptiest line
          if (k<K)
            goto fail

         let S = BFS(n,k)
         if (not all of S in same line)
            if (|S| == k)
               let L = emptiest(line,heap)
               move nodes in S into line L
            else
               handleSmallSubTree(S)
         while (n ∈ S)
            n=lookupStep(n)
   fail:
      while (1)
          move n to fullest(line,heap)
         n=lookupStep(n)
```

Figure 5.2: DICT: Pseudo-code for periodic1space, embedded1space+throttle=p% and embedded1space+throttle=p%+compact. The BFS(n,k) function returns the set of k nodes found by Breadth-First Search from node n. A node n is a child of set S iff n is not in S and the parent of n is in S. The body of the lookup loop is a macro lookupStep. The emptiest(line,heap) function returns the address of a line of maximal space in the heap, the fullest(line,heap) function returns the address of a line of minimal nonzero space in the heap. The handleSmallSubTree function deals with the re-laying out of the tree near the leaves (where less than k nodes are found by BFS).

uses a single space, allowing memory usage of $2 > m > 1$.

We start by observing that during the 'collection' (the movement of nodes from from- to to-space), large regions of empty space will arise in from-space. Our aim is to recycle these regions to form part of to-space. A sensible way to do this is to divide the two spaces into pages, and reuse any empty pages that arise in from-space. Note that this will produce a layout with the same L1/L2/TLB miss rates, and therefore this layout will be as efficient under a reasonable model of performance (recall that the hardware prefetcher does not fetch across page boundaries). To-space and from-space no longer exist as distinct 'physical' parts of the heap – the heap is now one 'physical' space split into two 'logical' parts during the execution of the collection. A further way of viewing this is to consider to-space and from-space as sets of pages, where the location of each page is irrelevant. In the former, each page is obtained from the system only when the optimisation attempts to move a node into it (a.k.a. 'lazy allocation' or 'allocate-on-use'). In the latter, the optimisation returns each page to the system as soon as all its nodes are moved out of it.

The number of pages in use (in to-space and from-space) give the memory requirements of the optimisation during collection, which varies as it progresses, starting and ending at one. Denoting the peak memory requirement m_{peak}, our aim is to arrange the order of node movement to minimise m_{peak}, which in general could be higher than 2 (i.e. worse than periodic2space).

A sensible way to reduce m_{peak} is to fill to-space one page at a time – by this we mean that all the k nodes that belong in a particular page in to-space are put there in k consecutive nodes movements. This procedure minimises the number of pages in use in to-space for a given amount of progress of the collection, where progress is expressed as the number of nodes in to-space. Alternately, for a given number of pages in use in to-space, the progress of the collection has been maximised. In the absence of any knowledge about the layout in from-space this is probably the best way to minimise m_{peak}.

Transforming MList's periodic2space optimisation to use this form of from-space/to-space is trivial because list are laid out in order in to-space, filling each page in to-space before allocation in the next. Dict's periodic2space-nested also deals in complete pages where a subtree has enough nodes. If a subtree doesn't have enough nodes, its nodes are moved into a specially denoted page used to store smaller subtrees. Thus at most two pages are being filled at once (one for subtrees with a page worth of nodes, and another for smaller subtrees), almost satisfying the condition stated above.

For Dict's periodic2space, which fills lines with nodes before moving onto the next, we have a choice. One method is to treat each group of *pageSize/lineSize* lines as one page, and then deal with incomplete subtrees as above. However, because page layout isn't very important for Dict (as shown in §3.4.2.1), it is better to treat from- and to-space as sets of *lines* instead. Lines are smaller than pages, and so in general they will arise in from-space sooner, which will reduce m_{peak} without significantly reducing layout quality, and so we will adopt this method. Likewise, the notion of using a smaller block size for to- and from-space can be applied to MList's optimisation to reduce m_{peak}, but this time with a more serious impact on layout quality, because MList is much more sensitive to page layout, as shown in §3.4.2.2.

In practice, an m_{peak} of around 1.35 is seen for Dict when periodic2space is converted to use one space, considerably lower than periodic2space's memory requirements. For

MList, m_{peak} depends on v, but is much less than 2. Note that greater overhead is incurred than with two spaces, because the memory manager must keep track of which cells are in use during the movement from from-space to to-space so it can maintain a free list of empty blocks.

Predicting or controlling m_{peak} is not simple, and so we now discuss how memory can be properly bounded to any value $m \in (1, 2]$. This is achieved by first claiming the space (a set of blocks) a factor of m larger than the space the RDS requires, and then allocating the RDS within them (as with reallocation). The body of the optimisations are changed so that instead of moving nodes into an empty block, nodes are moved into a block of maximal space in the space (hereafter referred to as the 'emptiest' block in the space). This is a trivial change for both benchmarks. As the memory allowance is lowered, the emptiest block will have less and less space in it, reducing the quality of layout produced. Thus memory is directly traded for performance. The memory manager from the previous chapter is reused to support finding the emptiest block in the heap (using the $\texttt{listset}_{*H}$ memory manager component). We denote this optimisation periodic1space, and pseudocode for Dict can be found in Fig. 5.2.

When investigating this optimisation for MList we may adjust block size, investigating intermediate sizes between line size and page size. We do this because the block size has the potential to affect the quality of the emptiest block in the heap[2]. For Dict, we will use blocks only the same size as the line size, because there is very little layout benefit to using larger blocks.

5.4 Incrementalisation

In this section, we discuss how the latency caused by the periodic2space and periodic1space optimisations can be reduced in a similar way to the incrementalisation of a garbage collector [39, 73].

5.4.1 Incrementalising periodic2space

Notice that periodic2space is very like a stop-the-world two-space copying collector. Periodically, it is invoked, and it traverses the heap, using a 'root set' as a starting point. For periodic2space we simply use the root node(s) of the RDS – the tree root for Dict and the head of each list for MList. During the traversal, nodes are moved into to-space, at which point from-space is entirely empty.

For Dict, periodic2space is incrementalised as follows. We maintain a queue of roots, whose descendants in the tree are still to be processed. The optimisation performs a constant amount of work W every K traversals. A node access in either from-space or to-space counts as one unit of work. Firstly K is set low enough so that pause time will be around $1ms$. Then, the work/operation ratio W/K is adjusted to its optimum value. Note that for a constant W/K (i.e. data movement rate), using a larger K will reduce overhead (because the optimisation doesn't have to stop and restart so often), but will cause greater pause time.

[2]If the block size is the heap size and the heap is of density d, then only a block of density d is available. If the block size is many times smaller, blocks of lower density are likely to exist given a favourable distribution of nodes.

When the optimisation is invoked, it pops the root queue, which initially just contains the root of the tree. It then gathers up to a line's worth of nodes, moves them to an empty line in to-space, and pushes the children of the cluster to the root queue. This process continues until the work allowed for this invocation has been performed. When to-space is empty, the spaces are flipped and the tree root is pushed to the root queue.

Like a two-space incremental GC, we have to ensure no nodes are missed if the 'mutator' (node insertion and deletion) changes the structure of the tree. Specifically, we use a write barrier. Any attempts to write a pointer to from-space into a node in to-space are captured. When this occur, the pointer is added to the root queue. Thus when a pointer assignment `x->c[i] = y` occurs, and x is in to-space and y is in from-space, the value `&(x->c[i])` is added to the root queue. When this pointer is loaded, its target and eventually all its descendants will be moved into to-space if they are not already there.

The value of the pointer may have been changed between its insertion in the queue and its use by the optimisation. For example, `x->c[i]` might now point to a different node z. In this case, there will be two copies of `&(x->c[i])` in the root queue. The approach taken to this problem is to ignore it – the second copy of `x->c[i]` will cause an additional unnecessary traversal of `x->c[i]`'s subtree. Other additional traversals may arise if a pointer in any descendant of `x->c[i]` is updated. Because of the distribution of pointer updates in DICT (uniformly over nodes), this conservatism does not actually cause a significant amount of extra work. Note however that given a pathological sequence of pointer updates (e.g. repeatedly reassigning a pointer of the root node) the collection would never terminate, and so the optimisation given here should be considered as demonstrations of the overheads involved in incrementalisation rather than being satisfactory for general use (termination could be guaranteed by redefining a unit of work as a node movement from from- to to-space, but pause time would now be significantly larger, negating the whole purpose of incrementalising the data movement).

Situations affecting correctness may also arise: if `&(x->c[i])` is fetched from the root queue and `x` no longer exists because it has been freed or already moved into to-space. The latter of these two can occur if x is a descendant of α, and a pointer of α is updated after a pointer of x is updated. The optimisation may pop α's root queue entry, moving α's subtree, which includes x, rendering the `&(x->c[i])` entry in the root queue invalid. The optimisation must therefore ignore invalid pointers, and so the memory manager must keep track of which cells are in use in from-space. Note that this additional overhead was not necessary for periodic2space.

Converting periodic2space into incremental2space is much simpler for MLIST because the data structure decomposes into many disjoint substructures. The time to re-layout one list is sufficiently low that we can simply halt normal program work, re-layout a list, then resume normal program work, without pauses larger than around $1ms$. We do not need to store a queue of roots, as with DICT, simply remember the last list that has been moved to to-space. The write barrier is also simpler. A node will only be missed by the collector if it is moved to a list that has already been processed. The write barrier fixes this by moving into to-space any node that the mutator tries to insert into an already-processed list. Note that the write-barrier is much simple for MLIST – indeed the collector does not need to be aware of pointer updates – precisely because the structure decomposes into small distinct substructures.

5.4.2 Incrementalising periodic1space

Here we discuss how the periodic1space optimisations for the two benchmarks may be incrementalised to produce the incremental1space optimisations.

As discussed above, incremental2space is really a specialised incremental two-space GC. It must visit all nodes before flipping spaces to preserve program correctness. For Dict this means that when the mutator updates a pointer, an entry is pushed into the root queue. This is a conservative strategy, and will result in extra work being performed by the collector, but ensures that all nodes are in to-space when the flip occurs.

For periodic1space, however, only one space is used, and so we have the option of completely removing the write barrier. When the mutator updates a pointer, there is a possibility that parts of the RDS will simply not be walked by the collector during its current traversal. This does not matter, because given a non-pathological sequence of pointer updates, the collector will walk those nodes eventually. The lack of a write barrier both reduces overhead and simplifies the application of the optimisation, at the possible expense of performance but not correctness.

Incrementalising periodic1space is also simple for MList – we simply interleave the re-laying out of lists with normal program work, and no write barrier is required. As with Dict, nodes are unlikely to always miss being re-laid out unless the pointer update sequence is pathological.

5.5 Embedding

The optimisations described so far (periodic2space, periodic1space, incremental2space and incremental1space) walk the data structure disjoint from normal program work. In this section we demonstrate how the data access overhead can potentially be reduced by embedding optimisation code into existing program loops.

An *embedded* optimisation moves parts of the structure close to those visited by normal program work, instead of walking the structure separately. Our intuition is that the overhead of a separate traversal of the structure is due mostly to the cost of data misses, not instruction costs, and so by hijacking existing node fetches it may be possible to improve the efficiency of an optimisation.

There are further advantages to embedding: data movement work is finely interleaved with normal program work, reducing latency. Our previous method of reducing latency – incrementalisation – required a write barrier, which carried an additional overhead. Although simpler than incrementalisation, in our experience, embedding code is still difficult because any stack references to nodes must be updated after data movement.

The main disadvantage of an embedded optimisation is that the optimisation has lost control of the order in which nodes are visited, indeed there is no guarantees that the optimisation will regularly visit – or ever visit – every node in the structure, which may make particular types of data movement difficult. In particular, we will explain that embedding periodic2space is possible but particularly ineffective. For other optimisations, this behaviour may actually be an advantage: if only a subset of the RDS nodes are being accessed, data movement will only be performed on the active subset, rather than on the whole RDS as would happen with a non-embedded optimisation.

A further potential disadvantage occurs for branching traversals. For Dict, without particular attention nodes near the top of the tree will be re-laid out exponentially more often than nodes near the bottom. In practice, this may prove to be an advantage, because they are visited exponentially more often than leaf nodes.

5.5.1 Embedding periodic2space

It is possible to embed periodic2space, but it is not very sensible. During DICT's lookup loop and MLIST's insert and delete loops, each node is inspected to see which space it is in, and if it is in from-space, a block's worth of nodes is found (by BFS for DICT or linearly for MLIST) and moved to to-space. Once from-space is empty, the spaces are flipped. Unfortunately, the flip occurs far too infrequently – for the DICT benchmark almost an order of magnitude less often than the optimal frequency for periodic2space. A real program would have less favourable access patterns, which makes embedding a two-space optimisation like periodic2space particularly unhelpful.

5.5.2 Embedding periodic1space

For DICT, the periodic1space optimisation is embedded in the lookup loop to produce embedded1space as follows (code is in Fig. 5.2). The body of the lookup loop is similar to the body of periodic1space. Firstly, the code inspects the emptiest line in the heap, finding a cluster of nodes to fill the line by using BFS from the current node. After the nodes are moved, periodic1space and embedded1space behave differently. The former recurses over all children of the cluster, whereas the latter performs lookup steps (inspecting the key and branching left or right or returning), until it exits the cluster, and the procedure is then repeated until the key is found or the bottom of the tree is reached. Thus, during a lookup, nodes close to the lookup path are moved. To make this optimisation practical, data movement may have to be throttled: two versions of the lookup loop are used, a normal and an optimised, with the optimisation version being used with some probability $p\%$, indicated as throttle=$p\%$ (e.g. periodic1space+throttle=10%).

For MLIST, code is embedded in both insert and delete traversals. The implementation is simple because the insert/delete traversals and the data movement traversal are identical – linear. The only difference is that the insert/delete traversals terminate before the end of the lists, whereas 1space's traversal does not. We evaluate three different options: first, the vanilla variety of embedded1space stops when the original traversal does. This produces a partially filled block. The second variety, embedded1space+toEndOfBlock, continues until the current block is filled. The third variety, embedded1space+toEndOfList, continues until the end of the list. As with DICT, we throttle embedded data movement using some probability $p\%$, indicated as throttle=$p\%$.

5.6 Imposing a minimum quality on new layout

The layout quality produce by the entire 1space family of optimisations depends only on the emptiness of the blocks found in the heap. A simple way to improve layout is therefore to reject any blocks below a certain threshold emptiness, indicated by +thresh=k in the optimisation name. In other words, the optimisation aborts part of its work, and does not start again until emptier blocks arise. DICT's embedded1space+thresh aborts data movement until the end of the lookup. MList's embedded1space+thresh similarly aborts until the end of the insert or delete traversal.

There are three ways in which a sufficiently empty block can be produced (or, even better, prevent the situation where there are no empty enough blocks). All three involve data movement which moves nodes with the aim of produce emptier blocks, rather than directly improving layout. We refer to this process as 'compaction'. The first form of compaction ('allocation site compaction') – making the correct choice of allocator at any allocation sites in the program – is a natural side-effect of applying the optimisation to the program, but is not always applicable. The second method is 'reallocation site compaction' – the addition of data movement code to reallocation sites. The third method is to perform data movement within the body of the optimisation ('optimisation compaction').

1. Allocation site compaction: When any of the optimisations in this thesis are applied, a custom memory manager is used (indeed, a memory manager produced in same way as in the previous chapter), and so any allocation sites must be changed to calls to the memory manager. This raises the question of which coallocator to use at allocation site. Although an allocator such as first_{CH} has the lowest overhead for general purposes, there seems little point in making the memory manager maintain another component when it is already maintaining $\texttt{bitfield}_{CB}$ and $\texttt{listset}_{BH}$ (to keep track of which cells are allocated, and to locate the emptiest line, respectively, where B is either L or P). Thus, a sensible choice of allocator would make use of these components. Allocation in the emptiest line will reduce the quality of lines available for the optimisation, and so allocation in the fullest line is the natural choice.

Recall that in our benchmarks the number of nodes stays constant, and thus an allocation site must have an associated `free` somewhere (and this is true for any sufficiently long-running program that experiences continual RDS layout degradation, because memory is finite). The node that has been freed doesn't come from any particular line, so we may assume the line has an average amount of empty space in it. The new allocation is in a line of minimal non-zero space, on average a more full line than the line the freeing is taking place in. And so the overall effect of the alloc-free pair is that the emptier line of the pair becomes emptier, the fuller line becomes fuller. Thus the optimisation will eventually have access to better lines. In our benchmarks, there are no allocation sites during the timed part (because the nodes from the delete operations are reused in the insert operations), but in general allocation compaction should be used at all allocation sites. In other words, transforming the program to use a sensible allocator at allocation sites (which may plausibly be done automatically) will achieve a compaction effect.

2. Reallocation site compaction: Similarly, we may explicitly move nodes to the fullest line at some selected reallocation sites, increasing the rate of compaction. We indicate this by +*site*=compact in the optimisation name, where *site* is the name of the site as given in §4.3. Despite the similarity of implementation, it is important to note the difference in meaning between compaction performed at allocation sites and at reallocation sites: in the first, we are simply *allocating* in the most sensible way possible (incurring no extra overhead, and not requiring the updating of any parent pointers), but in the second we are explicitly performing *data movement*, whose aim is not to improve layout (directly), but to produce emptier blocks which can later be used to improve layout. This incurs additional overhead and requires some mechanism to update parent pointers.

3. Optimisation compaction: Compaction is applied within the optimisation using a similar method to the above, which we indicate by appending compact to the optimisation's name. The code for DICT for the embedded1space+throttle=p%+compact+thresh=k is in Fig. 5.2. Observe how when no block above the threshold is found, the remainder of the lookup operation performs compaction. It is important to investigate the possibility of performing this compaction work within the optimisation's code, because it may mean the applier does not have to worry about locating and applying compaction at reallocation sites.

5.7 Summary

In this chapter we first described the simple stop-the-world two-space bulk data movement optimisation periodic2space. We then discussed several techniques to reduce memory usage, latency and overhead. We used an approximate data layout to reduce memory usage (periodic1space). We will improve the data layout created by this approximation by using compaction – additional data movement whose aim is to create emptier blocks, not to directly improve data layout. We then tackled latency, either by incrementalising the optimisation using techniques similar to those used for garbage collectors (incremental1space, incremental2space), or by embedding data movement code within existing program loops, which also may reduce overhead (embedded1space).

A summary of all optimisations including all optional parts can be found in Fig. 5.3.

Techniques:

periodic2space	Using two semi-spaces (§5.2)
periodic1space	Using one space (§5.3)
incremental	Incrementalisation (§5.4)
embedded	Embedding (§5.5)
throttle=p%	Throttling (§5.5)
thresh=k	Threshold (§5.6)
compact	Optimisation compaction (§5.6)
reallocSite=compact	Reallocation site compaction (§5.6)

All DICT optimisations:

periodic2space	
periodic2space-nested	
incremental2space	
periodic2space	
incremental2space	[+compact][+thresh=k][+throttle=p%][+insert=compact] [+deleteCut=compact] [+deleteMove=compact]
embedded1space	[+compact][+thresh=k][+throttle=p%][+insert=compact] [+deleteCut=compact] [+deleteMove=compact]

All MLIST optimisations:

periodic2space	
incremental2space	
periodic1space	
incremental1space	[+compact] [+thresh=k] [+throttle=p%] [+insert=compact] [+delete=compact]
embedded1space	[+toEndOfList |+toEndOfBlock] [+compact] [+thresh=k] [+throttle=p%] [+insert=compact] [+delete=compact]

Figure 5.3: Summary of bulk data movement optimisations. [] indicates optional parts of the optimisation.

Chapter 6

Perfect data movement

6.1 Introduction

In the previous two chapters we have presented two different types of optimisation. Both are relatively simple to understand and implement, but produce a data layout whose quality is hard to predict. Reallocation depends on there being space in the hint's block (line or page), and on the quality of the emptiest block in the heap. The $m < 2$ forms of bulk data movement similarly depend on the quality of the emptiest block in the heap to produce a good layout. Furthermore, both methods have to be tuned – reallocation by choosing the correct coallocation and search patterns, and the sites to use, and bulk by adjusting movement rates and thresholds.

In this chapter we investigate methods to restore a predefined data layout after every pointer update, which we will call *'perfect data movement'*. The data movement is 'perfect' in the sense that the layout is always restored, not that this is done with the minimum amount of effort, either per-update or amortised – re-laying out the entire RDS after every update is an example of a (very inefficient) perfect data movement optimisation.

The 'layout' that is perfectly restored can take a number of forms. It could be a specific *'map'* from nodes to memory locations (e.g. for MList, list are laid out end-to-end in order in memory), or express a family of maps, for example in terms of node groupings (e.g. lists are clustered into pages but the pages can be anywhere in memory). In this chapter we will sometimes use even less precisely defined layouts, for example using variable-sized groupings. For these less precisely defined layouts, the code that restores them must first implicitly or explicitly choose a particular map that satisfies the layout, and then carry out the data movement.

Applying perfect data movement involves considerably more changes to a program than either bulk data movement or reallocation. Read operations on the RDS stay the same, but all update operations must be changed substantially to restore data layout. The data movement code is far more tightly coupled to the program than in the previous two chapters. We should therefore view perfect data movement simultaneously in two ways: Firstly, as an optimisation applied to an existing simple RDS (as reallocation and bulk), and secondly, as a way of *synthesising* a cache-aware RDS from a cache-unaware one.

Viewed as an optimisation, we hope perfect data movement will produce improvements in execution times competitive with reallocation and bulk. By *relaxing* the layout, we can reduce the overhead of maintaining it, and therefore hopefully find a good balance. As well as any inherent value as an optimisation, perfect data movement gives us further insight about data movement at pointer update sites, occupying the opposite end of the spectrum to reallocation. The latter is opportunistic, with no guarantee over layout quality, but is simple to implement and has low overhead. Perfect data movement does guarantee a particular layout, but has higher overhead and is significantly harder to implement. Between these two approaches there is a large middle ground – more precise than reallocation, but easier to implement than perfect – into which the behaviour of perfect data movement may provide valuable insight.

Secondly, we can view perfect data movement as a method of producing cache-aware RDSs. These RDSs are unusual because they can be read in the same way as the original RDS, but updates must be performed in a different way to preserve data layout. It is not clear whether this method of producing a cache-aware RDS is any simpler or easier to automate than devising cache-aware RDS by normal methods, or whether the resulting RDS would be as efficient as a traditional cache-aware RDS. Although we don't consider it in this thesis, removing any pointers internal to clusters of nodes after applying perfect data movement might by useful to improve the performance (although read-only operations on the RDS are now different).

In summary, perfect data movement is the process of inserting code into the update operations of an RDS to restore a chosen layout (which may be relaxed, i.e. have some flexibility in it) after each update. By varying the layout we can reduce the overhead required to maintain it, and hopefully obtain good performance. We can view the application of perfect data movement as either an optimisation applied to an existing RDS, or as a way of synthesising a cache-aware RDS. Thus we investigate perfect data movement for three reasons: firstly as a method of optimisation (although the implementation difficulty is high), secondly to get a better grasp on pointer-update data movement (there is a large middle ground between perfect and reallocation), and thirdly to see if the synthesised cache-aware structures compare well to existing cache-aware RDSs (noting that perfect is at a disadvantage in terms of performance because it retains pointers internal to clusters).

In the rest of this chapter we describe the application of perfect data movement to the DICT and MLIST benchmarks.

6.2 DICT

The B-Tree – the canonical cache-aware structure for the dictionary problem – supports operations in less than

$$1 + \log_{B/24-1}(N) \quad = \quad 1 + \frac{\log_2(N)}{log_2(B/24-1)}$$

block transfers, where B is the block size in bytes[1]. By contrast, a binary search tree with poor layout requires at least $\log_2(N)$ block transfers. This bound is achieved to within one

[1]Nodes consist of a count of the number of keys ($k \leq max$), k keys, k values, $k+1$ pointers, and a field indicating if a node is a leaf. The node size in bytes is thus $4(3max + 3)$. Assuming a node size the same as the block size B, and assuming all nodes have the minimum number of keys (given by $k = (max - 1)/2$), we obtain $k = B/24 - 1$.

block transfer if the tree is perfectly balanced. The only known upper bound is N, because in the worst case the tree may become linearised. Furthermore, the expected height of the tree is not known either [20], so an expected number of block transfers cannot be given. In practice the balance we observe in the DICT benchmark is quite good – the height is about 25 for a tree with about 2^{20} nodes.

We will now consider the expected number of block transfers needed per operation to maintain the layout of a binary search tree under uniformly distributed pointer updates. We assume that the tree is *complete* and of height H. We assume that the layout uses complete clusters of height $h = \log_2(B/16)$ (16 bytes per node). We also assume that H is a multiple of h, and that H is large compared to h.

Layout is maintained simply by re-laying out the cluster the pointer update is in, and all clusters below in the tree. The analysis can be found in the appendix in §A.1. The number of block transfers needed for an operation (a delete then an insertion) is

$$\frac{4H}{h} + c \quad = \quad \frac{4\log_2(N)}{\log_2(B/16)} + c$$

whereas an unoptimised tree costs

$$2H + c'$$

where c and c' are small. Thus, provided $h > 2$, maintaining clusters is worthwhile. For a 128 Byte line size and 16 Byte node size, $h = 3$. The number of blocks accessed is the same $O()$ as the B-Tree, but about 3 times more numerically, so unless the instruction costs of the B-Tree are much higher than reclustering, we expect B-Trees to be more efficient than perfect data movement.

In summary, for DICT the simple approach of re-laying out the modified cluster and those below appears promising, although our analysis depended on the tree being well-balanced. This approach has the advantage that it is not very tightly coupled to the update code – all that is required is to identify which cluster the pointer update was in (the *'updated cluster'*), and then to re-lay it out, and all clusters below it. The disadvantage of this simple approach is that a large number of node movements may take place – at least a cluster's worth per update. Later in this section we will investigate better solutions that move the minimum number of nodes required for the first d levels of clusters below (and including) the updated cluster. The simple approach is the *'depth zero'* solution in the sense the data movement performed depends on the form of the update operation for the first zero levels of clusters – in other words, we resort to a simplistic re-laying out of clusters immediately. We show in this section that a *'depth two'* solution – performing 'update-sensitive' node movement for the updated cluster and the level below – achieves near-minimal total node movement experimentally. In the evaluation chapter we will discover whether minimising node movement is necessary to obtain good performance, and to what depth. In other words, we will determine how dependent on the form of the update the optimisation must be to perform well. We consider only small fixed depths because it is simple than infinite depth, and the distribution of pointer updates suggests that only a small depth is required (96% are in the bottom two clusters for a perfectly balanced tree[2], and we know that tree in DICT is quite well balanced).

[2]The tree consists of $1e6 \sim 2^{20}$ nodes, and is clustered from the top, so the bottom two levels of cluster correspond to the bottom five levels of the tree, if the tree is complete. Thus, $1 - 2^{(20-5)}/2^{20} = 96\%$.

This use of only a small depth is feasible precisely because the tree is well balanced, and the distribution of pointer updates is such that most updates are near the bottom of the tree. Thus, the results given for perfect data movement represent the upper bound on performance that can be obtained for a binary tree. It is quite plausible that data movement would be applied to unbalanced trees, in an attempt to use data movement to bridge the performance gap between a vanilla tree and a harder-to-implement self-balancing tree. By contrast, if pointer updates are not usually near the bottom of the tree, the fundamental assumption of perfect data movement is contradicted – namely that it is clearly not worthwhile maintaining layout for parts of the tree that are used too infrequently (the result of $2H + c'$ from the analysis on page 85 does not hold). Instead, a 'lazy' method of restoring a known data layout could be used (cf. Bulk data movement's technique of embedding data movement in a program's existing traversals, see §5.5).

In the rest of this section we discuss the two layout we will maintain using perfect data movement, and review the form of the pointer updates occurring to the tree. We will then describe the implementation of the optimisation for the two layouts.

6.2.1 Choice of layout

We will use two different layouts. Firstly, the '**BFS**' layout – each line contains a cluster of nodes obtain by breadth-first search, and the children of the cluster are dealt with recursively[3]. The second layout is called the '**fixedHeight**' layout and is formed as follows: Starting from the root x of the tree, we allocated x, x's children and x's grandchildren in a line. We then recursively repeat the same procedure starting from x's great-grandchildren. Note that with 16 byte nodes and 128 byte lines we can fit 7 nodes $- x$ and all its children and grandchildren – into one line, with room for some header fields. If this did not hold, we could use the first k of x's grandchildren. Note that this procedure leaves some unused space in lines if the tree is not complete and is thus not as effective a layout as BFS. However, it is simpler to maintain – for example the space in lines means that insertions are often very quick.

Note that both these layouts specify a unique grouping of nodes in cache lines – the data movement code only has to decide how to achieve the grouping, rather than deciding on a grouping. Other layouts are possible. We won't consider layouts that involve pages, because the investigation in §3.4.2.1 suggested that the benefits will be slight. We also do not consider relaxing node density (beyond the relaxation that occurs with the unused space with the **fixedHeight** layout).

In terms of static behaviour of the two layouts, experiments show that **fixedHeight** is at most 5% slower than **BFS** – which is unsurprising because the balance of the tree in DICT is good (maximum depth of 25 for $\sim 2^{20}$ nodes).

6.2.2 Pointer update operations

The insert and delete operations on the tree are composed of three different pointer updates, attach, cut and substitute:

- Insertion: node **attached** to null pointer field

[3]In the methodology chapter we referred to this as the '**goodLine**' layout, but here we call it to '**BFS**' to emphasise how the clusters are built.

		depth-zero	depth-one	depth-two
MF:	attach:	recluster	onAttach(recluster)	onAttach(merge)
	cut:	recluster	onCut(recluster,recluster)	onCut(split,merge)
MN:	attach:	recluster	onAttach(recluster)	onAttach(merge)
	cut:	recluster	copy, onCut(recluster,recluster)	copy, onCut(split,merge)
	substitute:	copy	copy	copy

Figure 6.1: Functions used for different depths of perfect data movement for the **BFS** and **fixedHeight** layout for DICT's MOVENODE (MN) and MOVEFIELDS (MF) variants. See §6.2 for details.

- Deletion (1-child case): node x with zero or one child **cut** out of tree
- Deletion (2-child case, MOVENODE): x's successor or predecessor (in key order), y, is found, and **cut** from the tree. Node y is **substituted** for x.
- Deletion (2-child case, MOVEFIELDS): x's successor or predecessor (in key order), y, is found, and **cut** from the tree. The key and value of y are moved into node x.

Note that MOVEFIELD's 2-child delete is equivalent to following MOVENODE's procedure, *and then freeing node x and moving node y to the same location in memory.* This is the difference between MOVENODE and MOVEFIELDS: With MOVEFIELDS we assume that we are allowed to free x during delete. In the MOVENODE variant, it is assumed that delete cannot free x, stopping delete from reusing its memory location, significantly increasing the rate of degradation to the tree. This means when repairing layouts using perfect data movement we cannot reuse x's cell.

6.2.3 Enforcing the BFS layout

Perfect data movement is achieved using a number of different functions (copy, recluster, onAttach, onCut, merge and split), whose implementation is different depending on the layout (**BFS** or **fixedHeight**). We write for example BFS.copy or fixedHeight.copy to talk about a particular function.

The functions are used as shown in Fig. 6.1 to implement depth-zero, depth-one and depth-two perfect data movement for DICT. (Recall the depth is the depth, measured in clusters, to which layout is restored using a minimal number of node movements. Below that depth naïve reclustering is used).

6.2.3.1 MOVEFIELDS

Recall that each line contains exactly zero or one clusters. This makes the implementation simpler and more efficient. The count of nodes in the line is the size of that line's cluster, so it is not necessary to use BFS to see how large a cluster is. Cluster boundaries are equivalent to line boundaries, and can be kept track of with low overhead as the tree is descended. Throughout this section, denote the maximum number of nodes that can fit in a line n.

Depth-Zero: The simplest way to to repair layout after the attach and cut operations is to use the recluster function:

> BFS.recluster(node *x) Creates a new cluster with x as root using copy, and then recurses over the tree by calling recluster(y) for all children y of the cluster (nodes who aren't in the cluster but whose parents are).

> BFS.copy(node *x) Creates a new cluster with x as root, by finding by BFS the $\leq n$ nodes that should be in x's cluster, and moving them all to an empty line.

Note that this function is identical to the body of the twoSpace bulk data movement optimisations for Dict. All that must be done for attach and cut is to identify the correct x to recluster from – this can be done by observing line transitions in the insert and lookup operations, respectively. Thus after a modification, the tree is always perfectly clustered. This approach works, but moves many more nodes than necessary – minimal-node-movement reclustering is performed to a depth of zero.

Depth-One: We will now discuss how we can reduce the number of node movements. The first step is to specialise based on whether an attach or a cut is performed, using the functions onAttach and onCut: (Example actions of these functions can be found in Fig. 6.4 and Fig. 6.5)

> BFS.onAttach(function fn1)(node *P, node *p, int i, node *x): This function updates node p's i^{th} child pointer to point to x, preserving clustering. P is the root of p's cluster. First the function discovers if x should be in cluster P or whether a new cluster should be created for it. If cluster P is not full – which is discovered in constant time by asking the memory manager how many nodes are in the line – x is moved to P's line. Otherwise, we obtain the n nodes that should be in cluster P by BFS. If x is not found, it is allocated in an empty line (a new cluster). Otherwise, the BFS is continued for one more node to find the node e in cluster P that should be evicted in favour of x. e is evicted by calling fn1(e), which moves it out of P's line, creating space for x.

> BFS.onCut(function fn1, function fn2)(node *X, node *x): This function is used to preserve clustering when x is cut from the tree. X is the root of x's cluster. The cut may bring some other nodes into the cluster and evict others. If X is not full before the cut (because the subtree below X's root contains less than n nodes), then nothing needs to be done when x is cut – cluster X shrinks by one node. If X is full before x is cut, then some node movement may occur. After x is cut, the n or n-1 nodes that should be in X are found by BFS. Any node y not already in the x is moved in and its children c_0, c_1 dealt with by fn1(c_0), fn1(c_1). The first incoming node reuses x's space. To create space for further nodes, the BFS is continued to find the node e that should be evicted from cluster X. This node is evicted by calling fn2(e). Note that there are a few special cases where a BFS is not required – such as when x was on the boundary of the cluster.

These functions repair the first cluster using a minimal number of node movements. After the first cluster is repaired, some clusters further down the tree may have to be repaired. This can be done most simply by calling recluster, giving a solution that performs minimal reclustering to a depth of one.

Depth-Two: To increase the depth of minimal-node-movement reclustering to almost two, the calls to recluster in the above are replaced with calls to merge and split (example actions of these functions can be found in Fig. 6.6).

These functions are used either to promote a node into the cluster above (*splitting* its current cluster), or to push it out the bottom of a cluster (*merging* the two clusters below it). The implementation of split performs the minimum number of node movements, but merge does not always (but is close in practice), and so these functions give a depth of almost two.

BFS.merge(node *X, node *x): This function evicts node x and any of its descendants from the cluster rooted at node X. On entry, the cluster at X may be smaller than n, but the tree is properly clustered below that cluster. If x is not on the boundary of the cluster, recluster(x) is used to build a new cluster rooted at x. Otherwise, x is on the boundary of the cluster, and its children are roots of two (possibly null) clusters. A new cluster is built rooted at x reusing one of the children's cluster's lines, chosen to minimise node movements. Below this recluster is used to recluster the tree.

BFS.split(node *x): x is the root of a cluster. This function splits this cluster into two new clusters rooted at x's children (reusing x's cache line if appropriate), then repairs clustering below that. The nodes that will form the two new clusters are found by BFS, and then the minimal numbers of nodes are moved to create them, reusing x's line. Below this, recluster is used to recluster the tree.

6.2.3.2 moveNode

Maintaining the **BFS** layout for moveNode is similar to moveFields, with two important differences. The use of the functions is given in Fig. 6.1.

Firstly, we cannot simply invoke onCut. Recall that in moveNode we are assuming that the node x that is deleted is used by the mutator for some purpose and so its cell cannot be reused by the Delete operation. Eventually the node is freed so memory is not exhausted. If the new cluster contains a line's worth of nodes, we clearly must move all the nodes into an empty line to cluster them. If the new cluster is smaller, it might be possible to leave the nodes in the old line, sharing it with the node x, but this increases the complexity and overhead of the optimisation because cluster size is now no longer the same as the number of nodes in the line – in particular it means that the cluster must be moved to a new line later on if it needs to grow. It also means that the time that it takes for the benchmark to free x becomes another parameter of the benchmark that can be adjusted. Thus even if the new cluster fits in the line, it is better to move it to an empty line using the copy function, before invoking onCut.

Secondly, during the delete 2-child case, recall that when deleting node x, its successor or predecessor y is substituted for it. As we argued above, reusing x's cell is not possible, and so to repair the cluster, it must be moved to a new empty line using copy, rather than simply moving y into x's cell.

6.2.4 Enforcing the fixedHeight layout

Enforcing the **fixedHeight** layout is much simpler than **BFS**. Lookup is changed to track cluster boundaries in a different way (using the depth from the root of the tree), but the fixedHeight.copy(node *x) Creates a new cluster with x as root. x and its children and grandchildren are moved to an empty line.

fixedHeight.recluster(node *x) Creates a cluster with x as root using copy, then calls itself for all children y of the cluster (nodes who aren't in the cluster but whose parents are).

fixedHeight.onAttach(function notUsed)(int level, node *p, int i, node *x): Updates node p's i^{th} child pointer to point to x, preserving clustering. level is the level of the tree the node x is on, which determines whether x should be in p's cluster. If so it is moved there (there is always space by construction), otherwise it is moved to an empty line.

fixedHeight.onCut(function fn1, function notUsed)(int level, node *x): x is cut from the tree, preserving clustering. level is the level of the tree the node x is on. When x is cut, all of its descendants are moved up by one node. One row of these may cross the cluster boundary, and are simply moved into the cluster's line (there is always space, by construction). No nodes leave the cluster when a cut occurs. The descendants of any node that crosses the cluster boundary and joins x's cluster are dealt with by calling fn1.

fixedHeight.split(node *x): Exactly the same as the split function for **BFS**, except instead of finding sets of nodes by calls to BFS, clusters are constructed using the root node x and its children and grandchildren.

fixedHeight.merge(node *x): Not needed for **fixedHeight**.

Figure 6.2: The functions used for perfect data movement using the **fixedHeight** layout for DICT. See §6.2.4 for details.

insert and delete functions do not need to be changed. The bodies of the onAttach, onCut, recluster and split functions are changed, and merge is no longer required. See Fig. 6.2 for descriptions of the functions.

6.2.5 Discussion

6.2.5.1 Overhead

For **BFS**, we can perform minimal node movements to almost a depth of two, and exactly a depth of two for **fixedHeight**. Because of the distribution of pointer updates, this provides almost minimal total node movements. We can deduce this by observing how many *potentially unnecessary* moves are being performed – ones that occur in functions which have not been shown to move the minimal number of nodes.

The functions onAttach, onCut, merge and split have been constructed so that any node movements they perform directly (not by calling another function) are necessary. Observe for example during split and merge that lines are reused in a way to minimise the total number of node movements. The function recluster does not move nodes itself, it uses the copy function. The copy function is also used during deletions for the MOVENODE variant, but the node movements it performs are necessary. Thus, we can count the number of

variant	depth	potentially unnecessary moves (%)	nodes moved /operation
moveNode	0	70	7.0
	1	10	3.2
	2	.75	3.0
moveFields	0	100	5.0
	1	33	1.0
	2	2	0.8

Figure 6.3: The effect of varying the depth to which cluster rebuilding is performed using the minimal node of movements, for the **BFS** layout. The third column shows the proportion of node moves that occur in functions that are not constructed to perform the minimal number of movements. Fourth column shows average number of nodes moved per operation.

potentially unnecessary moves (moves which may be unnecessary), we simply count the nodes moved by copy when called by the recluster function.

Fig. 6.3 shows how the number of potentially unnecessary node movements falls to 2% or less when the depth is increased to two, the majority of which is achieved by depth one. In other words, we have demonstrated that overhead measured in number of node movements can be minimised in practice by concentrating on only the first two levels of clustering using a few simple functions. Moreover, most of the reduction is obtained by only concentrating on intelligently reclustering the updated cluster. Thus, devising algorithms to performing minimal node movement to infinite depth is not necessary, simplifying the implementation of perfect data movement (and least while the assumptions regard tree balance and distribution of pointer updates hold, see the discussion in §6.2 on page 85).

6.2.5.2 Memory

Perfect data movement requires a supply of empty blocks, and thus does not have clearly stated memory usage – worst-case memory m_{worst} will probably be higher than the memory in use m. By contrast, reallocation and bulk data movement's 1space have memory strictly bounded by a value $2 \geq m > 1$ chosen by the applier, and bulk 2space uses a fixed value of $m = 2$. Worst-case memory as calculated in Appendix A.2 is in $[4, 4.2]$ for both layouts. Thus both methods have much larger m_{worst} than the values of m we intend to use for reallocation and bulk. We will investigate how m relates to m_{worst} in the chapter 7.

6.3 MList

The optimal layout for MList is **PcLco**, using the notation of §3.4.2.2. We will implement perfect data movement to maintain this layout, and four others: **PcLco**, **PcLm**, **PmLc**, **Pm** (in order of decreasing layout quality). We maintain these using three different components:

clustStrict(blockSize)	The nodes of each list are kept clustered in order within blocks of size blockSize bytes.
clustNonStrict(blockSize)	As above, except nodes are not kept in order within blocks.
min(smallBlockSize, largeBlockSize)	Supports allocation and freeing of blocks of size smallBlock, using the minimal number of blocks of size largeBlock, per list.

As with the cache-aware structures [6, 25, 60] to implement clustering we will relax node density. Relaxing memory to achieve minimisation isn't necessary; as we demonstrate later maintaining the exact minimum can be done with a small constant number of block transfers. The four layouts are maintained using the components as:

PcLco	clustStrict(pageSize)
PcLm	clustNonStrict(pageSize)
PmLc	clustStrict(lineSize) + min(lineSize,pageSize)
Pm	min(nodeSize,pageSize)

Notice that min is used for **Pm** to support allocation/freeing of nodes, but used for **PmLc** to support allocation/freeing of lines which are used by the clustStrict component.

Varying the page size used allows us to reduce memory usage, and to reduce the cost of updates (at the expense of layout) for strict clustering. Thus we will vary the size of pages used, between twice the machine page and equal to the line size (apart from where min(lineSize,pageSize) is used, where we require pageSize > lineSize). This gives four families of optimisation as summarised below – notice that the **Lm** and **Lc** layouts are also achieved by varying page size:

Shorthand	Set	Set includes...
min	$\{\text{min}(\text{cell}, \text{b}) : \text{b} \in \text{B}\}$	**Lm, PmLm**
clustStrict	$\{\text{clustStrict}(\text{b}) : \text{b} \in \text{B}\}$	**Lc, PcLco**
clustNonStrict	$\{\text{clustNonStrict}(\text{b}) : \text{b} \in \text{B}\}$	**Lc, PcLm**
clust[Non]Strict+min	$\{\text{clustNonStrict}(\text{line}) + \text{min}(\text{line}, \text{b}) : \text{b} \in \text{B}\backslash\{\text{line}\}\}$ $\cup$ $\{\text{clustStrict}(\text{line}) + \text{min}(\text{line}, \text{b}) : \text{b} \in \text{B}\backslash\{\text{line}\}\}$	**PmLc**

(for $\text{B} = \{2^7, 2^9, 2^{11}, 2^{12}, 2^{13}\}$, where lineSize= $2^7 = 128$ and pageSize= $2^{12} = 4096$)

In the rest of this section we discuss how minimisation and clustering is achieved, for the latter considering a simpler high-overhead version and a more complex low-overhead version (cf. DICT's depth=0 and depth=2).

6.3.1 Minimisation

Minimisation is performed using a simple algorithm. Recall that the aim of minimisation is to support the allocation and freeing of *small blocks* (cells or lines) using the minimal number of *large blocks* (line or page). In other words for each list, there are some number of large blocks who have all their small blocks allocated, and at most one large block with only some of its small blocks allocated.

To allocate a new small block, the partially-used large block P is used, if the list has one. If the list had no partially-used large block, an empty block is claimed and a small block is allocated in it. Thus under allocation the number of partially-used large blocks for a list is either zero or one.

Unless A happens to be the list's partially-used large block, or the list does not have a partially-used large block, freeing a small block from some large block A will mean that there are two partially-used large blocks associated with the list. This situation is prevented by moving one of the partially-used large block's small blocks into A.

Since any large block may become the list's partially-used large block, some mechanism for moving the small blocks must exist. Specifically, the parent of each small block must be known. If the small blocks are lines, this information can be simply stored in the line header with little space overhead and update overhead. If the small block is a cell, we must add a parent pointer to each cell, increasing M by a factor of 1.5. The parent pointers in cells should be considered not as a change of node definition, but as auxiliary information attached by the memory manager or optimisation to each cell. Maintaining parent pointers when small blocks are cells or lines does not involve any additional effort, because all pointer-update operations on the RDS already have optimisation code applied to them.

There certainly exist minimisation schemes that don't require a parent pointer to be added to each small block, and where nodes may have more than one parent, or the increase in M by a factor of 1.5 is undesirable, such a scheme may be more attractive. Such schemes probably achieve minimisation within some bound rather than exactly. We use the exact scheme here because the implementation is simple, consisting of only a few lines of code.

6.3.2 Clustering

Firstly, we will describe the layout that is maintained by strict and non-strict clustering, and then we will discuss two methods of maintaining it – simple and complex clustering.

With strict clustering, clustStrict(size), nodes are stored in order aligned to the bottom of each block, whose size in bytes is given by the size parameter. The number of nodes n in the block is stored in a field in the block header – the pointer to the next block is thus the pointer of the n^{th} node.

With non-strict clustering, clustNonStrict(size), nodes are stored in any order, possibly with gaps between them. The address of the last node x in the block is stored within the block header, thus the address of the next block is given by node x's pointer.

Note that in the strict case, pointers only have to be updated at block boundaries, because nodes are always stored in order, but in the non-strict case, pointers always have to be updated. Storing nodes in order increases node density and aids the hardware prefetcher (for large blocks) but inserting a node into a block is $\leq$size/8 node movements rather than one (eight bytes per node). Thus it is hard to tell when non-strict or strict clustering will be more effective.

Note also that blocks have been augmented with an *inter-block* pointer (directly for strict, indirectly for non-strict). Recall that the MList benchmark selects the place to perform the insertion, or the node to delete, by randomly choosing some $i \in [0, listLength)$ and then pointer-chasing to the i^{th} node in the list. Together with a count of the number of nodes in each block, the inter-block pointer could be used to very rapidly traverse to the correct node. However, as stated in §3.2.2, we explicitly assume that all nodes before the i^{th} *must* be accessed before the insertion and deletion (as if they were being checked one by one until the correct location for insertion or deletion is found). This means that the inter-block pointer cannot be used in the main program to reach the i^{th} node. The optimisation is allowed to use the inter-block pointers to scan forward in the list, because it is concerned only with layout of nodes (particularly the number of nodes in a block), not which nodes are found.

Various other clustering layouts beyond strict and non-strict are possible, which we do not investigate in this thesis. If a block contains k nodes, we may store them in cells $x, x+1, \ldots, x+k-1$, where x is allowed to vary. A dense, linear layout is obtained, but update costs are less because nodes may be popped or pushed from either end in $O(1)$. Secondly, we may store nodes in order, but allow gaps. When no gap is available for an insertion, simple $O(k)$ shuffling up is performed to create a gap. This scheme would perform well because it is clearly cheaper in terms of node movements than strict clustering, while achieving a (sparse) linearised layout. Update costs are still higher than non-strict clustering, however.

Now we describe two methods to preserve the clustered layout, of different sophistication and overhead, simple and complex clustering (c.f. depth=0 and depth=2 for DICT).

6.3.2.1 Simple clustering

This is possibly the simplest method of density-relaxed clustering. When a block overflows, a node is popped out of the current block into the next one, and so on, until an overflow doesn't occur or the end of the list is reached. Similarly, underflows are dealt with by filling gaps, again working only forward in the list. By adjusting the minimum density of the blocks, reclustering work may be reduced, at the expense of layout and memory usage.

Assume the list is of length N, and blocks hold between min and max nodes. The insert or delete operation plus the reclustering work accesses the whole list in the worst case, thus at most N/min blocks are accessed. This is assuming uniformly distributed updates, but worst-case reclustering. This is at most $16N/B$ block accesses per insert/delete+reclustering, where B is the block size in bytes and the nodes are 16 bytes.

6.3.2.2 Complex clustering

As discussed in §2.2.4.2, there are various cache-efficient linked list implementations, all based around relaxing node density. Using blocks that are at least 50% full is very simple, because when a block overflows or underflows, the correct density can be restored using the nodes of at most one adjacent block and one allocation or freeing of a block [6]. In the I/O model, again assuming uniform updates and worst-case reclustering, the traversal takes $N/2min$ block transfers, updates take 2 block transfers. min is fixed at $(B/8)/2 = B/16$, assuming 8-byte nodes, and thus $2 + 8N/B$ block transfers are required.

The more complicated 'VCL' scheme of Rubin et al. [60] allows a variable minimum density, which can thus be used to trade-off improved traversal performance for increased cost of updates. Blocks have between min and max keys in them, stored in order, apart from the last block in the list. The only pointers are between blocks. On insertion, if a block overflows, the next *min* blocks are inspected. If a block is found with less than *max* nodes, the overflow is dealt with by shuffling nodes until the block is reached. Similarly if the end of the list is encountered, shuffling occurs and a new block is allocated. If *min* blocks with *max* nodes are found, $max - min$ empty blocks are allocated, and the $min \times max + 1$ nodes evenly distributed over the *max* blocks. When underflow occurs during deletion, nodes are shuffled to restore density if a block is found close enough. If *max* blocks with *min* nodes are found, $max - min$ blocks are freed, and the nodes redistributed to even density in the remaining *min* blocks.

Using a VCL scheme to perform node clustering (retaining internal pointers), the number of blocks accessed is bounded by $min + max \leq max + max = B/8 + B/8 = B/4$ (scan forward max blocks and move to *min* blocks, or scan forward *min* blocks and move to *max* blocks. Each node is 8 bytes). This gives a cost of the insert or delete operation as $(N/2)/min + B/4 = (N/2)/(B/16) + B/4 = 8N/B + B/4$. This is worse than the half-full method by a small factor, but roughly half the cost of simple clustering, for long enough lists:

Clustering method	Block transfers
Simple	$16N/B$
Complex - 50% full blocks	$8N/B + 2$
Complex - VCL	$8N/B + B/4$

We use the VCL scheme rather than the 50% full scheme because it allows finer control over memory usage (particularly worst case), and because we found in §3.5.2 that is better in practice, at least when internal pointers are removed. Furthermore, schemes similar to VCL have been used practically, e.g. as a suggested cache-aware C++ STL list implementation [25].

6.3.3 Memory usage

For DICT, we use only line-sized clusters, based on the observation made in the methodology chapter that page performance is relatively unimportant, but incurs large overhead to maintain due to the increased number of nodes within a page. Furthermore, apart from choosing between either **fixedHeight** or **BFS**, we have no control over memory. For MLIST, we do have control over actual and worst-case memory usage, by varying the minimum number of nodes in a block when clustering, and by varying block size. Thus, for MLIST, we're interested not only in performance but the relationship of performance and memory. Worst-case memory can be calculated as follows:

clustStrict(*blockSize*), clustNonStrict(*blockSize*): At any given time, each of the v lists will have at most one block with fewer nodes than the minimum number allowed *min*. It is clear that to maximise the number of blocks in use all other blocks in the list must have *min* nodes in them. We can write the total number of nodes T in all the lists as $a.min + (T - a.min)$, where a is the total number of blocks at minimum density, and $T - a.min$ is the number of nodes that occupy blocks at less than minimum density. Note that $T - a.min \leq v.(min - 1)$, since each list can have at most one block with occupancy $min - 1$ or less. For given a, memory allowance is maximised if the $(T - a.min)$ nodes are spread between as many blocks as possible (up to one per list). The total number of blocks is therefore given as $a + least((T - a.min), v)$, which divided by the minimum number of blocks (T/max) gives the memory usage. For given T and v, the maximum memory usage is found by varying a, which can be done analytically or by computation.

min(*smallBlockSize,largeBlockSize*): As above, using block size *largeBlockSize*, and $min = max$.

6.4 Discussion

Perfect data movement is the process of modifying the update operations of an RDS so a chosen data layout is restored after each update. For Dict, we restore two clustered layouts: clusters are line-sized and are filled either by BFS (and are therefore full except near the leaves of the tree) or are of constant height (and therefore may have gaps in them). The first $d \in [0, 1, 2]$ levels of clusters, starting from the cluster the pointer update was in, are rebuilt using the minimal number of node movements. After this, clusters are rebuilt simply by moving the nodes to an empty line.

For MList, we maintain several different layouts of the form **PxLx** as described in §3.4.2.2. Nodes are clustered either in order and densely packed (strict clustering) or anywhere within the block (non-strict clustering). Clustering is performed using either a simple high-overhead method or a more complex lower-overhead method similar to Rubin et al.'s VCL structure [60]. Minimisation (maintaining a supply of small blocks using a minimal number of large blocks) is performed using a simple algorithm which requires augmenting each small block (cell or line) with a parent pointer.

We may view the application of perfect data movement to an RDS in two ways:

1. As an optimisation applied to an existing RDS
2. As a method of creating a new cache-aware RDS

Both are valid viewpoints depending on the details. Viewing as an optimisation is attractive because the shape of the RDS is the same, and hence read operations are unchanged. Although updates are far more complicated, they can still be decomposed into the part of the code that changes the shape of the RDS, and other code which moves nodes in memory – i.e. two distinct levels, one semantically significant to the program, and another level only related to performance. Furthermore, specific perfect data movement methods, such as depth-0 for Dict, do not require a great deal of implementation effort or deep understanding of the pointer update operation (just where it is occurring), and thus applying them automatically is plausible.

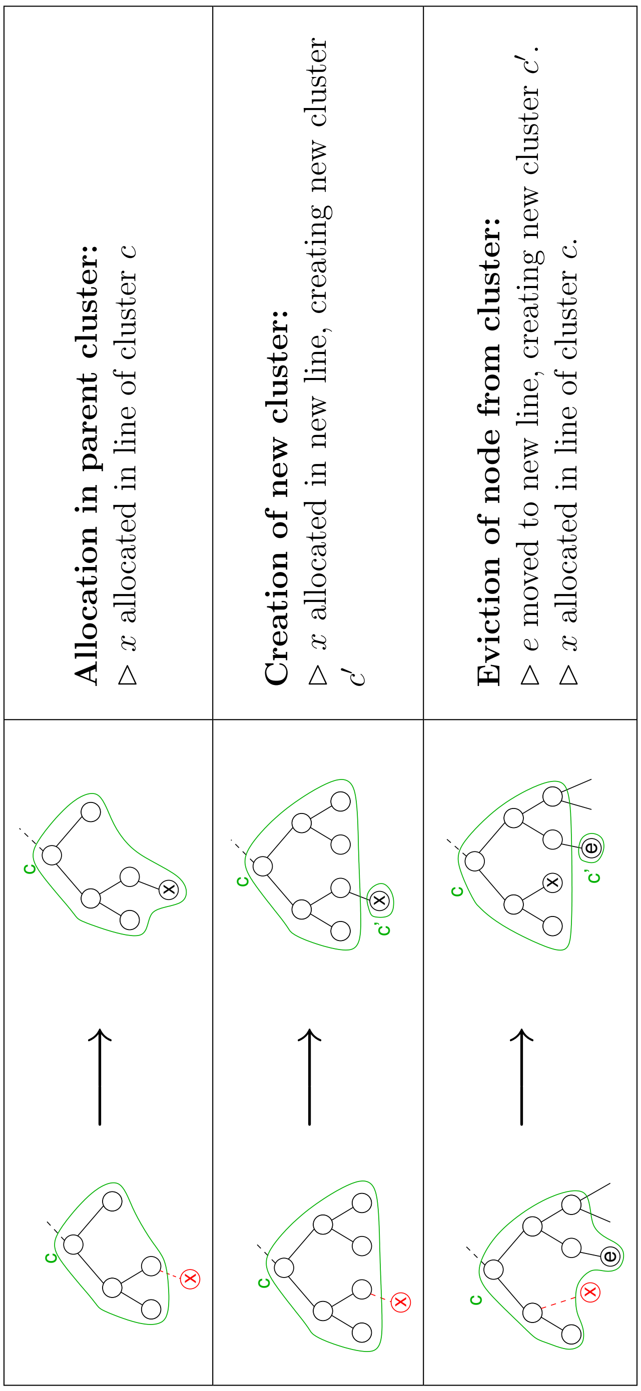
Alternatively, one may believe that this process is sufficiently complicated in general to view it as a method of producing a new RDS, not optimising an existing RDS. The depth-2 solution for Dict involves specific knowledge of the type of update occurring – attaching, cutting and substitution. It seems likely that any compiler or programmer able to perform this analysis and apply perfect data movement would be able to alter the program to use a cache-aware structure. Therefore we would rather view the application of perfect data movement to an RDS as a method of synthesising cache-aware RDSs, not optimisation.

Either of these two view points is valid, and we will address both during the evaluation. As an optimisation, if perfect data movement is less effective than reallocation, we may draw conclusions about the correct balance between layout quality and overhead, and perhaps also that low-overhead approximate methods are better than more precise ones. If bulk data movement out-performs reallocation and perfect, we might conclude that pointer-update layout maintenance is less powerful than en-masse layout maintenance.

As a method of constructing a new cache-aware RDS, we will investigate how close perfect data movement comes to the performance of the traditional cache-aware RDSs we considered in §3.5 (in other words, to what extent can the gap between a proper cache-aware structure and a vanilla structure augmented/optimised with data movement code be bridged). For MList, we are focusing on two levels of the memory hierarchy, and in particular are using minimisation, which for short lists can achieve similar miss rates to clustering, but with lower overhead.

We note that the investigation of Dict will give an upper bound on the performance of the perfect data movement optimisations (but such an investigation is still valuable): Firstly, pointer updates are uniformly distributed, meaning that most updates are near the bottom of the tree. Secondly, the tree has been shown to have good balance in practice. The second point is relevant when comparing perfect data movement against a B-Tree, because the latter must expend considerable effort keeping itself balanced (the clear next step is to investigate runtime data movement for self-balancing trees, but we do not consider this extension in this thesis).

Whether viewed as an optimisation or a new cache-aware structure, there are several points we will investigate in the evaluation chapter. Unlike reallocation and bulk, perfect data movement requires a supply of empty blocks, which produces different actual and worst-case memory usages. We will therefore investigate reducing the block size for MList to see if worst-case can be reduced without damaging performance (there are also probably performance reasons to use smaller blocks). We will also be interested in the relationship between performance and the layout quality that is maintained – for Dict, use of **fixedHeight** instead of **BFS**, and for MList, several different layouts that are not as good as **PcLco**. Finally, for both benchmarks we are interested in how closely coupled to updates the data movement code must be. For Dict we will investigate whether depth-0 is effective, and if not, what depth is required to get the best performance, and for MList we will compare simple and complex clustering.



Allocation in parent cluster:
▷ x allocated in line of cluster c

Creation of new cluster:
▷ x allocated in new line, creating new cluster c'

Eviction of node from cluster:
▷ e moved to new line, creating new cluster c'.
▷ x allocated in line of cluster c.

Figure 6.4: Examples of the operation of the BFS.onAttach function. The node being attached is labelled X and the pointer it is being attached to is dashed. Lines on the diagrams indicate the boundaries of clusters. If a set of nodes is encircled, this indicates that that set of nodes inhabit the same cache line and no other nodes inhabit that cache line. If the line on the diagram is not closed, this indicates that there may be other nodes in the cache line. All non-null pointers are shown using line-segments, but the node the pointer points to may be omitted – i.e. if a node has no line segments below it, it is a leaf node, and if a node a has a line segment that does not end in a node, this indicates that there is some subtree of at least one node attached to a's pointer. A *dashed* line segment above some node b that does not end in a node indicates that there may be nodes above b in the tree.

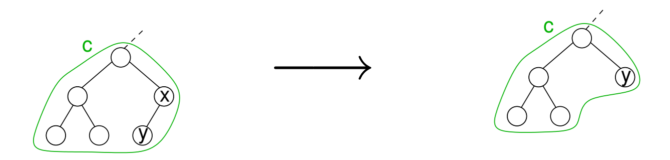
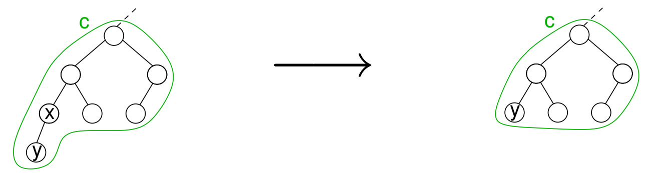
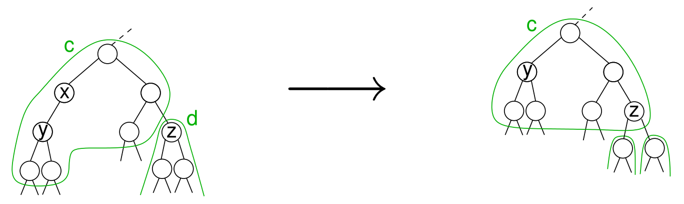
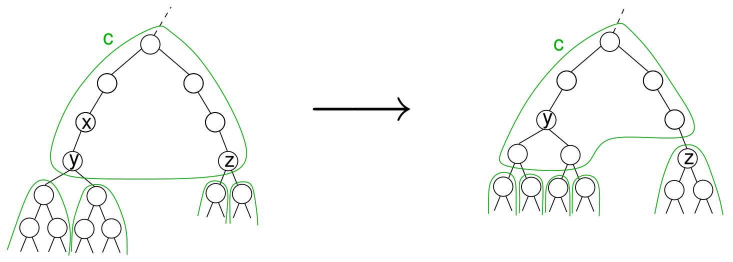
	**Cut in a partially-full leaf cluster:** ▷ x freed, cluster c shrinks by one node
	**Cut in a full leaf cluster:** ▷ x freed, cluster c shrinks by one node
	**Cut in full non-leaf cluster:** ▷ BFS.recluster or BFS.split (See Fig. 6.1) used to split cluster d into two clusters, and recursively recluster the tree below. If BFS.split is used, the line of cluster d will be reused for one of the new clusters. ▷ z moved into line of cluster c
	**Cut in full non-leaf cluster:** ▷ node z moved out of the line of cluster c using either BFS.recluster or BFS.merge (See Fig. 6.1), and the tree is recursively reclustered below. ▷ four new clusters are created with the grandchildren of y as roots, using either BFS.recluster or BFS.split, and the tree is recursively reclustered below. ▷ y and its children moved into the line of cluster c

Figure 6.5: Examples of the operation of the BFS.onCut function. See Fig. 6.4 for details of how to interpret these diagrams.

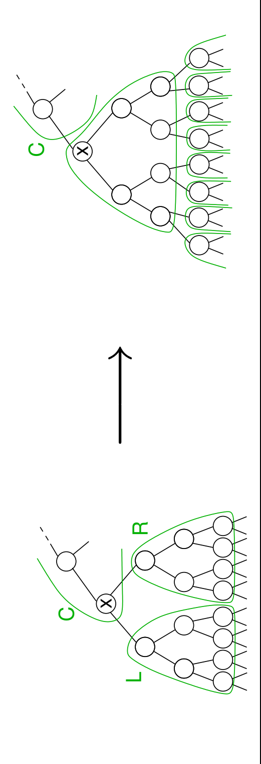
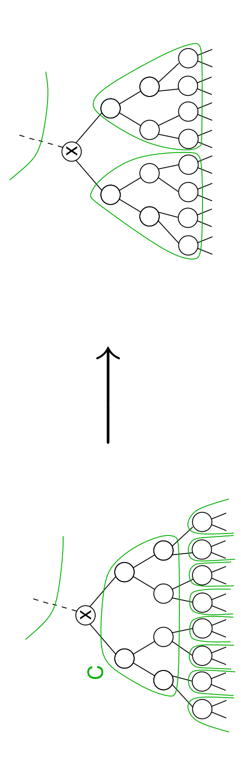
	**BFS.merge:** ▷ New cluster built with x as root, reusing either L's or R's line. ▷ The tree is recursively reclustered the new cluster using the recluster function. ▷ Cluster C has shrunk by one node
	**BFS.split:** ▷ Node x has been moved out of its previous cluster C by the caller ▷ Two new clusters built with children of x as roots, reusing line C ▷ The tree is recursively reclustered below using recluster

Figure 6.6: Examples of the operation of the BFS.merge and BFS.split functions. See Fig. 6.4 for details of how to interpret these diagrams.

Chapter 7

Evaluation

In this chapter we evaluate the performance of each method of optimisation, comparing each method to the others and to the cache-aware structures. We then discuss further work, including how the methods may be applied automatically.

7.1 Reallocation

In this section, we investigate what sort of searching and coallocators are suitable for the different (re)allocation sites in the benchmarks, and determine whether existing general coallocators (ccmalloc) perform well, or whether we have to specialise the coallocators and searches per-benchmark. The relative importance of performing data movement at the different sites is investigated. Finally we investigate whether focusing on lines or pages or both is more effective.

7.1.1 Dict

The performance of reallocation for Dict compared to the upper bounds (§3.4.3) is shown below:

v	maximum	reallocation	
		m=1.1	m=2.0
moveNode	55%	24	35
moveFields	36	13	19

In the rest of this section, we will consider which sites are useful for reallocation and how they interact. We will then investigate the value of attempting to improve TLB (page) performance as well as L1/L2 (line) performance, and finally consider the effect of memory.

7.1.1.1 Choice of sites and hints

Firstly, we will investigate which sites are most useful, and what form of searching is most effective. This investigation is performed by turning a subset of sites off (i.e. forcing a null strategy), and then varying the strategies used at all the other sites to find the best combination of strategies. We repeat this for all subsets of sites.

MOVENODES & MOVEFIELDS:

$$\{\mathsf{insert} := \langle \mathsf{L}; \mathsf{EL}\rangle \times \langle \mathsf{parent}_{\mathsf{p}_1}; \mathsf{t}_1\rangle$$
$$\wedge\ \mathsf{deleteCut} := \langle \mathsf{L}; \mathsf{EL}\rangle \times \langle \mathsf{parent}_{\mathsf{p}_2}, \mathsf{child}_{\mathsf{c}_2}; \mathsf{t}_2\rangle$$
$$\wedge\ \mathsf{deleteMove} := \langle \mathsf{L}; \mathsf{EL}\rangle \times \langle \mathsf{parent}_{\mathsf{p}_3}, \mathsf{child}_{\mathsf{c}_3}; \mathsf{t}_3\rangle$$
$$: \forall \mathsf{p}_1, \mathsf{p}_2, \mathsf{p}_3 \in \{0,1,2,4,7\}, \forall \mathsf{c}_1, \mathsf{c}_2 \in \{0,1,2\}, \forall \mathsf{t}_1, \mathsf{t}_2, \mathsf{t}_3 \in \{\mathsf{false}, \mathsf{true}\}\ \}$$

Figure 7.1: Reallocation strategies used for the different sites for DICT, used for the experiments in §7.1.1.1 and §7.1.1.2, and the results in Figs. 7.16, 7.18 and 7.19.

Chilimbi et al.'s experience with ccmalloc [15], and our preliminary experiments, suggest that ⟨L; EL⟩ is a good coallocator pattern to use for all sites. Formally, where a site is enabled, the coallocators we will evaluate are defined by the sets given in Fig. 7.1. Note that we do all parents before children because we expect this to be the most effective (see §4.3.1). Results can be found in Fig. 7.16 on page 141, including the values of the p's, c's and t's that gave the optimal strategy combination.

MOVENODE: Best performance is obtained from using only the deleteMove and insert sites – further investigation concludes that performance steadily decreases as the number of hints $(c+p)$ used at the deleteCut site increases. The deleteCut site is helpful only when the insert site is not in use. Observe that deleteMove is by far the most powerful site – by itself it obtains 16%, compared to the best of 24%. Contrast this to the insert site which yields only 6% if it alone is enabled. Thus we have seen that all three sites have very different behaviour, and worth. The reason for the different behaviour is possibly explicable, especially since the structure in question is simple[1], but for real programs no such prediction will be possible in general, and so brute force methods are probably the best way of determining which sites are valid.

MOVEFIELDS: For this benchmark it is preferable to perform data movement at both insert and deleteCut sites (13% vs 5%), a different behaviour to MOVENODE. For that variant when deleteMove is disabled it is unhelpful to perform data movement at both insert and deleteCut site – insert alone is preferable. There are several possible reasons for this behaviour.

Firstly, recall that the deleteCut site is actually the merging of two different sites, deleteOneCutBelow and deleteTwoCutBelow. The first is used when a node can be deleted simply by cutting it from the tree (because it has only one child), the second is for the MOVENODE benchmark only and occurs when a node with two children is deleted by substituting it by its successor or predecessor in the tree. The successor/predecessor is always removable from the tree by a cut, which corresponds to the deleteTwoCutBelow site. Thus, although the operation being performed on the tree is the same, the distributions of the reallocatees for the two sites are different (deleteTwoCutBelow's are further down the tree), and since only MOVENODE has the deleteTwoCutBelow site, this difference may cause the differing behaviour for the two variants of DICT. In other words, for MOVENODE it may be true that reallocation at deleteOneCutBelow (this site shared with MOVEFIELDS)

[1]e.g. deleteMove is more powerful because its reallocatee is accessed more often because it is higher up the tree than either insert (always at a leaf), or deleteCut (usually below a deleteMove).

is helpful, but performing the identical reallocation at the deleteTwoCutBelow site has a deleterious effect, so when merged as a single site deleteCut, it is more efficient to turn reallocation at the merged site off. We might hypothesise that as well as the sort of pointer update that occurs at a site, the distribution of where in the structure the site acts is important.

Secondly, for moveNode it may be the case that reallocation at deleteOneCutBelow is actually unhelpful when the tree is being updated by node substitution as occurs during deletion of a node with two children. We might hypothesise that the correct strategy for a site cannot be chosen without considering the form of pointer update occurring at all other sites, even if no reallocation is applied there.

In this thesis we merged the two cut sites into one site to reduce the search space, because we wished to evaluate a large number of different combinations of strategy. We have insufficient evidence to decide between these two plausible hypotheses. In practice, it seems advisable to not merge sites until it is certain that their behaviour is similar.

Conclusion: We have found that attempting reallocation at all sites is less effective than using the correct subset and that reallocation at some sites is much more effective (or damaging) than at others. Furthermore, we observe that the merged deleteCut site behaves differently for the two variants of Dict, the precise explanation for which requires further investigation. We hypothesised either that the addition of extra site(s) into a program may change the best strategies to use at the existing sites, and/or that the distribution of where in the structure a pointer update site acts is significant, not just the sort of pointer update that the site performs.

7.1.1.2 Interaction of sites

Next we will look in greater detail at the interaction of sites.

The experiment used is as described in Fig. 7.1. For moveNode, the previous section concluded that the best combination of strategies used only the insert and deleteMove sites, and so we present results obtained by varying the strategy at these two sites in combination with the null strategy at the deleteCut site. For moveFields, the strategies at the insert and deleteCut sites are varied independently. See Figs. 7.18 and 7.19.

moveNode: The choice of hints used makes a large difference to performance. The best reduction (24%) is obtained by using multiple hints, but if only one hint is used at each site no more than 18% is obtained. Moreover, if the wrong single hint is used, the reduction can be as low as 9%. When using multiple hints, we observe that using too many hints can reduce performance, even if we consider only hints gathered using low-overhead methods (i.e. $\text{parent}_{>1}$). Thus using the correct search pattern is important. Observe however that the data movement at the two sites appears to combine in a constructive fashion, and so finding the best search pattern need not be done by exhaustively trying all possibilities.

As with ccmalloc, we find that using EL is helpful (increasing reduction from 14% to 24%), but only at the insert site. Indeed, if no hints are used at this site, the use of EL alone will increase performance from 16% to 19%. This demonstrates how indirect layout improvement at one site (insert) can help direct layout improvement at another (deleteMove).

MOVENODES/MOVEFIELDS:

$\{\mathsf{insert} = \langle \alpha, \beta; \gamma \rangle \times \langle \mathsf{parent}_{\mathsf{p_1}}; \mathsf{t_1} \rangle$

$\wedge\ \mathsf{deleteCut} = \langle \alpha, \beta; \gamma \rangle \times \langle \mathsf{parent}_{\mathsf{p_2}}, \mathsf{child}_{\mathsf{c_2}}; \mathsf{t_2} \rangle$

$\wedge\ \mathsf{deleteMove} = \langle \alpha, \beta; \gamma \rangle \times \langle \mathsf{parent}_{\mathsf{p_3}}, \mathsf{child}_{\mathsf{c_3}}; \mathsf{t_3} \rangle$

$: \forall \alpha \in \{\mathsf{null}\} \cup \mathbb{SL}, \forall \beta \in \{\mathsf{null}\} \cup \mathbb{SP}, \forall \gamma \in \{\mathsf{null}\} \cup \mathbb{G}\}$

where the p_i, c_i and t_i are the best ones found by §7.1.1.1.

For MOVENODE: $\mathsf{p_1 = 1,\ p_2 = 1,\ p_3 = 0,\ c_2 = 2,\ c_3 = 0,\ t_1 = T,\ t_2 = F,\ t_3 = F}$.

For MOVEFIELDS: $\mathsf{p_1 = 1,\ p_2 = 1,\ p_3 = 0,\ c_2 = 0,\ c_3 = 0,\ t_1 = T,\ t_2 = F,\ t_3 = F}$.

Figure 7.2: Reallocation strategies used for the different sites for DICT, used for the experiment in §7.1.1.3 and the results in Figs. 7.20 and 7.21.

MOVEFIELDS: At least for the top third of the table, the general pattern is that the data movement at the two sites combines constructively rather than destructively. Observe also how the use of EL is different for the two sites: for insert, using EL is helpful (13% instead of 9%), but for deleteCut, using it is very damaging (4% instead of 13%). In other words, it is important to use EL, but it must be used correctly. We conclude that each site's strategy must be chosen individually, a per-program strategy will not do. The results also demonstrate the penalty for over eagerly using too many hints – the more hints are used, the smaller the reduction (using $p > 1$ at both sites yields 7%, compared to 13%).

Conclusion: Without use of EL only about three quarters of the optimal reduction is obtained. Using too many or too few hints or the wrong final coallocator strategy may significantly reduce performance, or not, depending on the benchmark. In the former situation, the choice of strategy must be made per-site, not per-program. The results suggest that finding the best combination of strategies for the sites doesn't require an exhaustive search; as long as coallocators are 'reasonable' the data movement at two sites combines constructively rather than destructively. However, because there are local maxima, some level of sophistication must be used when looking for the best combination of strategies.

7.1.1.3 Improving page layout

In this section we investigate whether performance can be improved by attempting to improve page layout as well as, or in place of, line layout. To do this experiment, we will use the optimal search patterns (given by p, c, t) from each site, and vary the coallocator pattern.

Formally, we use the reallocation strategies in given in Fig. 7.2. These strategies express many different approaches, for example $< \mathsf{P}; \mathsf{EP} >$ is simply the page analogue of $< \mathsf{L}; \mathsf{EL} >$. Patterns beginning with 'L, ELIP' or 'L, NLIP' attempt to improve page layout when improving line layout fails, and are dynamic forms of ccmalloc-newBlock. The full sets of dynamic forms of ccmalloc were given in Fig. 4.7.

Results for all coallocator patterns for MOVENODE and MOVEFIELDS are given in Figs. 7.20 and 7.21. A more detailed look at the ccmalloc sets can be found in Fig. 7.3.

	moveNode	moveFields
$< \mathsf{L}; \mathsf{EL} >$	24%	13%
ccmalloc-newBlock-extended	20	7
ccmalloc-newBlock-vanilla	17	4
ccmalloc-firstFit-extended	20	-8
ccmalloc-firstFit-vanilla	17	-16

Figure 7.3: How dynamic forms of ccmalloc compare against the best coallocator pattern $< \mathsf{L}; \mathsf{EL} >$, a summary of some of the results in Figs. 7.20 and 7.21. The definitions of the sets of different forms of ccmalloc are summarised in Fig. 4.7. See §7.1.1.3 for discussion of this figure.

moveNode: No attempt to involve page layout improves performance (the best obtains a 20% reduction in execution time, compared to $< \mathsf{L}; \mathsf{EL} >$'s 24%). Omitting the L stage immediately halves the improvement in layout obtained, and decreases the reduction in execution time by a factor of three. Some attempts to involve pages do not damage data layout, but do not improve it either, and because of increased overhead smaller reductions are obtained. Indeed, most choices of either β or γ can produce a good layout if (respectively) α, γ or α, β are chosen correctly, but few are useful in practice due to overhead. In other words, when using coallocators to perform runtime data movement, both layout and overhead are important, and this requires a more precisely chosen coallocator pattern.

Fig. 4.7 shows that ccmalloc-style coallocator patterns perform quite well, especially when the final coallocator is modified for dynamic use. All four ccmalloc sets have a member that produce a good layout and manage at least a 17% improvement (compared to $< \mathsf{L}; \mathsf{EL} >$'s 24%). Notice that the extended sets (those with final coallocator that select blocks based upon the number of nodes in them), despite the increased overhead, clearly achieve a better layout because they achieve at 20% reduction. We found no difference between newBlock and firstFit, nor did we find that modifying newBlock to use ELIP instead of NLIP helped.

moveFields: The results of Fig. 7.21 demonstrate that attempting to involve pages can significantly damage the layout produced. The best pattern involving pages halves the reductions in execution time obtained (13% $\rightarrow$ 7%), and most others perform much worse. Looking at Fig. 7.3 and the different forms of ccmalloc, only newBlock produces a positive reduction, with the extended form obtaining 7% and the vanilla only 4%.

Conclusion: Although statically improving page layout does help the Dict benchmark, as shown in the methodology chapter, it is hard to realise this dynamically through reallocation, and all our attempts to do so using admittedly quite simple strategies were unsuccessful. For moveNode layout isn't necessarily damaged, but overhead was higher. We conclude that combining line and page layout when doing reallocation might be possible in general, but is not easy, and in particular, where the static gains are slight (as they are with Dict), attempting to do so can be very damaging. Specifically, we have demonstrated that layout maintenance using coallocator patterns designed for good static layout (such as ccmalloc) may incur a larger overhead and generate a much poorer layout

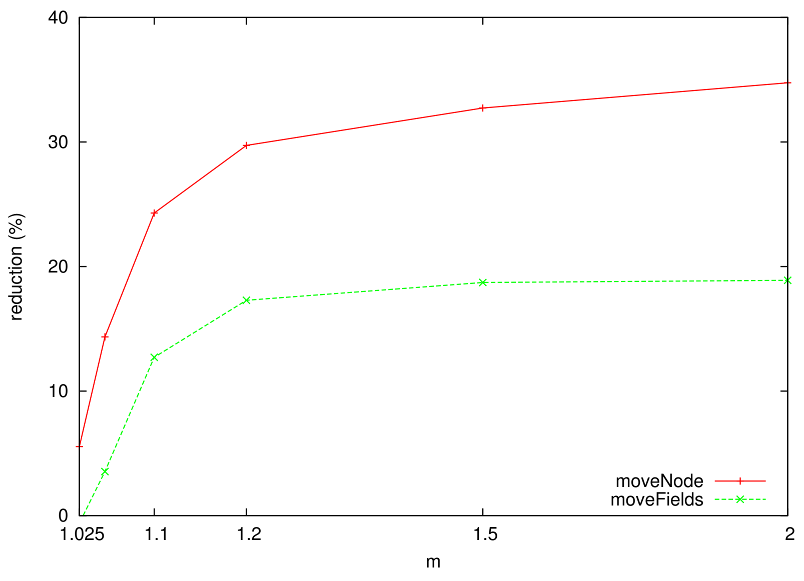


Figure 7.4: The effect of increasing m for DICT. For this graph, $M = 8/7 \times m$. See §7.1.1.4.

than coallocator patterns chosen on a per-benchmark basis.

7.1.1.4 Memory

Finally, we show the benefit of increasing memory allowance. The best strategy for $m = 2$ was found to be the same as the best at $m = 1.1$, for both benchmarks ($M = 2.28$ and $M = 1.25$, respectively). Results are in Fig. 7.4.

Increasing the memory allowance improves performance because the EL stage finds emptier lines, which means that the probability of the L stage succeeding increases, improving layout. This factor greatly outweighs the disadvantage of increasing heap size, namely increased working set size. Note that for these benchmarks, we get two thirds of the possible reduction with only 25% extra memory ($m = 1.1$, $M = 1.25$), but need 128% extra memory to get the remaining third ($m = 2.0$, $M = 2.28$). However, using 20% extra memory ($m = 1.05$, $M = 1.20$) halves the reduction. We chose $m = 1.1$ as the default memory allowance precisely because it was a good balance between low memory and useful performance.

7.1.2 MLIST

The performance of reallocation for MLIST compared to the upper bounds (§3.4.3) is shown below:

v	maximum	reallocation	
		m=1.1	m=2.0
2^7	11.7×	8.0	8.0
2^{11}	36.1	15.4	15.4
2^{15}	11.3	4.7	5.6
2^{19}	2.14	1.17	1.35

It was found that data movement at the delete site did not help to prevent data movement, either by itself or in combination with movement at the insert site. This is in contrast to the Dict benchmark, where the analogous site (deleteCut) was of some use to maintain a good layout. This behaviour may be explicable[2], because we understand the benchmark, but such observations give little insight into reallocation in general. Throughout this evaluation, only the insert site is used.

Based on its static layout properties, we expect MList to respond well to page-based reallocation, and so in the rest of this section we consider line, page and combined line+page reallocation, and then discuss how memory affects performance.

7.1.2.1 Line-focused reallocation

The *'line-focused'* coallocator patterns are the ones that don't allocate in the hint's page – apart from indirectly when allocating in the hint's line – and don't use a final coallocator that select pages based on the number of nodes in them. This is $\ll \{null\} \cup \mathbb{SL}; \{null\} \cup \mathbb{G}_{L\bar{P}} \cup \mathbb{G}_{\bar{L}\bar{P}} \gg$ (refer to §4.6 for description of the notation).

Of the line-focused coallocator patterns, the best is $< \mathsf{L}; \mathsf{EL} >$, as with Dict. Results for this pattern for different searches are in Fig. 7.22. Recall that where the subscript ≥ 1 is used, a range of different subscripts are searched, and the results from the best chosen – often, the subscript will be 1 or ∞ (For example in the $v = 2^{15}$ figure, $\mathsf{next}_{\geq 1}$'s subscript happens always to be ∞, which is why its speedups are the same as the next_∞ row. In the top figure, an intermediate value is better than either 1 or ∞, giving a speedup of 6.0×, compared to 1's 2.0× and ∞'s 4.4×).
Fig. 7.22 shows that the best search patterns depends on v (which may not be known until runtime). It also appears that the longer the list, the more sensitive performance is to getting the search pattern correct:

Short lists: For $v = 2^{15}$, a good layout may be obtained by either next_∞ or last_∞, but the former has lower overhead so produces a better speedup (4.7×, 3.9×). This is surprising. Recall that during traversal last_∞ inspects each new line to see if there is space, which incurs an instruction overhead but cannot cause a cache miss. However, next_∞ continues beyond the inserted node looking for a nonfull line, and so may cause a cache miss. In general, one would expect the former strategy to be less expensive, but the opposite is clearly true in this case. Thus, for $v = 2^{15}$ (and more generally, for RDSs made of many small structures), thinking of overhead purely in terms of line fetches may be too crude.

[2]Hypothesis: In Dict, if the reallocatee is in the same line as one of its children, it is worthwhile trying to move it to its parents line, because the branching nature of the tree means that the miss rate will be lowered. In other words, if node x has parent p and children c_1, c_2, and before reallocation node x and node c_1 share a line, a miss will be avoided when c_1 is accessed, but after successful reallocation of x, if x and node p share a line, a miss will be avoided when x is accessed, which occurs at least twice as often as accessing c_1.

For $v = 2^{19}$, the picture is similar, but we observe that first, which has very low overhead, manages to produce a good layout, and obtains the best speedup overall – because lists are only around 8 nodes long, the first node's line is likely to still be in the cache by the time the insertee is reached.

Long lists: For $v = 2^7$, the pattern last_∞ becomes the most effective, which is rather more intuitive. When lists are longer, the chance of finding a non-full line during traversal to the insertee is quite good. Using $\mathsf{next}_{\geq}1$ or next_∞ does produce a good layout, but incurs a large overhead. For $v = 2^{11}$, we observe behaviour somewhere between $v = 2^{15}$ and $v = 2^7$ – last_∞ and next_∞ are of equal value.

Conclusion: If the list length is known at compile time, either last_∞ or next_∞ can be used, and no further tuning is required. Obtaining *good* performance without knowing the list length before runtime can be done using last_∞, next_1 – for $v = 2^7$ and $v = 2^{11}$ the best speedup is obtained, for $v = 2^{15}$ it achieves 4.0 compared to 4.7, and for $v = 2^{19}$ it obtains 1.12 compared to 1.17.

To obtain the best performance without knowledge of the list length is not possible. Although the last_∞, next_k pattern is best for all list lengths[3], the subscript of the next stage depends on the list length – being 1 for $v = 2^7$ and $v = 2^{11}$, and ∞ for $v = 2^{15}$ and $v = 2^{19}$.

Thus although one can obtain good speedups with a static search pattern, obtaining the best speedup requires knowing the list lengths before execution, or varying the search pattern at runtime.

7.1.2.2 Page-focused reallocation

MList responds well to page-based static layouts, and so we now consider *'page-focused'* reallocation, using page-focused coallocator patterns. These patterns are the ones that don't allocate in the hint's line – unless by coincidence when allocating in the hint's page – and don't use a final coallocator that select lines based on the number of nodes in them. This is $\ll \{null\} \cup \mathbb{SP}_{\bar{L}}; \{null\} \cup \mathbb{G}_{\bar{L}P} \cup \mathbb{G}_{\bar{L}\bar{P}} \gg$.

Of the page-focused coallocator patterns, the best is $< \mathsf{P}; \mathsf{EP} >$, as with Dict. Results for this coallocator pattern are in Fig. 7.23.

Performance: Observe that at most two thirds of the speedup of line-focused reallocation ($< \mathsf{L}; \mathsf{EL} >$) is obtained. This is surprising because as observed in the methodology chapter, a page-clustered (**PcLm**) layout is about 1.5 times faster than a line-clustered (**Lc**) layout for MList. Indeed, for $v = 2^{19}$, speedups below 1 are seen – the overhead of page-focused runtime data movement is too high.

The poor performance of $< \mathsf{P}; \mathsf{EP} >$ compared to $< \mathsf{L}; \mathsf{EL} >$, can be explained in terms of overhead and layout quality – both are worse with page-based reallocation. Overhead is higher because to reallocate a node, the cache lines containing the page headers in the source and destination page must be touched, in addition to the cache lines that hold the source node and destination cell. For line-based reallocation the header information is in the same cache line as the source node/destination cell.

[3]Or extremely close for $v = 2^{19}$.

Layout is poorer for page-based reallocation because the layout maintained by it is only an *approximation* to **PcLm**. Using last_∞ does generate a layout roughly like dense clustering of pages, but it is apparently not as good as the layout that $< \mathsf{L}; \mathsf{EL} >$ achieves. This may be firstly because different lists share pages, reducing the cluster size, and secondly because for any given page the set of nodes inhabiting it are spread out in the list. This means that blocks run the risk of being evicted between uses, which cannot happen if page clusters are dense, as in **PcLm**.

Search patterns: Regarding search patterns, as with $< \mathsf{L}; \mathsf{EL} >$, either next_∞ or last_∞ work well for $< \mathsf{P}; \mathsf{EP} >$. Overall, less variation is seen between search patterns. Indeed, the worst search pattern for $< \mathsf{P}; \mathsf{EP} >$ produces up to twice as large a speedup as $< \mathsf{L}; \mathsf{EL} >$'s worst search pattern. Observe also that the use of last_1 next_1 (the nodes either side of the insertee) is fairly competitive for $< \mathsf{P}; \mathsf{EP} >$ compared to the best pattern, and 1.3–2.7 times better than last_1 next_1 for $< \mathsf{L}; \mathsf{EL} >$. For situations where the correct search type cannot be determined before the program is run, or it is not possible to implement longer searches, $< \mathsf{P}; \mathsf{EP} >$ may be a sensible conservative choice.

It is easier to select a reasonable search pattern for $< \mathsf{P}; \mathsf{EP} >$ because pages hold about 32 times as many nodes and so in general a page is more likely to have some space in it – shorter search distances are often appropriate. We also observe that the first strategy is more useful for $< \mathsf{P}; \mathsf{EP} >$ than for $< \mathsf{L}; \mathsf{EL} >$, because each list uses fewer pages and so the first nonfull page in the list is more likely to still be in the cache when the insertee is accessed (although other factors are involved such as the size and associativity of the TLB and L2 caches).

Conclusion: Page-focused reallocation underperforms despite the superiority of page-based layouts statically. This is because of overhead and because the layout achieved by reallocation is only approximate. However, when search length is limited (for example, in a hypothetical situation where the list is randomly accessed during updates, but linearly accessed during traversals), or when list length is unpredictable at runtime, pages appear to be a sensible conservative choice. The exception is for very short lists ($v = 2^{19}$), where speedups < 1 are obtained.

7.1.2.3 ccmalloc and combined line+page reallocation

We have found that line-focused reallocation outperforms page-focused reallocation, and now we investigate other coallocator patterns that attempt to focus on both lines and pages. We are particularly interested in the performance of the ccmalloc sets that were given in Fig. 4.7, but will use all coallocator patterns in $\ll \{null\} \cup \mathbb{SL}; \{null\} \cup \mathbb{SP}; \{null\} \cup \mathbb{G} \gg$, as for DICT.

The experiment is described formally in Fig. 7.5. Ideally, for each coallocator pattern we would like to perform an exhaustive search over all search patterns, to find the best possible performance of the coallocator pattern. This would take far too long, and so as a compromise we will use only four search patterns for each coallocator pattern. We use either a 'short' search pattern or a 'long' search pattern for pages and lines (independently). The short search pattern is last_1, next_1 – the neighbours of the inserted node. The long pattern is whatever was found to be optimal for the relevant block, from the previous sections' experiments.

$$
\begin{array}{l}
\forall\ \mathsf{pageLen} \in \{\mathsf{short}, \mathsf{long}\},\ \forall\ \mathsf{lineLen} \in \{\mathsf{short}, \mathsf{long}\} \\
\quad \forall\alpha \in \mathbb{SL}, \quad \forall\beta \in \mathbb{SP}, \quad \forall\gamma \in \mathbb{G} \\
\quad\quad \mathsf{insert} = \langle\alpha;\beta;\gamma\rangle \times \langle \mathsf{search(v, line, lineLen); search(v, page, pageLen); true}\rangle
\end{array}
$$

where:

$$
\begin{array}{l}
\mathsf{search}(*, *, \mathsf{short}) = \mathsf{last}_1,\ \mathsf{next}_1 \\
\mathsf{search}(2^7, \mathsf{line}, \mathsf{long}) = \mathsf{last}_\infty,\ \mathsf{next}_1 \\
\mathsf{search}(2^7, \mathsf{page}, \mathsf{long}) = \mathsf{last}_\infty \\
\mathsf{search}(2^{15}, \mathsf{line}, \mathsf{long}) = \mathsf{last}_1,\ \mathsf{next}_\infty \\
\mathsf{search}(2^{15}, \mathsf{page}, \mathsf{long}) = \mathsf{first},\ \mathsf{next}_1
\end{array}
$$

Figure 7.5: Reallocation strategies used for MList, used for the experiment in §7.1.2.3 and the results in Fig. 7.24.

Fig. 7.24 gives detailed results – for each coallocator pattern $\langle\alpha, \beta; \gamma\rangle$ the best speedup from the four search patterns is given. Performance appears to be determined almost entirely by layout, rather than overhead. The results show that combining lines and pages does not work – the best coallocator pattern is either $< \mathsf{L}; \mathsf{EL} >$ or $< \mathsf{P}; \mathsf{EP} >$, the latter is better when only short searches are allowed.

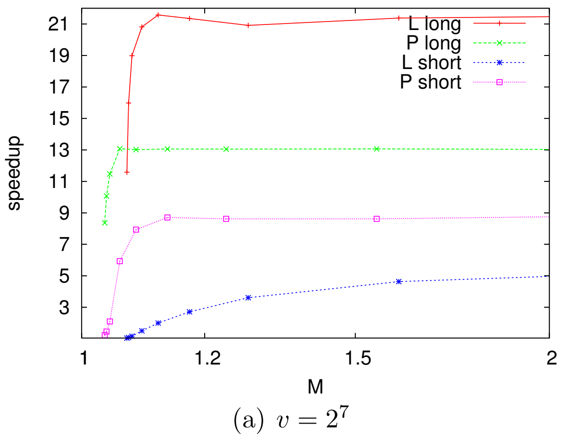
Fig. 7.25 summarises the performance of the ccmalloc sets, $< \mathsf{L}; \mathsf{EL} >$ and $< \mathsf{P}; \mathsf{EP} >$ for the four different search patterns. For the ccmalloc figures, the bracketed figure gives the speedup when the second stage of the coalloactor pattern is as originally described by Chilimbi et al. [15] – NLIP for ccmalloc-newBlock and FLIP for ccmalloc-firstFit. The unbracketed figures are the best when we also allow ELIP for ccmalloc-newBlock and P for ccmalloc-firstFit. The bracketed figure is omitted if it is the same as the unbracketed.

<u>Performance:</u> Observe from Fig. 7.25 that it is possible to almost equal $< \mathsf{L}; \mathsf{EL} >$'s performance using the extended versions of ccmalloc. When the second stage (coallocation in the same page) has a smaller chance of succeeding – i.e. when short searches are used for second stage – performance improves (For example, for $v = 2^7$, ccmalloc-firstFit obtains 7.6 with a long line search and short page, but 6.2 with both page and line long). This indicates that the second stage is unhelpful, although not as damaging for MList as for Dict-moveFields, for example. In general, Fig. 7.24 shows that for long lists anything coallocator pattern from the set

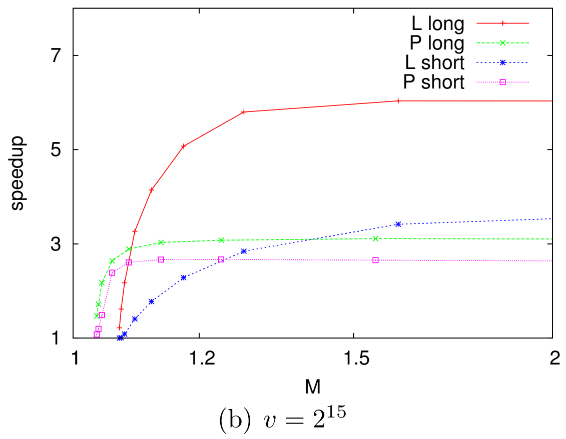
$$\{\langle \mathsf{L}; \alpha; \beta\rangle : \forall\alpha \in \{\mathsf{null}, \mathsf{P}, \mathsf{FLIP}, \mathsf{ELIP}, \mathsf{NLIP}\}, \forall\beta \in \{\mathsf{EL}, \mathsf{ELIEP}, \mathsf{FLIEP}\}\}$$

will perform pretty well, *provided* the page search length is correct.

<u>Conclusion:</u> Combining lines and pages is not more effective than either a line-focused or page-focused approach. The line-focused approach is better when the number of hints is large, but the page-focused approach is better when the number of hints is small. Although a good static coallocator like ccmalloc can be used for this benchmark (provided the tertiary stage is chosen properly) the stage that attempts to allocate in the same page is unhelpful, and the simpler $< \mathsf{L}; \mathsf{EL} >$ is more effective.



(a) $v = 2^7$

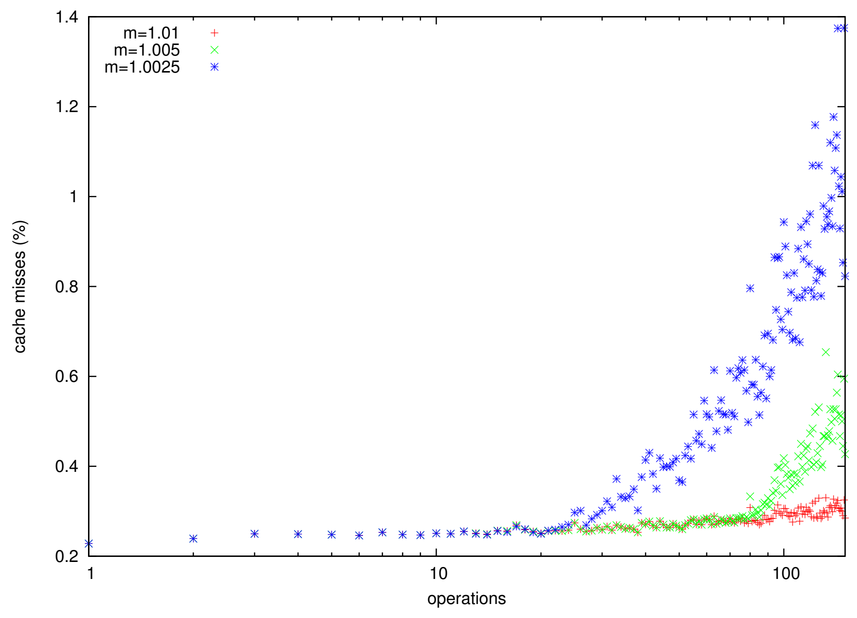


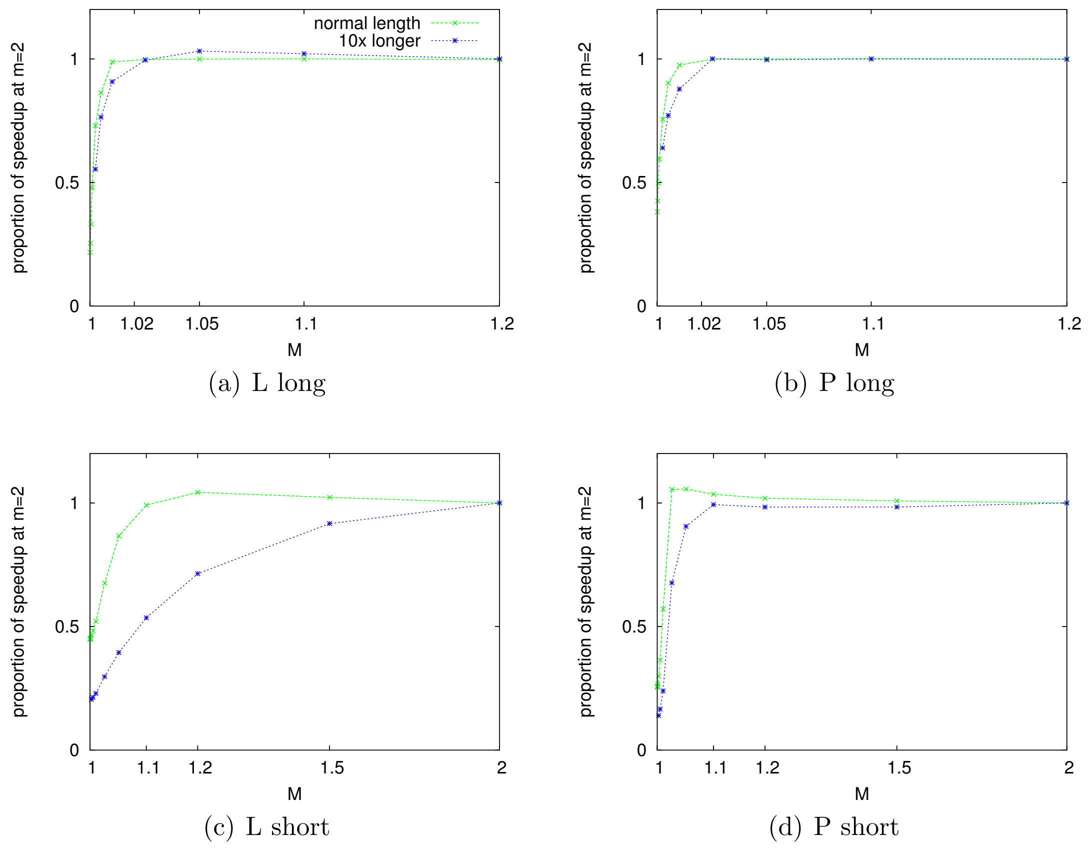
(b) $v = 2^{15}$

Figure 7.6: The effect of increasing M for MList. See §7.1.2.4.

7.1.2.4 Memory

A review of how blocks are used as an experiment progress: Initially, the structure is densely packed into a set of full blocks, and there is another set of empty blocks, a factor $m - 1$ larger than the set of full blocks. As the experiment progresses, it passes through several phases. In the initial phase, reallocation's EL or EP stage always finds an empty block. Two runs of the benchmark with different m have identical behaviour when they are both within this phase (apart from any small effect of working set size). As phase one progresses, the stock of empty blocks may decrease if m is small enough. We restrict our discussion to this situation because it is the situation observed in the experiments in this section. Phase two begins the first time EL or EP does not find an empty block, and the time of its onset depends on m:





(a) L long (b) P long

(c) L short (d) P short

Figure 7.7: The combined effect of M and experiment run length on the performance of reallocation for MList, $v = 2^7$. For each line, the vertical axis shows the speedup as a proportion of the speedup obtained for $m = 2$ (The $10\times$ longer experiments achieve speedups at least twice as large as the normal length).

During phase two the heap readjusts until an equilibrium is reached[4]. During phase 3, the distribution of blocks in the heap is stable, and hence layout quality and performance of the benchmark is stable.

As with Dict, we found that the best strategy was the same for $m = 2$ and $m = 1.1$. However, there is some evidence that the relative efficacy of strategies depends on memory (i.e. the availability of fairly empty blocks). In Fig. 7.6, we shown speedup against M for $m \in [1.0025, 2]$, for $v = 2^7$ and $v = 2^{15}$. The lengths of the experiments (measured in inserts and deletes) are identical, to allow direct comparison between the two list lengths. $v = 2^7$ has been run for ten times longer than the usual experiment length and has not reached equilibrium (extrapolation suggests that to do so is infeasible), $v = 2^{15}$ for 4 times shorter than usual (and has reached equilibrium). We give results for $< \mathsf{L}; \mathsf{EL} >$ and $< \mathsf{P}; \mathsf{EP} >$, for both short and long searches, as in the previous section. Note that line-focused reallocation and page-focused have different M/m because they have different memory manager space requirements.

The results show that, as expected, increasing m improves performance. It is encouraging that only around $M = 1.1$ is needed for $v = 2^7$. Note that the effect of increasing m is different for short and long searches and for line- and page-based strategies - for $v = 2^7$, only the combination of lines and a short search length benefits from $M > 1.1$. The behaviour is also different for the two values of v - for $v = 2^{15}$, both short and long line searches benefits from $M > 1.1$. Note also for $v = 2^{15}$ that if only short searches are allowed, the best strategy is page-focused for $M < 1.25$, but line-focused for $M > 1.25$. Thus in some situations, the best strategy depends on memory usage.

The effect of m depends also on experiment length, as shown in Fig. 7.7 for $v = 2^7$ (similar behaviour is seen for other values of v before equilibrium is reached). Observe that more memory is required for a certain speedup when experiments are run for ten times longer (about 100 minutes, unoptimised). Although it is not feasible to run $v = 2^7$ or $v = 2^{11}$ to equilibrium, the behaviour for $v = 2^{15}$ and $v = 2^{19}$ suggests that, although memory requirements increase as experiment length increases, the best speedups seen for shorter runs are still seen at equilibrium for reasonable values of m (< 2). In other words, although more memory is required for longer runs of the benchmark, long term performance is *not* damaged, provided m is large enough (and not more than 2)[5].

In practice, the terminal behaviour may not be relevant: Fig. 7.7 demonstrates that even small amounts of extra memory ($m = 1.05$) are suitable for runs of around 100 minutes for $v = 2^7$.

Conclusion: For MList we have demonstrated that increasing m improves performance, as seen with Dict. The effect of increasing m depends on the block size, search length, experiment length, and v. Larger blocks are more resilient to low memory, but provide

[4]The existence of which is confirmed for $v = 2^{19}$ and $v = 2^{15}$ and thus likely but not confirmed for $v = 2^{11}$ and $v = 2^7$ – recall that the time to reach equilibrium may be different in the unoptimised and optimised cases.

[5]Note that we can be certain that there exists a maximum value of m beyond which no further performance improvement is seen, irrespective of experiment length (ignoring for the moment the effect of increasing working set size, i.e. we assume the caches retain no blocks between individual list traversals). This maximum value is no more than the number of cells that fit in a block, because when m exceeds this value, there will always be an empty block available. The unanswered questions are firstly whether the maximum value is always a reasonable value (since 14 nodes fit into lines, and 502 into pages), and secondly failing that, how large m must be to get 'reasonable' speedups.

less speedup (representing a conservative choice). In practice, for a certain run time of the program (expressed in minutes not operations), smaller values of v (2^7) require far lower m than larger v (2^{15}) (1.05 compared to 1.5). The effect of memory at equilibrium for $v = 2^7$ is not known, but experiments with larger v suggest that good speedups will seen for reasonable m (< 2).

7.1.3 Conclusions

Although preexisting static coallocators such as ccmalloc [15] can be used to prevent layout degradation, it is more effective to choose coallocators on a per-site basis. For both our benchmarks, we found that the simple strategy of allocating in the same line as the hint, and failing that a line of maximal space in the heap ($<$ L; EL $>$) was the most effective. Obtaining the best performance from reallocation can achieved by concentrating on the factors below:

1. **Selection of (re)allocation sites**: There will typically be several sites in the program that update pointers within the RDS. For Dict, we found that although reallocation at any one site can be used to improve layout, better performance comes from using multiple sites, and use of all sites was sometimes less effective than a subset. For MList, we found that of the two sites available, only one was useful to improve data layout. Thus in general, finding the correct subset of sites at which to perform reallocation is necessary. We may speculate that certain sites are more suitable for reallocation than others. Sites that substitute one node for another or attach a node to a null pointer appear to be good candidates for reallocation, but cuts (e.g. deleteCut for Dict or delete for MList) are of less, but not necessarily zero, usefulness.

2. **Choosing hint object(s)**: Applied simply, reallocation can maintain a reasonable data layout using just the nodes closest to the (re)allocatee (its new parent and new child/children). For both benchmarks, we found that it is often beneficial to use more than one hint object. Hint addresses must be explicitly gathered, incurring overhead, and thus for each reallocation site there will be optimal number of hints to use. For MList, using more than the neighbouring nodes can increase speedup by a factor of 7, and we find that hints accessed before the reallocatee are better than those after. A hint can be found with relatively little overhead during traversal, simply by inspecting each new block for empty space. Using hints accessed after the reallocatee involves additional traversal of the list, which tends not to incur too much overhead because space is often found quickly, but produces a less effective layout. The best performance was obtained by using different hints for different list lengths, but quite good performance can be obtained using the same hints for all list lengths. For Dict, we find that usually only the nodes in the immediate vicinity of the pointer update are needed as hints. However, reallocation at substitutions works better when the reallocatee's grandchildren are also used as hints, and in the absence of reallocation at other sites, reallocation at insertion can be improved using a larger number of the reallocatee's ancestors. Thus both benchmarks can benefit from using more than one hint object, but using too many hints will incur a penalty.

3. **Block size**: For both benchmarks, we find that a purely line-focused reallocation strategy is almost always the most effective. While not surprising for Dict, given its

static layout behaviour, we might have expected MList to respond better to a page-based or combined line and page strategy. For the latter benchmark, we find that given the optimal choice of hints, pages provide only half the speedup of lines. When only the neighbouring nodes are used as hints, pages are more effective at low memory, but lines are better when memory is high. Regarding pre-existing static coallocators, we conclude that the dynamic analogues of ccmalloc often perform almost as well as a simple line-based approach *provided* the third stage (allocation when same-line and same-page fails) is chosen properly. For Dict-moveFields however, only half the reduction of line-based reallocation is obtained. Thus in general, we conclude that line-based reallocation is easier, safer and more effective than either page or line+page reallocation.

4. **Memory allowance:** Increasing the value of m means that the EL stage of reallocation will find emptier blocks, which improves the chance of the L stage succeeding, improving layout. For Dict, we find that most improvement is obtained using only $m = 1.2$, and m may be reduced to 1.05 with a loss of only half this reduction. For MList, we find that long lists are very resilient to low m even for runs of 100 minutes (unoptimised), requiring only $m = 1.05$ to give the full benefit of reallocation. For shorter lists, the speedup obtained at equilibrium increases by a factor of two as m is increased, with most being obtained by $m = 1.2$. Overall, reallocation works well for $M \in [1.23, 1.37]$ ($m = 1.2$), and still gives some improvement for $M \in [1.08, 1.20]$ ($m = 1.05$).

7.1.4 Summary for programmers

To apply reallocation, a memory manager must be found or written that supports allocation in the same line/page as a node and finding of a block of maximal space in the heap. This can be achieved simply and efficiently using a few linked lists and bitfields, when RDS objects are the same size (or padded or split to be the same size). The program must then be modified to use the memory manager for the RDS nodes.

Reallocation at a single site is likely to improve layout, but using multiple sites is much more effective. However, performing data movement at all sites may be less effective than at a subset of sites. As a starting point, it appears that sites where a node is attached to a null pointer or substituted for another nodes will respond well to reallocation. Sites where a node is cut from the structure (i.e. a node with single child and single parent is removed by updating the parent's pointer to point to the child) are not always useful. See §7.5 for a discussion of how the correct subset of sites to use might be found automatically.

Regarding coallocator patterns, the experiments in this thesis suggest that $<$ L; EL $>$ is the most effective (move to same line as hint, failing that move to a line of maximal space), even for structures whose static layout depends also on a good page layout as well as a good line layout. This conclusion may not hold for all architectures or memory hierarchies, and so $<$ P; EP $>$ should be tried as well.

Although using a single hint produces non-negligible improvements in data layout, both structures benefit from more hints. Trying a second hint at a site should require little extra effort, and if successful code to collect more hints can be inserted. In particular, for linear traversals, a good search type appears to be to inspect the line of each node seen during the traversal to the place in the structure where the pointer update occurs, and then use the most recent as a hint. For branching traversals, the parent node is a more useful hint than the child node(s), but using both parents and children (in that order) is most effective.

Increasing memory usage improves performance. To minimise memory manager overhead and maximise layout quality, allow the heap to be unbounded (i.e. fix the EL stage to always claim/reuse an empty line).

7.2 Bulk data movement

A summary of optimisations and techniques can be found in Fig. 5.3.

Where the parameters of an optimisation had to be adjusted to find the best performance, this was done so execution times were stable where possible – i.e. not noticeably increasing at the end of the experiment. In some situations it was possible to get better performance *in the short term* by using lower data movement rates, but this should be considered against the spirit of the work in this thesis.

7.2.1 DICT

A summary of results for low, medium and high memory usage can be found in Fig. 7.26. Note that simply periodically moving the entire RDS to a new space clustering nodes into lines (periodic2space) will achieve 41% and 22% reduction in execution time for MOVENODE and MOVEFIELDS, respectively, compared to the maximum possible reductions of 55% and 36%. The version of the optimisation that also clusters nodes into pages (periodic2space-nested) is about 2% worse, despite producing a better layout. This is due to the increased instruction overhead. As was shown in Fig. 5.1, because they are periodic both optimisation cause large pauses to normal program work at small scales (i.e. $1ms$), as well as using $m = 2.0$.

In terms of execution time, these two simple methods perform better than any of the other bulk data movement optimisations. In the rest of this section we will show that the more complicated optimisations can obtain reductions that are almost as large, but using far less memory and without large pauses to normal program work.

7.2.1.1 Reducing memory

The behaviour of periodic1space when m is reduced can be seen in Fig. 7.8. These results are for the best combination of compaction and threshold (we will talk in more detail about these two techniques in a later section).

The value of 0% for MOVEFIELDS for $m = 1.1$ and $m = 1.05$ are explained as follows: Recall that the periodic1space optimisation must be tuned for the optimal period, and that movement of nodes only takes place if there is a line with space for two nodes in it (since there is no point moving a node if the movement will not colocate two related nodes). The figures of zero arise because very little data movement is possible: this means that layout is scarcely improved, and the tuning therefore uses a very long period, reducing overhead to the point where the optimisation has no overall effect on execution times.

Note that at $m = 2$, periodic1space obtains smaller reductions than periodic2space (16 percentage points for MOVENODE, 10 for MOVEFIELDS). There are two reasons for this. Firstly, although the layout produced is identical in terms of the clustering of nodes into

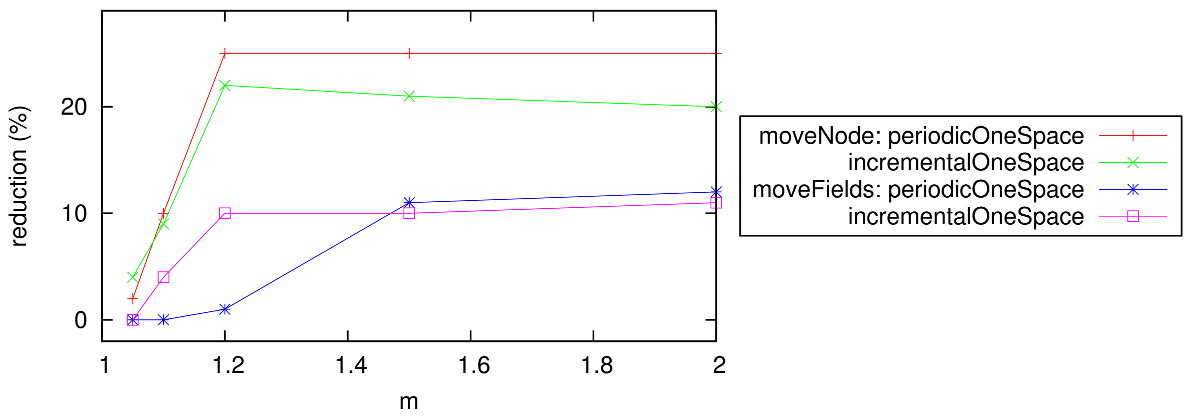


Figure 7.8: Reduction in execution time achieved by periodic1space and incremental1space for DICT-MOVENODE and DICT-MOVEFIELDS, for different m.

lines, the location of these lines in the heap is not. The traversal order of the tree in the periodic2space combined with the linear filling of to-space means that fairly reasonable page layout is produced. By contrast, the lines used by the periodic1space optimisation, although all empty (because m is sufficiently high), are drawn from a stack of empty lines, which in practice produces TLB miss rates similar to picking lines at random from the heap. In addition, the lines are densely packed into pages when two semi-spaces are used, unlike when a single space of size $m = 2$ is used, reducing the number of pages in use between swaps of the semi-spaces. Secondly, overhead is higher when one space is in use: periodic1space must maintain more complicated information about the space at runtime than periodic2space – in particular, whenever a node is allocated or deallocated, a line must move from one memory manager listset list to another. Some of this overhead might be reduced by more sophisticated memory manager design, but we do not consider this further here.

Once m is decreased below a certain value, the performance of the optimisation begins to decrease, eventually hitting zero. This value is between 1.1 and 1.2 for MOVENODE, and higher for MOVEFIELDS (between 1.2 and 1.5). This difference is possibly because the layout degradation that has occurred in the MOVEFIELDS variant after $1e8$ operations is less than the MOVENODE variant, which means that – because the overhead of data movement is the same in both variants – a much better layout must be achieved by MOVEFIELDS to get the same reduction, which requires higher quality lines, which requires larger m.

The resilience that periodic1space demonstrates to two-fold (or even five-fold) reductions in extra memory allowance is encouraging. In the next section we tackle the problems of latency by both incrementalisation and embedding, and also evaluate embedding as a method of reducing overhead.

7.2.1.2 Reducing latency and overhead

Incrementalising periodic2space to produce incremental2space incurs a penalty of 8–9 percentage points – MOVENODE's reduction goes from 41% to 32%, MOVEFIELDS's goes from 22% to 13%. There are several possible reasons for this. Splitting the traversal into about 500 separate chunks of work increases overhead – simply to resume the traversal each time. The write barrier incurs an overhead, and is likely to make the optimisation performs more traversing work than in the periodic version. Finally, it is possible that splitting the traversal into smaller chunks means that it incurs more cache misses because the mutator evicts nodes that the traversal will later reuse.

Incrementalising periodic1space: Incrementalising periodic1space appears to improve performance for lower values of m, particularly for MOVEFIELDS. Fig. 7.8 shows the reductions achieved by incremental1space and periodic1space for different m.

At $m = 2$ we incur a small overhead of 1–5 percentage points, as was seen previously, and we expect that some of the overheads have the same cause. The overhead is less than with incremental2space possibly because incremental1space does not use a write barrier. Again, it is hard and perhaps not necessary to understand the precise causes of overhead, but it is encouraging that the overhead for one-space incrementalisation is far lower than for two-space.

As m is lowered, the incremental version begins to outperform the periodic version, which can be seen most clearly for $m = 1.2$ for MOVEFIELDS. This improvement can be explained in terms of the availability of better quality lines, specifically because incrementalisation allows compaction to work better: when compaction is turned off, the incremental version performs worse than the periodic version. The compaction schemes we employ in this thesis work better when the movement of the tree is split into smaller chunks, because compaction work (block producing) is interleaved with re-laying out (block consuming), making it harder for the optimisation to exhaust the supply of good quality blocks. We discuss this in more detail in the next section. The increased availability of good quality blocks clearly outweighs the overhead of incrementalisation. Thus, we should view incrementalisation not merely as a method to fix an optimisation's high latency, but as method that combines well with compaction to enable more effective data movement.

Embedding periodic1space: Embedding periodic1space to produce embedded1space works remarkably well, as shown in Fig. 7.9.

When m is high, embedding either provides a method of reducing latency without performance loss, or actually improves performance. This was unexpected – although embedding reduces the number of cache misses incurred by the optimisation, it has the disadvantage of significantly increasing the complexity of the code within the traversal loop. Thus when m is high, we should view embedding not just as a method to reduce latency but as a method to increasing performance.

When m is low, far less performance is lost than with periodic1space or incremental1space. For $m = 1.1$, applying the periodic1space optimisation to MOVENODE produces a layout of 1.11–1.38μs per lookup (the lower is after movement, the higher just before), on average 1.24μs. Applying incremental1space produces a layout of 1.38μs, and embedded1space obtains 1.29μs. Thus embedded data movement has worse layout than stop-the-world data movement but incurs less overhead, and so is faster overall. Compared to incrementalised data movement, embedded both produces a better layout (because of more finely-interleaved compaction), and has lower overhead.

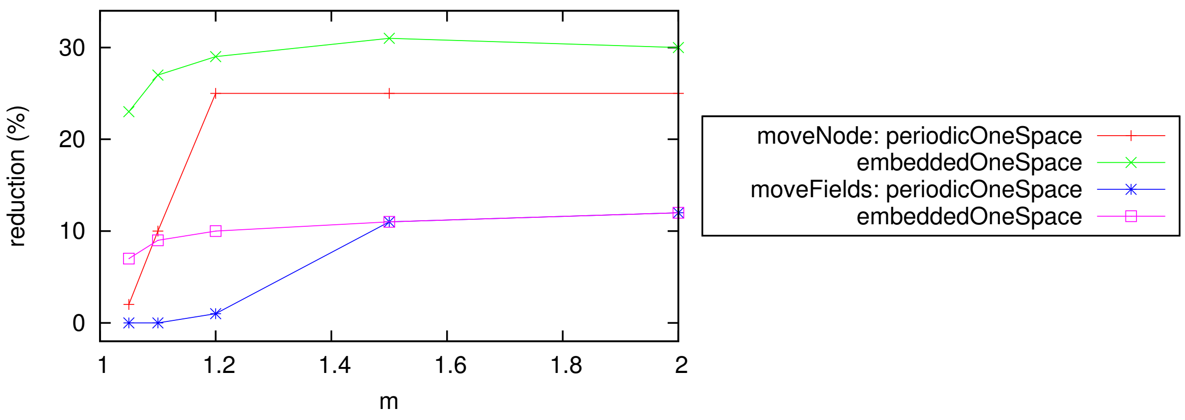


Figure 7.9: Reduction in execution time achieved by periodic1space and embedded1space for DICT-MOVENODE and DICT-MOVEFIELDS, for different m.

7.2.1.3 The role of threshold and compaction

In low memory conditions, embedded1space and incremental1space often perform far better than periodic1space because a combination of compaction and threshold allow them to create a far better layout. In this section we examine in more detail the role of compaction and threshold for these three optimisations.

periodic1space: Compaction gives a small increase in the quality of layout (5–10%) produced by the movement of the tree. However, due to overhead, performance decreases when compaction is used. Although the supply of empty lines has improved, this benefit does not outweigh the cost of compaction.

Regarding the two different types of compaction – compaction applied at reallocation sites, and compaction applied within the optimisation – recall that the optimisation will not move data unless there is a line with at least two empty cells in it (because it is not worthwhile moving nodes unless two or more nodes can be located in the same line). When compaction at reallocation opportunities is performed, the number of suitable lines available at the start of the RDS movement is increased, but runs out before the movement has finished, and thus parts of the RDS do not have their layout improved. If compaction is performed within the optimisation (during the movement of the RDS), again, some parts of the RDS do not have their layout improved because some nodes have been moved simply to compact lines rather than improve layout.

Thus neither of these simple methods of compaction is wholly successful for periodic1space, and a more sophisticated compaction scheme would help. We do not pursue this here because we do not expect even optimal zero-overhead compaction to enable periodic1space to significantly outperform embedded1space: observe that when empty lines are plentiful, periodic1space only obtains a 25% reduction, which is therefore an upper bound on the performance of this optimisation, scarcely larger than embedded1space's reduction of 23% for $m = 1.05$.

incremental1space: Compaction improves execution times slightly for MOVENODE when $m = 1.1$ (9% vs 6%). In terms of the quality of layout, for MOVENODE the layout is consistently 15% better with compaction, and for MOVEFIELDS consistently 5% better.

As with periodic1space, the overhead of compaction prevents most of this improvement translating into reduction in execution time.

embedded1space: For $m = 1.1$, for moveNode a 23% better layout is produced, and for moveFields, 8%. This translates into significant improvements in performance. The tables below demonstrate how compaction is most effective when combined with a threshold quality on the lines used: (i.e. any bad lines are rejected until a good one is produced by compaction)

		threshold	
		off	on
comp.	off	14%	0
	on	20	27

moveNode

		threshold	
		off	on
comp.	off	0%	0
	on	0	7

moveFields

In general, using compaction by itself makes sense because each node movement performed by compaction increases the probability of a good line being available. Using threshold by itself does not make sense, unless there is some reliable mechanism (i.e. node allocation and freeing in the main program) which can be relied upon to produce emptier lines (this is not the case for Dict, which does not have any allocation in the time part of the benchmark). Threshold and compaction work most effectively in combination.

The correct choice of threshold depends on the benchmark – for moveNode, it is best to use lines with at most 1 node in them, but for moveFields it is best to allow only empty lines. This is because the former benchmark has a higher rate of degradation than the latter, and so the compromise between overhead (cost of compaction to make emptier lines) and layout quality (threshold) shifts accordingly. When $m = 1.5$, a similar situation to the above is found, except that both benchmarks work best with empty lines only – this time because such lines are easier to produce because the heap is less dense.

It is usually more effective to perform compaction within the optimisation than at pointer update sites (e.g. for moveNode, 28% compared to 23%). The exception is for moveFields when $m = 2.0$ (3% compared to 12%). Thus both forms of compaction are useful in different situations, but compaction within the optimisation is usually better, which simplifies the application of the optimisation.

Conclusion: Threshold and compaction are vital to get the best performance out of embedded1space, allowing m to be reduced from 2.0 to 1.10 with reduction only falling from 30% to 27%, for moveNode. These two techniques are of some usefulness for incremental1space in terms of execution time, but ineffective for periodic1space. Furthermore, it appears unlikely that more sophisticated compaction would allow the incremental or periodic form to significantly outperform the embedded version, such are the benefits of embedded's reduced data access costs and fine interleaving with normal program work.

7.2.2 MList

The results for all optimisations are given in Fig. 7.27.

As with Dict, we are unlikely to improve upon periodic2space in terms of execution time, but there is definitely room for improvement in terms of memory usage, and latency.

Regarding latency, each movement of the RDS costs between 35 and 70ms, so even at the 10ms scale there is at least a 3.5× slow down, and even at 100ms there is a 1.35 to 1.7× slow down. At the 1s scale, no significant slowdown is seen.

7.2.2.1 Reducing memory

As shown in Fig. 7.27, using $m > 1.05$ can increase speedup. The trend is that increasing m is more valuable for larger v.

This is reflected more clearly by considering the layout achieved by an optimisation. We use the incremental1space optimisation. The figures below show the average layout quality of a list after it is re-laid out. Quality is expressed in the number of distinct blocks occupied by the list (Thus an optimal layout for the list, one that uses the fewest number of blocks, is max(1, length of list / maximum number of nodes per block)). The last column gives a ratio, expressing the relative quality of the two layouts.

v	$m = 1.05$	$m = 2$	$(m = 1.05)/(m = 2)$
2^7	65.906	65.758	1.0023
2^{11}	4.776	4.616	1.034
2^{15}	1.585	0.999	1.586
2^{19}	3.059	1.127	2.714

Observe that for $v = 2^7$ and 2^{11}, some improvement is seen for $m > 1.05$, although not significant enough to be seen as an increase in speedup. For $v = 2^{19}$ the $m = 2$ layout is almost three times better than the $m = 1.05$ layout (but still not optimal, implying that using $m > 2$ might produce a larger speedup).

Practically, we can say that for lists longer than around 2048 nodes, $m = 1.05$ is good enough. For lists shorter than this, larger values of m are needed, and $m > 2$ may be required to get the best speedup for lists of only eight nodes.

The increasing worth of using $m > 1.05$ as v increases has three causes. Firstly, the experiments are different lengths (in terms of insertions/deletions, but about the same wall clock time unoptimised). This effect is easily removed by running all experiments to $v = 2^7$'s length. The second is as follows. For smaller v, we note that the experimentally-found optimal list re-laying out frequency is lower (measured in node insertion/deletions). In other words the number of insertions/deletions between each list being re-laid out is larger for larger v. When a list is moved, the movement of the list into new blocks creates a supply of fairly empty blocks, which can be either used to re-layout later lists (or even the current list). If the list was laid out optimally before it was re-laid out, all the new blocks are empty. If the list is laid out with a pathological layout (number of blocks = length of list), all the new blocks have just a single space. Thus, the quality of the blocks decreases as the layout quality of the list decreases, in other words as the list relaying out frequency decreases.

Thus, in the figures above, larger v induces a slower re-laying out rate, which decreases the quality of blocks produced when a list is moved, which makes it harder to achieve a good layout. Increasing m thus reduces node density and makes better quality blocks available.

We can remove this effect by moving all lists at the same rate (we use $v = 2^7$'s, because it is fastest). These figures are seen:

v	$m = 1.05$	$m = 2$	$(m = 1.05)/(m = 2)$
2^7	65.906	65.758	1.0023
2^{11}	4.874	4.759	1.024
2^{15}	1.352	1.000	1.352
2^{19}	1.546	1.027	1.505

The difference between $m = 1.05$ and $m = 2$ is now smaller. Since nodes are moved at the same rate, the third and final cause is a structural one - how the 2^{22} nodes in the heap are split into distinct lists. For example, $v = 2^7$ requires at least 66 pages per list on average, whereas $v = 2^{15}$ only requires a quarter of a page. These partially used pages may explain the different ratio. Similarly, $v = 2^{11}$ requires five pages, one of which is only 8% used. For $v = 2^{19}$, recall that lines are used in place of pages, which may partially explain the large ratio, and in addition each list uses only 57% of a line on average.

Thus, although it is unclear why different average list lengths respond differently to increasing m, it is not surprising, since they make use of different numbers of blocks or different block sizes. Much of the discrepancy in practice is due to data movement rate, rather than the effect of list size, however. In practice we note that $m = 2$ works well, and often less memory is required.

7.2.2.2 Reducing latency and overhead

Here we discuss how incrementalisation and embedding perform for MList.

Incrementalising periodic2space usually incurs very little overhead – in contrast to when Dict's periodic2space was incrementalised. This is because MList's RDS is composed of distinct lists which are small enough that they can be processed in their entirety without significant pause to normal program work, avoiding the need for an expensive write barrier.

Similarly, incrementalising periodic1space incurs little overhead (and in one occasion improves performance slightly), in contrast to the equivalent process for Dict. For Dict, we attributed the overhead to a mixture of (i) the instruction overhead of resuming the traversal many times, (ii) increased cache misses due to the mutator and the optimisations traversals being interleaved, and (iii) the optimisation missing parts of the structure because the mutator had updated pointers. Recall that no write barrier is need for incremental1space's correctness, so we omitted it for increased efficiency and ease of implementation. Of these causes, only the first is applicable to MList – the latter two do not apply because each substructure is handled in its entirety before allowing the mutator to continue – which goes some way to explain why incrementalisation occurs little overhead.

In general we conclude that if an RDS can be decomposed into distinct parts with no (or perhaps few) connecting pointers, much of the overhead and implementation difficulty of incrementalisation may be removed.

In contrast to this positive result, embedding data movement in existing program loops is not very effective for MList. Performance is reduced, but it is not entirely clear why. The number of blocks transfered is demonstrably lower with embedding than with incrementalisation, and so the decrease in performance is probably due to instruction overhead or some superscalar effect. For execution time to be significantly affected by instruction overhead or a superscalar effect, the cost of data accesses must be small. This appears to hold for MList. Experiments show that the layout maintained by the bulk optimisation is actually very good, achieving L1/L2/TLB miss rates of only 7% *ignoring* the effect of the hardware prefetcher (which will work quite well because bulk moves nodes into a page in order). Thus it is at least plausible that some instruction or superscalar effect is significant compared to the cost of fetching data and is causing the reduction in performance.

7.2.2.3 Regarding block size

With very few exceptions, it is most efficient to use blocks the same size as pages, rather than lines or an intermediate size. Using pages instead of lines gives around three times the speedup for incremental1space, and four times for periodic1space. For the short lists we occasionally observe lines being more effective – for example, for $v = 2^{15}$, $m = 1.05$, lines give a 1.2 times larger speedup, but for large m pages become more effective.

This is in contrast to reallocation where lines were usually more effective than pages, and can be easily explained in terms of the overhead of moving nodes one-by-one (reallocation) or en masse (bulk data movement). Specifically, a node movement must access the destination block's header, which may be in another line when the block size is larger than a line. Bulk minimises the number of misses caused by this access by moving many nodes at once into the destination block.

7.2.3 Conclusions

Periodically copying the entire RDS to a new space (periodic2space) is by far the most effective method of bulk data movement, and is usually quite close to the maximum possible improvement – for Dict, we obtained 41% and 22% reduction for moveNode and moveFields, respectively, compared to maxima of 55% and 36%. For MList we obtain 10.6× and 1.60×, for $v = 2^7$ and 2^{19} respectively, with maxima 11.7× and 2.14×. However, these large improvements are only achievable using $m = 2$, and a large latency. These problems can be tackled by the following techniques:

1. Use of one space: Memory requirements can be reduced by using one space of size > 1 instead of two of size 1, and moving nodes not to empty blocks, but to a block of maximal space. Applying this method increases the overhead of runtime data movement, because the memory manager must maintain a set of linked lists, and move blocks between them as nodes are moved.

For Dict, using one space imposes an overhead (41% $\rightarrow$ 25%, 22% $\rightarrow$ 12%), but memory can be reduced to $m = 1.5$ without further loss of performance. Using lower values of m will reduce performance, but much of this can often be regained by using a threshold on the quality of blocks to which node are moved, in combination with compaction.

For MList, the overhead of using one space is smaller (10.6× $\rightarrow$ 9.56× and 1.60× $\rightarrow$ 1.24×, for $v = 2^7$ and $v = 2^{19}$, respectively). Using $m = 1.05$ caused no decrease in speedup for long lists, and relatively slight decrease for short lists (1.24× $\rightarrow$ 1.17×), but we believe that for a longer experiment length all values of v will suffer when m is lowered.

Concerning block size, we use only lines for Dict because investigation of stop-the-world data movement demonstrated that the additional overhead of improving page layout was larger than the benefit of the reduced TLB miss rate. For MList, unlike for reallocation, we find that using pages is far more successful than lines (up to three times faster) – lines are only ever useful for low memory and short lists, and then only slightly – because en masse movement of nodes reduces some of the costs of using larger blocks.

2. Threshold and compaction: For Dict, which suffers from a lack of sufficiently empty blocks when m is lowered below 1.5, we apply two techniques: firstly the bulk data movement optimisation stalls normal data movement until a block of a certain quality (amount of free space) arises, and secondly additional data movement is performed to explicitly compact blocks to produce blocks of increased quality. This movement can be performed either at pointer-update sites within the program (where reallocation would be applied), or within the optimisation, with the latter more effective for Dict.

Compaction is most effective when the movement of the RDS is split into small chunks, using the techniques of incrementalisation or embedding, which we summarise below. For example, splitting the traversal up using embedding and then applying compaction allows m to be reduced to 1.10 with little loss in performance (30% → 27%, 12% → 7%, for moveNode and moveFields respectively, compared to 14% and 0% without compaction). This demonstrates that greater sophistication of data movement – combining data movement that does not directly improve layout with movement that does – can be used to significantly improve performance.

3. Incrementalisation: Periodic movement of the entire RDS to another space (or within the same space) is rather like a stop-the-world GC, which we may make incremental using familiar techniques. This process is easy for MList, and incurs very little overhead, because the structure decomposes into many distinct lists, each of which can be moved in their entirety without a significant pause in normal program work. In general, if a compiler or programmer can decompose a larger RDS into smaller RDSs, latency may be reduced with little loss in performance. For Dict, this is not possible, and when two spaces are used a write-barrier is required for correctness (as in a GC) – this form of incrementalisation caused the reductions obtained to decrease by 8–9 percentage points (41% → 32% for moveNode, 22% → 13% for moveFields).

When one space is used, a write-barrier is not required, the optimisation is simply allowed to miss parts of the RDS until the next traversal – this decreases the reductions by only 1–5 percentage points.

In both situations, the data layout produced by incrementalisation is poorer than the stop-the-world version because the mutator changes the structure as it is being re-laid out. However, incrementalisation improves the performance of compaction, improving the resilience to low memory.

4. Embedding: An alternative approach to reducing latency is to embed data movement code in existing program loops, moving nodes in small groups as they are used in the normal course of execution. This also has the potential to reduce the number of data accesses generated by the data movement optimisation. For Dict, when $m = 2$ embedding the one-space data movement improves the reduction obtained (25% → 30% for moveNode), compared to incrementalisation which decrease it (25% → 20%). Furthermore for low memory ($m = 1.1$) the fine interleaving of data movement code with normal program work allows compaction to function more effectively than with incrementalisation (for moveNode embedded achieves 27%, incrementalisation achieves 9%). For MList, there is little scope for improvement over incrementalisation, and indeed embedding tends to reduce performance. Thus, we conclude that for some programs embedding is a very valuable technique, but must be applied carefully.

7.2.4 Summary for programmers

If twice as much memory can be used, and occasional large latencies are allowed, occasionally re-laying out the entire RDS in another space is the most effective bulk data movement (indeed, the best runtime data movement overall). This movement can be done every k operations, or when miss rates (or some other measure of layout quality) reach a certain level.

To reduce memory usage, a single space a factor of $m > 1$ larger than the memory required by the RDS should be used. This space is split into lines or pages, using a similar memory manager to that used for reallocation (see §7.1.4). The choice of line or page is likely to follow the static layout properties of the RDS. The code that moves the RDS should now move groups of nodes to lines/pages of maximal space, rather than empty lines/pages.

To reduce latency, two approaches may be used. Firstly, the movement of the RDS may be incrementalised, which is simple if one space is used because no write barrier is required. This could be achieved using another thread, provided sufficient care is taken to ensure the integrity of pointers. Secondly, data movement code may be embedded in existing program loops. This is harder than a one-space incremental movement of the RDS, but simpler than two-space. In some cases, embedding is much more effective than one-space incremental movement, particularly when m is small. Furthermore, data movement occurs with frequency proportional to the use of structure, which often means that little tuning of the optimisation is required.

When a single space is used, reducing m may also decrease layout quality. This effect can be partially prevented by putting a minimum threshold on the amount of space in the blocks used to re-lay out the structure, and by performing some additional node movement to produce better quality blocks (compaction). A simple compaction algorithm is as follows: when normal data movement cannot occur because no block with space above the threshold was found, move the nodes to block(s) of minimal non-zero space.

7.3 Perfect data movement

In this section we evaluate perfect data movement, investigating the quality of layout that should be maintained, the memory usage, and how significant reducing the number of node movements is when restoring data layout.

7.3.1 Dict

Recall that for perfect data movement for Dict we restore one of two layouts (**BFS** and **fixedHeight**), performing the minimal number of node movements to a depth of zero, one or two clusters.

Fig. 7.10 shows the effect of varying the depth of minimal node movement for **BFS**, including the percentage reduction in execution time obtained when $s = 1$ (the maximum modification rate of one insertion/deletion for every lookup). Note that this figure is the

variant	depth	potentially unnecessary moves (%)	nodes moved /operation	red'n (%)
moveNode	0	70	7.0	-4
	1	10	3.2	36
	2	.75	3.0	37
moveFields	0	100	5.0	-41
	1	33	1.0	19
	2	2	0.8	22

Figure 7.10: The effect of varying the depth to which cluster rebuilding is performed using the minimal node of movements, for the **BFS** layout. Third column shows the proportion of node moves that occur in functions that do not necessarily perform minimal movements. Fourth column shows average number of nodes moved per operation. Final column shows the reduction in execution time obtained by the optimisation.

same as Fig. 6.3, but with the reductions included. The results show that the depth-0 solution – naïvely reclustering the updated cluster and those below it – is too slow, obtaining large negative reductions in execution time. However, depth-1 solutions are all that are needed to get close to the best performance. In other words, the updated cluster should be repaired with the minimal number of node movements, but below that naïve reclustering can be used.

Fig. 7.11 gives reductions in execution times for **BFS** and **fixedHeight**, for different values of s. Also included is the time for the read-only lookup operation, which is a measure of the layout quality of the tree. The reductions obtained are competitive with reallocation and bulk data movement, and we compare them in more detail in the next section. Actual memory usage m is around half of $m_{worst} \in [4, 4.2]$; a steady $m = 2.33$ for **fixedHeight** and $m = 1.94$ for **BFS**, irrespective of s.

The relation between **BFS** and **fixedHeight** is as expected. When modification rate is high ($s = 1$), it better to maintain a slightly worse layout with lower overhead (**fixedHeight**). When modification rate is low, more time is spent reading the structure than updating it, and so it is better to maintain a better layout (**BFS**). The difference between the two layouts is not very significant, no more than 4 percentage points. This is a positive conclusion, because **fixedHeight** is significantly easier than **BFS** to implement.

Thus, we see that by use of a simpler layout (**fixedHeight**) and simpler restoring (depth-1), performing effective perfect data movement in practice does not require an especially complicated implementation. In particular, it is our opinion that the implementation is simpler than a B-Tree.

7.3.2 MList

The two clustering functions clustStrict(*blockSize*) and clustNonStrict(*blockSize*), and the single minimisation function min(*smallBlockSize*, *largeBlockSize*) (which is used to allow the allocation and freeing of small blocks using a minimum number of large blocks) can be used to maintain several different layouts for MList by perfect data movement. We treat these as four families of optimisations (min, clustStrict, clustNonStrict

		reduction (%)		μ seconds/lookup	
variant	s	**BFS**	**fixedHeight**	**BFS**	**fixedHeight**
moveNode	1	37	43	0.94	0.89
	0.1	49	49	0.93	0.89
	0.01	41	38	0.95	0.91
moveFields	1	19	22	0.91	0.87
	0.1	14	13	0.90	0.87

Figure 7.11: Perfect: Dict: Reduction in execution time for perfect data movement for **BFS** and **fixedHeight**, for Dict benchmark.

and clust[Non]Strict+min), as given in the table on page 92.

7.3.2.1 Overview

Unlike Dict, we have (indirect) control over actual (m, M) and worst-case memory usage (m_{worst}, M_{worst}) when performing perfect movement for MList. This is achieved by varying the block size and the minimum number of nodes allowed in a block. In this section we consider actual memory usage *including* all memory manager headers, M. This is more relevant than m because varying block size alters the value of M/m.

In Fig. 7.29 for $v \in \{2^7, 2^{11}, 2^{15}, 2^{19}\}$ we plot speedup against M for the four families of optimisations. Results use the best out of simple and complex clustering. For each line, these graphs show the improvement (reduction or speedup) obtained for M *at most* the value of the horizontal axis. In other words, each line is non-decreasing, even for optimisations whose performance degrades as M increases. For values of M where there is no line for a particular optimisation, the optimisation cannot be used with that value of M.

Apart from for very short lists ($v = 2^{19}$), the most effective of the four families is simple strict or non-strict clustering. Non-strict is a little better that strict when lists are short (5.5× for non-strict, 4.5× for strict), but for longer lists strict gives twice the speedup. For longer lists strict clustering can exploit larger block better because it keeps nodes in traversal order, allowing the hardware prefetcher to work. For short lists, the cost of update is more important than the traversal cost, so the $O(1)$ update that non-strict clustering provides gives a slight improvement over strict. For $v = 2^{19}$ the overhead of strict and non-strict clustering is too high – the large movement of nodes that occurs when a block overflows or underflows is unnecessary when the lists are short.

The hybrid strategies in clust[Non]Strict+min are ineffective, either slower than strict/non-strict clustering, or requiring far more memory.

The min(cell, b) optimisation performs well for an average list length of 8 nodes ($v = 2^{19}$). Note that simple minimisation min(cell, b) requires at least $m = 1.5$ because each node has to be augmented with a parent pointer. Although the layout maintained is not as good as strict and non-strict clustering, the much lower overhead translates to a respectable 1.6× speedup. For average list length of 128, min(cell, b) also performs relatively well when m is high enough. For longer lists the optimisation is ineffective, which is as expected because blocks are unlikely to stay in the cache long enough when lists contain thousands of nodes. Thus we conclude that min(cell, b) is a good technique either when lists are short enough or memory is high enough, or when a simpler implementation

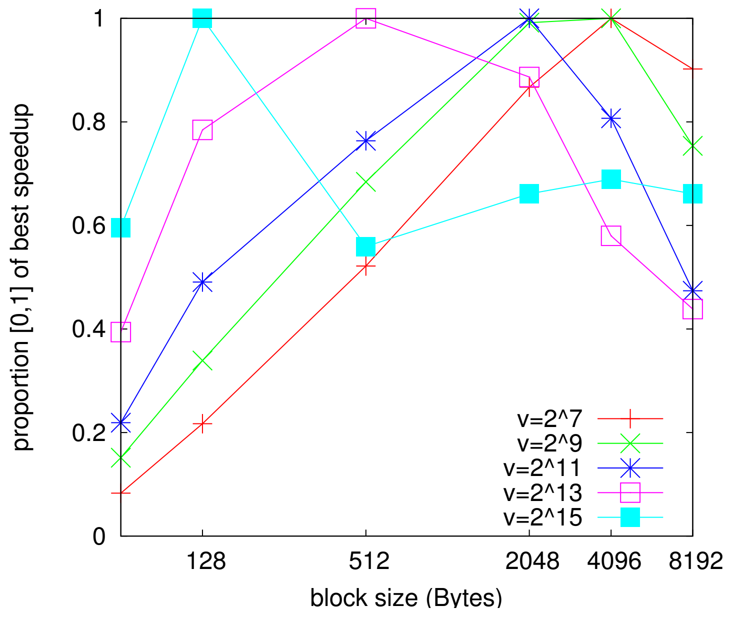


Figure 7.12: Perfect: MList: clustStrict(*blockSize*), for varying *blockSize* and v. On the vertical axis, for each value of v, results are scaled linearly so the maximum speedup obtained is 1 and a speedup of $1\times$ is 0. In more detail, proportion$_v$(blockSize)=(speedup$_v$(blockSize)-1)/(bestSpeedup$_v$-1).

is favoured.

7.3.2.2 Simple vs. complex clustering

Here we compare the the two different methods of clustering: simple, which moves a node (or gap) forward in the list until a suitable block is found, and a more complex scheme similar to Rubin et al.'s VCL structure [60]. Recall from §6.3.2 that in the worst case the former makes roughly twice as many block accesses as the latter, for long enough lists[6]. Fig. 7.28 shows the best of the two families clustStrict and clustNonStrict, for simple and complex clustering, plotted against M.

As is expected from the worst case block accesses, complex clustering is more effective than simple for long lists ($v = 2^7$). For shorter lists, simple clustering is occasionally a little better because it has lower overhead, but the difference is slight. Thus, in practice, the simple method of clustering is sufficient to perform effective perfect data movement when the average list length is around 2048 or less ($v \geq 2^{11}$).

7.3.2.3 Choosing the correct block size

Here we discuss the effect of block size on the performance of perfect data movement for MList. Using a larger block size has the potential to improve layout, firstly because fewer pages are accessed when a list is traversed (hence fewer TLB misses), and secondly because the hardware prefetcher may remove L1/L2 misses, if nodes are kept in order within blocks. However, using large blocks may increase the number of pages, increasing working set size, increasing miss rates.

[6]About 1.99 times as many block accesses for $v = 2^7$, and 1.88 times as many for $v = 2^{11}$.

Now considering overhead, for strict clustering using larger blocks increases the length of any node shuffling within a block, but will reduce the chance of a block overflowing or underflowing, and thus the effect of block size on overhead is not easily predictable. A further effect is that the longer the lists the more time is spent traversing to the point at which the update occurs, and so the overhead of updates is less significant.

How the performance of an optimisation relates to blocksize is thus not easy to predict. Fig. 7.12 shows the effect of block size on the performance of the best optimisation, simple strict clustering (clustStrict). The vertical axis is scaled so that for each value of v the best speedup corresponds to 1 and a speedup of $1\times$ corresponds to 0 (i.e. $y=$ (speedup$_v(x)$-1)/(bestSpeedup$_v$-1), where x denotes the horizontal axis and y the vertical).

Experiments show that for all lists, increasing the block size increases the quality of data layout, but as shown in the figure this does not translate to improved performance, because increasing block size also increases update cost. For $v = 2^{15}$, where lists are on average 128 nodes (8 cache line's worth), the best balance between maximising layout quality and minimising update cost is to use blocks the same size as cache lines, rather than just using a block size large enough to contain each list within it. As v is increased, the best block size increases, and thus the increase in overhead is less significant than the improvement in layout that larger blocks give. However, it is never most efficient to use a block size that will contain the whole list.

In terms of the tuning of the perfect data movement optimisation, we note that there is no single block size that works reasonable well for the list lengths covered here. Using a block size either half or twice as large as the best block size, can cause up to a 0.45 loss in speedup (using the scaling from the figure), usually around a 0.20 loss. Using only either line or page will cause a 0.25 loss, and furthermore if the incorrect choice of these two options is used, large losses are possible: For example, when $v = 2^7$, using lines instead of pages gives a 0.8 loss. Thus a conservative approach that attempts to minimise memory wastage for all list lengths by using small blocks[7] may remove the majority of the speedup if lists turn out to be long when the program runs.

In conclusion, we have shown by varying blocksize that best performance often comes from using perfect data movement to restore a *reasonable* layout rather than the best layout. The perfect data movement optimisation in this thesis are effective, but must be tuned based on list length, which may only be possible at runtime. More complex schemes (e.g. adjusting the block size on a list-by-list basis using a memory manager that provides blocks of several sizes) might allow the optimisation to adjust to changes in list length at runtime.

7.3.2.4 Worst-case memory

Unlike reallocation or bulk, perfect data movement does not have a specified memory usage, but we may calculate the upper bound on memory usage (the 'worst-case' memory usage), as we did for MList in §6.3.3. The results from Fig. 7.29 are shown against worst-case memory usage in Fig. 7.30.

In general, worst-case memory is larger relative to actual memory when blocks are large and/or the minimum number of nodes in a block (i.e. the minimum number of nodes for clustering) gets smaller. For $v < 2^{19}$, it appears that the best optimisations

[7]For $v = 2^{15}$, using pages gives $m = 4$, compared to using lines which gives $m = 1.2$.

keep blocks at least three quarters full, and don't use large blocks when to do so would waste a lot of memory. This prevents worst-case memory usage from being much larger than actual memory usage.

The most significant difference is for long lists, where the minimum amount of memory needed to get any improvement (or the best improvement) increases from close to 1 to around 1.2. This is not a large amount of extra memory in real terms. However, for $v = 2^{19}$, worst-case memory is rather higher – $m_{worst} = 3.2$ is required to get a 1.6× speedup, compared to the actual memory usage $m = 2.3$.

In conclusion, for average list length larger than a line, perfect data movement for MList is suitable for applications where memory must be properly bounded, provided the minimum number of nodes per blocks is kept high, which appears to happen naturally when the best optimisation is chosen and tuned to maximise improvement in execution time. This is in contrast to Dict where worst-case memory is at least 4, and actual memory usage was between 1.9 and 2.4. For lists less than a line, the worst-case memory requirements are even higher than the already high actual memory usage.

7.3.3 Conclusions

Here we summarise a few observations on the construction of an effective perfect data movement optimisation:

1. Choice of layout: There is considerable evidence that using a layout worse than optimal is necessary to get the best performance. For Dict, when the update rate s is high it is more effective to use the slightly worse **fixedHeight** layout because is cheaper to maintain, but for low s using the better **BFS** layout is more effective. For MList layout relaxation takes three forms: reducing memory density, reducing block size, and using a layout involving non-strict clustering and/or minimisation. Reducing memory density is necessary to reduce cost of update from linear in list length to linear in block size. Regarding block size, the best performance comes from using a block size in the range [lineSize,pageSize], depending on the list length, whereas the best layout is always produced by using pages. Non-strict clustering is slightly better than strict for $v \geq 2^{15}$, and use of min(cell,line) is very effective for $v = 2^{19}$, but in all other situations the most effective optimisation was strict clustering (the analogue of the cache-aware solution).

2. Memory usage: Depending on the benchmark, the memory demands of perfect data movement may be high – both practically and worst-case. For Dict, $m \in [1.9, 2.4]$ is required in practice, but the worst-case memory is somewhere in $[4, 4.2]$, far too high for memory-constrained applications. For Mlist, provided lists are larger than a line, the amount of memory used is comparable to reallocation and bulk, and the worst-case memory is also reasonable.

3. Implementation difficulty: Perfect data movement maintains a chosen layout, but does not need to do so with the minimum number of node movements. This makes perfect data movement a little more attractive as a source of optimisations. For Dict, we combine a few simple functions and rely on the observation that most modifications are near the bottom of the tree, rather than always performing the minimal number of node movements. We observe that it is only necessary to perform the minimal number of node movements within the updated cluster to achieve good performance. Simply rebuilding the updated cluster from scratch after a pointer update is far too expensive, and thus perfect data movement must be applied using some knowledge of the pointer update that has occurred. Furthermore, we have shown that if the tree is well-balanced, we can use a simple fixed cluster height, instead of defining clusters by BFS, which greatly simplifies the implementation. For MList, we have shown that simple clustering (shuffling a node or gap forward until space is found) performs as well as more complex clustering schemes, when the average list length is smaller than a few thousand nodes.

7.3.4 Summary for programmers

Perfect data movement is harder to implement than either reallocation or bulk data movement, but is often more effective. In some situations it may be simpler than using a cache-aware structure, and for branching traversals our results suggest that its performance is competitive with the cache-aware structure. In addition perfect data movement may be preferable to using a cache-aware structure because read accesses of the structure are unchanged.

Unlike reallocation or bulk data movement, worst-case memory usage is different from typical-case memory usage. Worst-case memory is reasonable for linear structures, but may exceed four times the space required for the unoptimised structure for branching structures.

For linear traversals, perfect data movement can be achieved simply by using a good cache-aware structure, but retaining pointers between nodes. Such cache-aware structures typically keep nodes grouped into blocks, stored either in-order or out-of-order (the latter is cheaper to update, but the former has a better layout). When an insertion or deletion occurs, nodes are moved between blocks to keep the number of nodes in each block below a minimum value. The minimum number of nodes per block and the block size (between the size of a line and size of a page) should be varied to find the best balance between overhead, layout quality and memory usage. For sufficiently short lists (in our experiments, a few thousands nodes), simply shuffling nodes (or gaps) forward in the list is an effective way of restoring blocks to the correct number of nodes (given a non-pathological sequence of updates). For longer lists, more complicated schemes may be more effective (for example the 'VCL' scheme of Rubin et al. [60]).

For branching traversals, perfect data movement can be achieved by clustering nodes into blocks. We did not find any value in using blocks larger than pages. Clusters may be filled either using breadth-first search or be a rigid shape. The latter is simpler to implement but for sparse trees may waste memory and create a poorer layout. On insertion and deletion, the updated cluster and all clusters below it in the tree are repaired. This requires a favourable distribution of pointer updates – uniformly distributed insertions and deletions in a binary tree work well, but if updates were too heavily weighted to the root of the tree the amount of reclustering per update would be sufficiently high to prevent the improvement in layout translating into an improvement in performance. The results in this thesis suggest that at least the updated cluster and possibly the first level of clusters below should be repaired intelligently, rather than just moving all nodes to an empty line. This is significantly more complicated for clusters filled using breadth-first search than rigidly-shaped clusters.

7.4 Comparison

We now compare the three different families of optimisation, concentrating on four different *'environments'*, formed by two binary choices.

Firstly, we consider either high or low latency. Recall that latency is defined by looking at subsequences of operations that take about $1ms$ in the unoptimised benchmark. An optimisation is said to have low latency when no subsequence's performance is significantly decreased when the optimisation is applied. We summarise the latency of the different optimisations and cache-aware structures below:

Reallocation: Low latency, since only a single node moved per operation.

Bulk data movement:
- periodic: High latency due to movement of whole structure.
- incremental: Low latency, because program work only has to be paused for long enough to move a block's worth of nodes. Note however, that the larger the pause the more efficient incrementalisation will be.
- embedded: Low latency, because data movement is only turned on occasionally and is interleaved with normal program work.

Perfect data movement: Low latency in practice, due to distribution of pointer updates in the structure, but high latency in pathological situations. e.g. for DICT, multiple updates near the root of the tree, causing reclustering of a large part of the structure.

Cache-aware structures:
- DICT: Low latency.
- MLIST: Low latency in practice, but high in pathological situations, as for perfect data movement.

In this evaluation we will consider only the latency observed in practice –i.e. only bulk data movement's periodic optimisations have high latency.

Secondly, we consider either actual memory usage (m, M) or worst-case memory usage (m_{worst}, M_{worst}). Both bulk data movement and reallocation have $M = M_{worst}$, but the cache-aware RDSs and perfect can have significantly larger M_{worst} than M.

A summary of the latency and memory properties of the different optimisations and structures can be found in Fig. 7.13. The results for all optimisations for the four environments for the two variants of DICT and four list lengths for MLIST are given in Figs. 7.31–7.36. For cache-aware and reallocation the best results obtained are given. For perfect data movement we give only the best result for MLIST, but for DICT we show both perfect optimisations (**BFS** and **fixedHeight**). For bulk data movement we show the best one-space optimisation and the best two-space optimisation. The value of the maximum improvement possible from runtime data movement (as calculated in §3.4.3), is also shown.

For each series, these graphs show the improvement (reduction or speedup) obtained for memory *at most* the value of the horizontal axis. In other words, each line is non-decreasing, even for optimisations whose performance degrades as memory increases (e.g.

	latency		
optimisation/structure	observed	patho'l	memory
reallocation	low	low	$M = M_{worst}$
bulk: periodic	high	high	$M = M_{worst}$
bulk: incremental, embedded	low	low	$M = M_{worst}$
perfect	low	high	$M < M_{worst}$
cache-aware: DICT	low	low	$M < M_{worst}$
cache-aware: MLIST	low	high	$M < M_{worst}$

Figure 7.13: For the three optimisations, and cache-aware RDSs, this figure shows whether actual memory M is different to worst-case memory M_{worst}, and the observed and pathological latency.

due to increasing working set size). If the line for a particular optimisation does not exist for some value of M, it means the optimisation cannot be used with that value of M. Finally, note that no results is given within the shaded box, which is between $[1, 1.2]$ on the horizontal (memory) axis. This is because to show accurately how the improvements decreased to zero (or lower) as memory tended to one would require a large number of experiments, but we don't consider this behaviour important – it is simpler just to omit the results.

In the rest of this section we will firstly compare the easier to apply techniques (reallocation and bulk data movement) and then compare them to perfect data movement. Secondly, we discuss the long-term stability of each optimisation technique, and how they may be tuned. Finally, we compare the optimisation to other methods of preventing layout degradation in RDSs (cache-aware RDSs and layout-improving garbage collectors).

7.4.1 Easily applied optimisations: reallocation and bulk

Here, note that $M = M_{worst}$, and only bulk data movement's periodic optimisations have high latency in practice.

DICT: When low latency is required, reallocation gives 1.5–2 times larger reductions than bulk data movement for MOVEFIELDS, and is a few percentage points better for MOVENODE. For low enough memory, bulk is better than reallocation – giving a 23% reduction for MOVENODE compared to reallocation's 15%. When high latency is allowed, bulk is 5–7 percentage points better than reallocation provided M is larger than ~ 2 (when the periodic2space optimisation can be used). The periodic optimisations do not beat reallocation when low latency is required.

Thus, both optimisation techniques are valuable, with the best depending on both memory allowance and whether high latency is allowed.

MLIST: For M around 1.5, bulk gives a larger speedup than reallocation (but usually only by a factor of 1.1), apart from for very short list ($v = 2^{19}$) where bulk is a little worse than reallocation (a factor of 0.9). For lower values of M, reallocation's performance decreases, but only by a significant amount for $v = 2^{15}$, where bulk data movement obtains almost twice the speedup of reallocation for $M = 1.2$. For M greater than around 2, the periodic bulk data movement optimisation is by far outstanding because the two-space optimisations can be used, producing speedups between 1.1 and 1.8 larger than reallocation.

Thus, for MList, bulk data movement is more powerful in general. However, for mid-range values of M (around 1.5) the difference between reallocation and bulk is relatively small. Unlike with Dict, allowing high latency does not have a significant effect on the absolute or relative performance of the two techniques.

Discussion: Bulk data movement and reallocation are both easily applied, but are orthogonal approaches to data movement: the former corrects the degradation of every individual pointer-update, the latter corrects many degradations en masse. We might expect bulk data movement to be more effective because many of the memory manager costs can be reduced, but the results demonstrate that reallocation is often equally good, and sometimes better, depending on the benchmark, memory allowance, and whether high latency is allowed.

7.4.2 Perfect data movement

Here, note that perfect data movement has $M \neq M_{worst}$, and only bulk data movement's periodic optimisations have high latency in practice.

Dict: Both methods of perfect data movement either equal or exceed reallocation, by 9 percentage points for moveNode and 3 for moveFields. However, they do so a cost of $M \geq\sim 2.2$, and $M_{worst} \in [4.57, 4.80]$, whereas reallocation obtains most of its reduction by $M = M_{worst} = 1.35$. Perfect always outperforms bulk.

MList: Ignoring the min(c,l) perfect optimisation, which is extremely effective for the lists of length 8 of $v = 2^{19}$, the results show that perfect data movement is better than reallocation for long list ($v = 2^7$), but worsens as lists shorten, eventually becoming slower. For $v = 2^7$, the performance of the perfect optimisation is only slightly better than bulk, but worse than bulk for larger v.

Discussion: We have observed that perfect data movement can outperform reallocation for both benchmarks, and thus we may conclude that performing more complex data movement at pointer update sites is worthwhile – and there may be a middle ground of optimisations more effective than reallocation but simpler to apply than perfect. For MList, we observe that neither form of pointer-update data movement approaches bulk's simple twoSpace optimisations, which suggests that for some benchmarks correcting data movement en masse is far more effective.

The relative performance of reallocation, bulk and perfect suggest that well-tuned approximate optimisations (bulk, reallocation) will perform fairly well compared to, and sometime exceed, more precise data movement (perfect). Thus, implementation effort may be traded for tuning effort.

7.4.3 Stability

The experimental results given in this thesis are for the improvement in execution time seen for a small group of operations at the end of the experiment, whose length is around ten minutes. For bulk and perfect, the optimisations that achieve the largest improvement according to this measure are also stable – in other words the improvement does not

	improvement	
benchmark	normal length	10× longer
Dict-moveNode†	31% S	31% S
Dict-moveFields	18% U	27% U
MList $v = 2^7$	8.11× U	20.7× U
MList $v = 2^{11}$	14.9× U	9.64× U
MList $v = 2^{15}$	5.65× S	5.07× S
MList $v = 2^{19}$	1.34× S	1.27× S

Figure 7.14: The stability of reallocation. The reduction or speedup obtained using the normal experiment length, and ten times longer, are shown for the different benchmarks. 'S' indicates stable, 'U' unstable.

decrease if the experiment is run for longer. In some cases the improvement will increase if the unoptimised benchmark's data layout continues to degrade.

In bulk data movement's case the stability is aimed for when tuning parameters. For perfect data movement for MList, it often takes most of the experiment length to achieve a stable time – this is because the structures are built by insertion, but updated using insertion and deletions, and so the block densities take some time to reach an equilibrium.

Reallocation is not always stable, but good improvements are still seen when the experiments are run to ten times normal length, as shown in Fig. 7.14. More detail of MList's stability can be found in §7.1.2.4.

In conclusion, all the optimisations in this thesis, even the most ad-hoc, can be relied on to produce improvements in both the short- and long-term.

7.4.4 Tuning

Here we demonstrate the effect on performance of mistuning bulk data movement optimisations. Similar results for reallocation were given in Figs. 7.16–7.25 (varying the subset of sites, varying the coallocators, and varying the block size), and for perfect in Fig. 7.12 (varying block size for MList).

Fig. 7.15 shows how the optimal data movement rate depends on the list length (for MList) or the value of s (for Dict) – observe that if the list length or s are different at run time to what was tuned for, a lot of performance can be lost.

For Dict-moveNode and the embeddedOneSpace optimisation, Fig. 7.37 shows in more detail how the optimal values of data movement rate and threshold depend on both s and m. Observe that some of the behaviour is unexpected – when $m = 2$ memory is plentiful, and one would expect optimisation to be easier, but the penalty for mistuning is far higher than for $m = 1.1$.

7.4.5 Comparison to other solutions

Here we compare the three optimisation families to two other methods of preventing RDS data layout degradation: cache-aware RDSs and layout-improving garbage collectors.

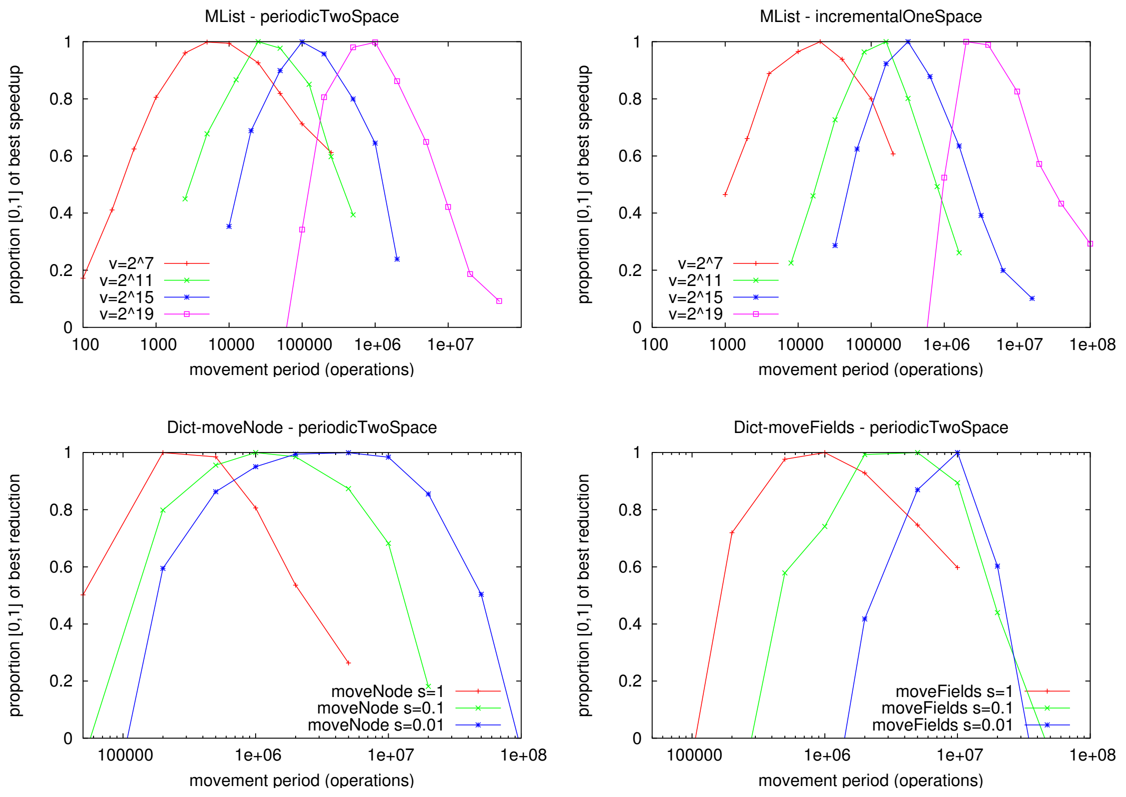


Figure 7.15: The effect of mistuning data movement rates, for both benchmarks, for the periodicTwoSpace and the best low-latency oneSpace optimisation. The horizontal axis shows the period of movement of the whole RDS, measured in operations. The vertical axis shows proportion of best reduction for DICT, and proportion of best speedup for MLIST (as in Fig. 7.12).

7.4.5.1 Cache-aware RDSs

The results in Figs. 7.31–7.36 are used.

DICT: For MOVENODE, the cache-aware solution is better than the best easily applicable solution (reallocation) but not as good as the better perfect data movement solution (**BFS**). Regarding implementation effort it is debatable whether **BFS** is harder to implement than a B-Tree, but the latter has far lower worst-case memory requirements, so is probably preferable.

For MOVEFIELDS, cache-aware performs badly, obtaining only half the reduction of reallocation. This difference in performance is perhaps because a B-Tree maintains a well-balanced tree unlike the binary search tree used for DICT, or perhaps because the B-Tree has higher instruction costs than the optimisations despite maintaining a better layout.

In conclusion, we may view the application of reallocation, bulk or perfect to DICT as a way of *synthesising* cache-aware data structures that are more effective than B-Trees in some situations (e.g. for MOVEFIELDS, or when m can be high), but harder to reason about analytically.

MLIST: We do not expect perfect data movement to exceed the cache-aware RDSs, because apart from for $v = 2^{19}$, the former is just clustStrict, which is simply a density-relaxed, higher overhead form of the latter. Indeed, the removal of internal pointers within blocks allows cache-aware to far outperform all our data movement optimisations – even exceeding the maximum speedup we could obtain by runtime data movement for $v \leq 2^{11}$. Memory usage M is also very low, often below one, something too that is not achievable by applying runtime data movement to a simple linked list structure.

The exception to cache-aware's dominance is for the short lists of $v = 2^{19}$, where bulk's periodic optimisations are either competitive with (high latency not allowed) or better than (high latency allowed) the cache-aware RDSs.

Although the cache-aware solutions are usually better than the runtime data movement optimisations, the former is never better by more than a factor of 1.5. This is achieved for example for $v = 2^{11}$, where the cache-aware solution achieves a speedup of 40× and the best runtime data movement optimisation achieves 27×. Compared to the unoptimised benchmark, this may not be significant difference; it is conceivable that a programmer might find using a simple linked list with the incremental2space optimisation and $m = 2$ a better compromise between performance and implementation effort than the cache-aware RDS. Thus we conclude that runtime data movement allows most of the benefit of using a cache-aware solution to be realised, through methods which may be easier to implement.

7.4.5.2 Locality-Improving GCs

The bulk optimisations can be seen as modelling the sort of runtime data movement an LIGC might achieve (for results, refer to Figs. 7.26 and 7.27). For both benchmarks periodicTwoSpace achieves the best improvements, and thus we might expect that if a programmer isn't concerned with memory usage or latency, a stop-the-world two-space LIGC will achieve similar improvement, provided the rate of data movement is correct (refer to §7.4.4 and Fig. 7.15), the overhead of updating parents can be reduced and some mechanism for producing a good layout in to-space is used (either by the programmer explicitly choosing a layout as in Novark et al. [51], or by profiling [12, 17, 35, 66]).

For low-latency situations, our results suggest that the performance hit of using an incremental GC is not too severe[8].

For low memory for MList, our results suggest that using one space and moving to the best block available allows the reduction of memory without significant impact on performance. Locating the best block is simpler when all objects are the same size, and so if the RDS can inhabit its own region (a transformation which can be done automatically [44]) both layout and overhead will be reduced. This process may allow the possibility of walking the RDS heap at a different rate to the rest of the heap (cf. generational garbage collection). If the whole heap must be walked to re-layout the RDS, the performance overhead will be large.

For low memory for Dict, we found that periodic or incremental bulk data movement was far less effective than embedding data movement code in traversal loops. The implication of this is that for low memory, runtime data movement in a GC may be ineffective, and a solution more closely coupled to the program will work more effectively – such as bulk's embeddedOneSpace, reallocation or perfect data movement.

[8]This is for Dict. For MList, we relied on the fact that the RDS can be split into small distinct parts, avoiding the need for a write barrier. This will not be possible for an LIGC without some help from the applier to identify distinct parts of the RDS. However, we believe that, like Dict, the overhead of the write barrier would not be prohibitive.

In conclusion, our results demonstrate that GC-based runtime data movement can be effective for high memory and high latency, provided the cost of updating parents is low and the data movement rate is correctly adjusted, but for tougher conditions, non-GC techniques such as embedding data movement traversals, reallocation or perfect data movement are likely to be more effective.

7.4.5.3 Summary

In reallocation, this thesis demonstrates a novel pointer-update-time ad-hoc approach to data movement, which produces good improvement with low latency and copes naturally with low memory. In perfect data movement, the limits of pointer-update data movement have been investigated, synthesising a new cache-aware structure for trees that outperforms B-Trees in practice. In bulk data movement, this thesis demonstrates the performance that a good LIGC may achieve, and identifies the limitations, and suggest refinements such as embedding data movement and compaction, providing new insight which may guide the design of these collectors.

The techniques of this thesis come close to and on one occasion exceed the performance of traditional cache-aware structures. We observe here that the results of this thesis represent the upper bound on the performance that may be achieved by runtime data movement (due for example to the lack of parent pointers in structures, and the uniformity of pointer updates), but note also that the benchmarks used are representative of real problems and have certainly not been contrived to justify particular optimisations. Thus, we believe that there is much potential to make use of these techniques in practice. We conclude that although a programmer should always use a structure designed for good cache behaviour, in many situations this is not practical[9], and the techniques of thesis will be of great use to improve performance.

7.5 Further work

All the optimisations require some mechanism to update pointers, different approaches to this problem were discussed in §2.3.2.1. For realistic use, optimisations should be able to cope with multiple node sizes. The memory manager could be rewritten to allow heterogeneous nodes, but it is more efficient if the memory manager is homogeneous. This can be achieved by a combination of separating differently-sized objects into distinct regions, padding objects so they are the same size and splitting the hot and cold parts (using current techniques, see §2.2.5). We note also that the heap may be grown and shrunk dynamically, which we avoided in this thesis because we did not want the performance of the system memory manager to be a factor.

Reallocation: Given some mechanism to update parents, reallocation is a practical optimisation, and all that is required is to apply it automatically. To do this, we must choose the correct sites to use, and the sort of reallocation to use at each site. A method such as the following may be used.

[9]e.g. 1. No such cache-aware analogue of a ‘hand-rolled’ structure exists, 2. The requirements are not known in advance, 3. The implementation effort of the cache-aware structure is too great, 4. It is not initially known that data layout problems are a significant cause of poor program performance.

Each of the k sites has either (i) no reallocation applied (ii) coallocator pattern $\langle \mathsf{L}; \mathsf{null} \rangle$, with one hint chosen in the immediate vicinity of the realloctee (iii) coallocator pattern $\langle \mathsf{L}; \mathsf{EL} \rangle$, with hint as before, or (iv) coallocator pattern $\langle \mathsf{null}; \mathsf{EL} \rangle$. The space to be searched is 4^k, and methods such as those used in Chow et al. [19] can be used to reduce it. Once this search has been performed, the number of hints used can be increased at each site, noting that we expect the number of hints to have an optimal value, since using more hints imposes a higher overhead and may degrade layout quality.

Finding the pointer-updates sites for an RDS is within the reach of static analysis, and we expect it to be relatively easy to find one hint object by static analysis as well. Increasing the number of hints will probably require the insertion of code into program loops, for example as was done with the last_∞ search type for MList, thus a semi-automatic approach may be more feasible.

Bulk: Deploying bulk data movement practically could be done it two ways – either using the techniques within an LIGC (apart from embedded), or applying them automatically as optimisations. Regarding the former, with the incremental variants any help the programmer can give to reduce the cost of or remove the write barrier will help to improve performance. Applying the embedded optimisations automatically may be made easier using automatic tools to unwind loops etc, but it is not clear how simple this process could be made.

For both directions, finding the correct parameters for bulk data movement is critical. As was shown in Fig. 7.15, if the behaviour of the program is not what was tuned for, significant performance can be lost. Furthermore, the behaviour may change at runtime. Tuning parameters statically has been tackled in the past (E.g Chow et al. [19], Bodin et al. [9]) – our optimisations have at most four dimensions, of which only two take more than a half dozen values.

Dynamic tuning (choosing parameters at runtime) may be possible in a number of ways. Note that while changing movement rate, thresholds and turning compaction on and off at runtime may be easy, changing block size will require the cooperation of the memory manager. The ideal method of tuning is for the optimisation to adjust its parameters online in response to the execution time for a group of operations. The aim would be to minimise execution time, subject to layout quality staying constant (i.e. there is no point minimising time now if layout will degrade in the future). This may be possible for programs whose behaviour changes slowly over time, or changes infrequently. A second method is to construct a map from observable behaviour of the program – pointer update rate, average list length, etc – to the value of parameters to use. This requires statically finding the optimal set of parameters for many different variants of the program. This allows the optimisation to select the best parameters if the behaviour of the program changes at runtime, but the procedure must be repeated for different machines.

The above is clearly extremely speculative, and current work in this area is still in its infancy. For example, even in Chen's state-of-the-art locality-improve garbage collector [12] a very simple heuristic is used to determine the data movement rate: a layout-improving collection is activated when miss rates exceed a threshold. As we have shown, the key to efficient runtime data movement is doing it at the correct rate, thus balancing overhead and layout, and so such a scheme requires a properly tuned program- and machine-dependent threshold to achieve the best performance.

Perfect: Applying perfect automatically is probably not possible, but we may view the results of this thesis as insight into cache-aware RDSs. For MList, we found that the existing best cache-aware RDS was usually the best source of perfect data movement optimisation, and so the work doesn't provide much insight. However, we observe here that the actual and worst-case memory usage of these schemes may be reduced by adapting block size based on list length.

For Dict both **BFS** and **fixedHeight** can outperform a B-Tree implementation, and we expect that the new cache-aware structure produced from perfect data movement using these layouts by removing any pointers internal to a cluster would be even more effective. The worst-case memory usage is the main disadvantage to such a structure, but this could be tackled in a number of ways, for example by allowing small subtrees to share lines (a subtree is moved to a new line if it cannot grow within the line).

The tuning of perfect data movement and the new cache-aware structures can be achieved in a similar way to bulk (refer to Fig. 7.12 for the effect of MList's average list length on optimal block size).

site enabled			
deleteMove	insert	deleteCut	reduction (%)
✓	✓	(✓)	24
✓	✓		24
✓		✓	17
	✓	(✓)	6
✓			16
		✓	1
	✓		6
			0

values of p, c, t							
deleteMove			insert		deleteCut		
p	c	t	p	t	p	c	t
1	2	F	1	T	0	0	F
1	2	F	1	T			
1	2	T			0	0	T
			7	F	0	0	F
1	2	T					
					1	0	F
			7	F			

(a) DICT-MOVENODE

site enabled		
insert	deleteCut	reduction (%)
✓	✓	13
✓		5
	✓	5
		0

values of p, c, t				
insert		deleteCut		
p	t	p	c	t
1	T	1	0	F
7	T			
		1	0	F

(b) DICT-MOVEFIELDS

Figure 7.16: Reallocation: DICT: The effect of only enabling a subset of sites. The reduction in execution time obtained is given, for all different subsets of sites enabled. In the left-hand figures, the '✓' symbol indicates that a site is enabled and is performing data movement. The '(✓)' symbol indicates that a site is enabled, but the best combination of strategies used a null strategy at that site (i.e. where $p = c = t = 0$). In the right-hand figures, the values of p, c and t that gave the best performance are given (in the t column, T=true, F=false). Refer to §7.1.1.1 for more description and evaluation of these results, and Fig. 7.1.

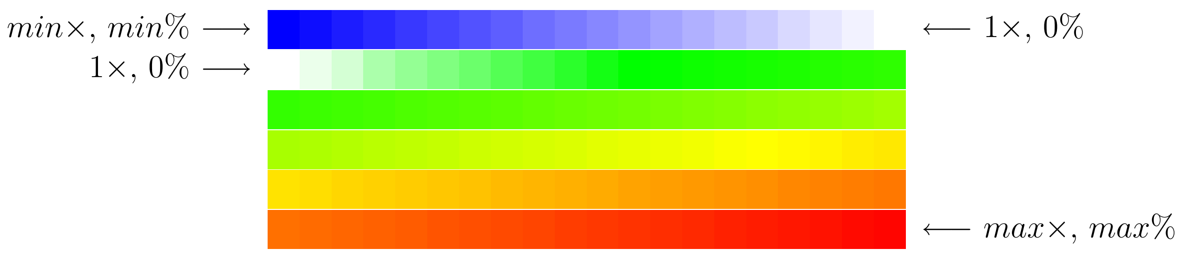

$min\times$, $min\%$ $\longrightarrow$
$1\times$, 0% $\longrightarrow$
$\longleftarrow$ $1\times$, 0%
$\longleftarrow$ $max\times$, $max\%$

Figure 7.17: Read from top to bottom, and within each row from left to right, this figure shows the palette used for Figs. 7.18–7.24 and 7.37. Negative improvements – reductions less than zero (%), or speedups less than one (×) – use a color scheme when blue corresponds to the worst negative improvement in the figure. No improvement corresponds to white. Positive improvements – reductions greater than zero (%), or speedups greater than one (×) – pass through green, yellow and orange arriving at red, which corresponds to the best positive improvement in the figure.

					insert			
		p	1	>1	0	1	>1	0
p	c	t	t	t	t	f	f	f
deleteMove								
1	>1	f	24	22	19	14	14	11
1	>1	t	24	22	19	20	20	16
1	1	f	23	21	16	13	13	9
1	1	t	22	20	17	19	19	16
>1	>1	f	22	21	17	14	13	9
>1	>1	t	21	20	17	19	19	16
>1	1	f	21	20	15	13	12	8
>1	1	t	20	19	15	18	18	15
>1	0	f	18	18	11	11	11	6
1	0	f	18	18	11	11	11	6
>1	0	t	17	16	10	16	16	13
1	0	t	17	16	10	16	16	13
0	>1	f	14	14	11	11	11	9
0	>1	t	13	14	9	14	13	12
0	1	f	13	13	9	9	9	7
0	1	t	12	11	8	11	11	10
0	0	f	-0	5	-4	2	6	0
0	0	t	-3	-3	-5	-1	-2	0

Figure 7.18: Reallocation: DICT-MOVENODE. Reduction in execution time when the strategy used at the insert and deleteMove sites are varied independently, with the deleteCut strategy null. The strategies used for insert and deleteMove are given in Fig. 7.1. Where p's value is > 1, the best result obtained for $p \in \{2, 4, 7\}$ is used. The palette used is in Fig. 7.17.

deleteCut			insert					
		p	1	> 1	1	0	0	> 1
p	c	t	t	t	f	t	f	f
1	0	f	13	9	9	6	5	5
> 1	0	f	10	7	6	5	1	1
1	1	f	10	6	7	5	3	3
1	2	f	7	3	5	2	2	1
> 1	1	f	7	4	4	4	0	0
> 1	2	f	5	2	3	1	0	0
1	2	t	3	-1	4	0	3	0
1	1	t	3	-2	2	0	4	0
> 1	2	t	2	-3	2	0	1	0
> 1	1	t	1	-3	2	-1	0	0
1	0	t	1	-4	1	-2	3	1
> 1	0	t	0	-4	1	-3	0	1
0	0	f	-0	5	2	-6	0	1
0	1	f	-3	2	0	-8	2	1
0	2	f	-4	0	-2	-10	3	-2
0	2	t	-7	-11	-6	-14	6	-9
0	1	t	-7	-10	-8	-12	6	-9
0	0	t	-14	-16	-10	-16	9	-13

Figure 7.19: Reallocation: DICT-MOVEFIELDS. Reduction in execution time when the strategy used at the insert and deleteCut sites are varied independently. The strategies used are given in Fig. 7.1. Where p's value is > 1, the best result obtained for $p \in \{2, 4, 7\}$ is used. The palette used is in Fig. 7.17.

α, β \ γ	EL	EP	-	F	ELIEP	FP	FLIEP	FLIFP	ELIFP	FL
L	24	17	15	14	10	5	3	3	0	0
L, NLIP	20	9	15	14	-	17	-	-	-	-5
L, P	12	20	20	15	7	17	12	10	5	-7
L, ELIP	4	11	12	7	12	8	-	-	9	-11
L, FLIP	4	10	10	5	-	5	9	6	-	-8
	0	-8	0	3	-14	-3	-9	-3	-7	1
P	-1	7	7	2	-5	1	0	-3	-10	-6
FLIP	-3	1	3	0	-	-2	1	-1	-	-7
NLIP	-8	-15	-3	-4	-	-8	-	-	-	-26
ELIP	-9	-3	-1	-6	-4	-8	-	-	-6	-10

30	25	12	14	24	10	12	6	7	-2
28	21	17	18	-	26	-	-	-	-3
29	29	28	29	27	26	27	24	24	-1
30	29	28	28	29	26	-	-	27	-2
23	23	21	21	-	18	22	18	-	-2
-0	-3	0	0	-5	-2	-3	-3	-3	-2
16	16	15	15	15	11	15	9	9	-2
15	15	14	13	-	9	13	9	-	-3
-3	-3	-3	-3	-	-2	-	-	-	-20
15	14	14	14	13	9	-	-	8	-3

Figure 7.20: Reallocation: DICT-MOVENODE: The effect of using pages as well as lines. The coallocator pattern $\langle \alpha, \beta; \gamma \rangle$ is varied as shown in the table, and figures for reduction in execution time (%) are given in the top table. The bottom table gives the percentage reduction in underlying time (i.e. a measure of how much layout has been improved ignoring the overhead of runtime data movement). Refer to §7.1.1.3 for more description and evaluation of these results, and Fig. 7.2 for the reallocation strategies used. The palette used is in Fig. 7.17.

α, β \ γ	EL	EP	-	F	ELIEP	FP	FLIEP	FLIFP	ELIFP	FL
L	13	6	9	7	−3	−11	−6	−9	−14	−2
L, NLIP	7	−4	6	4	-	−16	-	-	-	−5
L, P	−13	−4	−6	−12	−21	−15	−13	−16	−22	−15
L, ELIP	−25	−17	−15	−23	−15	−23	-	-	−19	−24
L, FLIP	−20	−13	−12	−17	-	−19	−13	−16	-	−16
	−7	−15	0	−1	−20	−9	−17	−9	−14	−2
P	−21	−13	−8	−16	−27	−19	−22	−17	−23	−18
FLIP	−24	−19	−14	−19	-	−19	−16	−16	-	−18
NLIP	−16	−29	−8	−10	-	−18	-	-	-	−10
ELIP	−31	−24	−19	−26	−22	−24	-	-	−21	−21

α, β \ γ	EL	EP	-	F	ELIEP	FP	FLIEP	FLIFP	ELIFP	FL
L	25	20	10	11	19	0	9	−2	−2	0
L, NLIP	22	16	13	13	-	0	-	-	-	3
L, P	15	15	12	11	12	3	12	0	0	8
L, ELIP	14	14	10	10	14	0	-	-	1	5
L, FLIP	11	10	7	8	-	−1	9	−1	-	7
	0	0	0	0	−2	0	−2	−2	−2	0
P	9	9	8	7	6	0	7	−1	−1	6
FLIP	6	7	5	5	-	−1	6	−2	-	5
NLIP	−3	−2	−3	−3	-	−2	-	-	-	−2
ELIP	6	6	6	5	6	−1	-	-	−1	−2

Figure 7.21: Reallocation: DICT-MOVEFIELDS: The effect of using pages as well as lines. The coallocator pattern $\langle\alpha, \beta; \gamma\rangle$ is varied as shown in the table, and figures for reduction in execution time (%) are given in the top table. The bottom table gives the percentage reduction in underlying time (i.e. a measure of how much layout has been improved ignoring the overhead of runtime data movement). Refer to §7.1.1.3 for more description and evaluation of these results, and Fig. 7.2 for the reallocation strategies used. The palette used is in Fig. 7.17.

after \ *before*	-	last$_1$	last$_{\geq 1}$	last$_\infty$	first
$v = 2^7$					
-		1.9	7.9	7.9	2.6
next$_1$	2.0	2.1	8.0	8.0	2.6
next$_{\geq 1}$	6.1	6.0	8.0	8.0	2.6
next$_\infty$	4.4	4.4	4.4	4.0	2.3
$v = 2^{11}$					
-		2.2	15.3	15.3	2.9
next$_1$	2.2	2.6	15.3	15.3	2.9
next$_{\geq 1}$	15.0	15.0	15.4	15.4	9.0
next$_\infty$	15.0	15.0	15.0	10.5	9.0
$v = 2^{15}$					
-		1.7	3.9	3.9	3.1
next$_1$	1.7	2.0	4.0	4.0	3.1
next$_{\geq 1}$	4.7	4.7	4.7	4.7	4.7
next$_\infty$	4.6	4.7	4.7	4.7	4.7
$v = 2^{19}$					
-		1.06	1.12	1.12	1.16
next$_1$	1.03	1.09	1.12	1.12	1.14
next$_{\geq 1}$	1.17	1.14	1.15	1.15	1.15
next$_\infty$	1.17	1.14	1.15	1.15	1.14

Figure 7.22: Reallocation: MList: Speedup figures for line-based reallocation (see §7.1.2.1). We use insert := $\langle L; EL\rangle \times \langle$ *before*, *after*; true$\rangle$, where the search types *before* and *after* are varied independently, and delete := null. Where a value of ≥ 1 is indicated, it means that the optimal value in $[1, \infty]$ is used. For example when *before*=last$_{\geq 1}$ and *after*=next$_{\geq 1}$, the best pattern in the set of patterns $\{\langle \text{last}_\alpha, \text{next}_\beta; \text{true}\rangle : \forall \alpha \in [1, \infty], \forall \beta \in [1, \infty]\}$ is used. The figures for no hints (*before*=*after*=-) are omitted because using just EL yields no layout improvement and incurs overhead. The palette used is in Fig. 7.17.

	before				
after	-	last$_1$	last$_{\geq 1}$	last$_\infty$	first
-		4.0	4.8	4.8	2.3
next$_1$	3.9	3.9	4.7	4.7	2.3
next$_{\geq 1}$	4.8	4.8	4.8	4.7	2.8
next$_\infty$	4.6	4.6	4.6	4.1	2.7

$v = 2^7$

	before				
after	-	last$_1$	last$_{\geq 1}$	last$_\infty$	first
-		7.0	10.3	10.3	4.6
next$_1$	7.2	7.2	10.3	10.3	4.6
next$_{\geq 1}$	10.9	10.6	10.7	10.7	9.1
next$_\infty$	10.9	10.6	10.6	10.0	9.1

$v = 2^{11}$

	before				
after	-	last$_1$	last$_{\geq 1}$	last$_\infty$	first
-		2.3	2.8	2.8	3.3
next$_1$	2.6	2.6	2.8	2.8	3.3
next$_{\geq 1}$	2.7	2.8	2.9	2.9	3.3
next$_\infty$	2.7	2.8	2.9	2.9	3.2

$v = 2^{15}$

	before				
after	-	last$_1$	last$_{\geq 1}$	last$_\infty$	first
-		0.93	1.00	1.00	1.00
next$_1$	0.97	0.99	0.99	0.99	0.99
next$_{\geq 1}$	0.97	0.99	0.99	0.99	0.99
next$_\infty$	0.97	0.99	0.99	0.99	0.99

$v = 2^{19}$

Figure 7.23: Reallocation: MList: Speedup figure for page-based reallocation (see §7.1.2.2). We use insert := $\langle P; EP\rangle \times \langle$ *before*, *after*; true$\rangle$, where the search types *before* and *after* are varied independently, and delete := null. Where a value of ≥ 1 is indicated, it means that the optimal value in $[1, \infty]$ is used. For example when *before*=last$_{\geq 1}$ and *after*=next$_{\geq 1}$, the best pattern in the set of patterns $\{\langle \text{last}_\alpha, \text{next}_\beta; \text{true}\rangle : \forall \alpha \in [1, \infty], \forall \beta \in [1, \infty]\}$ is used. The figures for no hints (*before*=*after*=-) are omitted because using just EP yields no layout improvement and incurs overhead. The palette used is in Fig. 7.17.

	γ									
α, β	EL	ELIEP	FLIEP	EP	-	F	FL	FP	FLIFP	ELIFP
L	8.0	8.0	6.9	2.0	1.0	1.0	1.0	1.0	1.0	1.0
L P	7.9	7.9	7.0	4.6	1.0	1.0	1.3	1.5	1.8	1.8
L NLIP	7.8	-	-	7.1	1.2	1.2	1.2	1.2	-	-
L FLIP	7.6	-	7.0	4.4	1.4	1.7	1.9	1.5	1.8	-
L ELIP	7.6	7.9	-	6.9	1.7	2.2	1.9	1.8	-	1.8
ELIP	4.0	4.4	-	4.4	1.4	1.6	1.5	1.5	-	1.5
FLIP	4.0	-	4.3	4.3	1.4	1.6	1.5	1.4	1.4	-
P	3.6	4.4	4.3	4.8	1.0	1.0	1.0	1.5	1.4	1.5
-	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
NLIP	1.0	-	-	1.0	1.0	1.0	1.0	1.0	-	-

$v = 2^7$

α, β										
L	4.6	4.2	3.3	1.7	1.0	1.0	0.9	0.9	0.9	0.9
L P	4.4	4.2	4.5	3.4	0.9	0.9	0.9	0.9	0.9	0.9
L NLIP	4.4	-	-	4.3	0.9	0.9	0.9	0.9	-	-
L FLIP	4.0	-	4.1	3.2	1.0	1.0	1.0	0.9	0.9	-
L ELIP	4.0	4.4	-	4.3	1.0	1.0	1.0	0.9	-	0.9
ELIP	2.8	2.9	-	2.9	1.0	1.0	1.0	1.0	-	1.0
FLIP	2.8	-	3.1	3.1	1.0	1.1	1.0	1.0	1.0	-
P	2.9	2.9	3.1	3.2	1.0	1.0	1.0	0.9	0.9	0.9
-	1.0	0.9	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
NLIP	1.0	-	-	0.9	1.0	1.0	1.0	1.0	-	-

$v = 2^{15}$

Figure 7.24: Reallocation: MList: The effect of using pages as well as lines. Shown are the speedups in execution time as the coallocator pattern $\langle\alpha, \beta; \gamma\rangle$ is varied. The search patterns are given in Fig. 7.5, with evaluation of these results given in §7.1.2.3. The palette used is in Fig. 7.17.

search dist.		coallocator pattern					
				ccmalloc-newBlock-		ccmalloc-firstFit-	
line	page	P EP	L EL	vanilla	extended	vanilla	extended
s	s	4.0	2.1	1.0	3.7(2.0)	1.0	3.9(3.7)
s	l	4.7	2.1	1.4(0.9)	4.4(1.6)	1.5	4.7(4.2)
l	s	4.0	8.0	1.3(1.2)	7.8	1.2	7.6
l	l	4.8	8.0	2.2(1.0)	7.0	1.9	6.2
best		4.8×	8.0	2.2(1.2)	7.8	1.9	7.6

$v = 2^7$

search dist.		coallocator pattern					
				ccmalloc-newBlock-		ccmalloc-firstFit-	
line	page	P EP	L EL	vanilla	extended	vanilla	extended
s	s	2.4	1.8	0.9	2.8(1.8)	1.0	2.9(2.6)
s	l	3.1	1.9	1.0	3.3(1.6)	1.0	3.4(3.2)
l	s	2.5	4.6	0.7	4.4	0.7	4.4(4.0)
l	l	3.2	4.6	0.8(0.7)	4.4(4.3)	0.7	4.5(4.1)
best		3.2×	4.6	1.0	4.4	1.0	4.5(4.1)

$v = 2^{15}$

Figure 7.25: Reallocation: MLIST: The effect of using pages as well as lines. Refer to §7.1.2.3 for more description and evaluation of these results, and Fig. 4.7 for the reallocation patterns used for ccmalloc.

variant	optimisation	m=2	1.5	1.1
MOVENODE (max. = 55%)	periodic2space	41%	-	-
	incremental2space	32	-	-
	periodic1space	25	25	10
	incremental1space[,thresh][,compact]	20	22	9
	embedded1space[,thresh][,compact][,throttle]	30	31	27
MOVEFIELDS (max. = 36%)	periodic2space	22	-	-
	incremental2space	13	-	-
	periodic1space	12	11	0
	incremental1space[,thresh][,compact]	11	10	0
	embedded1space[,thresh][,compact][,throttle]	12	11	7

Figure 7.26: Bulk data movement: DICT: Results for different optimisations, expressed as reduction in execution time. All figures given are for the optimal combination of threshold, compaction, throttling, data movement rate, etc. The 'max.' figure is the maximum possible reduction, as calculated in §3.4.3.

optimisation	m	$v = 2^7$	2^{11}	2^{15}	2^{19}
max.	N/A	11.7×	36.1	11.3	2.14
periodic2space	2.00	10.6	27.1	8.45	1.60
incremental2space	2.00	10.5	26.9	8.09	1.48
periodic1space	1.05	9.56	21.6	5.16	1.17[a]
incremental1space	1.05	9.72	19.5	5.09[b]	1.06[c]
embedded1space	1.05	7.11	12.8[d]	3.64[e]	1.17[f]

a: 1.17× → 1.24× as $m \to 1.1$

b: 5.09× → 5.68× as $m \to 2.0$

c: 1.06× → 1.23× as $m \to 1.5$

d: 12.8× → 13.7× as $m \to 1.1$

e: 3.64× → 4.61× as $m \to 2.0$

f: 1.17× → 1.24× as $m \to 1.1$

Figure 7.27: Bulk: MLIST: Results for different optimisations. All figures given are for the optimal combination of threshold, compaction, throttling, data movement rate, etc. The 'max.' figure is the maximum possible reduction, as calculated in §3.4.3.

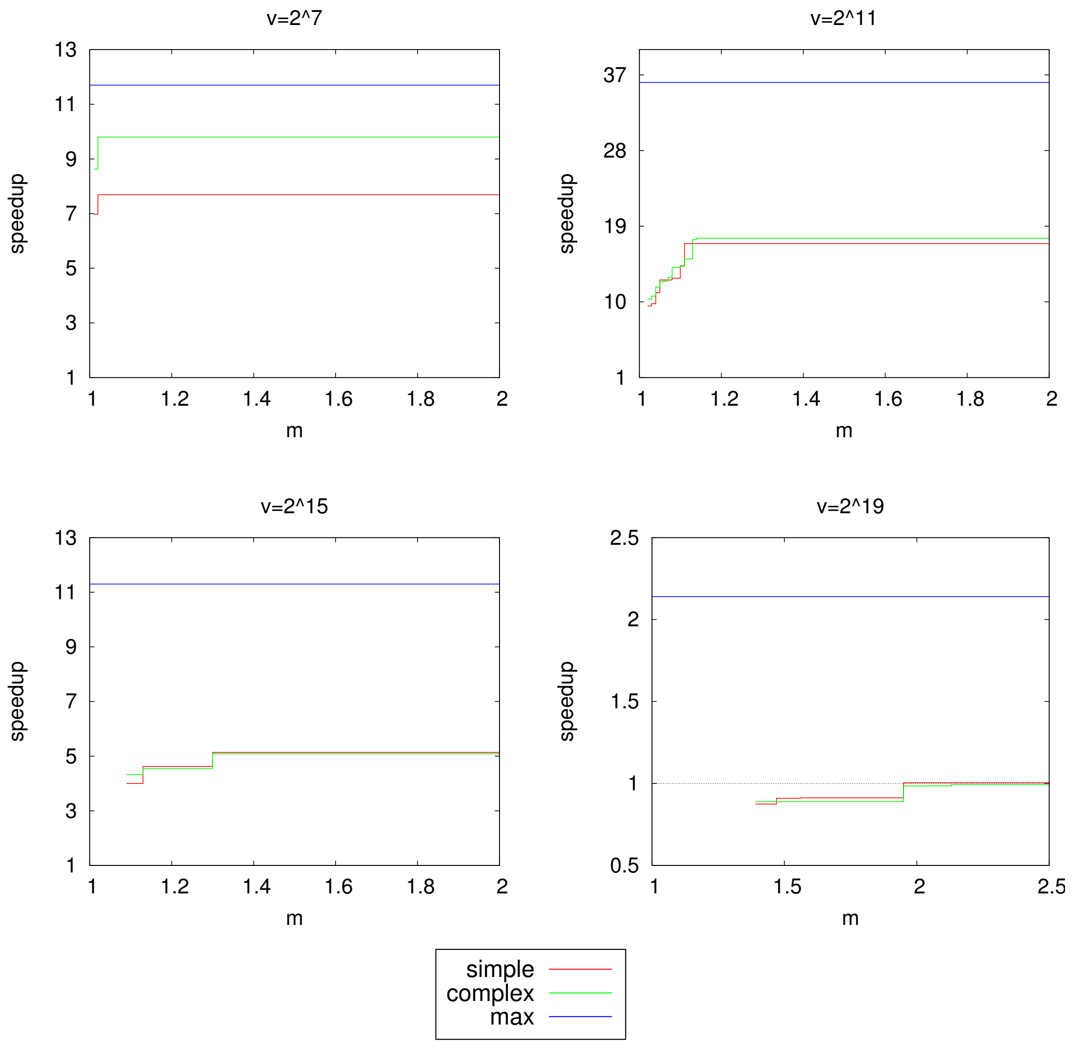


Figure 7.28: Perfect: MList: This figure shows the best of the clustStrict and clustNonStrict perfect data movement optimisations, using either complex or simple clustering.

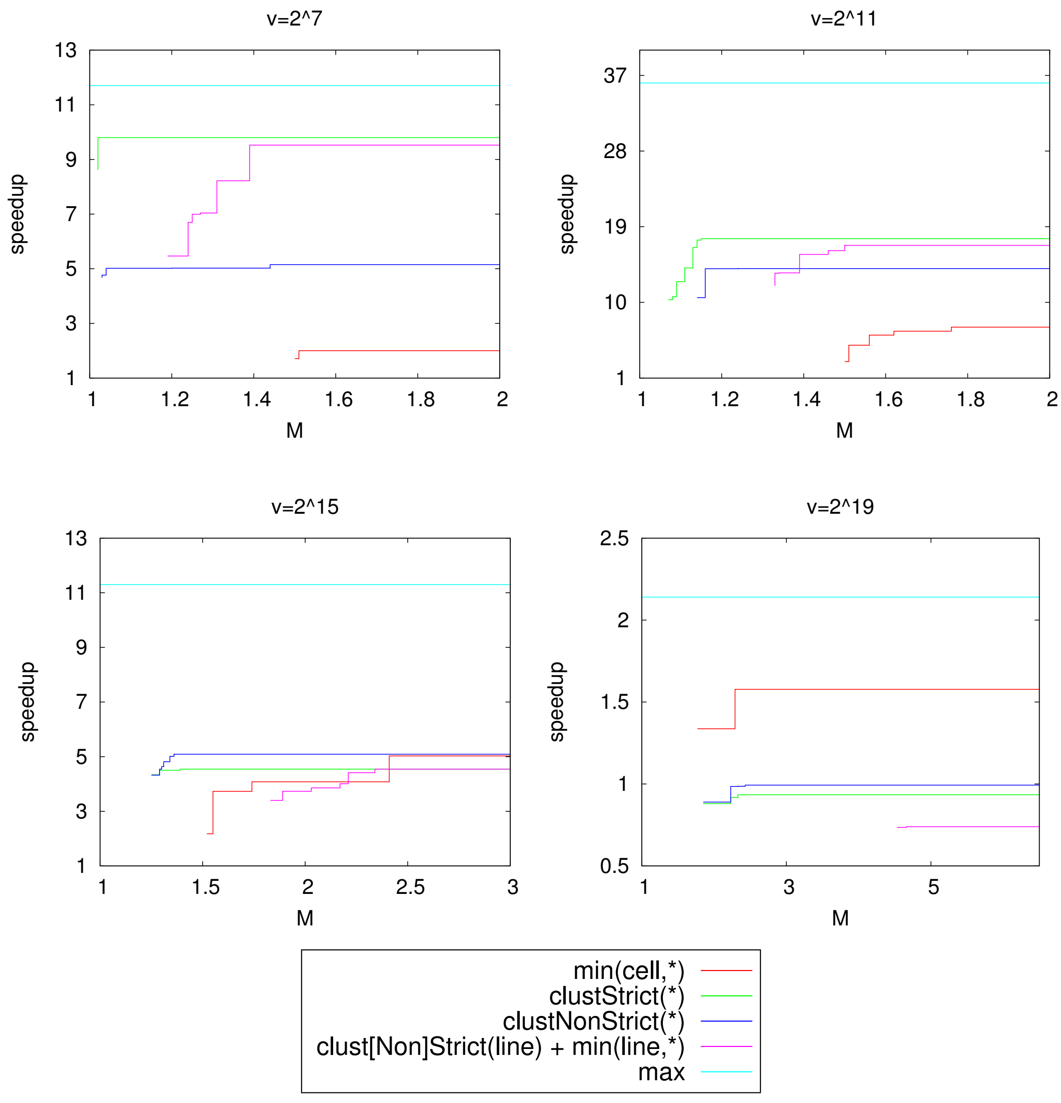


Figure 7.29: Perfect: MList: The speedup given by the different forms of perfect data movement, plotted against actual memory usage M. The 'max' line is the theoretical maximum speedup obtainable if the optimal layout were maintained with zero overhead.

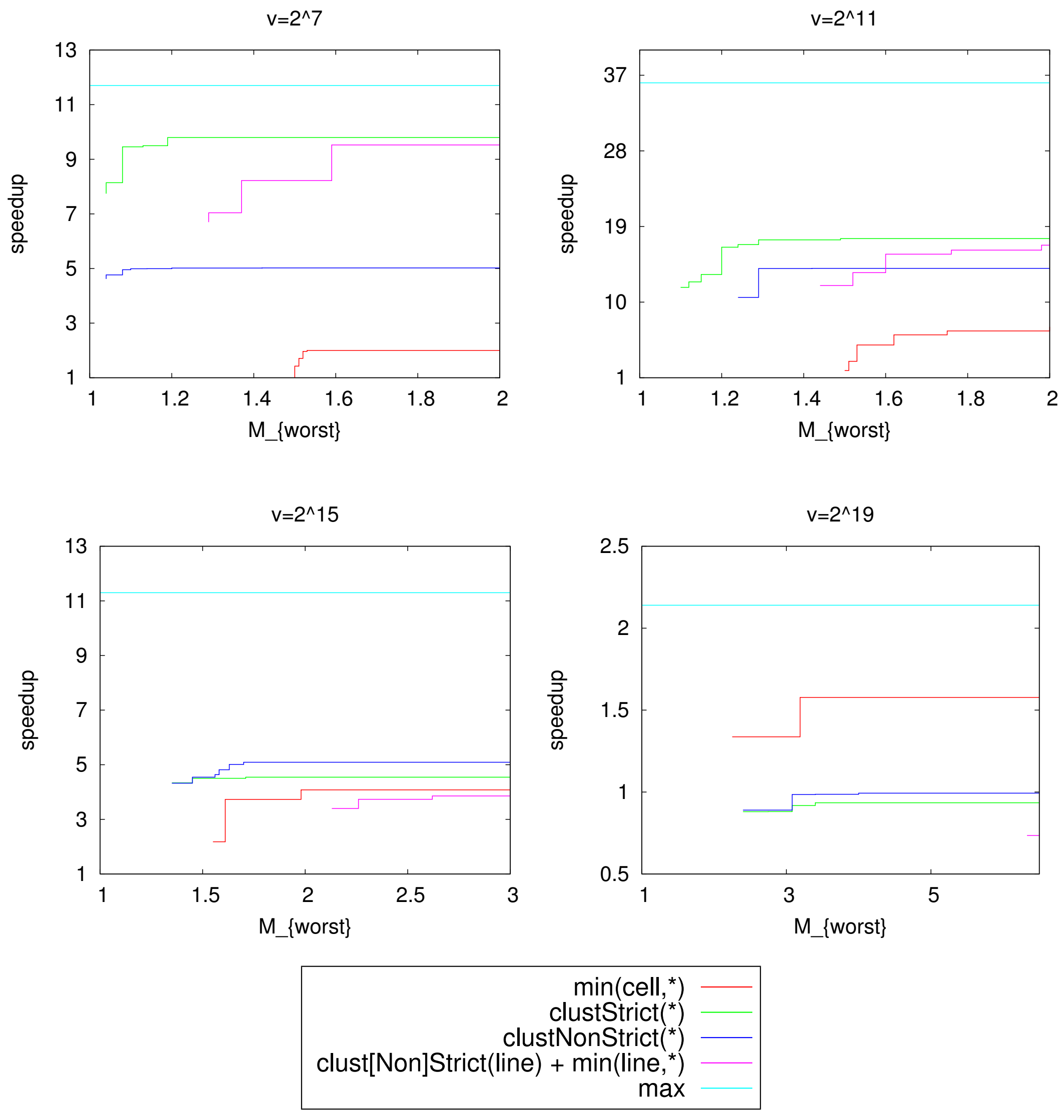


Figure 7.30: Perfect: MList: As Fig. 7.29, but *worst-case* memory usage is used on the horizontal axis.

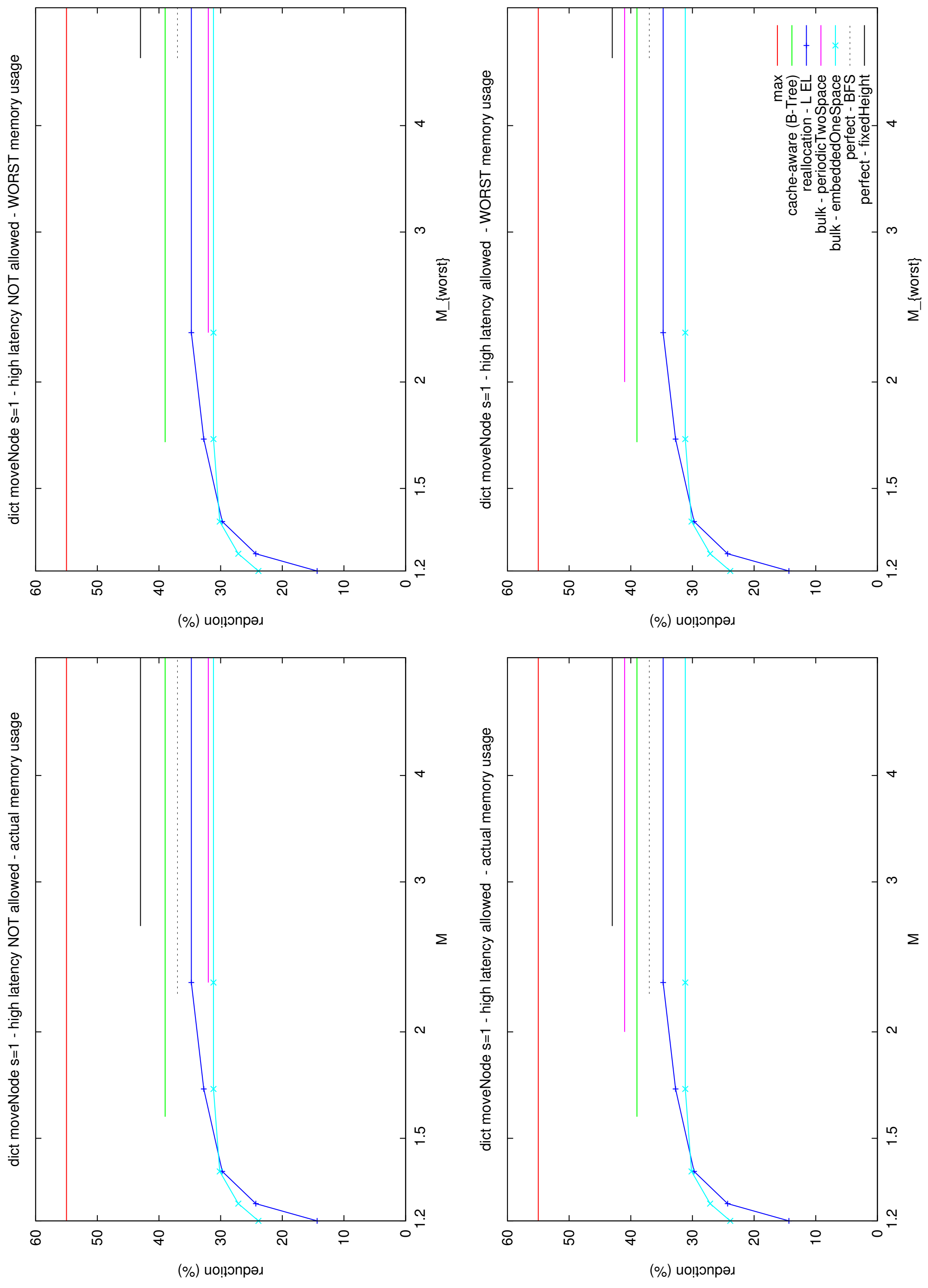


Figure 7.31: DICT-MOVENODE, all results. See §7.4.

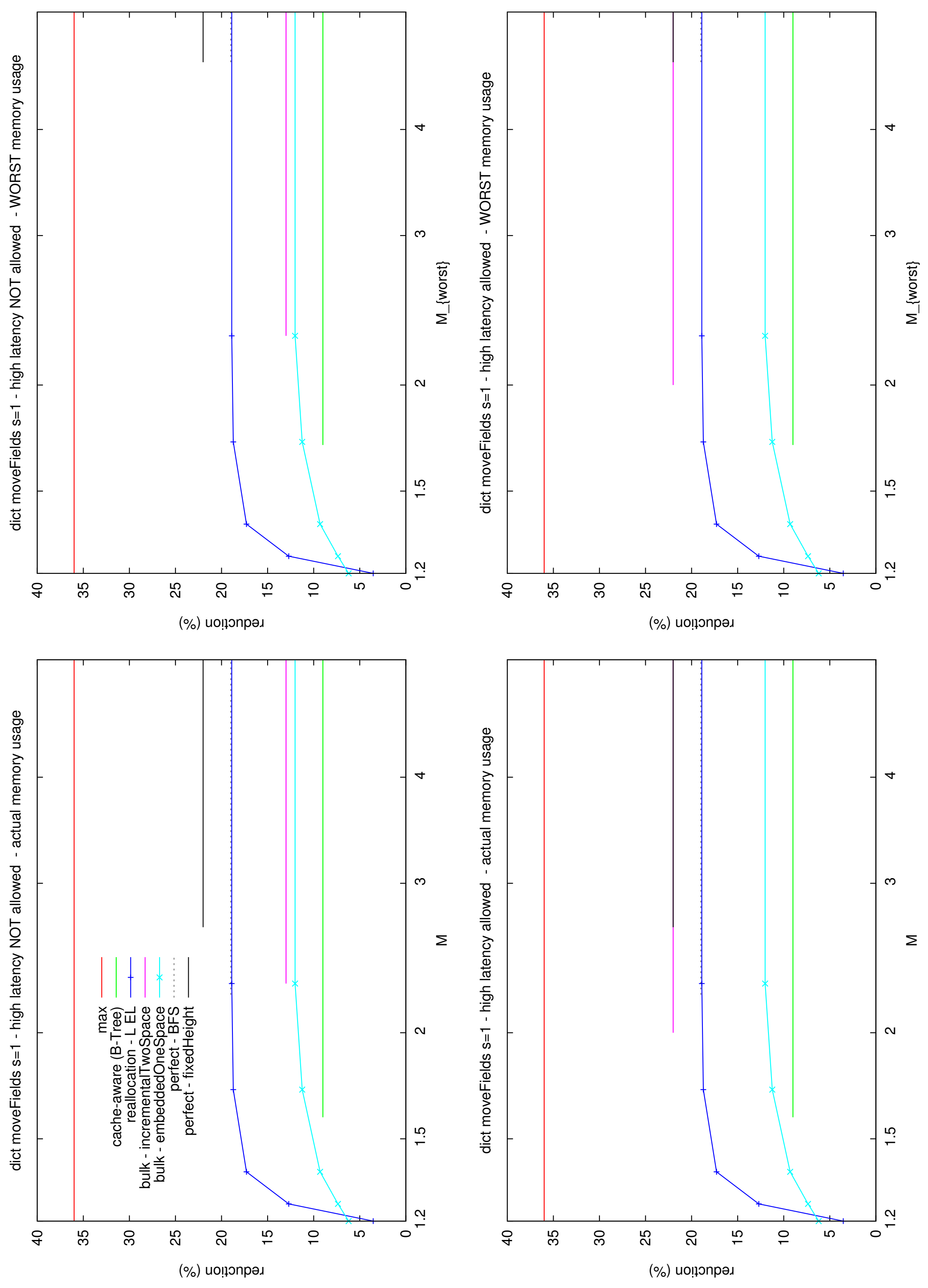


Figure 7.32: Dict-moveFields, all results. See §7.4.

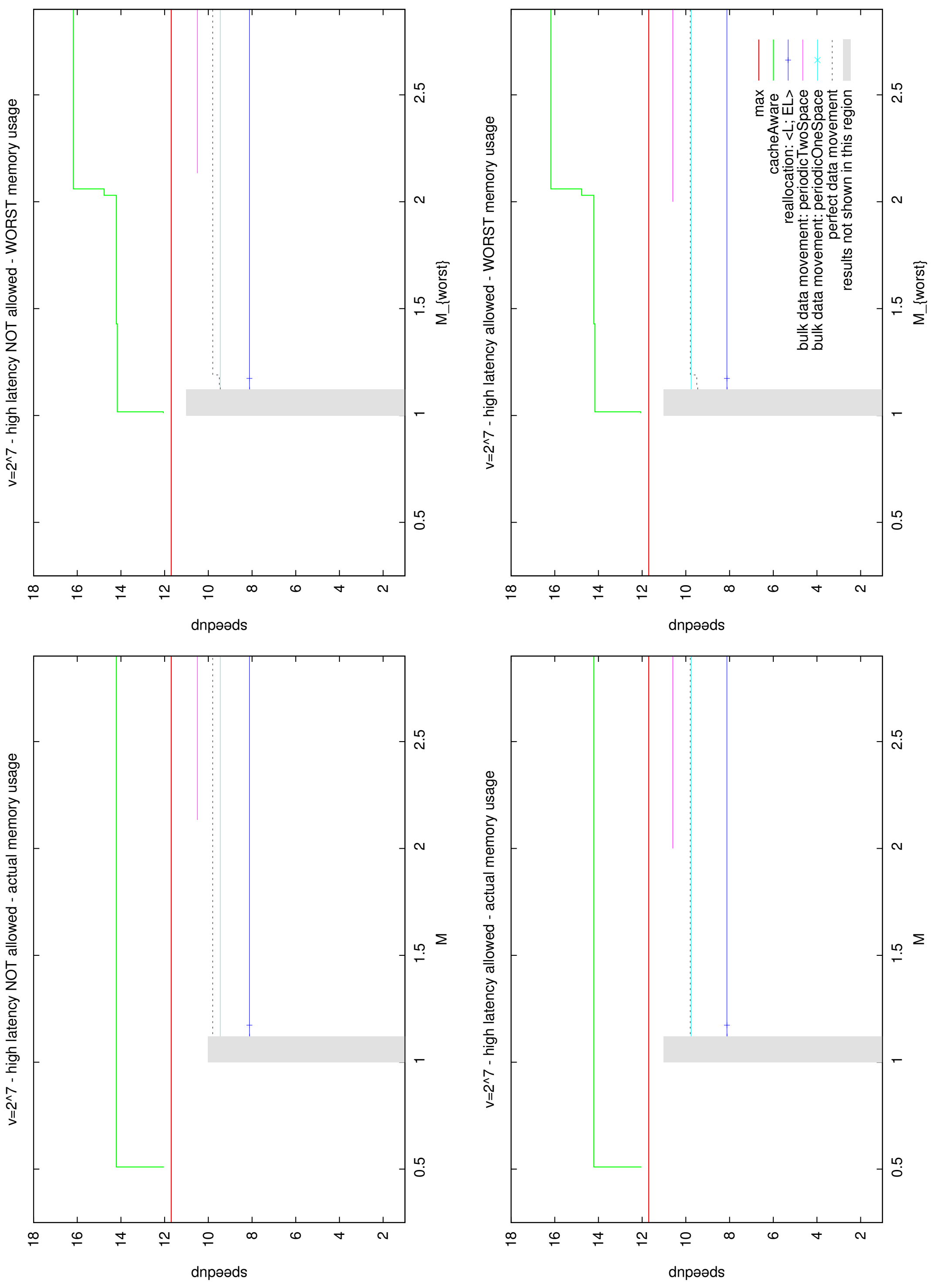


Figure 7.33: MList $v = 2^7$, all results. See §7.4.

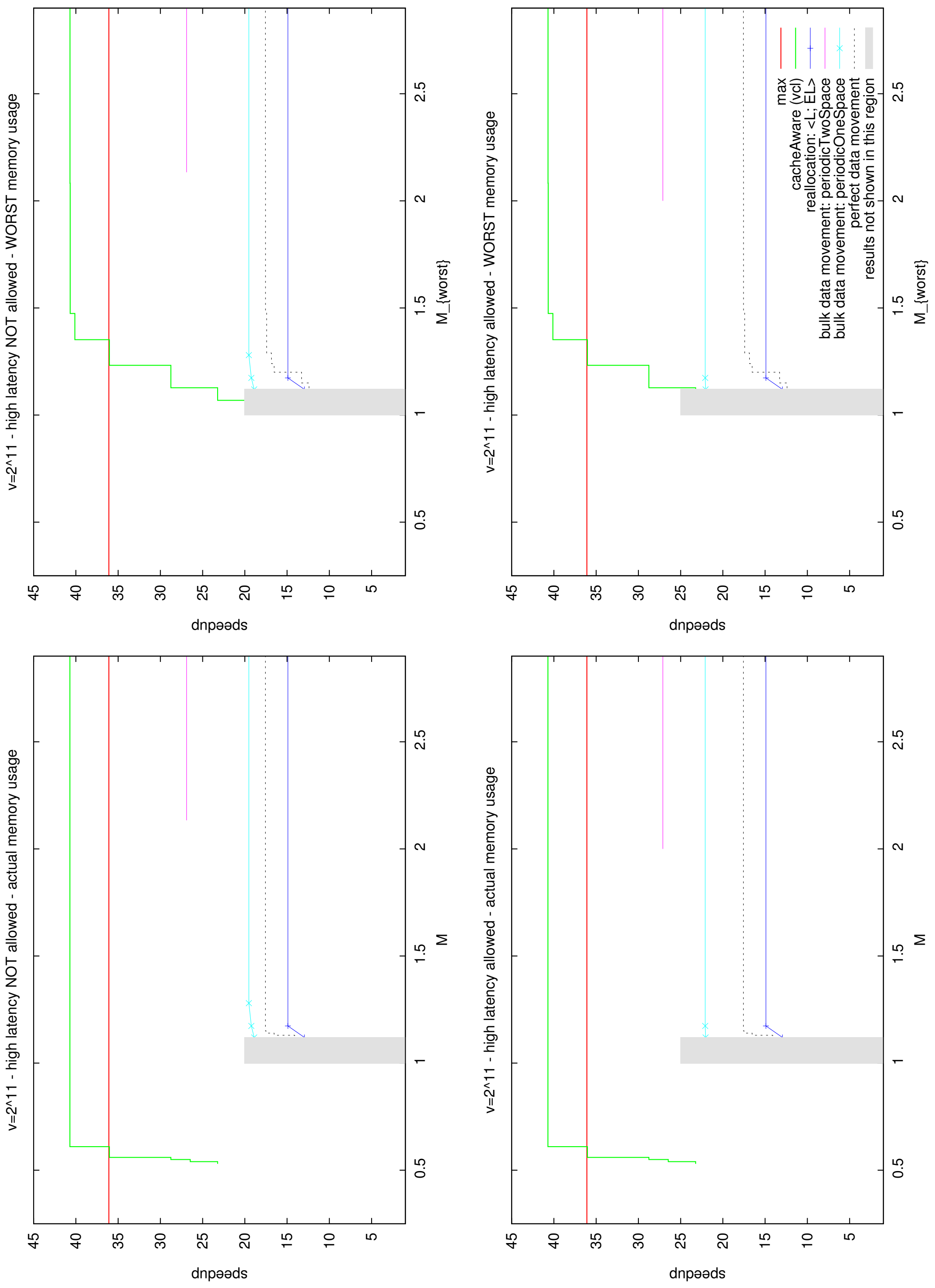


Figure 7.34: MLIST $v = 2^{11}$, all results. See §7.4.

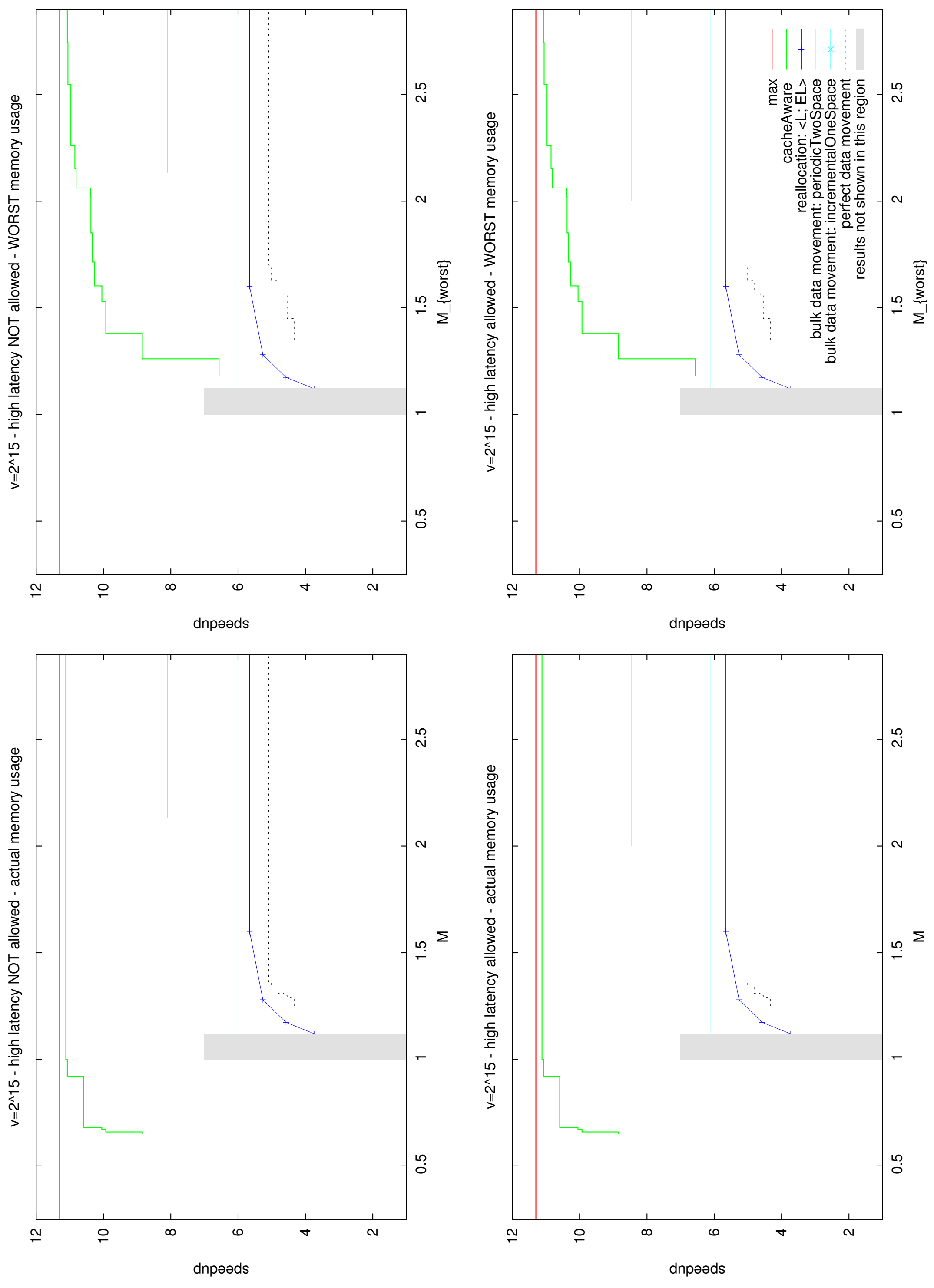


Figure 7.35: $\text{ML}\textsc{ist}$ $v = 2^{15}$, all results. See §7.4.

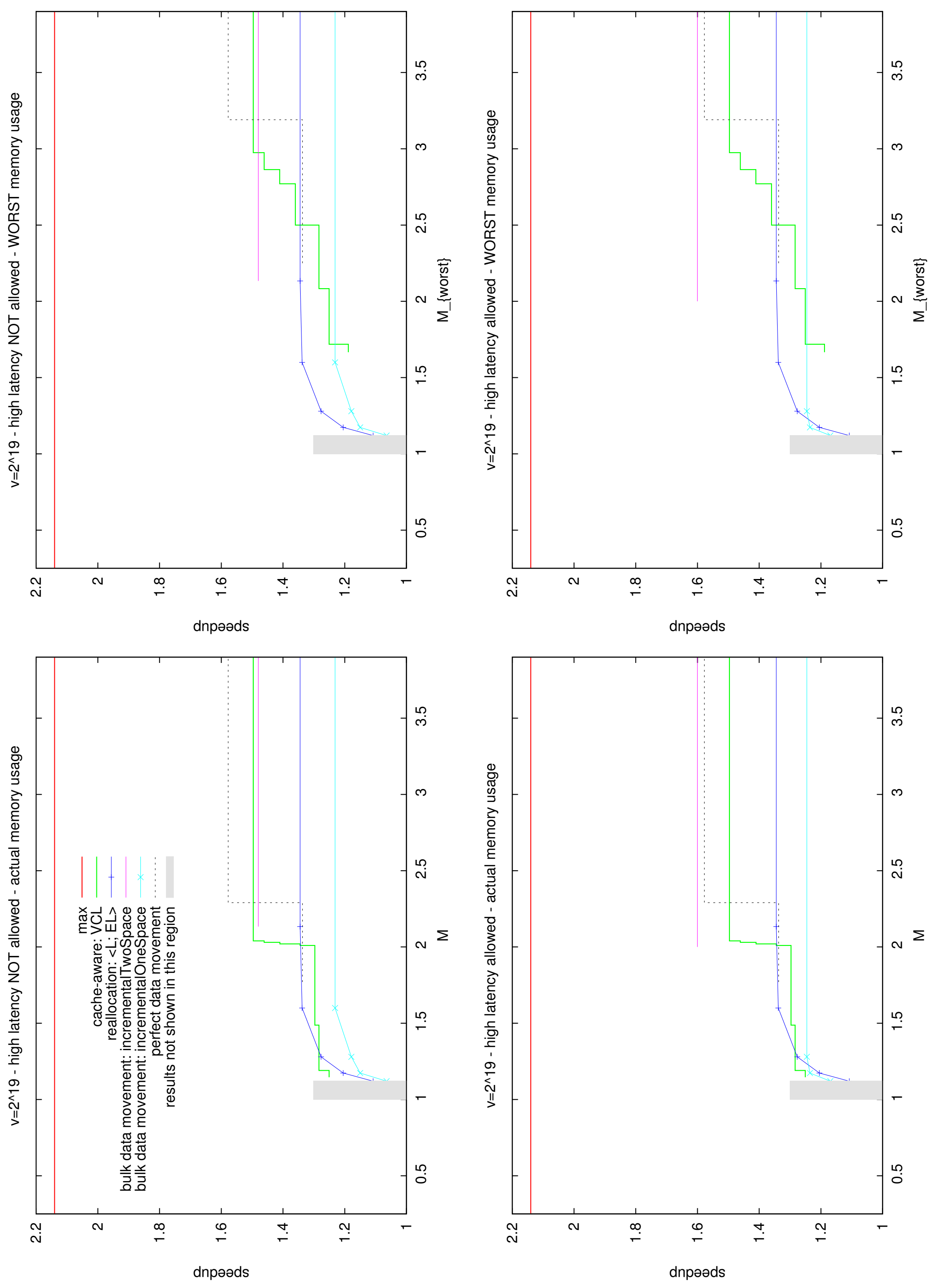
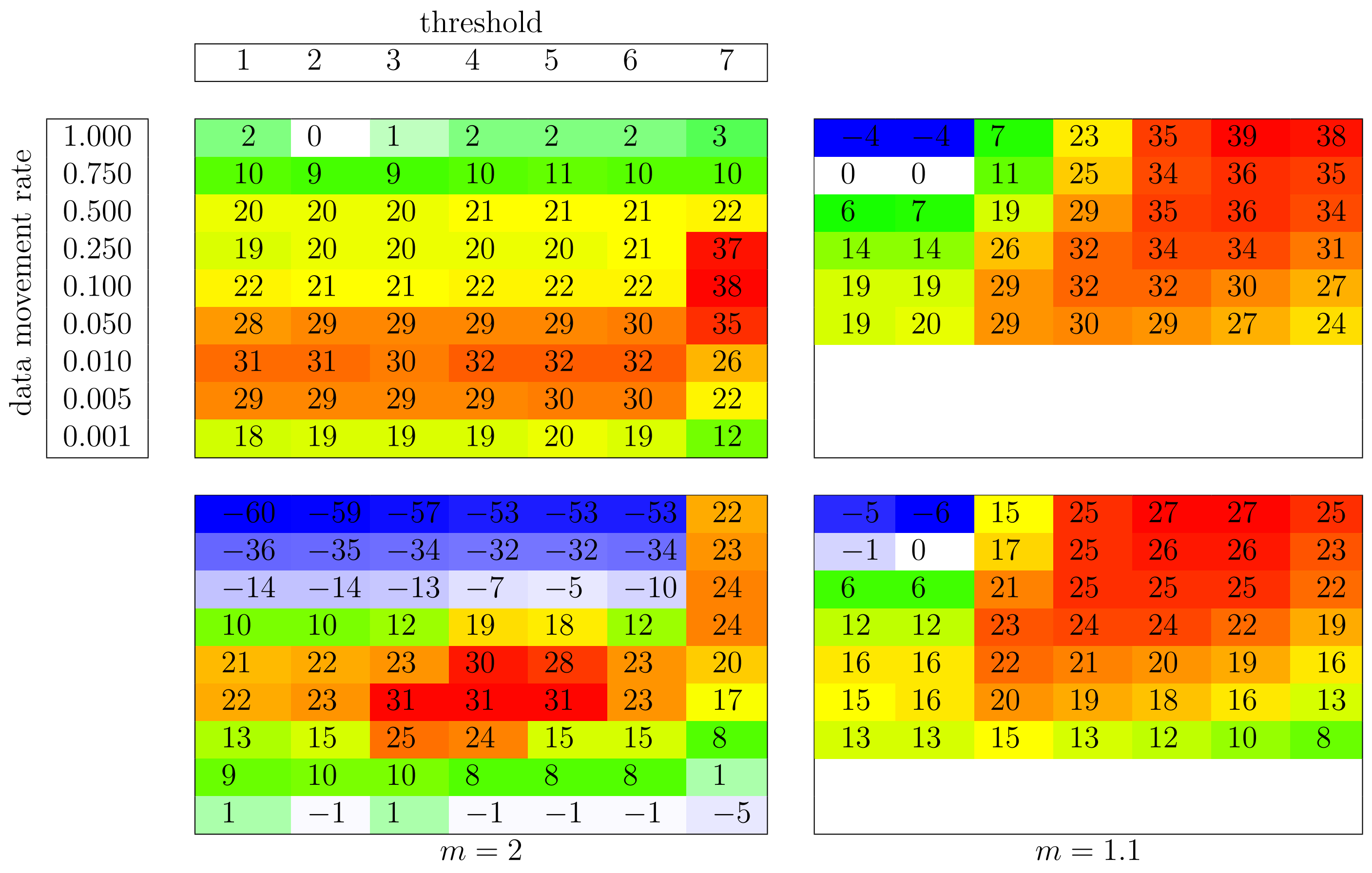


Figure 7.36: MList $v = 2^{19}$, all results. See §7.4.



$s = 0.1$, $m = 2$

data movement rate \ threshold	1	2	3	4	5	6	7
1.000	2	0	1	2	2	2	3
0.750	10	9	9	10	11	10	10
0.500	20	20	20	21	21	21	22
0.250	19	20	20	20	20	21	37
0.100	22	21	21	22	22	22	38
0.050	28	29	29	29	29	30	35
0.010	31	31	30	32	32	32	26
0.005	29	29	29	29	30	30	22
0.001	18	19	19	19	20	19	12

$s = 0.1$, $m = 1.1$

data movement rate \ threshold	1	2	3	4	5	6	7
1.000	−4	−4	7	23	35	39	38
0.750	0	0	11	25	34	36	35
0.500	6	7	19	29	35	36	34
0.250	14	14	26	32	34	34	31
0.100	19	19	29	32	32	30	27
0.050	19	20	29	30	29	27	24
0.010							
0.005							
0.001							

$s = 1$, $m = 2$

data movement rate \ threshold	1	2	3	4	5	6	7
1.000	−60	−59	−57	−53	−53	−53	22
0.750	−36	−35	−34	−32	−32	−34	23
0.500	−14	−14	−13	−7	−5	−10	24
0.250	10	10	12	19	18	12	24
0.100	21	22	23	30	28	23	20
0.050	22	23	31	31	31	23	17
0.010	13	15	25	24	15	15	8
0.005	9	10	10	8	8	8	1
0.001	1	−1	1	−1	−1	−1	−5

$s = 1$, $m = 1.1$

data movement rate \ threshold	1	2	3	4	5	6	7
1.000	−5	−6	15	25	27	27	25
0.750	−1	0	17	25	26	26	23
0.500	6	6	21	25	25	25	22
0.250	12	12	23	24	24	22	19
0.100	16	16	22	21	20	19	16
0.050	15	16	20	19	18	16	13
0.010	13	13	15	13	12	10	8
0.005							
0.001							

Figure 7.37: The effect of adjusting threshold and data movement rate for the DICT-MOVENODE benchmark using the embeddedOneSpace optimisation, and two values of s and two values of m. Data movement rate is expressed as proportion of lookup traversals that have data movement enabled. Threshold=7 corresponds to an empty line. The palette used is in Fig. 7.17.

Chapter 8

Conclusion

Recursive data structures provide great flexibility to the programmer, but their performance often depends on the location of RDS objects in memory. Even if the initial location of an object is good, data layout is likely to degrade as the RDS is used.

8.1 Contributions

This thesis is an exploration of three complementary optimisation techniques which use runtime data movement to prevent data layout degradation in two fundamental recursive data structures (sets of singly linked lists, and binary trees). Particular attention was paid to reducing the memory usage used by optimisations, and to reducing the latency of normal program work.

To maximise the number of different methods of runtime data movement investigated by this thesis, optimisations were applied by hand, rather than by a compiler. Nevertheless, two of the three techniques involve only small changes to a program, and we conclude that it is likely that tools could be built to greatly reduce the amount of work required of the programmer. The third technique provides valuable insight into the limitations of runtime data movement.

Below, we summarise the three techniques.

8.1.1 Reallocation

The technique of Chapter 4 is the insertion of data movement code into some subset of the points in a program which alter pointers in the RDS. This code moves a single object to the same line or page as one of a number of 'hint' objects identified by the applier of the optimisation. Reallocation produces significant performance improvements using only 35% extra memory[1].

This work is a development of previous work on memory 'coallocators' – memory allocators which can be used to improve heap data layout at allocation time only [15]. These functions can be easily adapted to move data at runtime, but we observe that better performance may be obtained with functions designed for runtime data movement. In particular, for dynamic usage, overhead is far more important, and so often a compromise

[1] $M = 1.35$, in notation of this thesis.

between layout quality and the difficulty of maintaining it must be made. For example, we may focus optimisation on the performance of the L1 and L2 cache (cache lines), even for a structure whose performance also depends heavily on TLB performance (pages).

8.1.2 Bulk data movement

The technique of Chapter 5 is the infrequent moving of large numbers of nodes, rather than the movement of individual nodes at pointer-update sites. The simplest optimisation of this form is to periodically move the entire RDS to a different area of memory, restoring layout to some known quality. For applications where extra memory usage can be as high as 100%, and where large infrequent pauses of program work are irrelevant, this is often the most effective runtime data movement optimisation.

The memory requirement of an optimisation may be reduced by re-laying out an RDS within the area of memory it currently occupies, rather than using a new area of memory. This incurs a small overhead, but allows extra memory usage to be reduced to as low as 12–74% (depending on the benchmark), with no further loss of performance. Further reduction in memory usage requires ‘compaction’ – the explicit movement of nodes to create emptier contiguous areas of memory, rather than to directly improve layout. This may be more effective carried out within an optimisation or at a program’s pointer-update sites, depending on the benchmark.

The latency may be reduced by incrementalising the movement of the RDS in a similar way to that used to produce an incremental garbage collector. Incrementalisation incurs either a write-barrier overhead when the RDS is moved to an empty area of memory, or a layout quality overhead when the RDS is moved within its original area of memory, but these overheads are not large. Latency may also be reduced by ‘embedding’ data movement code in a program’s existing traversal of the RDS. For some benchmarks this is far more effective than incrementalisation, even for only 25% extra memory. This is because merging the re-laying out of the RDS with a programs usage of the RDS reduces the data access overhead of runtime data movement, and also because the fine interleaving of data movement work and normal program work enables more effective compaction.

One might expect bulk to be always better than reallocation, because many of the overheads of runtime data movement can be reduced by moving nodes in small groups, but when memory usage and latency are taken into account, neither bulk data movement nor reallocation is universally superior.

8.1.3 Perfect data movement

The technique of Chapter 6 is similar to reallocation, in that data movement code is inserted into every point of the program that updates a pointer in the RDS, but differs in the number of nodes moved and the complexity of the code that is inserted. Specifically, after each pointer update a chosen data layout is ‘perfectly’ restored, unlike reallocation’s ad-hoc single-node movements, which produce a layout of unpredictable quality (although often good in practice).

As with previous optimisations we conclude that good performance depends on finding the best compromise between overhead and layout quality, observing that sub-optimal layouts are sometimes more effective, and furthermore reducing layout quality can be used to reduce memory usage.

Perfect data movement often outperforms reallocation, but may require a lot more memory. We may consider perfect data movement and reallocation as occupying opposite ends of a spectrum, and their relative performance provides insight into the performance of any optimisations that may lie in between – those that create a more predictable layout than reallocation, but are less complicated to implement than perfect data movement, and require less memory. For other benchmarks, bulk data movement is more effective than both reallocation and perfect data movement, suggesting that in some situations the overhead of performing single node movements at pointer-update sites is simply too high compared to bulk data movement's en-masse node movements.

8.1.4 Discussion

Although the work in this thesis has produced viable optimisations, it has also provided insight into other methods of rectifying or avoiding RDS layout degradation – using a layout-improving GC or a cache-aware RDS, respectively.

In comparison with a GC, the behaviour of the bulk data movement optimisation suggest that properly-tuned layout-improving GCs can be effective when memory is high and large pauses are allowed, but for tougher conditions, embedding data movement in existing traversals, or ad-hoc pointer update optimisations (e.g. reallocation) will be more effective.

The complexity of the code required for a perfect data movement optimisation is sufficiently high that it maybe be more sensible to regard it not as an optimisation but as a method of *synthesising* a new cache-aware RDS, particularly when combined with the removal of any pointers internal to cache blocks (which we did not consider in this thesis). Indeed, we find that perfect data movement applied to a binary search tree exceeds the performance of a well-tuned B-Tree. Even the opportunistic data movement of reallocation sometimes equals the performance of the B-Tree. It should be noted, however, that for a number of reasons the results of this thesis provide an upper bound on the performance that may be achieved using these techniques, rather than the typical case; a programmer should use a cache-aware structure if one exists.

We therefore prefer to view runtime data movement as a useful compromise between performance and difficulty of implementation. This behaviour is observed for the multiple linked-list benchmark – although cache-aware RDSs are about 1.5 times faster than the best bulk data movement or reallocation optimisation, the latter is between *5–20 times faster* than the unoptimised benchmark.

8.2 Summary

The application of runtime data movement to prevent the data layout degradation of recursive data structures has not been studied, apart from in a small number of layout-improving garbage collectors, where the emphasis is on practicality, not performance. RDS layout degradation may be avoided by use of a much more complex cache-aware structure. In this thesis, we have investigated the application of runtime data movement to simple RDSs, but unlike GCs have prioritised performance, and have on occasion come close to, or exceeded, the performance of cache-aware RDSs. The relative performance of the techniques of different complexity in this thesis suggest that, in general, more sophisticated data movement will result in better performance. In the future, as the cost of a cache miss continues to rise relative to cycle time, more sophisticated runtime data movement – implemented either explicitly, within a GC or an optimisation, or implicitly, as a new cache-aware RDS – will become both more practical and more worthwhile.

Appendix A

Derivations for Dict perfect data movement

This appendix contains derivations of results for perfect data movement for the Dict optimisation. See Chapter 6.

A.1 The cost of depth-zero reclustering

The derivation below gives the expected number of block accesses for a *complete* binary tree, when perfect data movement is used to restore layout by repairing all clusters below a pointer update. Clusters are rebuilt naïvely, i.e. by moving all nodes to an empty line, aka 'depth-zero' perfect data movement.

Consider a complete binary tree of height H, with clusters that are also complete and of height h. We assume that H is large and a multiple of h. Here we work out the cost of repairing the layout below uniformly distributed pointer updates. The probability p_k of updating a pointer at depth $k \in [0, H-1]$ is $\frac{2^k}{2^H-1}$. When a pointer is updated, the cluster its node is in is moved to a new line, and all clusters below are similarly rebuilt. During a cut or substitute operation, the tree remains well-clustered in the cluster below the one that contains the pointer update. The number of blocks below and including the i^{th} *cluster* from the top of the tree is $1 + 2^h + (2^h)^2 + \ldots + (2^h)^{(H/h-i)} = \frac{2^{H-hi}-1}{2^h-1}$. To repair the clusters will access twice that amount of blocks (source + destination blocks), plus at most one extra block if a node has been substituted. Including the i additional blocks accessed during traversal to the node whose pointer is updated, the expected amount of block transfers per pointer update is thus:

$$\sum_{i=0}^{\frac{H}{h}-1} \sum_{j=0}^{h-1} p_{hi+j} \left(\frac{2(2^{(H-hi)}-1)}{2^h-1} + 1 + i \right) = \frac{1}{2^H-1} \sum_i 2^{hi} \left(\left(i + 1 + \frac{2(2^{H-hi}-1)}{2^h-1} \right) \sum_{j=0}^{h-1} 2^j \right)$$

$$= \frac{2^h-1}{2^H-1} \sum_i i 2^{hi} + \frac{2^h-1}{2^H-1} \sum_i 2^{hi} + \frac{\frac{H}{h} 2^{H+1}}{2^H-1} - \frac{2}{2^H-1} \sum_i 2^{hi} = A + B + C + D$$

$$\sum_{i=0}^{n-1} ix^n = \frac{x(1-x^{n-1})}{(x-1)^2} + \frac{(n-1)x^n}{x-1} \Rightarrow A = \frac{2^h-1}{2^H-1}\left(\frac{2^h(1-2^{h(H/h-1)})}{(1-2^h)^2} - \frac{(H/h-1)2^H}{(1-2^h)}\right)$$

$$= \frac{2^h\left(1-2^{H-h}\right)}{(2^H-1)(2^h-1)} + \frac{(H/h-1)2^H}{2^H-1} \to \frac{1}{1-2^h} + (H/h-1) \quad \text{for large } H/h$$

$$B = \frac{2^h-1}{2^H-1}\frac{2^H-1}{2^h-1} = 1 \qquad C \to \frac{2H}{h} \quad \text{for large } H/h$$

$$D = -\frac{2}{2^H-1}\frac{2^H-1}{2^h-1} = \frac{2}{2^h-1}$$

Thus the expected number of block accesses *for uniformly distributed pointer updates* is $\frac{3H}{h} + \epsilon'$, for small ϵ'. The expected number of block accesses for the pointer updates due to the delete operation is therefore less than this, because (i) substitutions are uniformly distributed, (ii) cuts are below uniformly distributed locations in the tree.

Insertion always takes at most $\frac{H}{h} + 1$ accesses ($\frac{H}{h}$ for the traversal, and 1 for the possible creation of a new cluster). Thus the number of blocks accessed for an insert/delete pair is no more than $\frac{4H}{h} + \epsilon$, for small ϵ. □

A.2 DICT perfect data movement, worst-case memory

Below we derive the value of m_{worst} for perfect data movement for DICT using the **BFS** and **fixedHeight** layouts.

A.2.1 BFS

Assume each cluster lives in its own block, which can store at most 7 nodes. Let $c(T,k)$ be the number of clusters with k nodes in tree T, and $n(T)$ be the number of nodes in the tree T, and $m(T)$ be the memory used by tree T, where $m(T) = (c(T,1) + \ldots + C(T,7))/(n(T)/7)$.

Firstly we show that for any tree T there is another tree T' with $c(T',i) \leq 5, \forall i \in [2,6]$ and $n(T) = n(T')$ and $m(T') \geq m(T)$. Given the tree T, the tree T' is produced as follows. Nodes are moved around the tree to change the size of clusters. E.g. any two 6-clusters can be changed into a 7-cluster with five 1-clusters attached to it:

6+6	$\to$	7+1+1+1+1+1
5+5	$\to$	7+1+1+1
4+4	$\to$	7+1
3+3+3	$\to$	7+1+1
2+2+2+2+2+2	$\to$	7+1+1+1+1+1

This rearrangement of nodes from 2-,3-,4-,5-,6-clusters is valid because such clusters do not have children, and because the number of 1's produced is less than 8 (the maximum number of possible children of the cluster with 7 nodes). Note that the total number of clusters does not decrease, hence the memory usage does not decrease.

Therefore, a tree T of maximal memory usage has the property $n(T) = 7c(T,7) + c(T,1) + \delta$, where δ is the insignificant number of nodes that live in clusters with > 1 and < 7 nodes.

Note that $c(T,1) \leq 8c(T,7)$, because (i) each 1-cluster must have a parent cluster if $n(T) > 1$ and (ii) only 7-clusters can have child clusters and (iii) each 7-cluster has at most 8 child clusters. Using $n(T) = 7c(T,7) + c(T,1) + \delta$ we get $c(T,1) \leq \frac{8(n(T)-\delta)}{15} \leq \frac{8(n(T)}{15}$. The memory allowance $m(T)$ is given by:

$$\frac{c(T,1)+c(T,7)}{n/7} = \frac{c(T,1)+(n-c(T,1))/7}{n(T)/7} = \frac{n(T)+6c(T,1)}{n(T)} \leq 4.2$$

A complete tree of height $3k+1$, has $c(T,7) = (8^k - 1)/(8-1)$, and $c(T,1) = 8^k$, and $n(T) = 2^{3k+1} = 2 \cdot 8^k$, giving $m(T) = \frac{(8^k-1)/7+8^k}{n(T)/7} = \frac{8^{k+1}-1}{2 \cdot 8^k} \approx 4$.

Thus m_{worst} is in $[4, 4.2]$. □

A.2.2 fixedHeight

The derivation is similar to above. Recall that clusters are of fixed maximum height 3. Let $C(T,i,j), i \in [1,7], j \in \{0,2,4,6,8\}$ be the number of clusters that have i nodes and *can have* at most j child clusters (denote this an (i,j)-cluster). Valid clusters are: $(1,0),(2,0),(3,0),(3,2),(4,2),(4,4),(5,4),(6,6),(7,8)$.

We rearrange nodes so the tree has only an significant number of $(1,0)$ and $(7,8)$ clusters:

$$\begin{array}{rcl}
4 \times (2,0) & \to & (7,8) + (1,0) \\
3 \times (3,0) & \to & (7,8) + 2 \times (1,0) \\
3 \times (3,2) & \to & (7,8) + 2 \times (1,0) \\
2 \times (4,2) & \to & (7,8) + (1,0) \\
2 \times (4,4) & \to & (7,8) + (1,0) \\
2 \times (5,4) & \to & (7,8) + 3 \times (1,0) \\
4 \times (6,6) & \to & 3 \times (7,8) + 3 \times (1,0)
\end{array}$$

Note as for **BFS** that number of nodes stays the same, and the number of possible child clusters doesn't decrease, and the number of clusters doesn't decrease. Now we may write $n(T) = 7C(T,7,8) + C(T,1,0) + \delta$, for some small δ.

Note that the tree and clusters are now the same shape as in **BFS**'s derivation, thus giving m_{worst} is also in $[4, 4.2]$ also. In both cases, memory is maximised when clusters with as many clusters with seven nodes and clusters with one node are used, which are the same shape for both clustering methods, giving the same memory bounds. □

Bibliography

[1] Agaram, K. K., and Keckler, S. W. The memory behavior of data structures in C SPEC CPU2000 benchmarks. `http://www.spec.org/Workshops/2006/papers/14_akkartik-spec06.pdf`, 2006.

[2] Aggarwal, A., and Jeffrey, S. V. The input/output complexity of sorting and related problems. *Communications of the ACM 31*, 9 (1988).

[3] Annavaram, M., Patel, J. M., and Davidson, E. S. Data prefetching by dependence graph precomputation. *SIGARCH Computer Architecture News 29*, 2 (2001).

[4] Badawy, A.-H. A., Aggarwal, A., Yeung, D., and Tseng, C.-W. Evaluating the impact of memory system performance on software prefetching and locality optimizations. In *Proceedings of the 15th International Conference on Supercomputing* (2001).

[5] Belady, L. A study of replacement algorithms for a virtual storage computer. *IBM Systems Journal 5*, 2 (1966).

[6] Bender, M., Cole, R., Demaine, E., and Farach-Colton, M. Scanning and traversing: Maintaining data for traversals in memory hierarchy. In *Proceedings of the Annual European Symposium on Algorithms* (2002).

[7] Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P. W., Wies, T., and Yang, H. Shape analysis for composite data structures. In *Proceedings of the 19th International Conference on Computer Aided Verification* (2007).

[8] Berger, E. D., Zorn, B. G., and McKinley, K. S. Reconsidering custom memory allocation. In *Proceedings of the 17th Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2002).

[9] Bodin, F., Kisuki, T., Knijnenburg, P., O'Boyle, M., and Rohou, E. Iterative compilation in a non-linear optimisation space. In *Proceedings of the Workshop on Profile and Feedback Directed Compilation* (1998).

[10] Cahoon, B., and McKinley, K. S. Data flow analysis for software prefetching linked data structures in java. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques* (2001).

[11] Calder, B., Krintz, C., John, S., and Austin, T. Cache-conscious data placement. *SIGPLAN Notices 33*, 11 (1998).

[12] Chen, W., Bhansali, S., Chilimbi, T., Gao, X., and Chuang, W. Profile-guided proactive garbage collection for locality optimization. In *Proceedings of the 27th Conference on Programming Language Design and Implementation* (2006).

[13] Cheney, C. J. A nonrecursive list compacting algorithm. *Communications of the ACM 13*, 11 (1970).

[14] Chilimbi, T. M., Davidson, B., and Larus, J. R. Cache-conscious structure definition. *SIGPLAN Notices 34*, 5 (1999).

[15] Chilimbi, T. M., Hill, M. D., and Larus, J. R. Cache-conscious structure layout. *SIGPLAN Notices 34*, 5 (1999).

[16] Chilimbi, T. M., and Hirzel, M. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the 23rd Conference on Programming language design and implementation* (2002).

[17] Chilimbi, T. M., and Larus, J. R. Using generational garbage collection to implement cache-conscious data placement. In *Proceedings of the 1st International Symposium on Memory Management* (1998).

[18] Chilimbi, T. M., Larus, J. R., and Hill, M. D. Improving pointer-based codes through cache-conscious data placement. Tech. Rep. CS-TR-98-1365, University of Wisconsin-Madison, 1998.

[19] Chow, K., and Wu, Y. Feedback-directed selection and characterization of compiler optimizations. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture, Second Workshop on Feedback-Directed Optimization* (November 1999).

[20] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. *Introduction to Algorithms*, first ed. MIT Press, 1990.

[21] Courts, R. Improving locality of reference in a garbage-collecting memory management system. *Communications of the ACM 31*, 9 (1988).

[22] Feng, Y., and Berger, E. A locality-improving dynamic memory allocator. Tech. Rep. TR09-05, Department of Computer Science, University of Massachusetts Amherst, 2005.

[23] Fix, J. D. The set-associative cache performance of search trees. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (2003).

[24] Franz, M., and Kistler, T. Splitting data objects to increase cache utilization. Tech. Rep. 98-34, Department of Information and Computer Science, University of California, Irvine, 1998.

[25] Frias, L., Petit, J., and Roura, S. Lists revisited: Cache-conscious stl lists. In *Proceedings of the 5th International Workshop on Experimental Algorithmics* (2006).

[26] Gay, D., and Aiken, A. Memory management with explicit regions. *SIGPLAN notices 33*, 5 (1998).

[27] Gay, D., and Aiken, A. Language support for regions. *SIGPLAN notices 36*, 5 (2001).

[28] Hallberg, J., Palm, T., and Brorsson, M. Cache-conscious allocation of pointer-based data structures revisited with hw/sw prefetching. In *Proceedings of the 2nd Annual Workshop on Duplicating, Deconstructing, and Debunking, in conjunction with the 30th International Symposium on Computer Architecture* (2003).

[29] Hennessy, J. L., and Patterson, D. A. *Computer Architecture: A Quantitative Approach*, second ed. Morgan Kaufmann, 1996.

[30] Henning, J. SPEC CPU2000. In *IEEE Computer* (2000).

[31] Hertz, M., and Berger, E. D. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the 20th Annual Conference on Object Oriented Programming, Systems, Languages, and Applications* (2005).

[32] Hill, M. D., and Smith, A. J. Evaluating associativity in CPU caches. *IEEE Transactions on Computers 38*, 12 (1989).

[33] Hinton, G., Sager, D., Upton, M., Boggs, D., Carmean, D., Kyker, A., and Roussel, P. The microarchitecture of the pentium 4 processor. *Intel Technology Journal* (Q1, 2001).

[34] Hirzel, M., Henkel, J., Diwan, A., and Hind, M. Understanding the connectivity of heap objects. In *Proceedings of the 3rd International Symposium on Memory Management* (2002).

[35] Huang, X., Blackburn, S., McKinley, K., Moss, E., Wang, Z., and Cheng, P. The garbage collection advantage: improving program locality. In *Proceedings of the 19th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2004).

[36] Inagaki, T., Onodera, T., Komatsu, H., and Nakatani, T. Stride prefetching by dynamically inspecting objects. In *Proceedings of the 24th Conference on Programming language design and implementation* (2003).

[37] Intel. AP-485 Intel processor identification and the CPUID instruction. `http://www.intel.com/design/processor/applnots/241618.htm`.

[38] Intel. How to choose between hardware and software prefetch on 32-Bit Intel architecture. `http://software.intel.com/en-us/articles/how-to-choose-between-hardware-and-software-prefetch-on-32-bit-intel-architecture`.

[39] Jones, R., and Lins, R. *Garbage Collection: Algorithms for automatic dynamic memory management.* Wiley, 1999.

[40] Karlsson, M., Dahlgren, F., and Stenstrom, P. A prefetching technique for irregular accesses to linked data structures. In *Proceedings of the 6th International Conference on High Performance Computer Architecture* (2000).

[41] Kistler, T., and Franz, M. Automated data-member layout of heap objects to improve memory-hierarchy performance. *ACM Transactions on Programming Languange Systems 22*, 3 (2000).

[42] Ladner, R. E., Fix, J. D., and LaMarca, A. Cache performance analysis of traversals and random accesses. In *Proceedings of the ACM SIAM Symposium on Discrete Algorithms* (1999).

[43] Lam, M. S., Wilson, P. R., and Moher, T. G. Object type directed garbage collection to improve locality. In *Proceedings of the International Workshop on Memory Management* (1992).

[44] Lattner, C., and Adve, V. Automatic pool allocation for disjoint data structures. In *Proceedings of MSP '02: 2002 Workshop on Memory System Performance* (2002).

[45] Luk, C.-K., and Mowry, T. C. Compiler-based prefetching for recursive data structures. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems* (1996).

[46] Luk, C.-K., and Mowry, T. C. Memory forwarding: enabling aggressive layout optimizations by guaranteeing the safety of data relocation. In *Proceedings of the 26th Annual International Symposium on Computer Architecture* (1999).

[47] Luk, C.-K., Muth, R., Patil, H., Weiss, R., Lowney, P. G., and Cohn, R. Profile-guided post-link stride prefetching. In *Proceedings of the 16th International Conference on Supercomputing* (2002).

[48] Manegold, S. Cache calibrator (v0.9e), a cache-memory and TLB calibration tool. `http://monetdb.cwi.nl/Calibrator/calibrator.shtml`.

[49] Mehta, D. P., and Sahni, S. *Handbook of Data Structures and Applications*. Chapman and Hall, 2005.

[50] Moon, D. A. Garbage collection in a large LISP system. In *Proceedings of the ACM Symposium on LISP and Functional Programming* (1984).

[51] Novark, G., Strohman, T., and Berger, E. D. Custom object layout for garbage-collected languages. Tech. Rep. UM-CS-2006-007, University of Massachusetts, Amherst, 2006.

[52] Petrank, E., and Rawitz, D. The hardness of cache conscious data placement. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages* (2002).

[53] Rabbah, R. M., Sandanagobalane, H., Ekpanyapong, M., and Wong, W.-F. Compiler orchestrated prefetching via speculation and predication. *SIGOPS Operating Systems Review 38*, 5 (2004).

[54] Raman, E., and August, D. I. Recursive data structure profiling. In *Proceedings of the Workshop on Memory System Performance* (2005).

[55] Rao, J., and Ross, K. A. Cache conscious indexing for decision-support in main memory. In *Proceedings of the 25th International Conference on Very Large Data Bases* (1999).

[56] Rao, J., and Ross, K. A. Making B+ trees cache conscious in main memory. *SIGMOD Record 29*, 2 (2000).

[57] Rogers, A., Carlisle, M., Reppy, J., and Hendren, L. Supporting dynamic data structures on distributed memory systems. In *ACM Transactions on Programming Languages and Systems* (1995).

[58] Roth, A., Moshovos, A., and Sohi, G. S. Dependence based prefetching for linked data structures. *SIGOPS Operating Systems Review 32*, 5 (1998).

[59] Roth, A., and Sohi, G. S. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 8th International Conference on Architectural Symposium on Computer Architecture* (1999).

[60] Rubin, S., Bernstein, D., and Rodeh, M. Virtual cache line: A new technique to improve cache exploitation for recursive data structures. In *Proceedings of the 8th International Conference on Compiler Construction* (1999).

[61] Sair, S., and Charney, M. Memory behavior of the SPEC2000 benchmark suite. Tech. rep., IBM T. J. Watson Research Center, 2000.

[62] Seidl, M. L., and Zorn, B. Predicting references to dynamically allocated objects. Tech. Rep. CU-CS-826-97, Department of Computer Science, University of Colorado at Boulder, 1997.

[63] Seidl, M. L., and Zorn, B. Segregating heap objects by reference behavior and lifetime. In *Proceedings of the 8th International Conference on Architectural support for programming languages and operating systems* (1998).

[64] Seidl, M. L., and Zorn, B. Low cost methods for predicting heap object behavior. In *Proceedings of the 2nd Workshop on Feedback Directed Optimization* (1999).

[65] Seidl, M. L., and Zorn, B. Implementing heap-object behavior prediction efficiently and effectively. *Software Practice and Experience 31*, 9 (2001).

[66] Shuf, Y., Gupta, M., Franke, H., Appel, A., and Singh, J. P. Creating and preserving locality of java applications at allocation and garbage collection times. In *Proceedings of the 17th Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2002).

[67] Stoutchinin, A., Amaral, J. N., Gao, G. R., Dehnert, J. C., Jain, S., and Douillet, A. Speculative prefetching of induction pointers. In *Proceedings of the 10th International Conference on Compiler Construction* (2001).

[68] TOFTE, M. A brief introduction to regions. In *ISMM '98: Proceedings of the 1st International Symposium on Memory Management* (1998).

[69] TRUONG, D. Considerations on dynamically allocated data structure layout optimization. In *Proceedings of the Workshop on Profile and Feedback Directed Compilation* (1998).

[70] TRUONG, D., BODIN, F., AND SEZNEC, A. Accurate data layout into blocks may boost cache performance. In *Proceedings of the 2nd Workshop on Interaction between Compilers and Computer Architecture* (1997).

[71] TRUONG, D. N., BODIN, F., AND SEZNEC, A. Improving cache behavior of dynamically allocated data structures. In *Proceedings of the 6th International Conference on Parallel Architectures and Compilation Techniques* (1998).

[72] WHITE, J. L. Address/memory management for a gigantic LISP environment or, GC considered harmful. In *Proceedings of the ACM Conference on LISP and Functional Programming* (1980).

[73] WILSON, P. R. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management* (1992).

[74] WILSON, P. R., LAM, M. S., AND MOHER, T. G. Effective static-graph reorganization to improve locality in garbage-collected systems. In *Proceedings of the 12th Conference on Programming Language Design and Implementation* (1991).

[75] WU, Y. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. *SIGPLAN notices 37*, 5 (2002).

[76] WU, Y., SERRANO, M. J., KRISHNAIYER, R., LI, W., AND FANG, J. Value-profile guided stride prefetching for irregular code. In *Proceedings of the 11th International Conference on Compiler Construction* (2002).

[77] ZHANG, H., AND MARTONOSI, M. A mathematical cache miss analysis for pointer data structures. In *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing* (2001).

List of terms

This list gives the section in which a term is defined.