LEABHARLANN CHOLÁISTE NA TRÍONÓIDE, BAILE ÁTHA CLIATH | TRINITY COLLEGE LIBRARY DUBLIN
Ollscoil Átha Cliath | The University of Dublin

Terms and Conditions of Use of Digitised Theses from Trinity College Library Dublin

Copyright statement

All material supplied by Trinity College Library is protected by copyright (under the Copyright and Related Rights Act, 2000 as amended) and other relevant Intellectual Property Rights. By accessing and using a Digitised Thesis from Trinity College Library you acknowledge that all Intellectual Property Rights in any Works supplied are the sole and exclusive property of the copyright and/or other IPR holder. Specific copyright holders may not be explicitly identified. Use of materials from other sources within a thesis should not be construed as a claim over them.

A non-exclusive, non-transferable licence is hereby granted to those using or reproducing, in whole or in part, the material for valid purposes, providing the copyright owners are acknowledged using the normal conventions. Where specific permission to use material is required, this is identified and such permission must be sought from the copyright holder or agency cited.

Liability statement

By using a Digitised Thesis, I accept that Trinity College Dublin bears no legal responsibility for the accuracy, legality or comprehensiveness of materials contained within the thesis, and that Trinity College Dublin accepts no liability for indirect, consequential, or incidental, damages or losses arising from use of the thesis for whatever reason. Information located in a thesis may be subject to specific use constraints, details of which may not be explicitly described. It is the responsibility of potential and actual users to be aware of such constraints and to abide by them. By making use of material from a digitised thesis, you accept these copyright and disclaimer provisions. Where it is brought to the attention of Trinity College Library that there may be a breach of copyright or other restraint, it is the policy to withdraw or take down access to a thesis while the issue is being resolved.

Access Agreement

By using a Digitised Thesis from Trinity College Library you are bound by the following Terms & Conditions. Please read them carefully.

I have read and I understand the following statement: All material supplied via a Digitised Thesis from Trinity College Library is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of a thesis is not permitted, except that material may be duplicated by you for your research use or for educational purposes in electronic or print form providing the copyright owners are acknowledged using the normal conventions. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone. This copy has been supplied on the understanding that it is copyright material and that no quotation from the thesis may be published without proper acknowledgement.

Practical and Architectural Aspects of Sorting and Searching

Thesis submitted for the degree of Doctor in Philosophy

Submitted: September 2009
Defended: December 2009
Approved: January 2010

Nicholas C.A. Nash, B.A. (Mod.)

Declaration

This thesis has not been submitted as an exercise for a degree at this or any other university. It is entirely the candidate's own work. The candidate agrees that the Library may lend or copy the thesis upon request. This permission covers only single copies made for study purposes, subject to normal conditions of acknowledgement.

Thesis 9

Summary

The work in this dissertation was motivated by a desire to understand as well as to improve the performance of certain algorithms and data structures in practice. The performance of algorithms in practice is influenced by many interacting factors, of which one of the most important is the architecture of the machine where the algorithm or data structure is used. In this dissertation we consider architectural as well as other practical factors influencing the performance of sorting algorithms and data structures.

We begin by studying experimentally the interaction between branch prediction and sorting algorithms. Even for simple and otherwise well-understood algorithms such as insertion sort, selection sort and bubble sort we show they have very different branch prediction characteristics. Our experiments lead to a simple analysis showing that these algorithms cause asymptotically different numbers of branch mispredictions on average. We continue to analyse more efficient classic algorithms such as quicksort, mergesort and heapsort. We show through simple experiments and their analysis that quicksort has, in a certain sense, the most desirable branch prediction properties of the efficient algorithms. This demonstrates for the first time a new strength of the classic, popular algorithm.

A problem closely related to the sorting problem is searching. We continue by studying dynamic, ordered data structures experimentally, paying attention to their interaction with architectural features such as branch prediction and caching. We consider traditional comparison-based search trees as well as less commonly used data structures that do not rely solely on key comparisons to operate. We note that such data structures cause far fewer branch mispredictions as a result, and show that in circumstances where cache performance is not dominant, branch mispredictions can be. As a result of our experimental study, we propose a new, hybrid data structure for integer keys based on the burst trie from the string sorting literature. We show that this data structure requires less space and, on a wide range of test data, operates more efficiently than all the other data structures considered. Our experiments include traditional search trees such as a red-black tree, B-tree, as well as data structures

specialized to integer keys such as a stratified (van Emde Boas) tree and a hybrid trie structure inspired by the Willard's q-fast trie. We validate our results on different architectures, compilers and data sets (both synthetic and naturally occuring) showing that our $LPCB$-trie data structure performs well in practice.

In an additional contribution, still grounded in the realm of providing algorithms that perform well in practice, we consider the maximum independent set problem for a class of graph known as circle graphs, which have a number of applications. We compare and improve the performance of the two best approaches and finish by presenting the first output-sensitive algorithm for the problem. We then validate the performance of our algorithm experimentally showing that it performs substantially faster than previous approaches.

Acknowledgements

The support of a number of people enabled me to complete this dissertation. It is a great pleasure to finally thank my supervisor David Gregg. David is an ideal supervisor: always available, full of enthusiasm, ideas, and much useful advice.

I am also very grateful to Sid Touati and Albert Cohen for organizing my very pleasant three month visit to *INRIA*-Saclay. This visit provided me with a wonderful learning experience.

All my collaborators provided generosity in time and ideas, for which I am very grateful. I would like to especially thank Julian Seward for interesting discussions, as well as his assistance in extracting data sets from Valgrind for our work on integer data structures, which proved interesting and important. I also thank Julian for his patience: an implementation will come!

I am very lucky to have had great office-mates over the years, many of whom are now also my friends. I am sure that I will miss the regular discussions and friendly atmosphere created by such pleasant people. Many of my other good friends from High School have helped me through the tougher times of research, whether they are aware of it or not.

I am enormously grateful to my family for unquestioning support. To my sisters Elise and Mary, and to my brother Oliver. Oliver deserves a special mention: he taught me to program. His many hours of instruction were also great fun. To my parents, I can only say that I owe you debts of time, support, kindness and love that I can never repay.

Table of Contents

1 Introduction **1**

1.1 Background and Motivation 1

1.2 Our Thesis 2

1.3 Previously Published Work 3

1.4 Dissertation Structure and Contributions 3

2 Background **6**

2.1 Instruction Level Parallelism 6

2.1.1 Pipelining 6

2.1.2 Branch Prediction 8

2.2 Memory Hierarchies 11

2.2.1 Caches 12

2.2.2 Virtual Memory 13

2.3 Notation 14

3 Sorting and Branch Prediction **16**

3.1 Introduction 16

3.2 Elementary Sorting Algorithms 19

3.2.1 Branch Prediction Model 19

3.2.2 Insertion sort 22

3.2.3 Selection sort 24

3.2.4 Bubble sort 25

3.2.5 Discussion 29

3.3 Efficient Sorting Algorithms 32

3.3.1 Comparing Quicksort, Mergesort and Heapsort 32

3.4 Related Work 39

3.4.1 Skewed Quicksort 41

3.4.2 Insertion Mergesort 42

3.5 Conclusion 45

4 Comparing Integer Data Structures 48
4.1 Introduction 48
4.2 Burst Tries 51
4.3 Engineering Burst Tries 53
4.3.1 In-Node Data Structures 54
4.3.2 Bucket Data Structures 56
4.3.3 Level and Path Compressed Tries 58
4.3.4 Engineering Level Compression 60
4.3.5 Operations 62
4.4 Experimental Comparison 65
4.4.1 Burst Trie Configuration 65
4.4.2 Burst Trie Comparison 79
4.5 Related Work 99
4.6 Future Work 104
4.7 Conclusion 106

5 Maximum Independent Sets in Circle Graphs 108
5.1 Introduction 108
5.2 Background 109
5.3 Experimental Setup 111
5.4 Overview of Algorithms 112
5.4.1 Supowit's Algorithm 112
5.4.2 More Efficient Approaches 113
5.5 Apostolico et al's Algorithm 114
5.5.1 Set Construction 116
5.5.2 Optimizations 118
5.5.3 Type-I Results 120
5.5.4 Type-II Results 122
5.6 Valiente's Algorithm 123
5.6.1 Optimizations 129
5.6.2 Type-I Results 130
5.6.3 Type-II Results 133
5.7 Comparison 137
5.7.1 Type-I Results 137
5.7.2 Type-II Results 141
5.8 Register Allocation Results 142
5.9 Faster Algorithms 143
5.9.1 Introducing Ouput Sensitivity 143

5.9.2 A Combined Algorithm 148
5.9.3 Experimental Comparison 150
5.10 Conclusion and Future Work 154

6 Final Thoughts **157**
6.1 Why Experiment? 157
6.2 Future Directions 158
6.3 Theory and Practice 159
6.4 Conclusion 161

Bibliography **161**

A Additional Experimental Results for Integer Data Structures **175**
A.1 Alternative Machine Configurations 175
A.2 Growth Factor Results 184

Chapter 1

Introduction

1.1 Background and Motivation

The work in this dissertation was motivated by a desire to understand as well as to improve the performance of certain algorithms and data structures *in practice.*

The performance of algorithms in practice is influenced by many interacting factors. A key factor is architectural. There are certain architectural features that have become so wide-spread that it is important that the behaviour of implementations of the basic algorithms of Computer Science be understood when executing on machines with these architectural features. Some of these architectural features, such as memory bottlenecks and caching have long been recognized in an algorithmic context. For example, the External Memory Model of Aggarwal and Vitter [1988] provided a simplified model of computation to take memory operations into account, more recently the Cache Oblivious Model of Frigo *et al.* [1999] has attempted to provide a robust model for analysing the cache performance of algorithms.

Others architectural features, such as the interactions of algorithms with instruction level parallelism (ILP) and specifically branch prediction, have only been studied for the first time much more recently, [Biggar and Gregg 2005; Brodal and Moruz 2005] despite ILP's enduring presence in our machines.

Another important factor influencing the performance of algorithms and data structures in practice is simply their *implementation*, some constructions used in algorithms and data structures are extremely difficult to produce practical realizations of. An example is the stratified tree of van Emde Boas [1977]. This is a dynamic, ordered data structure offering all operations in $O(\log w)$ worst-case time, where the keys are w-bits in length. The first general purpose implementation of this data structure that out-performed comparison-based search trees was the work of Dementiev *et al.* [2004] –

nearly 40 years after the data structure was proposed. Even then, the implementation is quite limited and requires a perhaps impractical amount of space, details of which can be found in Chapter 4 of this dissertation.

The desire to genuinely understand the behaviour of algorithms and data structures in practice has given rise relatively recently to the discipline of Algorithm Engineering. This discipline is driven by the desire to provide efficient algorithms, as indeed is the classical and extremely successful theory of algorithms. The goal in Algorithm Engineering however is typically focused on achieving an implementation of an algorithm that performs well on real machines. This performance is validated through experimentation, and as a result often leads to the term Experimental Algorithmics being used to also describe this field of study. It is within this field of study that the work in this dissertation falls. Details of some of the successes and contributions of Algorithm Engineering can be found in the excellent recent survey of Sanders [2009]. Moret [2001] and McGeoch [2008; 2007] also discuss the field.

1.2 Our Thesis

We believe that practical and architectural aspects of algorithms and data structures in Computer Science can be effectively studied in an experimental manner, yielding important insights into their behaviour and providing insight into their performance.

To investigate our thesis, this dissertation provides a study of the behaviour of sorting and searching algorithms on real machines. We experimentally study comparison-based sorting algorithms and their interaction with branch predictors, an architectural feature of almost every current computer. We then consider the performance of data structures whose operations do not rely solely on comparisons. This study of data structures leads us to compare traditional comparison-based data structures with purely integer data structures – performing no comparisons between keys – as well as with hybrid data structures. This study shows that the cache performance of these data structures is often the most important factor influencing their performance in practice, although other factors such as branch prediction characteristics are occasionally dominant.

In an additional experimental study, we examine the performance of a graph algorithm with a number of practical applications: finding a maximum independent set in a circle graph. In this case, the construction of efficient implementations of these algorithms leads us to propose new *output sensitive* algorithms for this problem.

We outline the contributions of these studies in the following Section 1.4.

1.3 Previously Published Work

Portions of this dissertation have appeared in work that has been previously published. Chapter 3 is based on a paper published with Paul Biggar, Kevin Williams and David Gregg:

- An Experimental Study of Sorting and Branch Prediction, *ACM Journal of Experimental Algorithmics* 12:1.8 (June 2008).

Paul Biggar did the initial experiments for this paper and documented them in a technical report [Biggar and Gregg 2005]. My contribution to this work was a substantial number of additional experiments, together with the analysis of the experimental results.
Chapter 4 is based on two papers published with David Gregg:

- Comparing Integer Data Structures for 32 and 64-bit Keys, *ACM Journal of Experimental Algorithmics*, Special issue devoted to Selected papers from WEA 2008 (to appear).

- Comparing Integer Data Structures for 32 and 64-bit Keys, in Proceedings of WEA 2008 (7th International Workshop on Experimental Algorithms, Provincetown, Cape Cod, MA, USA, 30 May - 2 June 2008), C. McGeogh (ed.). LNCS 5038, pp28–42. Springer, 2008.

Chapter 5 is based on a paper published with Sylvain Lelait and David Gregg:

- Efficiently Implementing Maximum Independent Set Algorithms on Circle Graphs, *ACM Journal of Experimental Algorithmics*, 13:1.9 (February 2009).

In this paper, Sylvain Lelait's contribution was the correction of the error in Apostolico *et al.*'s [1992; 1993] algorithm. My contributions were the experimental evaluation of Apostolico *et al.*'s algorithm, Valiente's [2003] algorithm, and the $\Theta(n \min\{n, \alpha \log n\})$ time output sensitive algorithm (and its $\Theta(n \min\{n, \alpha\})$ refinement), together with its experimental evaluation.

1.4 Dissertation Structure and Contributions

The material in this dissertation is organized as follows:

Chapter 2 provides background material necessary for the material described in subsequent chapters. This material principally comprises a description of the architectural features that have important algorithmic implications. For readers familiar with basic computer architecture, this chapter serves to simply set the context, and provide terminology as well a small amount of notation used throughout this dissertation.

Chapter 3 explores the interaction between classical sorting algorithms and *branch prediction*. We find that the branch prediction characteristics of sorting algorithms such as insertion sort, selection sort and bubble sort are very different, despite the fact that these algorithms are simple and otherwise well-understood. In the case of efficient sorting algorithms, such as quicksort, heapsort and mergesort, we show for the first time another strength of quicksort: its branches are naturally more predictable than the other algorithms.

Chapter 4 explores the data structures for searching that do not rely solely on *comparisons* between input keys. Despite the widespread popularity of comparison-based data structures based around search trees, we show that a data structure obtained by carefully engineering a *burst trie* data structure performs excellently in practice. We show experimentally that our engineered data structure requires less space than many alternative data structures, as well as often having better cache behaviour. In addition we note that, in situations where cache misses do not dominate performance, the small number of branch mispredictions incurred by our data structure improves its performance relative to traditional search trees.

Chapter 5 presents a contribution separate from the preceding chapters, but firmly within the remit of Algorithm Engineering. We first present an experimental comparison of two algorithms that are equally efficient (asymptotically) for finding a maximum independent set of a circle graph. These are the algorithms of Apostolico *et al.* [1992; 1993] and Valiente [2003]. Despite operating in the same asymptotic time, Apostolico *et al.*'s algorithm seems – on paper – to be far less efficient than Valiente's. We show that with a suitable improvements it out-performs an implementation of Valiente's algorithm. We then focus on developing a highly efficient variation of Valiente's algorithm, and begin by describing a variant that is several times faster on average than a direct implementation of the algorithm. This culminates in a description of an *output sensitive* algorithm for finding a maximum independent set in an unweighted circle graph. Previous algorithms for this problem operated $\Theta(n^2)$ time for an n vertex dense circle

graph, we describe an algorithm operating in time $\Theta(n \min\{n, \alpha \log n\})$ where α is the independence number of the circle graph. We also present experiments demonstrating that our algorithm is highly efficient in practice, and a sketch of a more complicated algorithm with slightly better asymptotic behaviour.

Chapter 2

Background

In this chapter, we provide an account of the relevant background material necessary for this dissertation. The material in this chapter is principally an account of the basic computer architecture that we consider the algorithmic implications of in this dissertation. The account we provide is very brief and focuses architectural concepts rather than their implementation details. Further details can be found in texts on computer architecture [Hennessy and Patterson 2006; Zargham 1996]. For readers familiar with this computer architecture the present chapter serves to simply introduce some terminology and remind the reader of a small amount of standard notation.

2.1 Instruction Level Parallelism

Since the 1960s*, computer processors have attempted to speed up computations of sequences of instructions by executing independent parts of the sequence simultaneously, or *in parallel*. This is referred to as *instruction level parallelism*.

2.1.1 Pipelining

In this section we provide an overview of a technique for achieving instruction level parallelism known as *pipelining*. Suppose the execution of an instruction for adding two operands such as ADD X, Y, Z can be divided into 5 phases:

1. Instruction fetch (IF): The operation code (op-code) for the instruction is retrieved from memory.

*The first general purpose pipelined processor was the IBM "Stretch" 7030 introduced in 1961, Hennessy and Patterson [2006] provide a brief historical account.

Instruction number	Cycle 1	2	3	4	5	6	7	8	9
1	IF	ID	OF	EX	WB				
2		IF	ID	OF	EX	WB			
3			IF	ID	OF	EX	WB		
4				IF	ID	OF	EX	WB	
5					IF	ID	OF	EX	WB

Table 2.1: A 5-stage pipeline: the stages are instruction fetch (IF), instruction decode (ID), operand fetch (OF), execution (EX) and write-back (WB). The parallelism is visible vertically in the diagram. For instance, at cycle number 5, 5 distinct instructions are being processed.

2. Instruction decode (ID): The op-code is translated by the processor into a particular instruction (for example, an integer add instruction).

3. Operand fetch (OF): The necessary operands for this instruction are retrieved from memory.

4. Execution (EX): The instruction is executed upon its operands.

5. Write-back (WB): The result of the instruction is recorded, or "written back" to a register or memory.

Assuming each phase takes a single cycle, 5 cycles will be required in total to process an instruction. The idea of pipelining is to allow one instruction to be in each of these phases of execution. When the first instruction in a sequence finishes its IF phase a second instruction can immediately take its place. As a result, at best, a new instruction can be issued on every cycle instead of once every 5 cycles. Figure 2.1.1 shows an illustration of a *full* pipeline, where one instruction is in each phase of execution, or *pipeline stage*. With pipelining, the latency to complete an instruction remains 5 cycles, however, the throughput is increased by 5 times: one instruction can complete on each cycle.

In practice, there are many complications in the implementation of pipelining. These complications are often referred to as hazards. There are several important classes of hazard such as structural hazards, data hazards and control hazards. Structural hazards result when instructions are unable to execute in parallel due to an over-demand for hardware resources. Data hazards are caused when the results of a previous instruction in the pipeline are required by a currently executing instruction. In both cases, the pipeline must stall – delaying the execution of certain instructions

```
if(a[i] < a[min_idx])
{
    min_idx = i;
}
i++;
```

Figure 2.1: Some *C* code demonstrating a control hazard.

in order to resolve the hazard. In this section, we focus on control hazards because they are the most relevant to the material in Chapters 3 and 4.

Control hazards are the result of instructions that change the flow of control in a program. In order to maintain a full pipeline the next instruction in the sequence of instructions to be executed must be available. However, in general, it is simply not possible to know what the next instruction is, as the *C* code in Figure 2.1 demonstrates. In Figure 2.1, which instruction will follow the execution of the comparison, the assignment or the increment? We refer to the class instruction generated by a construct like the `if` statement above as a branch instruction. If the condition tested by the branch is true, then we say the branch is taken, otherwise we say the branch is not-taken. When a branch instruction is encountered, a simple solution is to simply stall the pipeline until it is known whether the branch is taken or not-taken. However, in order to maintain the utilization of the pipeline at a reasonable level, processors employ some form of prediction in the outcome of branch instructions. We explore these techniques in the following section.

2.1.2 Branch Prediction

We refer to the dedicated hardware in a processor that attempts to predict the direction of a branch as a *branch predictor*. Consider the *C* code of Figure 2.1. We regard the repeated execution of the `if` statement as resulting in the processing of a branch instruction that presents a stream of values to the branch predictor

$$\text{T}, \text{T}, \text{NT}, \text{T}, \text{T}, \text{T}, \text{NT} \ldots$$

Here T denotes a taken branch, whereas NT denotes a not-taken branch. The purpose of a branch predictor is to provide the next value in this sequence. The processor then fetches the next instruction from the address predicted by the branch predictor without needing to stall the pipeline. When the branch prediction provides an incorrect prediction we say a *branch misprediction* has occurred. This can be very costly:

potentially causing the computation of all other instructions in the pipeline to be abandoned, to allow the processor to fetch instructions from the actual target of the branch instruction. As an extreme example, the Intel Pentium 4 [Hinton et al. 2001] Prescott processor had an instruction pipeline of 31 stages, making branch mispredictions very expensive.

The prediction techniques for branch instructions can be divided into three classes: static, semi-static and dynamic. Static prediction schemes produce a prediction that is independent of the history of the branch. When the first pipelined processors appeared, a typical approach was for a static prediction scheme was to simply do nothing and continue fetching instructions [McFarling and Hennessey 1986]. An example of more accurate static technique is to predict forward branches as not-taken and backward branches as taken. This static prediction technique is motivated by the programming construct of a loop: programmers tend to use loops for computations that execute several times, thus, the static predictor will only predict wrongly when the loop exits. Semi-static branch predictors support a *hint* bit, that allows the compiler to provide the predicted direction of the branch. The IBM CELL SPE [Eichenberger et al. 2006] is an example of a modern processor using semi-static branch prediction.

Almost all modern desktop processors use dynamic branch prediction schemes. A simple dynamic branch prediction technique is to maintain a small table of counters indexed by the low-order bits of the addresses of branch instructions. Each time a branch instruction is executed, its corresponding counter is looked-up in the table and used to make a prediction. Many branch instructions may correspond to the same index in the prediction table, and thus their behaviour may interfere with each other. The mapping of two or more distinct branches to the same index in the prediction table is known as aliasing, and generally reduces prediction accuracy. These counting predictors were described by Smith [1981].

The number of bits in the counters in the table play an important role in the way the prediction scheme operates. In a so-called 1-bit scheme, the branch is predicted to follow the direction it followed on its previous execution. 1-bit predictors achieve between 77% and 79% accuracy, these statistics as well as those in the following paragraphs are taken from those derived by Uht *et al.* [1997] over the SPECint92 benchmarking suite. Typical table sizes are $2^{12} = 4096$ counters.

Adding more bits to the counters can improve accuracy. A 2-bit predictor, also commonly referred to as a bimodal predictor, maintains a table of 2-bit counters. The counter associated to a particular branch is incremented each time the branch is taken and decremented each time it is not-taken, saturating at the minimum and maximum

```
for(i = 0; i < n; i++)
{
    if(i & 1)
    {
        do_odd_processing();
    }
    finish_processing();
}
```

Figure 2.2: Some *C* code providing a motivating example for a correlating branch predictor compared to a table of 2-bit predictors.

values. When the leading bit of the counter is equal to zero, the branch is predicted not-taken, otherwise, the branch is predicted as taken. Although in theory n-bit counters could be used, the improvement they offer over 2-bit counters is not enough to justify their use [Smith 1981; Hennessy and Patterson 2006]. 2-bit predictors with table sizes of 4096 achieve 78% to 89% accuracy [Uht et al. 1997].

More accurate dynamic branch predictors can be obtained by exploiting correlations in branch outcomes. A *two-level adaptive* predictor [Pan et al. 1992] maintains a *branch history register*. This register records the outcomes of previous branch instructions. For example, a register contents of `110010` indicates that the last 6 branch outcomes were taken, taken, not-taken, not-taken, taken and not-taken. A typical length for a history register is 10-12 bits. For each branch, this register is used to index a table of bimodal predictors. The address of the branch instruction can also be included in this index, either concatenated or XORed with it. Two-level adaptive predictors have typical accuracies of about 93% [Uht et al. 1997]. A simple example of the strength of a two-level adaptive predictor can be seen in Figure 2.2. The `if` statement is taken on every second iteration of the loop. For a bimodal branch prediction table simply indexed by the low order bits of the branch instruction address, this would cause the bimodal predictor to oscillate between predicting taken and not-taken, resulting in approximately 50% accuracy. With a two-level adaptive predictor the history register will have a history register ending in `0` or `1`, allowing the odd and even iterations of the loop to be distinguished, and for the branch to be predicted with close to 100% accuracy.

There are many more complicated schemes for branch prediction than those discussed above. However, the above provides the necessary conceptual background for the algorithmic considerations surrounding branch prediction. We also again emphasize that the account of branch prediction provided above is greatly simplified, and

neglects many of its other interactions with pipelining. An excellent, although still relatively high-level account of branch prediction and other micro-architectural details of the Alpha 21264 processor is provided by Kessler *et al.* [1998].

It is noteworthy that the prediction schemes described above were devised in an experimental manner, strongly guided by the available hardware resources. However, prediction problems such as this have received many decades of attention. Predicting the values of sequences is at the heart of many data compression techniques such as Prediction by Partial Match (PPM) [Bell et al. 1990] or Context Tree Weighting [Willems et al. 1995], and less explicitly, in dictionary-based data compression techniques such as LZ77 and the large family of related algorithms [Ziv and Lempel 1977; Bell et al. 1990]. It appears that correlating branch predictors described in this section were developed independently of data compression techniques, and it is worth noting that the correlating predictors described above are quite similar to the finite-context modelling technique of PPM data compressors. More recently, data compression algorithms, especially PPM, have been used to construct extremely accurate branch predictors in software. Due to their complexity, they are far from suitable for hardware implementations however [Srinivasan et al. 2007].

2.2 Memory Hierarchies

In this section we consider another aspect of the design of computers that has algorithmic implications: the memory hierarchy. Typically processors require data at a speed much faster than main memory can provide it. This would place an extremely seriously bottleneck on the speed at which computations could be performed due to the difference in speed between main memory and the CPU were it not for the *principle of locality.* This is an experimental observation about the behaviour of programs noting that programs have a strong tendency to reference "nearby" data, or data within their "locality". Here the locality can be *spatial* or *temporal.* Spatial locality refers to the fact that if a program accesses memory address ADDR then it is likely to also access other memory addresses with a small difference from ADDR also. Temporal locality refers to the fact that programs tend to reuse values that they have recently accessed. Early accounts of locality of reference (in the context of virtual memory systems, see Section 2.2.2.) were provided by Belady [1966] and Denning [1968]. Denning [2005] describes a historical account of the principle's development. This locality of reference that programs exhibit motivated the introduction of *cache memories*, or caches.

2.2.1 Caches

A cache is a small, fast memory that holds a subset of the values in main memory. When a program makes a request for a value from memory, the value can either be retrieved rapidly from the cache – a cache hit, or retrieved from main memory and stored in the cache for later use – a cache miss. The cache is designed to exploit the principle of locality resulting in an access time to memory that is close to that of the cache rather than that of main memory. Wilkes [1965] provides the earliest description of the idea, under the name of "slave memory". The abstract of his paper [Wilkes 1965, 2000] reads

> The use is discussed of a fast memory of, say, 32 000 words as a slave to a slower core memory of, say, one million words in such a way that in practical cases the effective access time is nearer that of the fast memory than that of the slow memory.

Shortly afterwards, the term "cache" was adopted in favour of "slave memory" [Liptay 1968].

When a word of memory is required by a program, caches exploit spatial locality by also loading surrounding words of memory into the cache. The total number of words transfered in a single transfer between main memory and cache is refered to as the cache block size or cache line size. On current desktop machines, typical cache line sizes are 32, 64 or 128 bytes†. Caches exploit temporal locality by prioritizing the eviction of cache lines, typically using an approximation to a Least Recently Used (LRU) scheme. That is, when a cache line must be evicted (since the cache is smaller than main memory) the cache line that was made use of the most distantly in the past is evicted.

In general there is not just a single cache, but generally at least two. This gives rise to what is referred to as the *memory hierarchy*. The fastest memory can be regarded as the processor registers, followed by the first cache (referred to as the level 1 cache), followed by the level 2 cache, main memory, and then magnetic and solid-state storage devices. As this hierarchy of memories is descended away from the processor, the size and latency increase, while cost decreases. On current desktop machines (see Table 4.1 and Section 4.4 for details), level 2 cache sizes of 2MB or 4MB are common, compared to main memory that is often 4GB or larger. Registers can typically be accessed by the processor within a single cycle. In Chapter 4 we present experimental results for

†These cache line lengths can be measured using a tool such as `cpuid`, see `http://www.cpuid.org`. Hennessy and Patterson [2006] also provide some example lengths for certain processors.

machines where we have measured a level 2 cache hit (or level 1 cache miss) to cost up to 20 processor cycles, and a level 2 cache miss to cost up to 200 processor cycles. See Section 4.4 for details.

There are many implementation details involving caches that we omit, since the simple discussion above is enough to appreciate the algorithmic considerations concerning memory access. We finish by noting that the need for a processor to access a large, fast memory was recognized as early as 1946 in the design of computer systems [Burks et al. 1946; Burks 1989].

> Ideally one would desire an indefinitely large memory capacity such that any particular word would be immediately available. We are forced to recognize the possibility of construct a hierarchy of memory, each of which has greater capacity than the preceding but which is less quickly accessible.

2.2.2 Virtual Memory

Computer systems tend to allow the processor to serve several processes at once, and so it is important that processes can utilize the main memory in a simple, efficient manner. It is also important that processes be protected from one another, so that one process may never access memory belonging to another. An additional concern faced when executing a process is the total amount of memory it requires – what should be done if the process requires more memory than there is main memory available to it?

Virtual memory was proposed as a technique to manage these concerns. Virtual memory was introduced in 1972 for the IBM System/370 computers, described by Case and Padegs [1978]. In a virtual memory system, each process is regarded as executing in and having access to a contiguous region of memory whose size can exceed the amount of main memory available, this is called the *virtual address space* of the process. The processor together with the operating system (OS) manage the mapping from the *virtual addresses* used by the process to the *physical addresses* required to access the main memory (and other memory) that is physically available.

The virtual memory system views memory as consisting of fixed sized contiguous regions called *pages*. When the process requires memory, the processor and OS provide it with page-sized regions from the physical memory. The mapping of memory addresses used by the process for pages of memory, i.e. the *virtual addresses*, to the physical address to which the page corresponds is managed via an in-memory data structure called the *page table*. For example, with a 4GB $= 2^{32}$ byte virtual address space, and 4KB $= 2^{12}$ byte page size, the page table requires 2^{20} entries. This page table is stored

in main memory. As a result, a memory access by a process first requires a main memory access to look-up the page table to perform the virtual address to physical address mapping, followed by a memory access to retrieve the desired data. This is an unacceptable overhead for the virtual memory system, and as a result, another cache is introduced, referred to as the *translation lookaside buffer*[‡] or TLB. The TLB is a small cache of frequently accessed entries of the page table. When a virtual memory address requires an entry of the page table not in the TLB (a TLB miss) the page table in main memory must be accessed, otherwise, the physical address retrieved from the TLB can be requested directly. We have measured the penalty of a TLB miss to be approximately 20 cycles on some of the machines we present experimental results for, Section 4.4 provides details.

As with our discussions of pipelining, branch prediction and caching above, we omit many important details of virtual memory design since the details above are enough for the algorithmic considerations in this dissertation.

2.3 Notation

We use only the most well-known standard asymptotic notation in this dissertation, following the definitions of standard texts [Cormen et al. 2001]. For a function $f : \mathbb{R} \to \mathbb{R}$, we use the usual definitions for upper, lower and tight bounds on that function:

- We denote by $O(g(n))$ the set of functions such that there exist positive constants c, n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.
- We denote by $\Omega(g(n))$ the set of functions such that there exist positive constants c, n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$.
- We denote by $\Theta(g(n))$ the set of functions such that there exist positive constants c_1, c_2, n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$.

We also occasionally make use of the standard stronger lower and upper bounds:

- We denote by $o(g(n))$ the set of functions such that for any positive constant c there exists a positive constant n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.
- We denote by $\omega(g(n))$ the set of functions such that for any positive constant c there exists a positive constant n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$.

[‡]This term was introduced by Case and Padegs [1978] in their description of the virtual memory system of the IBM System/370 computers.

We use $\lg(n)$ to denote $\log_2(n)$. We denote the nth harmonic number as $H_n = \sum_{k=1}^{n} 1/k = \ln n + \gamma + O(1/n)$, where $\gamma = 0.577\ldots$ is Euler's constant [Knuth 1997a].

Chapter 3

Sorting and Branch Prediction

3.1 Introduction

The sorting problem can be stated as follows: Given as input a sequence $\sigma_1, \ldots, \sigma_n$, provide as ouput a permutation of that sequence such that $\sigma'_1 \leq \sigma'_2 \leq \ldots \leq \sigma'_n$. Sorting is a fundamental problem that is one of the most well studied in Computer Science. Many good algorithms are known that offer trade-offs in efficiency, simplicity, memory use and other factors. Classical analyses of these algorthms, such as the RAM model used for establishing asymptotic bounds, or Knuth's MIX machine code make drastically simplifying assumptions about the cost of different machine instructions [Knuth 1997b].

More recently it has been recognized that on modern computers the cost of accessing memory can vary dramatically depending on whether the data can be found in the first-level cache, or must be fetched from a lower level of cache or even main memory. This has spawned a great deal of research on cache-efficient searching and sorting [Nyberg et al. 1994; Agarwal 1996; LaMarca and Ladner 1996, 1997; Xiao et al. 2000; Rahman and Raman 2001; Wickremesinghe et al. 2002; Frigo et al. 1999].

Another type of instruction whose cost can vary dramatically is the conditional branch. Modern pipelined processors depend on *branch prediction* to improve their performance. If the direction of a conditional branch is correctly predicted ahead of time, the cost of the conditional branch may be as little as the cost of, say, an integer add instruction. If, on the other hand, the branch is mispredicted the processor must flush its pipeline, and restart from the correct target of the branch.

Fortunately, the branches in most programs are very predictable, so branch mispredictions are usually rare. Indeed, prediction accuracies of greater than 90% are typical [Uht et al. 1997] with the best predictors.

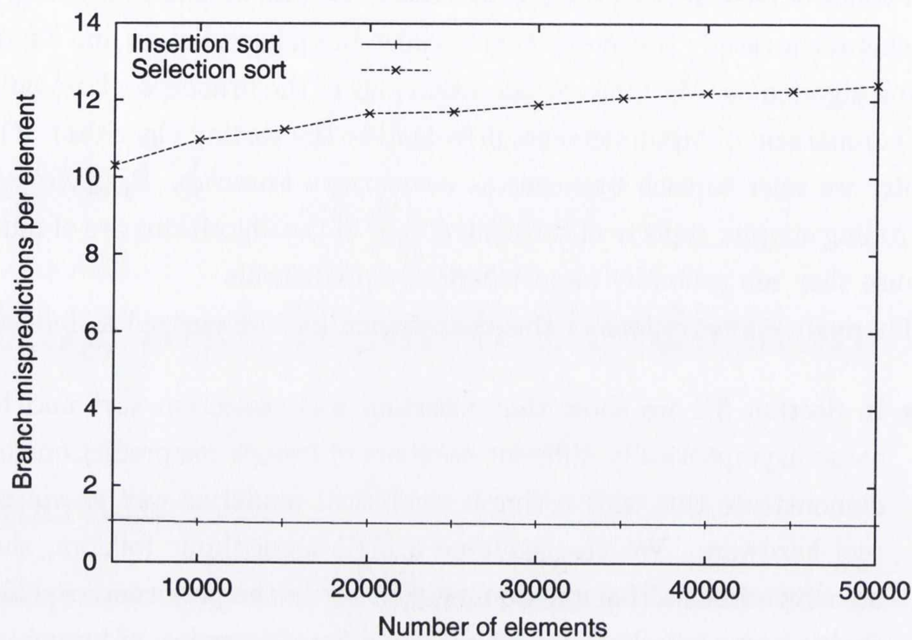
The cost of executing branches is particularly important for sorting because the inner-loops of most sorting algorithms consist of comparisons of items to be sorted. In this chapter we study the interaction between branch prediction and a number of classic sorting algorithms. We focus on the behaviour of the branches whose outcome depends on a comparison of input elements presented to the sorting algorithm. Throughout this chapter we refer to such branches as *comparison branches.* Branches associated with controlling simpler aspects of the control flow of the algorithms are of much less interest because they are generally almost perfectly predictable.

The main contributions of this chapter can be summarized as follows:

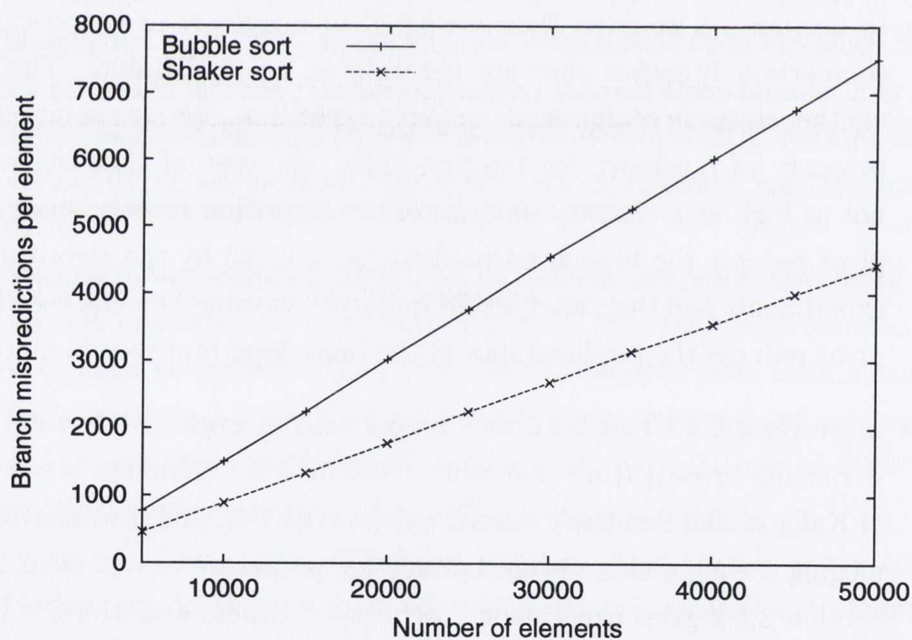
- In Section 3.2 we show that insertion sort, selection sort and bubble sort all cause asymptotically different numbers of branch mispredictions on average. We demonstrate this with a simple analytical model as well as via experiments on real hardware. We also examine a little algorithmic folklore, showing by suitable experiments that the dominant factor in the performance difference between shaker sort and selection sort is their different number of branch mispredictions.

- In Section 3.3 we show that compared to mergesort or heapsort, quicksort has comparison branches that are naturally more predictable. This demonstrates another strength of the classic, popular algorithm. We also point out an appealing property of quicksort for the first time: the cost of choosing a poor pivot is not as high as a classical analysis of the algorithm reveals, since an unbalanced pivot reduces the branch mispredictions incurred by the algorithm. We present experiments and their analysis for quicksort showing how the use of a median-of-3 pivot reduces the predictability of the inner-loop branches in quicksort.

- In Sections 3.4.1 and 3.4.2 we consider related work and perform additional experiments investigating that work. Section 3.4.1 examines a practical variation of Kaligosi and Sanders's skewed quicksort [2006], and provides experiments cataloging the algorithm's branch prediction properties as well as its spatial locality. Section 3.4.2 gives experimental results for Brodal and Moruz's insertion d-way mergesort algorithm [2005]. We provide experiments demonstrating their technique improves the spatial locality of mergesort while simultaneously improving its branch prediction properties, finally we note their algorithm has an analogue in terms of quicksort.



(a)



(b)

Figure 3.1: This figure shows the number of branch mispredictions caused by some classic $\Theta(n^2)$ worst-case time sorting algorithms, while operating on uniform random data. The data points are the number of branch mispredictions caused by a particular input, divided by the number of elements in that input. These results were gathered on an Intel Core 2 Duo 2.13GHz processor using PAPI [Dongarra et al. 2003].

3.2 Elementary Sorting Algorithms

We begin by presenting experimental results for a number of classic elementary sorting algorithms: insertion sort, selection sort and bubble sort. Knuth [1998a] provides a very detailed account of these algorithms. Each of these algorithms executes $\Theta(n^2)$ branch instructions on average (and at worst), and indeed their implementations all consist of a very simple pair of nested loops. Surprisingly, Figure 3.1 shows that they clearly cause very different numbers of branch mispredictions.

Figure 3.1(a) shows that insertion sort causes just a single branch misprediction per element. Figure 3.1(a) also shows that selection sort causes a substantially larger number of branch mispredictions, although the rate of growth is slow. Figure 3.1(b) shows the number of branch mispredictions per element incurred by bubble and shaker sort. The number of mispredictions for both algorithms grows in a similar fashion. Both algorithms also cause enormously more branch mispredictions than selection and insertion sort.

Motivated by these experimental results, we now provide a number of simple calculations giving the average number of branch mispredictions caused by these algorithms. In particular, we show that despite the fact that each algorithm executes $\Theta(n^2)$ branches, somewhat surprisingly they cause asymptotically different numbers of branch mispredictions. In particular:

- Insertion sort causes $\Theta(n)$ branch mispredictions.
- Selection sort causes $\Theta(n \log n)$ branch mispredictions.
- Bubble sort causes $\Theta(n^2)$ branch mispredictions.

These results correspond to the experimental results of Figure 3.1. In the following section we outline the assumptions used to derive these results, and the general approach used in our analysis of these algorithms.

3.2.1 Branch Prediction Model

Section 2.1 discussed instruction level parallelism and branch prediction from a hardware point of view. In this section, we describe the model used when analysing the experimental results presented in this chapter.

We assume the repeated execution of a *C* statement like `if(x < y)` presents a stream of values to the branch predictor, such as

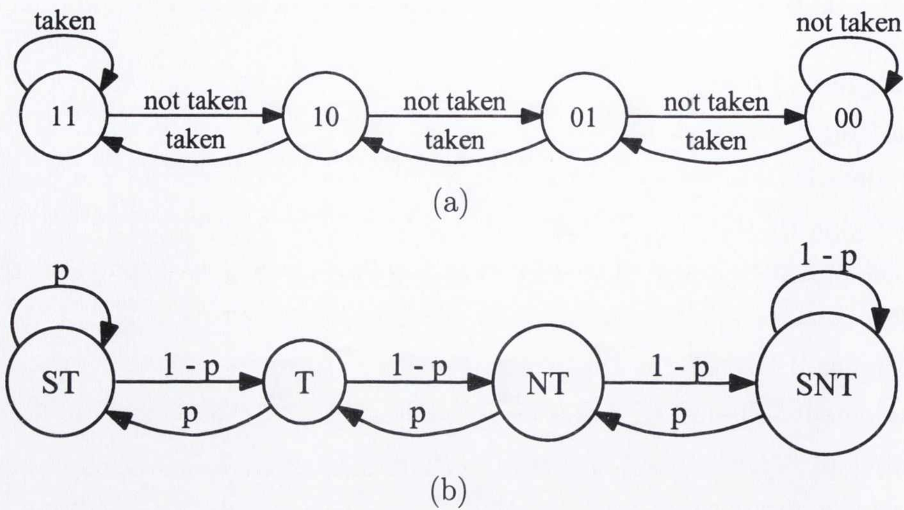


Figure 3.2: (a) Shows a 2-bit saturating counter predictor. When the predictor is in state 00 or 01 the associated branch is predicted not-taken. When the predictor is in state 10 or 11 the associated branch is predicted taken. (b) Shows the Markov chain associated to the predictor, where the branch has a probability p of being taken. Here we name the four states as follows: strongly taken (ST), taken (T), not taken (NT) and strongly not taken (SNT).

$$\text{T}, \text{T}, \text{NT}, \text{T}, \text{T}, \text{T}, \text{NT} \ldots \tag{3.1}$$

Here T denotes a taken branch, and NT denotes a not-taken branch. We neglect aliasing effects described in Section 2.1, and assume there is a dedicated branch predictor for each branch instruction. The job of the branch predictor is to predict the next value of this sequence.

The experiments considered in this chapter are designed to reveal the average number of branch mispredictions caused by sorting algorithms. By average, we mean assuming the usual average case model in sorting where each input permutation of $\{1, \ldots, n\}$ is equally likely [Cormen et al. 2001]. As a result, our experiments are performed over uniform random inputs to the sorting algorithms and our analysis of experimental results is performed for the case of a simple bimodal predictor rather than more complex predictors that are designed to exploit dependence between branch executions (see Section 2.1). Moreover, we regard Sequence 3.1 as a sequence of Bernoulli trials [Feller 1968] with p being the (unknown) probability of observing T.

Under these assumptions, the optimal strategy to predict a whether the next branch is taken or not is to simply keep a counter, incrementing it for each T observed, and decrementing it for each NT observed. The sign of the counter is then used to predict

the next value in the sequence. After n executions of the branch storing such a counter requires $\lceil \lg n \rceil$ bits. In hardware, the number of bits in this counter is fixed, and the counter *saturates* at its minimum and maximum value. The result of this saturation is of course the simple counting branch predictors described in Section 2.1. Figure 3.2(a) shows a saturating 2-bit branch predictor. An early examination of this classic problem of estimating a probability with finite memory is provided by Leighton and Rivest [1983]. A discussion of the problem in the context of branch prediction is given by Michaud [2004].

After repeated execution of a branch instruction, the probability that a saturating 2-bit predictor correctly predicts a branch instruction converges to

$$C_{2bit}(p) = \frac{3p^2 - 3p + 1}{2p^2 - 2p + 1} \tag{3.2}$$

We refer to the function C_{2bit} as the *characteristic function* of the branch predictor. This function is easily derived by modelling the 2-bit predictor associated to a branch instruction as a Markov chain, shown in Figure 3.2(b). The transition matrix of this Markov chain is

$$M = \begin{bmatrix} p & p & 0 & 0 \\ 1-p & 0 & p & 0 \\ 0 & 1-p & 0 & p \\ 0 & 0 & 1-p & 1-p \end{bmatrix}$$

The characteristic function can be derived by solving for the steady state probabilities of the Markov chain, i.e. the vector s such that $Ms = s$. These steady state probabilities are those resulting from a random walk in a Markov chain with reflecting barriers, and is considered in the general case in standard texts [Feller 1968]. If the saturating counter has $2k$ bits, then its characteristic function is given by [Michaud 2004]:

$$C_k(p) = p - \frac{2p - 1}{1 + (\frac{p}{1-p})^k}$$

Independent of k, if $p = 1/2$ then $C_k(p) = 1/2$. For $p > 1/2$, the value of $C_k(p)$ converges to p as $k \to \infty$, and for $p < 1/2$ the value of $C_k(p)$ converges to $1 - p$ as $k \to \infty$. Thus, if the number of bits in the counter is not limited, then the probability of a correct prediction is given by

$$C_{perfect}(p) = \max\{p, 1 - p\}$$

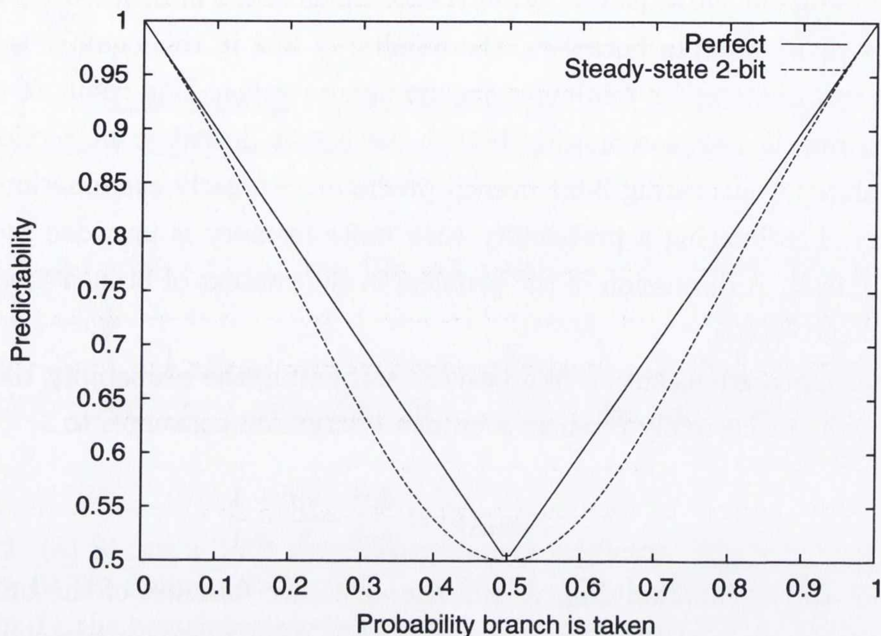


Figure 3.3: This figure shows the characteristic functions of a perfect static branch predictor and a 2-bit saturating counter predictor. The values of these functions at a particular probability is the probability the branch with that probability will be correctly predicted, which is what we refer to as the branch's *predictability*.

We refer to a (fictional) branch predictor with this characteristic function as a *perfect static* predictor. Figure 3.3 shows a plot of $C_{perfect}$ and C_{2bit}, showing the approximation that C_{2bit} provides to $C_{perfect}$. We will make use of both of these functions when analysing the experimental results presented below. In general, the analyses in the following sections all follow the same very simple pattern. We begin by making use of the assumption that all inputs are equally likely to derive the probability that a particular branch will be taken. We then make use of this probability assuming either a perfect static or two-bit predictor, giving the expected number of branch mispredictions on average.

3.2.2 Insertion sort

Insertion sort is a simple, classic, $\Theta(n^2)$ worst (and average) case time sorting algorithm. Here we show insertion sort causes $\Theta(n)$ branch mispredictions. The inner loop of insertion sort, operating on an input `a[0..n-1]` is

```
item = a[k];
while(item < a[k - 1])
{
   a[k] = a[k - 1];
   k--;
}
a[k] = item;
```

This loop is iterated $n - 2$ times over the input.* When executing the loop for a particular value of k, $1 < k < n$, `a[0..k - 1]` is already sorted, allowing `a[k]` to be placed in the correct position as shown in the inner-loop above. As Figure 3.1(a) shows, insertion sort has excellent branch prediction characteristics. Examining the inner-loop code above it is easy to see why this is so. The comparison branch `item < a[k - 1]` is only not-taken exactly once in each inner loop iteration. Intuitively, the single untaken execution causes the single branch misprediction per element, as observed in Figure 3.1(a).

Although the analysis here is very simple, we give the details since the analysis of our experimental results in the following sections follows the same pattern. Let Q_l^k denote the probability that the branch comparing `a[l]` to `a[l - 1]`, $1 < l \leq k$, is not-taken in the inner loop iteration that is positioning `a[k]`, $1 < k < n$. Defining $B(k)$ as the average number of mispredictions caused by the execution of the inner-loop when positioning `a[k]`, we have:

$$B(k) = \sum_{l=2}^{k} \min\{Q_l^k, 1 - Q_l^k\} \tag{3.3}$$

Here we assume a perfect static branch predictor (as discussed in Section 3.2.1), which causes a misprediction with probability $\min\{p, 1-p\}$ for a branch that is taken (or not-taken) with probability p. Since we assume all inputs equiprobable, in each execution of the inner-loop `item` is equally likely to belong in any of `a[1]`, ..., `a[k]` — causing the corresponding branch to be not-taken — we have simply $Q_l^k = 1/k$.

The total number of branch mispredictions incurred by insertion sort is thus simply $\sum_{k=1}^{n-1} B(k)$, which is clearly $\Theta(n)$ and explaining the single branch misprediction per element observed in Figure 3.1(a).

*We assume for simplicity that `a[0]` initially contains the minimum element.

3.2.3 Selection sort

Selection sort is another simple sorting algorithm operating in $\Theta(n^2)$ average, best and worst-case time. The inner loop of selection sort operating on an input `a[0..n-1]` is

```
min = k;
for(l = k + 1; l < n; l++)
   if(a[l] < a[min]) min = l;
swap(a[k], a[min]);
```

Here $0 \leq k < n-1$ is the outer loop counter. Figure 3.1(a) shows the experimentally observed branch mispredictions for selection sort. The number of branch mispredictions incurred by selection sort can be analyzed simply as we now sketch, the analysis is a simple variation on that provided by Knuth [1997b], adapted to the context of branch prediction.

The comparison branch `a[l] < a[min]` is taken only if `a[l]` is the minimum of `a[k..l]`. Thus, in any given inner-loop iteration, the lth execution of the comparison branch is taken with probability $1/(l+1)$ [Knuth 1997b]. As with insertion sort in the previous section, we assume a perfect static branch predictor. The expected number of branch mispredictions $B(k)$ on inner-loop iteration k is then

$$B(k) = \sum_{l=1}^{n-k} \min\left\{\frac{1}{l+1}, 1 - \frac{1}{l+1}\right\} \quad (3.4)$$

$$= H_{k+1} - 1 \quad (3.5)$$

The total branch mispredictions on average is then simply $\sum B(k)$, that is

$$M(n) = \sum_{k=1}^{n-1}(H_{k+1} - 1) \quad (3.6)$$

$$= (n+1)H_{n+1} - 2n - 1 \quad (3.7)$$

Here $H_n = \sum_{i=1}^{n} 1/i$ is the nth harmonic number. Since $H_n = \Theta(\log n)$ we have $M(n) = \Theta(n \log n)$. The quantity $M(n)$ here is in fact essentially the average number of changes to the right-to-left maxima in the input, provided by Knuth [1998b].

Thus although selection sort executes $\Theta(n^2)$ branches it causes $\Theta(n \log n)$ mispredictions, with respect to a perfect static predictor. Re-examining Figure 3.1(a), the logarithmic growth rate in the mispredictions per element is evident (consider for example

the data points at $n = 10000, 20000, 40000$). We see that for $n = 50000$ selection sort causes approximately 12 branch mispredictions on real hardware, naturally the analysis for a perfect static predictor slightly underestimates this, indeed $M(50000) \approx 9$.

3.2.4 Bubble sort

Bubble sort is another simple sorting algorithm, although notoriously inefficient. Bubble sort works works by iterating the loop shown below at most $n-1$ times on an input `a[0..n - 1]`.

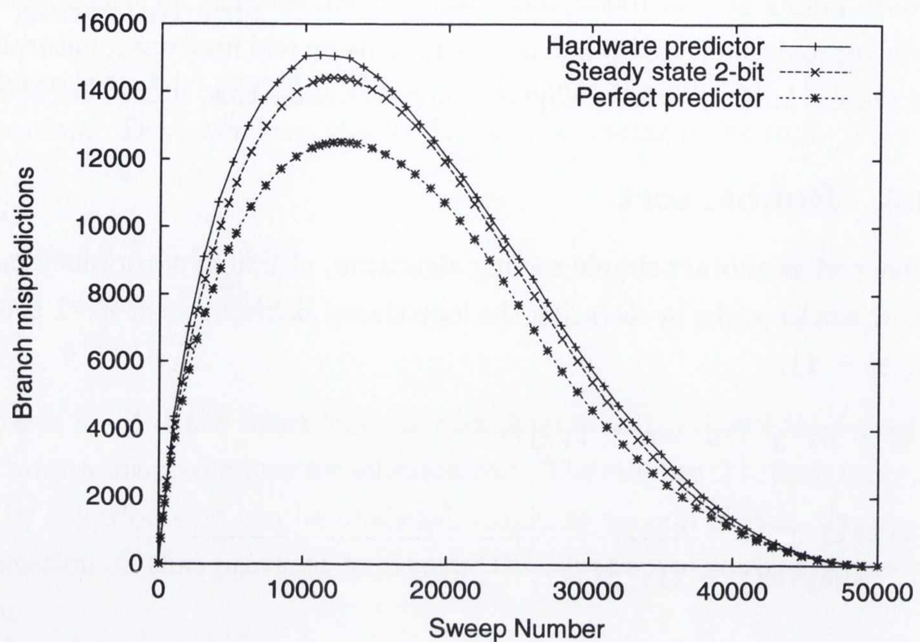
```
for(j = 0; j < n - i - 1; j++)
{
    if(a[j + 1] < a[j])
        swap(a[j + 1], a[j]);
}
```

In this code i is the outer-loop iteration counter, and begins at 0 and counts towards $n - 1$. After a full iteration of the inner-loop shown larger elements have moved to the right, closer to their final location.

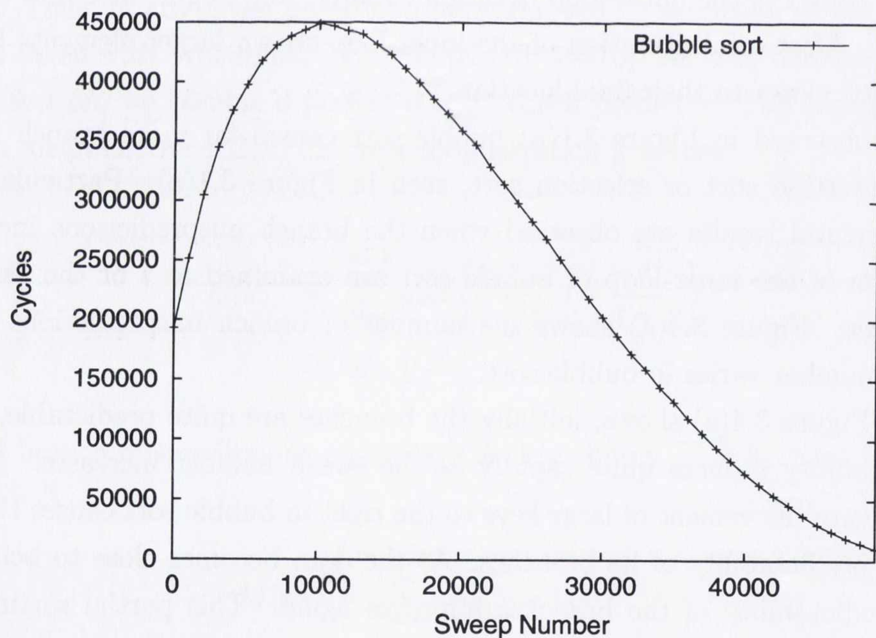
As observed in Figure 3.1(b) bubble sort causes far more branch mispredictions than insertion sort or selection sort, seen in Figure 3.1(a). Particularly interesting experimental results are observed when the branch mispredictions incurred by each iteration of the inner-loop of bubble sort are examined as i or the "sweep number" increases. Figure 3.4(a) shows the number of branch mispredictions caused as the sweep number varies in bubble sort.

As Figure 3.4(a) shows, initially the branches are quite predictable, however their predictability reduces quite rapidly as the sweep number increases. Intuitively, the incremental movement of large keys to the right in bubble sort causes this degredation in the predictability of its branches. As the data becomes close to being fully sorted the predictability of the branches improves again. This partial sorting that bubble sort performs allows it to potentially converge to fully sorted data in fewer outer-loop iterations than selection sort. We can detect that the data is sorted early if the branch shown in the inner-loop above is never taken for a whole iteration. However, this partial sorting also greatly increases the number of branch mispredictions bubble sort incurs, substantially slowing it over selection sort or insertion sort.

As Figure 3.4(b) shows, the number of cycles per sweep closely mimics the number of branch mispredictions, indicating that they are the dominant factor in the time required by bubble sort.



(a)



(b)

Figure 3.4: (a) Shows the number of branch mispredictions incurred per inner-loop of bubble sort as the outer-loop iterates or the "sweep number" increases. The curve labelled "hardware" is the actual number of branch mispredictions observed in hardware. The other two curves are plots of the functions $S(k)$ (the perfect static approximation) and $S'(k)$ (the bimodal approximation) defined in Section 3.2.4, and show that the approximations made in the analysis of bubble sort's branch mispredictions give results close to what is observed in hardware. (b) Shows the number of cycles (i.e. time) per inner-loop of bubble sort as the sweep number increases. This shows that the number of cycles is dominated by the number of branch mispredictions caused by bubble sort.

To understand the high misprediction rate of bubble sort more precisely we define Q_l^k† as the probability that l^{th} inner-loop comparison branch is taken on the kth outer-loop iteration, $1 \leq k < n$. It turns out that

$$Q_l^k = \begin{cases} \frac{l}{l+k} & \text{if } l \leq n-k \\ 0 & \text{otherwise} \end{cases} \tag{3.8}$$

This is a result of the fact that the lth comparison branch of the inner-loop is taken only if `a[l]` is not larger than all of `a[0..l - 1]`. For this to be the case, the $(l+1)^{th}$ branch in the previous outer-loop iteration must have been taken, since otherwise the previous iteration of the outer-loop left the maximum of `a[0..l - 1]` in `a[l]`. The probability that the $(l+1)^{th}$ branch of the previous outer-loop iteration is taken is Q_{l+1}^{k-1}. Given that the $(l+1)^{th}$ branch of the previous outer-loop iteration was taken, the probability that `a[l]` is not larger than all of `a[0..l - 1]` is $1 - 1/(l+1)$, thus,

$$Q_l^k = Q_{l+1}^{k-1}\left(1 - \frac{1}{l+1}\right) \tag{3.9}$$

The bases $Q_l^1 = 1 - 1/(l+1)$ for $1 \leq l < n$ and $Q_n^1 = 0$ give the solution shown in Equation 3.8. It is easy to see that in the first outer-loop iteration ($k = 1$) the comparison branch is very likely to be taken, whereas in the last outer-loop iteration ($k = n-1$), the comparison branch is very unlikely to be taken. Moving between these two extremes causes the intermediate branches to be unpredictable.

It is straightforward to show that bubble sort incurs $\Theta(n^2)$ branch mispredictions with respect to a perfect static branch predictor, as we now outline. Let $S(k)$ be the number of branch mispredictions caused by bubble sort on its $k^{th}, 1 \leq k \leq n$ outer-loop iteration:

$$S(k) = \sum_{l=1}^{n-k} \frac{1}{l+k} \min(l, k) \tag{3.10}$$

Again, we assume a perfect static branch predictor, and fill in the probability that the comparison branch is taken from Equation 3.8. With a series of rather involved but standard manipulations [Graham et al. 1994] we obtain:

$$S(k) = \begin{cases} (1 + H_n + H_k - 2H_{2k})k & \text{if } k \leq n/2 \\ n - (1 + H_n - H_k)k & \text{otherwise} \end{cases} \tag{3.11}$$

†Knuth [1997b; 1998b] considers a quantity related to Q_l^k, namely $\Pi_{l=1}^{n-k} Q_l^k = k^{n-k}k!/n!$ although he does not consider or derive Q_l^k itself.

Figure 3.4(a) shows a plot of $S(k)$, compared to the actual number of branch mispredictions observed in hardware. As expected, $S(k)$, the number of branch mispredictions expected using a perfect static predictor slightly underestimates the actual number of branch mispredictions observed in hardware. However, the trends in the number of branch mispredictions are very similar. This figure also shows a plot of the function:

$$S'(k) = \sum_{l=1}^{n-k} \left[1 - C_{2bit}\left(\frac{l}{l+k}\right)\right] \tag{3.12}$$

Recall that C_{2bit} is the characteristic function of a two-bit branch predictor described in Section 3.2.1. As Figure 3.4(a) shows, the number of branch mispredictions caused by this approximation of a two-bit finite state predictor, $S'(k)$, also slightly underestimates the actual number of branch mispredictions observed in hardware, although to a lesser extent than the analysis according to a perfect static predictor.

Finally, the total number of branch mispredictions incurred by bubble sort with respect to a perfect static predictor is simply $M(n) = \sum_{k=1}^{n-1} S(k)$. For simplicity of notation we assume n is even. Through a variety of rather involved but standard manipulations [Graham et al. 1994], we have

$$\begin{aligned} M(n) &= \frac{n^2}{4}\left[\frac{1}{2} - \frac{1}{n+1} - \frac{1}{n+2}\right] \\ &+ \frac{n}{2}\left[\frac{5}{4} - \frac{1}{n+1} - \frac{1}{n+2}\right] \\ &+ \frac{1}{4}\left[\frac{H_{n/2+1}}{2} - H_{n+2} + 1\right] \end{aligned}$$

Thus $M(n) = \Theta(n^2)$, showing that bubble sort causes substantially more branch mispredictions than insertion sort or selection sort. Recall that as shown above, the latter two algorithms cause $\Theta(n)$ and $\Theta(n \log n)$ branch mispredictions respectively. Bubble sort is often noted for its undesirability compared to other elementary sorting algorithms, for example Knuth [1998a] states:

> In short, the bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems.

Our results add further weight to Knuth's point of view.

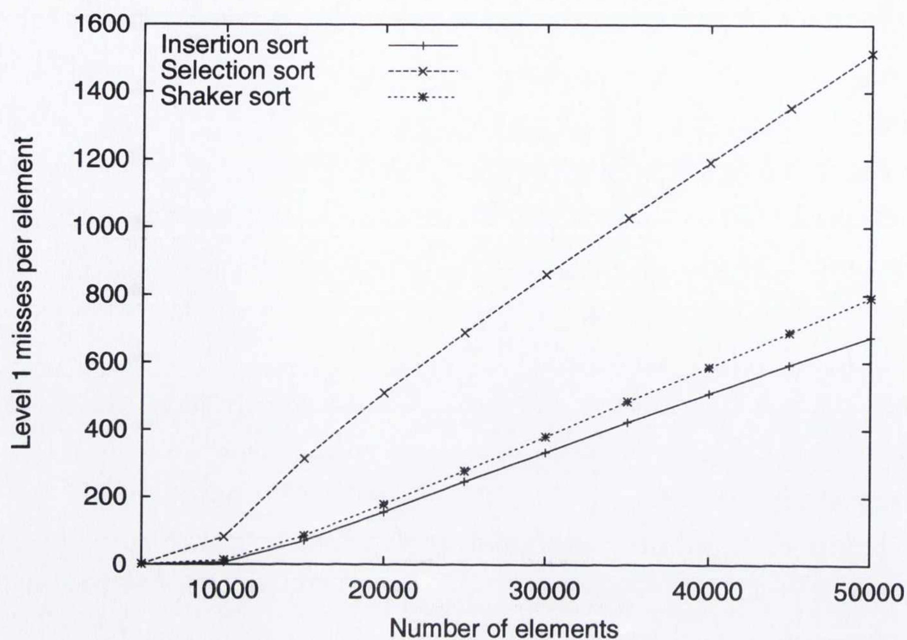
3.2.5 Discussion

The preceding sections have shown that three of the simplest well-known elementary sorting algorithms cause asymptotically different numbers of branch mispredictions. The results above show that simple, similar pieces of code can have very different branch prediction properties. In particular we have seen that bubble sort has the worst branch prediction properties. The deficiencies of bubble sort are often noted compared to other elementary $\Theta(n^2)$ worst-case time sorting algorithms. The apparent popularity of bubble sort despite its weaknesses are examined in detail by Astrachan [2003]. We now demonstrate briefly via additional experimentation that the principle weakness of bubble sort and its variations is their branch mispredictions, an aspect of the algorithms overlooked by Astrachan [2003] and Knuth [1998a].

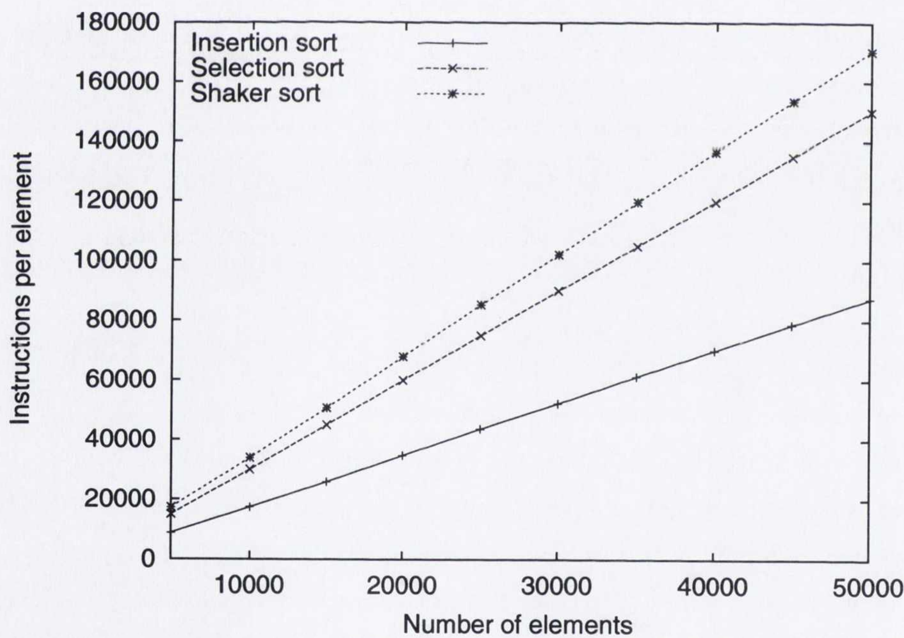
The principle objection to bubble sort is that it is substantially less efficient than other elementary sorting algorithms, and (arguably) no simpler than algorithms such as insertion sort or selection sort. Bubble sort has several refined variants. For example, shaker sort is a variation of bubble sort in which there are two inner-loops that are alternated. One is the inner-loop shown for bubble sort in the previous section. The other inner-loop scans right-to-left moving small elements to the left in the same manner as larger elements are moved to the right in bubble sort's inner-loop shown above. Although shaker sort has the same worst-case time as bubble sort, it is generally more efficient in practice.

Figure 3.5(a) shows the level 1 data cache misses caused by insertion, selection and shaker sort. Figure 3.5(b) shows the number of instructions executed by these algorithms. Finally Figure 3.6 shows the number of cycles taken by these algorithms. These results demonstrate the conventional wisdom, that insertion sort is the most efficient quadratic-time algorithm. Interestingly, Figure 3.5(a) shows that shaker sort causes substantially fewer level 1 data cache misses than selection sort. Figure 3.5(b) shows that shaker sort only executes slightly more instructions than selection sort. However, Figure 3.6 shows that shaker sort is much less efficient than selection sort. This is due to the enormous number of branch mispredictions incurred by shaker sort, as shown in Figure 3.1(b), compared to the number incurred by selection sort, in Figure 3.1(a).

In summary, these experimental results show the real reason bubble sort and its variations such as shaker sort are doomed to inefficiency on modern architectures is because they incur enormous numbers of branch mispredictions compared to the alternative algorithms.



(a)



(b)

Figure 3.5: (a) Shows the number of level 1 data cache misses per element for the elementary sorting algorithms. We do not present leve 2 data cache misses because the inputs fit within the level 2 data cache. We note that shaker sort has better cache performance than selection sort. (b) Shows the number of instructions per element executed by the algorithms. These results were gathered on an Intel Core 2 Duo 2.13GHz processor with a 2MB level 2 cache, using PAPI [Dongarra et al. 2003]. The data is uniform random data generated using the *C* function `random`, and each data point is averaged over ten executions of the algorithms on the data.

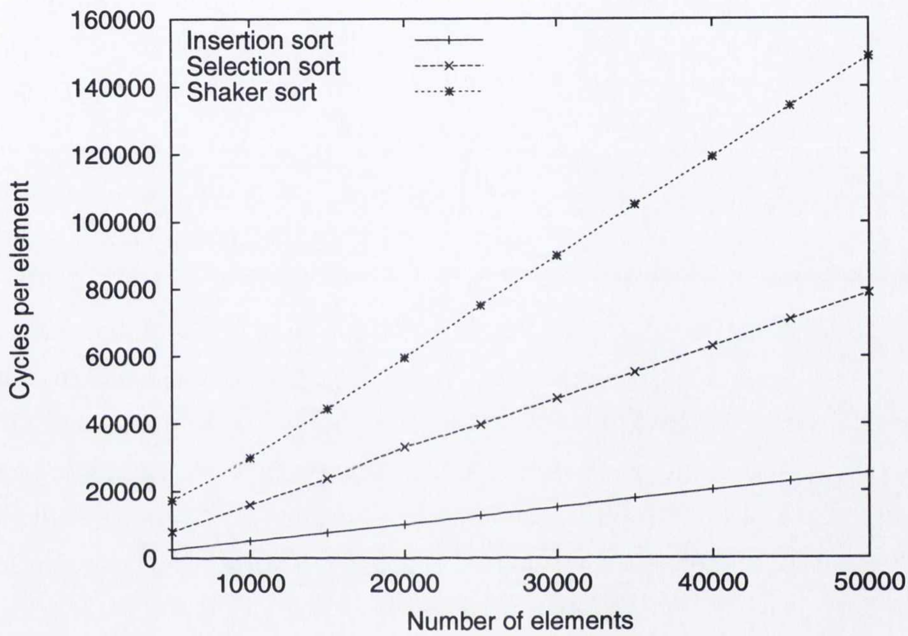


Figure 3.6: This figure shows the number of cycles per element required by the elementary sorting algorithms. It is noteworthy that while shaker sort is much less efficient than selection sort, this is not fully accounted for by its cache performance or instruction count, seen in Figure 3.5. These results were gathered on an Intel Core 2 Duo 2.13GHz processor with a 2MB level 2 cache, using PAPI [Dongarra et al. 2003]. The data is uniform random data generated using the *C* function `random`, and each data point is averaged over ten executions of the algorithms on the data.

3.3 Efficient Sorting Algorithms

The preceding sections have shown that well-known elementary sorting algorithms, all executing the same worst and average case number of comparisons cause asymptotically different numbers of branch mispredictions on average. In this section we consider the branch prediction characteristics of efficient sorting algorithms, that is, those that execute $\Theta(n \log n)$ comparisons in the worst case, or on average.

This upper bound matches the well-known $\Omega(n \log n)$ lower bound for any sorting algorithm that progresses by only making use of comparisons between its input elements [Cormen et al. 2001]. Unlike the worst-case $\Theta(n^2)$ algorithms of the previous section, we cannot expect any difference in the asymptotic number of branch mispredictions caused by these efficient algorithms. An intuitive, informal explanation of why this is so can be obtained by visualizing the decision tree [Cormen et al. 2001] associated to a sorting algorithm. We imagine inputs entering the tree at the root, and then flowing down either to the left or right child depending on the out-come of the comparison at each node. Eventually, each input flows to its unique associated leaf, at which point it is correctly sorted. In this view, a branch predictor must assign a labelling to the nodes of the tree, according to whether it predicts the input to flow to either the left or right child. Assume that at each node some proportion $\alpha \in (0, 1)$ of the inputs flow to the left child and $1 - \alpha$ inputs flow to the right child. As a result, the labelling of the branch predictor is wrong $\beta = \min\{\alpha, 1 - \alpha\}$ of the time at each node. Moreover, since the decision tree has $n!$ leaves, it has depth $\Omega(\log_\beta n!) = \Omega(n \log n)$. Resulting in $\Omega(\beta n \log n)$ branch mispredictions on average. We discuss a theorem of Brodal and Moruz [2005] which essentially results from a formalization of this argument in Section 3.4. Of course, when the decision tree is unbalanced many nodes have a child that only a small number of inputs flow down to, and the proportion of inputs for which the branch predictor's labelling is correct can be higher. Indeed, as we saw above insertion sort causes only $\Theta(n)$ branch mispredictions on average.

Despite the fact that all $\Theta(n \log n)$ (worst or average case) time sorting algorithms must cause $\Omega(n \log n)$ branch mispredictions, there is of course still the possibility of differences in the constant factors for branch mispredictions of the algorithms. We examine this in the following section.

3.3.1 Comparing Quicksort, Mergesort and Heapsort

In this section we study the branch prediction characteristics of three classic sorting algorithms: quicksort, mergesort and heapsort [Knuth 1998b; Cormen et al. 2001].

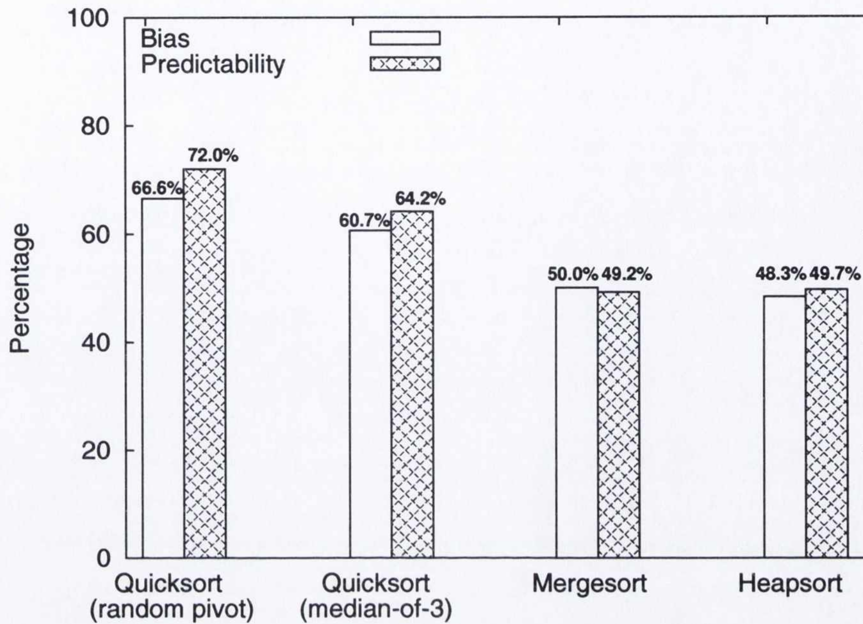


Figure 3.7: The bias and predictability of the principal comparison branches in classic efficient sorting algorithms.

```
pivot = a[l];
while(true)
{
    while(a[++i] < pivot) ; // i-loop
    while(a[--j] > pivot) ; // j-loop
    if(i >= j) break;
    swap(a[i], a[j]);
}
swap(a[j], a[l]);
```

Figure 3.8: The partition step of quicksort. We refer to the two inner loops as the *i-loop* and *j-loop*.

We pay particular attention to quicksort. We show that the comparison branches of quicksort are naturally more predictable than those in mergesort and heapsort, and examine the influence of the pivot choice on the behaviour of quicksort's comparison branches.

We begin by recalling briefly how these algorithms operate.

Quicksort [Hoare 1962] is an algorithm that selects an element from its input called the pivot and then partitions the other elements into two sets, one set containing elements at most equal to the pivot and another containing elements at least equal to the pivot. The keys of these sets should appear respectively to the left and right of pivot in sorted order. Therefore they can be sorted independently, each using quicksort. Quicksort operates in $\Theta(n^2)$ time in the worst case, and $\Theta(n \log n)$ time on average. In studying the branch prediction characteristics of quicksort, we are interested in the branch that compares elements to the pivot during partitioning. Code for the partitioning step of quicksort is shown in Figure 3.8. We refer to the comparison branches of quicksort as the i-loop and j-loop branches.

Mergesort [Knuth 1998b] is another $\Theta(n \log n)$ worst case time sorting algorithm, which works by repeatedly merging pairs of sorted lists into a single sorted list of twice the length. These merges can be done very simply in linear time and space. Mergesort's input can be thought of as n sorted lists of length one. To simplify the description, we assume the number of keys, n, is a power of two. Mergesort makes $\lg n$ passes over the input where the ith pass merges sorted lists of length 2^{i-1} into sorted lists of length 2^i. In studying the branch prediction characteristics of mergesort, we are interested in the comparison branch of the merge operation, which compares the two elements at the head of the two lists being merged. We refer to this branch as the *merge* branch.

Heapsort Another well known general purpose sorting algorithm is heapsort [Williams 1964]. Heapsort's running time is $\Theta(n \log n)$ in the worst case. Heapsort begins by constructing a *heap*: a binary tree in which every level except possibly the deepest is entirely filled, with the deepest level filled from the left. In addition, a heap must also satisfy the *heap property*: An element is associated with each node that is always at least as large as the elements associated with all its children nodes. The sequence of elements resulting from a breadth-first traversal of a heap's nodes can be used to represent it unambiguously in an array. Using the approach of Floyd [1964] an unordered array can be transformed into a heap in

$\Theta(n)$ worst-case time [Knuth 1998b]. Given a node whose children are heaps but who may itself violate the heap property, an iteration of Floyd's approach swaps the node with the larger of its children, and then repeats the same procedure with the subtree rooted at that child, until the heap property is satisfied or a leaf is reached. We refer to this as "sifting down" a node. In studying the branch prediction characteristics of heapsort, we are interested in the predictability of this *sift-down* branch, which compares the two children of a node to determine the larger child. To build the entire heap, we first sift-down the deepest nodes having children, followed by their parents, and so on until we reach the root. Once the heap is built, we gradually destroy it in order to sort. We iterate $n - 1$ times. On the ith iteration we swap the largest element, `a[0]`, with `a[n - i]`. After each swap, the heap property is restored by *sifting down* the new root of the heap (an operation with worst case time $\Theta(\log n)$). After iteration i the heap contains $n - i$ elements, and is found in `a[0..n - i - 1]`. Meanwhile `a[n - i..n - 1]` contains the i largest elements of the input from smallest to largest.

Figure 3.7 shows the experimentally observed average ***bias*** and ***predictability*** associated with the i-loop, *merge* and *sift-down* branches of quicksort, mergesort and heapsort respectively. We define the bias of a comparison branch to be the proportion of the time the branch is taken on average. We define the predictability of a comparison branch to be the proportion of the time the branch is correctly predicted, in this case by a two-bit saturating counter predictor. These results have been gathered by instrumenting the individual comparison branches in question with a simulated two-bit saturating predictor. For quicksort we show the behaviour of two variations of the algorithm. Firstly, a quicksort that chooses a random pivot, and secondly a quicksort that chooses a pivot using the "median-of-3" [Singleton 1969], where the median element of the first, middle and last element of the input is used as a pivot.

It is noteworthy that Figure 3.7 shows that quicksort is experimentally observed to have the most biased and predictable branches of the algorithms. In the case of a random pivot, the i-loop branch is observed to be approximately 66.6% biased, and 72.0% predictable (the j-loop branch behaves symmetrically, see Figure 3.8). We discuss this slightly counter-intuitive result — that the branches of quicksort are more predictable than biased — below. Where the median-of-3 technique is used to choose a pivot, the i-loop branch is approximately 60.7% biased and 64.2% predictable. On the other hand, mergesort and heapsort are both observed to have branches that exhibit bias and predictability of approximately 50%.

Quicksort generally offers superior performance to mergesort and heapsort for a

variety of reasons, notably its better spatial locality and lower instruction count. These results show us another strength of quicksort: it has naturally better branch prediction properties than mergesort or heapsort.

The observed bias of the *merge* branch in mergesort can be explained very simply as follows: In merging any two nondecreasingly ordered lists x and y (which we assume both have ∞ as their last element for simplicity) the *merge* branch is taken when $x_i < y_j$ and not taken otherwise. Clearly to exhaust both lists the branch must be taken $|x|$ times and not-taken $|y|$ times. In mergesort, the difference in the length of these lists is at most one, and hence the 50% bias of the *merge* branch. It is more difficult to give a simple argument explaining the average bias of the *sift-down* branch in heapsort, despite Figure 3.7 showing empirically that the bias and predictability appear to be close to 50%.

We now analyse the experimental results observed for the comparison branches in quicksort in detail. As with the analysis of our experimental results for the quadratic time algorithms in Section 3.2, this analysis is simple and similar to the average case analysis of quicksort, as is provided in many textbooks [Cormen et al. 2001].

The predictability of the i-loop of quicksort (or, symmetrically the j-loop, see Figure 3.8) branch depends on the rank of the pivot. When the pivot has rank q, the i-loop is executed (but not necessarily taken) a total of q times, since there are exactly $q-1$ elements that should appear to the left of the pivot after partitioning. With this rank, the i-loop branch is taken with probability $(q-1)/(n-1)$. Assuming the pivot element is selected randomly from the input the probability the pivot has rank q is $1/n$. Thus, the number of times the i-loop branch is taken on average is given by

$$T_n = \frac{1}{n(n-1)} \sum_{q=1}^{n} (q-1)q = \frac{1}{3}(n+1) \tag{3.13}$$

The i-loop is executed on average $1/n \sum_{q=1}^{n} q = (n+1)/2$ times, and thus we see that, on average, the i-loop is taken $2T_n/(n+1) = 2/3$ of the time that it is executed, matching the experimental result of Figure 3.7. This natural bias in quicksort's inner loop comparison branches makes them substantially more predictable than those of the other efficient sorting algorithms such as mergesort or heapsort, as noted above. Assuming a perfect static branch predictor, the probability that an i-loop is correctly predicted is given by

$$I_n = \frac{1}{n} \sum_{q=1}^{n} \max\left\{\frac{q-1}{n-1}, 1 - \frac{q-1}{n-1}\right\} = \frac{3n}{4n-4} - \frac{1}{2(n-1)} \tag{3.14}$$

Here we assume n is even for simplicty of notation. Thus, as n grows on average the i-loop is predicted correctly 75% of the time by a perfect static branch predictor. As expected this is more accurate than we observe experimentally in Figure 3.7. In this case it is also straightforward to estimate the predictability when a saturating two-bit counter is used

$$I'_n = \frac{1}{n}\sum_{q=1}^{n} C_{2bit}\left(\frac{q-1}{n-1}\right)$$

Here C_{2bit} is the steady-state predictability function of a saturating two-bit counter, described in Section 3.2.1. As $n \to \infty$ this sum approaches the value of the integral

$$I'_\infty = \int_0^1 \frac{3p^2 - 3p + 1}{2p^2 - 2p + 1}\, dp = \frac{3}{2} - \frac{\pi}{4} \tag{3.15}$$

Thus we see that, on average for large n, the predictability of the i-loop branch with a saturating two-bit counter branch predictor is approximately 71%, closely matching the experiment of Figure 3.7. We noted above that this branch is approximately 67% biased (that is, it is taken approximately 67% of the time). The fact that the branch is more predictable than biased is because the bias is an average. Note that for pivots of large rank (i.e. when q approaches n) the number of executions of the i-loop is also large, and that this is also the most predictable case — the probability that the branch is taken approaches 1.

Although quicksort's comparison branches are naturally more predictable than those of heapsort or mergesort, it is notable that this comes at the cost of risking quadratic time performance. To reduce the chances of quadratic time behaviour, the pivot in quicksort can be chosen more carefully. As described above, one particularly common method for choosing a pivot in quicksort is via the median-of-3. Analysing the experimental results in this case is analagous to the case just described, when a random pivot is used. The only difference lies in the probability that the pivot has a particular rank. When chosen via the median-of-3, it is easily shown that the probability the pivot has rank q is given by $R(n, q) = (q - 1)(n - q)/\binom{n}{3}$ [Sedgewick and Flajolet 1996]. The probability that the i-loop is taken in this case is then obtained in the same fashion as in Equation 3.13, replacing the $1/n$ term with $R(n, q)$:

$$T_n = \frac{1}{(n-1)}\sum_{q=1}^{n} R(n, q)(q-1)q = \frac{3n^2 - n - 4}{10(n-1)}$$

Again, the i-loop is executed on average $(n + 1)/2$ times in total, and so the bias is

given by $2/(n+1)T_n = (3n-4)/(5n-5)$. Thus as n increases the bias approaches $3/5 = 60\%$, as was observed experimentally in Figure 3.7. This is less than the the 66% bias when a random pivot is used. This is to be expected, since the median-of-3 pivot is designed to reduce the total number of comparisons performed. The predictability of the i-loop with respect to a perfect static predictor is also obtained analgously to the random pivot case. For the case of a perfect static predictor, we have

$$\begin{aligned} I_n &= \sum_{q=1}^{n} R(n,q) \max\left\{\frac{q-1}{n-1}, 1-\frac{q-1}{n-1}\right\} \\ &= \frac{11n^2}{16(n-1)^2} - \frac{11n}{8(n-1)^2} + \frac{1}{2(n+1)^2} \end{aligned}$$

Thus as n becomes large a perfect static predictor correctly predicts $11/16 \approx 69\%$ of the i-loop branches in a quicksort using a median-of-3 pivot, exceeding the accuracy observed experimentally in Figure 3.7. These experimental results correspond to the case of a 2-bit saturating predictor, and the mispredictions can be estimated by the sum

$$I'_n = \sum_{q=1}^{n} R(n,q) C_{2bit}\left(\frac{q-1}{n-1}\right)$$

Analagously to the case of a random pivot, when $n \to \infty$ this sum approaches the value of an integral

$$I'_\infty = \int_0^1 \frac{3p^2-3p+1}{2p^2-2p+1} W(p)\, dp$$

Here W is the continous analogue or "shape function" corresponding to R above, and is given by $W(p) = 6p(1-p)$ [Martínez and Roura 2001]. With the result that $I'_\infty = 3 - 3\pi/4$, or approximately 64% predictability, closely matching the experimental result of Figure 3.7.

Although widely used because of its efficiency in practice, quicksort nonetheless operates in $\Theta(n^2)$ time in the worst case. This worst-case time can result, for example, when a series of pivots are chosen such that only a constant number of elements reside on one side of the pivot at each partitioning step. Clearly it is not desirable for quicksort to "go quadratic" in this manner. However, quicksort is widely used in practice because it performs well on average, essentially because of the fact that its inner-loop is efficient in practice compared to other $\Theta(n \log n)$ average case sorting algorithms. Our

experimental results show an interesting contributing factor to the efficiency of quicksort's inner-loop for the first time, namely that the on average slightly unbalanced partitioning it performs improves its branch prediction properties, effectively reducing the cost of each comparison instruction it executes. As we have demonstrated in this section, quicksort's branches become less predictable as the quality of the pivot used is improved via the median-of-3 technique. In work current with that of this thesis, Kaligosi and Sanders [2006] investigated the performance of quicksort when the pivot quality is deliberately degraded. We discuss this work in detail in Section 3.4.1.

3.4 Related Work

As early as 1972 Knuth commented on the "ever-increasing number of 'pipeline' or 'number crunching' computers that have appeared in recent years" whose "efficiency deteriorates noticeably in the presence of conditional branch instructions unless the branch almost always goes the same way" (the same comment appears in [Knuth 1998b]). Most likely, Knuth was referring to the IBM 7030 "Stretch" computer (released in 1961 with a four-stage pipeline), Seymour Cray's Freon-cooled CDC 6600 supercomputer (released 1964), or other early supercomputers which used pipelining in the execution of general-purpose instructions.

A number of other authors have considered optimized implementations of sorting algorithms for more recent single-processor machines. This work however is most often related to the caching properties of sorting algorithms. An early example is the work of Nyberg *et al* [1994]. They present a quicksort-based algorithm for RISC architectures. They found that the main limit on performance was cache behaviour, and do not consider or refer to branch prediction. More recently, the study of cache-oblivious algorithms [Frigo et al. 1999] has led to cache-oblivious general purpose sorting algorithms that may be practical [Brodal et al. 2007].

In their classic work, Bentley and McIlroy [1993] implemented a highly-tuned quicksort, to be used in the `qsort` function of the standard C library. Again, they did not consider branch prediction, perhaps because the first desktop processors with dynamic branch predictors (such as the DEC Alpha 21064, MIPS R8000 and Intel Pentium) were only just appearing around that time.

Sanders and Winkel [2004] investigate the use of *predicated* instructions on Intel Itanium 2 processors. In addition to its normal inputs, a predicated instruction takes a predicate register input. If the value in the predicate register is true, the instruction takes effect; otherwise it does not. Predication is normally implemented by the

instruction being executed regardless of whether the predicate is true. However, its result is only *written back* to a register or memory if the predicate is true. Although predicated instructions use execution resources regardless of whether they are allowed to write back, they allow conditional branches to be eliminated. Sanders and Winkel show how the partition step in quicksort can be rewritten to use predicated instructions on an Itanium 2, a highly instruction-level parallel machine. The Itanium 2 provides large numbers of parallel functional units and registers which make such trade-offs worthwhile.

Mudge *et al* [1996] examine the behaviour of the i and j comparison branches in randomized quicksort, as an example to demonstrate a general method for estimating the limit on the predictability of branches in a program. Their work provides an early discussion of the relationship between branch prediction, sorting and data compression.

They note that — as described in Section 3.2.1 — without any a priori knowledge of the data to be sorted, the optimal predictor for these branches would keep a running count of the total number of array elements examined so far that are greater than the pivot. If the majority of the elements examined so far are greater than the pivot, the next element is predicted as greater than the pivot, and vice versa.

This simple technique described by Mudge *et al* could be made use of on architectures featuring semi-static branch predictors. In such a situation the hint-bit of the appropriate i and j branches in quicksort could be dynamically adjusted according to their simple scheme just described. A modern architecture where this may be practical is IBM's CELL SPE architecture. On this architecture there is an 18-cycle branch misprediction penalty, and the branch predictor is semi-static [Eichenberger et al. 2006]. The CELL SPE also features predicated instructions, and hence the performance comparison between the two techniques would be interesting. In fact, in a situation such as this, where it is possible to hint branches optimally, a semi-static predictor could out-perform dynamic predictors.

Brodal *et al* [2005] examine the adaptiveness of quicksort. They find that although randomized quicksort is not adaptive (i.e. its asymptotic analysis does not improve with respect to some measure of presortedness), its performance is better on data with a low number of inversions. They justify this by showing that the expected number of element swaps performed by randomized quicksort falls as the number of inversions does. The number of branch mispredictions is roughly twice the number of element swaps, because two branch mispredictions tend to occur when the two while loops of the partition code exit (see Figure 3.8). Furthermore they provide an empirical relation between the presortedness of the input, randomized quicksort's performance, and the

number of branch mispredictions.

In closely related work, Brodal and Moruz [2006] explore skewed binary search trees. That is, a binary search tree that is deliberately constructed to have more nodes in its left subtrees than its right sub-trees. During a search, this biases the comparison branches executed at each node, improving their predictabilty. For certain static layouts of the trees, deliberate skewing results in a performance improvement.

Finally, in the following two sections we explore experimentally two pieces of important work [Brodal and Moruz 2005; Kaligosi and Sanders 2006] that examine variations of quicksort and mergesort designed to have improved branch prediction properties.

3.4.1 Skewed Quicksort

In work concurrent to and independent of our own, Kaligosi and Sanders [2006] investigated the effect of the choice of pivot on the performance of quicksort. They found that an artificially skewed pivot gives better performance than using a random pivot or the median-of-3, due to the reduction in branch mispredictions. This performance increase is despite the increased instruction count such an artificially skewed pivot causes. Kaligosi and Sanders examine α-skewed pivoting, that is, where the pivot has rank $\lfloor \alpha n \rfloor$ when sorting n keys. They examine the performance of quicksort when the pivot is determined randomly, from the median-of-3, or is the true median (i.e. a (1/2)-skewed pivot). They show that using a (1/10)-skewed pivot gives a performance increase over all the aforementioned pivot choices. In this section, we provide an additional experimental evaluation of their work. In particular, we attempt to construct a more realistic implementation of a skewed quicksort, by improving the spatial locality of the algorithm and choosing the pivot in a practical manner.

In their experiments, Kaligosi and Sanders sort random permutations of $\{1, \ldots, n\}$, and they take advantage of this to compute an α-skewed pivot very simply, i.e. as $l + \alpha(r - l)$, where the end-point indices of the sorting sub-problem are l and r. In practice however, a skewed pivot must be chosen via sampling. In order to choose an, on average, $1/(s+1)$-skewed pivot it is necessary to select the minimum of a sample of s elements as the pivot. This can be seen easily since the probability that the minimum of a sample of s elements has rank q in an array of n elements is $\binom{n-q}{s-1}/\binom{n}{s}$. And so the average rank of the pivot is given by $\sum_{q=1}^{n-s+1} q \binom{n-q}{s-1}/\binom{n}{s} = (n+1)/(s+1)$.
In our implementations, sampling is used to select a skewed pivot. Choosing a skewed pivot adversely affects the spatial locality of quicksort, since larger subproblems are processed until a deeper depth of recursion than when a better balanced pivot is used. As a result, to improve performance our skewed quicksort implementation only begins

to use skewed pivots when the remaining subproblem fits within the level 2 cache.

Figure 3.9 shows experimental measurements of our skewed quicksort implementation in practice. We also compare to a traditional median-of-3 quicksort implementation, which chooses its pivot via the median-of-3. this is labelled "Base Quicksort" in Figure 3.9. In fact, this is the `std::sort` function of the GNU *C++* Standard Template Library [Stroustrup 1997] implementation, and the other implementations differ only in the code that choose their pivot. Figure 3.9(a) shows that as expected, the number of branches executed increases as the skew of the pivot is increased – this increase is traded for the improvement in the predictability of the resulting branches, visible in Figure 3.9(c). Despite only skewing when the number of remaining elements fits within the level 2 cache, the skewed quicksorts cause slightly more level 2 cache misses than the base quicksort implementation. Furthermore, although not shown, they also cause a larger number of level 1 cache misses. Finally Figure 3.9(d) shows the best performing algorithm is the unskewed base quicksort implementation.

The disparity between the results we observe and those reported by Kaligosi and Sanders, is likely due to difference in instruction pipeline lengths in the experimental setups. They report a speed-up on a Pentium 4 Prescott, which has a 31-stage pipeline. Whereas the Core 2 processor we experimented with has only a 12-stage pipeline. Our results show that skewing is a somewhat fragile technique for achieving a speed-up in a quicksort implementation, even when spatial locality is taken into account.

We believe it was important to describe and examine our variant of skewed quicksort adjusted to realistically choose a pivot and account for spatial locality, since in performance critical situations sorting algorithms can be tuned for specific architectures, and even to data sets. A detailed account of automatically constructing sorting algorithms tuned to a particular machine is provided by Bida and Toledo [2007]. An earlier account is also provided by Li *et al* [2005].

3.4.2 Insertion Mergesort

Brodal and Moruz [2005] have shown that any deterministic comparison based sorting algorithm performing $O(dn \log n)$ comparisons must incur $\Omega(n \log_d n)$ branch mispredictions — this is under the assumption that each key comparison determines the direction of an immediately following branch instruction. Their theorem results essentially from the formalization of the decision tree argument given in Section 3.3.

In Section 3.2.2 we showed that insertion sort causes $\Theta(n)$ branch mispredictions, appealing to the result of Brodal and Moruz with $d = n/\log n$ we see that insertion sort is optimal in the sense that any sorting algorithm performing $O(n^2)$ must cause

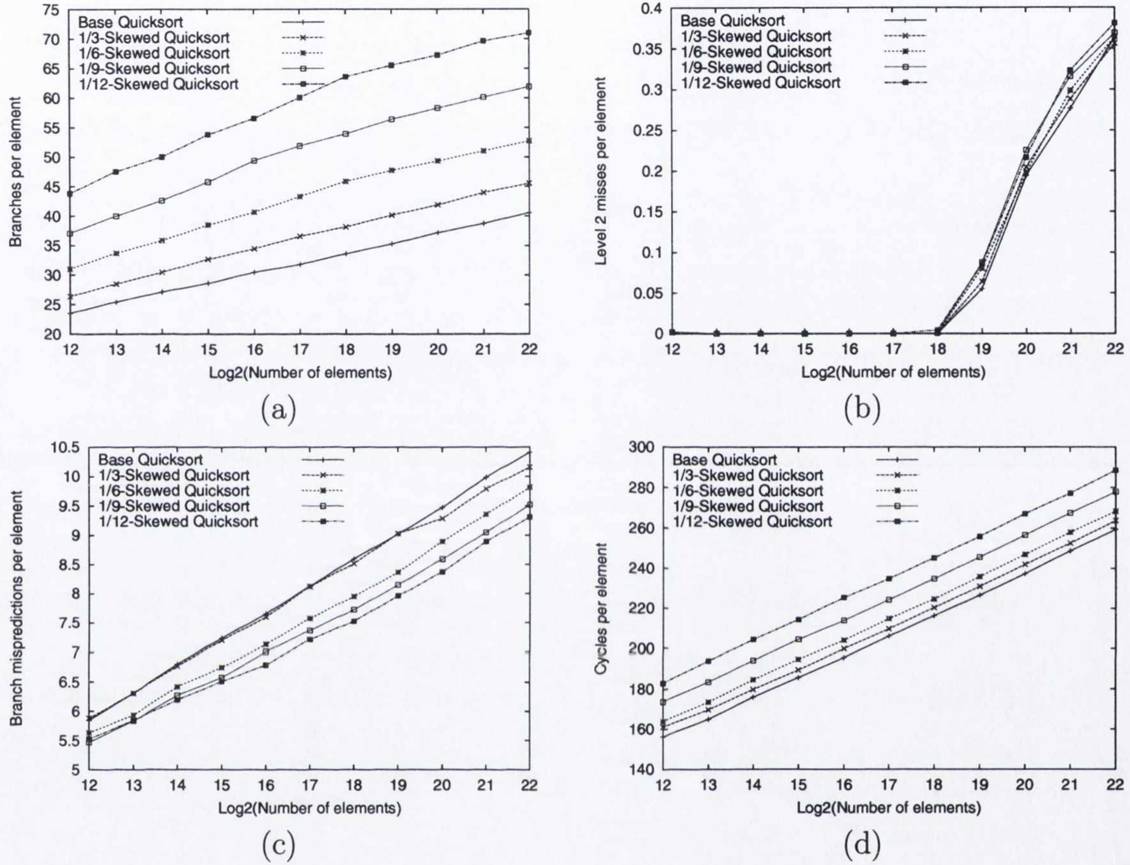


Figure 3.9: This figure shows performance results for skewed quicksorts as well as a traditional median-of-3 quicksort, labelled "Base Quicksort". (a) Shows that the skewed quicksorts execute more branches than base quicksort, with the number of executed branches rising with the skew. Although not shown, the number of executed instructions grows in a similar fashion. (b) Shows that the skewed quicksorts also have slightly worse spatial locality, despite only resorting to skewing when the data fits inside the level 2 cache. (c) Shows that the number of branch mispredictions does indeed decrease as the skew increases, however (d) shows that the best performing algorithm is base quicksort. These results were gathered on an Intel Core 2 Duo 2.13GHz processor with a 2MB level 2 cache, using PAPI [Dongarra et al. 2003]. The data is uniform random data generated using the *C* function `random`, and each data point is averaged over ten executions of the algorithms on the data.

$\Omega(n)$ branch mispredictions.

Brodal and Moruz also described *insertion d-way mergesort*, a multi-way mergesort algorithm that is optimal in the sense that it incurs $O(n \log_d n)$ branch mispredictions when performing $O(dn \log n)$ comparisons. Brodal and Moruz do not provide performance results for their algorithm. We now describe their algorithm and briefly present performance results for it.

Insertion d-way mergesort is an out-of-place multi-mergesort algorithm operating in $O(dn \log n)$ time on an input of n keys. For the simplicity of the description assume n is a power of d. The algorithm performs $\log_d n$ passes over its input, with the ith pass performing d-way merges of sorted lists of length d^{i-1} into sorted lists of length d^i. These d-way merges operate as follows: d sorted subarrays of an array a of m keys are merged in $O(dm)$ time. The algorithm maintains a vector $i = (i_1, \ldots, i_d)$ of indices into the sorted subarrays, together with a permutation $\pi = (\pi_1, \ldots, \pi_d)$ of $\{1, \ldots, d\}$ such that $(a[i_{\pi_1}], \ldots, a[i_{\pi_d}])$ is sorted. To perform a step of the merge, $a[i_{\pi_1}]$ is appended to the output array and i_{π_1} is incremented. The inner-loop of insertion sort (see Section 3.2.2) is then used to update π to ensure in $O(d)$ time that $(a[i_{\pi_1}], \ldots, a[i_{\pi_d}])$ is again in sorted order. This process is repeated m times, when all the keys in the subarrays will have been exhausted.

Figure 3.10(a) shows the number of instructions per element for our insertion mergesort variations. Increasing d from 3 to 6 causes a significant reduction in instruction count, however increasing d beyond 6 does not give further reductions in instruction count, as can be seen in Figure 3.10(a).

Figure 3.10(b) shows that as d is increased in the insertion d-way mergesort there is a corresponding reduction in the number of branch mispredictions. It is also notable that as d increases the cache performance of the algorithm also improves, as shown in Figure 3.10(c).

Figure 3.10(d) shows the cycles per element for each of the insertion mergesort variations. Our results indicate that Brodal and Moruz's [2005] insertion merge is a practical technique, although the base quicksort implementation of Figure 3.9 comfortably out-performs these mergesort variations. One desirable property of the insertion merge technique is that increasing d improves locality while also reducing branch mispredictions. Moreover, it may be possible to substantially mitigate the high instruction count of the technique by varying the value of d depending on the number of keys which remain to be sorted. In addition, for small values of d the insertion merge should be special-cased. It is also likely that the cache performance of the algorithm could be substantially improved by copying blocks of keys (for example, as many keys as fit in

a cache-line) to small buffers when appending keys from subarrays to the destination buffer. We also note that the insertion merge could also be used in a multi-quicksort implementation. That is, a quicksort implementation where d pivots are selected at each partitioning step.

3.5 Conclusion

This chapter has described how a ubiquitous hardware artifact: branch prediction, interacts with a number of classic sorting algorithms. We believe it is important that the everyday algorithms of Computer Science be understood in detail, and their interaction with the machines we use be carefully examined. Indeed, dynamic branch predictors have been present in some form in processors for nearly 40 years. In his description of the MU5 Instruction Pipeline in 1971, Ibbett [1971] notes

> The Instruction Buffer Unit is therefore designed to reduce the number of occasions on which this delay is incurred by predicting the result of the control transfer. The prediction technique is based on the use of an associative store containing addresses of instructions which have previously caused control transfers to occur and a corresponding conventional store containing the addresses of instructions to which control was actually transferred.

Ibbet appears to be describing a combination of two concepts now separated in computer architecture: a branch predictor and branch target buffer [Hennessy and Patterson 2006].

We have shown that elementary algorithms such as insertion sort, selection sort and bubble sort cause asymptotically different numbers of branch mispredictions. Of particular note here is that the large numbert of the unpredictable branch instructions executed by bubble sort and its variants such as shaker sort dominate its execution time.

In the case of efficient algorithms, we have shown that quicksort's inner-loop comparison branches are naturally more predictable than those of heapsort or mergesort. This demonstrates another advantage of quicksort for the first time. In a sense, it shows that choosing the pivot in a somewhat cavalier manner in quicksort has certain advantages, so long as the quadratic time worst-case of the algorithm is not encountered. Quicksort's comparison branches are not only more predictable than those of mergesort or heapsort, but when pivot choosing does go badly and partitioning becomes unbalanced, the inner-loop branches become more predictable. This offers quicksort a certain

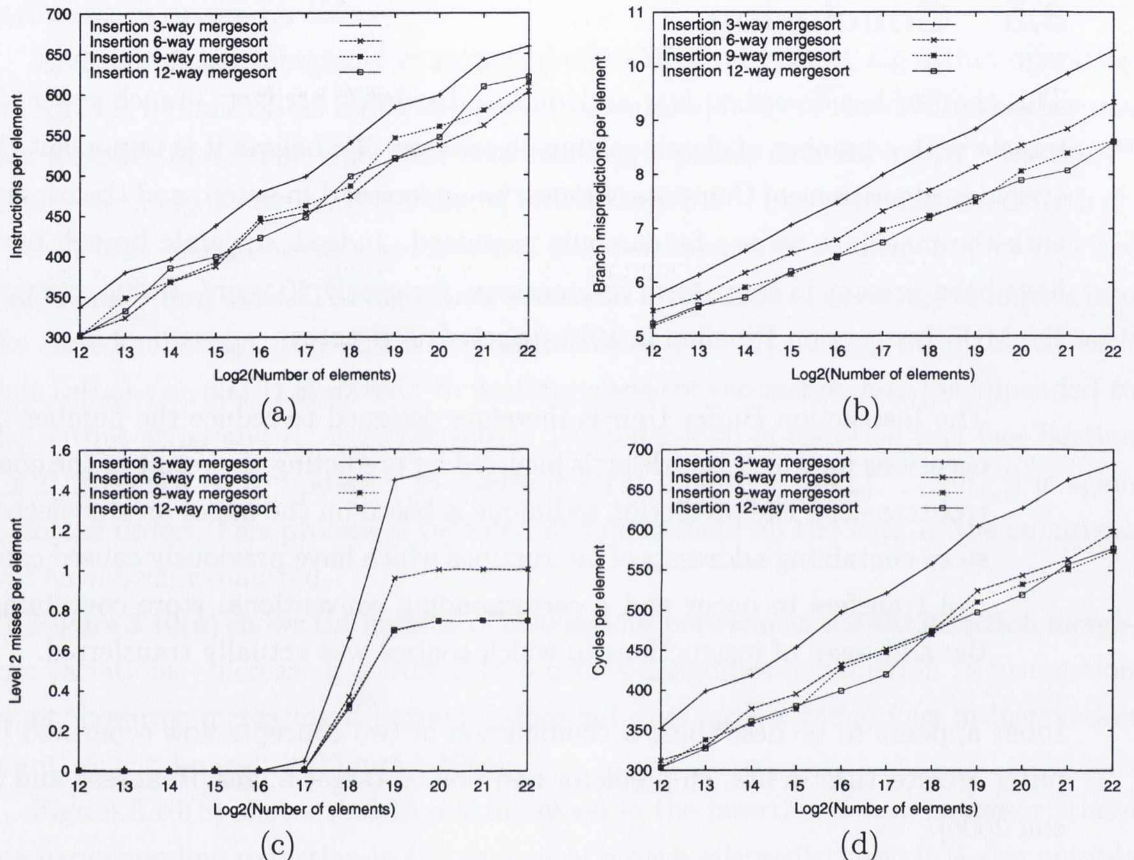


Figure 3.10: (a) Shows the instruction counts for the insertion d-way mergesort algorithms, for a variety of values of d. (b) Shows the branch mispredictions per key for the algorithms. (c) Shows the cache performance of the algorithms, we note that increasing d also improves spatial locality. The lack of improvement in spatial locality for $d = 9$ and $d = 12$ is a result of the fact that $d = 9$ suffices to ensure all sub-problems fit within the level 2 cache for the problem sizes shown. Finally (d) shows the cycles per key of the algorithms, showing how the results of (a), (b) and (c) translate into true performance. These results were gathered using PAPI [Dongarra et al. 2003] on an Intel Core 2 Duo 2.13GHz with a 2MB level 2 cache.

amount of natural resistance to poor pivots: although the instruction count increases, the branch misprediction penalty decreases. We have also shown that skewing quicksort is a somewhat fragile technique for achieving a speed-up, even when implemented carefully. Moreover, it is our view that skewing is unlikely to be an effective technique on future processors. The current processor trend is towards increased parallelism, and steps away from so-called super-pipelined designs, where very long pipelines enable increases in the clock-speed of the processor to be reported [Hennessy and Patterson 2006]. We have also presented experimental results for a mergesort variation that is asymptotically optimal in the number of branch mispredictions it causes. In addition, we have examined the spatial locality of the algorithm experimentally. In this line, we have also noted possible improvements to the algorithm, and that there is an analgous insertion multi-quicksort algorithm.

An important practical observation is that algorithms such as radix sort that do not compare their input elements in order to sort them are not subject to the resultant problem of branch mispredictions. This observation was made by Knuth as early as 1972, with the same observation appearing in the latest revision of his classic text [Knuth 1998b]. Comparison based sorting and searching data structures are more general than radix based approaches, although many objects of interest can be ordered based on their digital representation. Indeed, in the following chapter we explore the performance of data structures that do not rely only on comparisons, and demonstrate that in certain situations their superior performance is a result of the small number of branch mispredictions they incur (see Section 4.4.2). A second important practical observation is that where comparison based sorting and searching are genuinely necessary (examples occur in certain geometric algorithms [Bentley and Ottmann 1979]) predicated instructions, and good compiler support for them may be important for improving performance.

Finally, further experiments concerning sorting and branch prediction, as well as the branch prediction properties of certain cache-aware sorting algorithms can be found in the paper upon which this chapter is based [Biggar et al. 2008].

Chapter 4

Comparing Integer Data Structures for 32 and 64-bit Keys

4.1 Introduction

Maintaining a dynamic ordered data structure over a set of ordered keys is a classic problem, and a variety of data structures can be used to achieve $O(\log n)$ worst-case time for insert, delete, successor, predecessor and search operations when maintaining a set of n keys. Examples of such data structures include AVL trees [Knuth 1998a], B-trees [Bayer and McCreight 1972; Knuth 1998a] and red-black trees [Cormen et al. 2001]. Red-black trees in particular see widespread use via their GNU *C++* STL implementation [Stroustrup 1997].

Where the keys are known to be integers, different asymptotic results can be obtained by data structures that do not rely solely on key comparisons. For example, the stratified trees of van Emde Boas [1977] support all operations in $O(\log w)$ worst-case time, while Willard's q-fast tries [1984] support all operations in $O(\sqrt{w})$ worst-case time. Such data structures are attractive because of their potential to offer better performance than comparison-based data structures. However, it is a significant challenge to construct implementations that reveal their asymptotic performance, especially without occupying a large amount of extra space compared to comparison-based data structures. For example, Dementiev *et al.* [2004] provide a stratified tree implementation based on the variation described by Mehlhorn and Näher [1990]. While they achieve superior performance in time against comparison-based data structures, their data structure occupies more than twice as much space and is restricted to 32-bit keys.

In this chapter we experimentally evaluate the performance of a variety of data structures when their keys are either 32 or 64-bit integers. We emphasize that although

we refer to the keys as integers, the keys may be any set of bit-strings all of some fixed length. A classic example is that the keys may also be floating point numbers, since their order is preserved when their bit representation is interpreted lexicographically [IEEE 2008]*.

We find that a carefully engineered variant of a *burst trie* [Heinz et al. 2002] generally provides excellent performance in time and space compared to all the alternative data structures. We experimentally compare the performance of our burst trie variant in both time and space to red-black trees and B-trees. Aside from these commonplace general purpose data structures, we also experimentally examine the performance of two slightly more ad-hoc data structures [Dementiev et al. 2004; Korda and Raman 1999] that are tailored for the case of integer keys, and have been shown to perform well in practice. We describe these two data structures in the remainder of this section.

Dementiev *et al.* [2004] describe the engineering of a data structure based on stratified trees [van Emde Boas 1977] and demonstrate experimentally that it achieves superior performance to comparison-based data structures. Their data structure is highly specialized to the case of 32-bit keys. Their data structure is a hierarchy of tables. The root table, r, contains 2^{16} pointers, if there is a key in the data structure with its highest 16 bits equal to i, then $r[i]$ points to a second level dynamic hash table. Assuming a level two table, $L2$, exists and there is a key with bits $8 \ldots 15$ equal to j then $L2[j]$ contains a pointer to a third and final dynamic hash table, indexed by the lowest 8 bits of the keys. These hash tables grow in power of two sizes from a minimum of 4 to a maximum of 256 entries. To allow the data structure to support operations such as predecessor and successor, an analogous hierarchy of bit vectors is maintained together with the tables at each level. This hierarchy of tables is a recursive example of *exponential range reduction*, originally used in a data structure devised by van Emde Boas. We provide further details in Section 4.5.

We refer to this structure of Dementiev *et al.* as an S-tree. Although highly efficient in time, the S-tree is tailored around keys of 32-bits in length and generalizing the data structure to 64-bit keys would not be feasible in practice because of the large amount of space required to maintain efficiency. For example, it would be infeasible for the root table r mentioned above to contain 2^{32} pointers. Thus, it would have to become a dynamic hash table, with a significant impact on performance. As our experiments will show, even for 32-bit keys the data structure requires more than twice as much space as a typical balanced search tree.

*Actually, minor modifications are required to ensure this: the most significant bit is complemented for non-negative floating numbers, all bits are for complemented negative floating point numbers.

Korda and Raman [1999] describe a data structure similar to a q-fast trie [Willard 1984] and experimentally show that it offers performance superior to comparison-based data structures. Unlike the S-tree data structure engineered by Dementiev *et al.* this data structure is not restricted to 32-bit keys and requires less space in practice. We now briefly describe the features of Korda and Raman's data structure relevant to our discussion. We refer to their data structure as a Q-trie. A Q-trie consists of a path compressed trie containing a set of *representative keys*, $K_1 < K_2 < \cdots < K_m$. Associated with each representative key K_i is a bucket data structure B_i containing the set of keys $\{k \in S : K_i \leq k < K_{i+1}\}$ for $i < m$, and $\{k \in S : k \geq K_m\}$ for $i = m$, where S is the entire set of keys in the data structure.

Each bucket contains between 1 and $b - 1$ keys. When a new key is inserted into the data structure the compressed trie is first searched for its predecessor key, giving a representative key K_i. If the associated bucket B_i already contains $b - 1$ keys, a new representative key is added to the compressed trie that partitions the bucket into two new buckets containing $b/2$ keys each. Deletions operate in a similar manner to insertions, except that when two adjacent buckets B_i and B_{i+1} contain fewer than $b/2$ keys in total the keys of B_{i+1} are inserted into B_i and K_{i+1} is deleted from the trie. A search in the data structure is accomplished by a predecessor query in the compressed trie, followed by a search in the relevant bucket data structure.

There are many other non-comparison-based data structures in addition to the two just mentioned, both practical and theoretical. We provide an overview in Section 4.5.

The contributions of our work are as follows:

1. *We provide a thorough experimental comparison of dynamic data structures over 32 and 64-bit integer keys.* We provide time and space measurements over uniform and non-uniform random data as well over data sets that arise more naturally, and over several different machine architectures, configurations and compilers. We feel that previous experimental work on integer data structures has been focused on engineering an efficient variant of a particular data structure, rather than placing that data structure in context with other comparison-based and non-comparison based structures in time and space. For example, Korda and Raman [1999] provide experimental results comparing the Q-trie described above where the bucket data structure is varied. However, they do not include space measurements, use only random data, and do not include any classic comparison-based data structures in their comparison. Dementiev *et al.* [2004] compare their engineered data structure only with comparison based data structures, do not provide space measurements and only examine performance on artificial data

sets. Nilsson and Tikkanen [2002] provide a comparison of several variants of their relaxed level compressed trie structure with comparison based data structures. Their experiments include time and space measurments over both random and naturally occuring data. Unfortunately they do not include certain other integer data structures (e.g. a Q-trie) in the comparison, and amongst their comparison-based structures they lack a B-tree — often the most efficient comparison-based structure for searching. In addition, all of the experiments above were performed on a single machine architecture. While all of the work described above has made an important experimental contribution, we hope to contribute by building upon it in this chapter by providing a more thorough experimental evaluation of integer data structures.

2. *We show that burst tries extend efficiently to a dynamic ordered data structure.* The work of Heinz *et al.* [2002] examining burst tries focuses on the problem of *vocabulary accumulation*, where the keys are variable length strings. The only operations performed are insert and search, with a final in-order traversal of the burst trie. In contrast, we consider the case of integer keys with all the operations usually associated with a dynamic ordered data structure. We show how all the operations usually associated with a dynamic ordered data structure can be implemented efficiently through careful engineering. The burst trie variant we describe is a combination of a level compressed trie [Andersson and Nilsson 1993] with a burst trie. To the best of our knowledge, this combination has not been studied experimentally in the past.

3. *We show that our burst trie variant performs excellently in practice.* Although the full picture is only available by examining our experimental results (see Section 4.4 and Appendix A), we note that our burst trie variant usually offers better performance than both comparison-based and the other integer data structures in time, and occupies less space than even space efficient implementations of comparison-based search trees.

4.2 Burst Tries

In this section we provide the definition of a burst trie and some basic background information regarding the data structure.

Definition A string w is the v-suffix of a string u if $u = vw$.

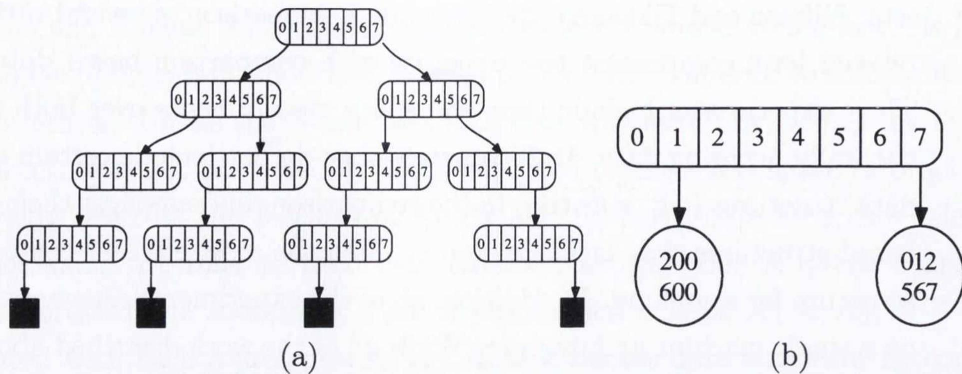


Figure 4.1: (a) Shows a trie holding the keys 1200, 1600, 7012 and 7567. The leaves of the trie (black squares) hold the satellite data associated with the keys. A corresponding burst trie, with bucket capacity 2, is shown in (b).

Definition A *burst trie* with bucket capacity c containing n keys is a tree with the following properties:

1. If $n = 0$, the burst trie is empty.
2. If $n \leq c$, the burst trie is a bucket data structure containing the n keys and their associated values.
3. If $n > c$, the burst trie consists of an internal node with 2^i children, $i \geq 1$. For each binary string x of length i, there is a child burst trie containing all the x-suffixes of the keys.

Figure 4.1(a) shows an example of a trie while Figure 4.1(b) shows a burst trie corresponding to it. Although we refer to what has just been described as a burst trie, using some kind of bucketing in a trie is an old technique. Sussenguth [1963] provides an early suggestion of the technique, while Knuth analyses bucketed tries [1998a]. In addition, Knessl and Szpankowski [2000b; 2000a] analyse what they refer to as b-tries — tries in which leaf nodes hold up to b keys.

We use the term burst trie of Heinz *et al.* [2002] because their work was the first to provide a large scale investigation of alternative bucket data structures, the time and space trade-offs in practice resulting from bucketing, and the *bursting* of bucket data structures during insertions, which we describe below.

Searching in a burst trie is similar to searching in a conventional trie. The digits of the key are used to determine a path in the trie that either terminates with a NIL pointer, in which case the search terminates unsuccessfully, or a bucket is found. In the latter case, the search finishes by searching the bucket data structure for the key.

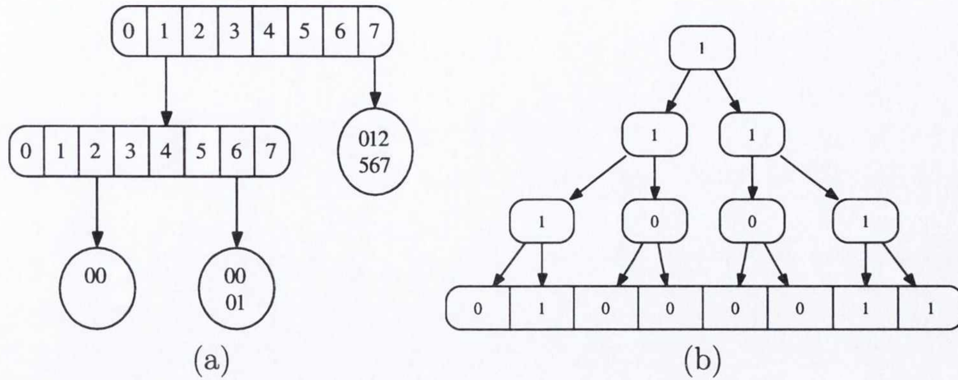


Figure 4.2: (a) Shows the burst trie of Figure 4.1(b) after inserting the key 1601. Assuming the buckets can hold at most two key suffixes, inserting the key 1601 causes the left bucket shown in Figure 4.1(b) to burst. In (b) an OR-tree is shown, a possible in-node data structure for implementing a burst trie.

Insertion of a key into a burst trie is also straightforward. The digits of the key are used to locate a bucket where the key should be stored. If no such bucket exists, one is created. On the other hand, if a bucket is found and it contains fewer than c keys it need not be burst and the key is simply added to that bucket. Otherwise, if the bucket already contains c keys, it is burst. This involves replacing the bucket with a trie node and distributing the keys of this bucket into new buckets descending from this new trie node. Figure 4.2(a) shows an example of a burst operation occuring on the burst trie of Figure 4.1(b). It is possible that all keys from the burst bucket belong in the same bucket in the newly created node. In this case, the bursting process is repeated.

Deleting a key k from a burst trie is performed by first searching for the bucket where k is stored, as described above. If there is no such bucket, no deletion need occur. Otherwise, k is deleted from some bucket b at a node x. If b is then empty, it is deleted from x. If x then has only NIL child and bucket pointers x is deleted from the trie. This step is repeated, traversing the path from x to the root of the trie deleting ancestors encountered with only NIL child or bucket pointers. The traversal terminates when either a node with a non-nil pointer is encountered, or the root of the trie is reached.

4.3 Engineering Burst Tries

Although the burst trie data structure described in the Section 4.2 leads to a highly efficient data structure, especially for strings, as shown by Heinz *et al.* [2002], care must be taken when engineering it for the case of an ordered data structure for integer keys.

Our variant of a burst trie makes use of three data structures for which we consider the engineering concerns: (1) The trie data structure itself. We describe the use of a trie making use of level and path compression. We note that the combination of level compression and bucketing in tries has not been examined experimentally in the past. (2) The bucket data structures at the leaves of the trie, and (3) the data structures inside the nodes of the burst trie. We describe the alternatives for this latter data structure in the next section.

4.3.1 In-Node Data Structures

Given a node x in a trie-based data structure with branching factor b, and an index i, $0 \leq i < b$, it is often necessary to find $\textsc{Succ}(i)$, that is, the smallest $j > i$ such that $x[j] \neq \textsc{nil}$. This is the bucket or child node pointer directly following $x[i]$. It is also often required to find $\textsc{Pred}(i)$, the largest $j < i$ such that $x[j] \neq \textsc{nil}$. These operations upon nodes are required, for example to support queries on the trie involving in-order iteration over its keys. We elaborate on the use of these operations in Section 4.3.5.

Many data structures can be used to support these predecessor and successor operations on the trie node [Demaine 2003]. We experimented with several in-node data structures for our burst trie variant. The simplest data structure supporting these predecessor and successor operations is just a linear search over a bit-vector. This data structure requires only $\Theta(1)$ worst-case time (all bounds in this section should be understood as worst case) when a new bucket or child is added or removed from the node, however, $\textsc{Pred}$ and $\textsc{Succ}$ are inefficient, requiring $\Theta(b)$ time.

An alternative in-node data structure is an OR-tree. Figure 4.2(b) shows an example of this data structure. A breadth-first traversal of an OR-tree can be laid out in an array inside each node, requiring an additional $\Theta(b)$ space compared to a simple bit-vector approach. However, an OR-tree offers all operations in $\Theta(\log b)$ time.

Another simple solution is to implement $\textsc{Pred}$ and $\textsc{Succ}$ using $\lceil\sqrt{b}\rceil$ counters. Where the ith counter, $0 \leq i < \lceil\sqrt{b}\rceil$ holds a count of the non-zero bits in the range $[i\lceil\sqrt{b}\rceil, i\lceil\sqrt{b}\rceil + \lceil\sqrt{b}\rceil - 1]$ (except perhaps for the last counter, which covers the range $[b - \lceil\sqrt{b}\rceil, b-1]$). This data structure allows insertions and deletions in $\Theta(1)$ time and supports $\textsc{Pred}$ and $\textsc{Succ}$ in $\Theta(\sqrt{b})$ time, requiring at most $\lceil\sqrt{b}\rceil$ counters to be examined followed by at most $\lceil\sqrt{b}\rceil$ bits.

Of course, for large enough inputs the OR-tree will out-perform the counter search, because it executes asymptotically fewer instructions. However, even for the largest experiments we conducted in this chapter (see Section 4.4), which include data sets of $2^{27} \approx 1.3 \times 10^8$ keys, the branching factor of any node in the burst trie did not exceed

2^{20}. Moreover, it is clear that due to its breadth-first layout, the spatial locality of the OR-tree is worse than that of the counter search. Drawing on insights from Chapter 3, the OR-tree is also likely to incur more branch mispredictions, since intuitively one expects that an algorithm executing an asymptotically smaller number of branches extracts more information from each branch, thus making each branch less predictable. The spatial locality of the OR-tree can be improved by avoiding the use of the breadth-first layout, instead a cache oblivious layout can be used [Bender et al. 2000; Brodal et al. 2002]. We use the simple indexing algorithm of Kasheff [2004] to implement our cache oblivious OR-tree. Recall that the basic idea behind a cache oblivious layout of a complete tree of height H is to recursively divide the tree vertically, with the result that if the cache line length, B, is such that a complete sub-tree of height at most h fits within a cache line, there exist complete sub-trees of height $H/2^m \leq h < H/2^{m-1}$ which are stored contiguosly. As a result loading a sub-tree of height h can cause at most 2 cache misses (or, 4 cache misses due to alignment issues), resulting in $\Theta(H/h)$ cache misses in a traversal of the tree from root to leaf. Our cache oblivious OR-tree on n nodes has $H = \lceil \lg n \rceil$, and clearly $B = \alpha 2^h$ for some $1 \leq \alpha < 2$, giving the optimal $\Theta(H/h) = \Theta(\log_B n)$ cache misses in the cache oblivious OR-tree.

Figure 4.3 shows experimental measurements for the OR-tree in breadth-first and cache oblivious layouts, and the counter search as the size of the bit-vector increases. The bit-vector in this case consists of all zeros except that its right-most bit is set to a one. The searching operation here is to find the successor of the left-most bit in the bit-vector. Figure 4.3(a) shows the cycles (i.e. time) per search operation. At the smaller input sizes, $b < 2^{13}$, the counter search is a maximum of approximately 30% faster than the breadth-first layout OR-tree (although this is not easily visible in Figure 4.3(a)). We note that the OR-tree out-performs the counter search from around $b = 2^{13}$ onwards. This is due to the rapidly increasing instruction count of the counter search compared to the OR-tree, shown in Figure 4.3(b). The smaller number of cache misses and branch mispredictions incurred by the counter search, shown in Figure 4.3(c) and (d) respectively do not result in a performance improvement for the counter search, because they are insignificant compared to the number of instructions executed, even accounting for their higher cost compared to the average instruction. For the range of input sizes considered here, the improved spatial locality of the cache oblivious layout OR-tree is not enough to compensate for the increase in instruction count and branch mispredictions that the more complicated indexing of the cache oblivious layout causes. It should be noted that the cache misses shown in Figure 4.3 are level 1 misses, and are less expensive than the usually considered level 2 misses.

In our burst trie variant, we have chosen to use the OR-tree as the in-node data structure. This engineering decision is based on the data just presented. It should be noted that the experiments above consider the case where the trie nodes are very sparse, and the bits are not distributed uniformly. If one expects denser nodes and the burst trie to contain data with a high degree of randomness then the counter search is preferable. For example, note that if a bit-vector of any number of bits has only 0.1% non-zero bits and these are uniformly distributed then successor queries on the in-node structure can be regarded as operating on bit-vectors of 1000 bits long on average, which is well within the region for which the counter search performs comparatively well (see Figure 4.3(a)), and moreover it requires only $\Theta(\sqrt{b})$ rather than $\Theta(b)$ extra space.

4.3.2 Bucket Data Structures

The choice of data structure used for the buckets of a burst trie is critical in achieving good performance. Heinz *et al.* [2002] concluded that unbalanced binary search trees holding at most 35 strings offered the best performance as a bucket data structure. They also experimented with linked lists and splay trees [Sleator and Tarjan 1985]. Since the maximum number of keys stored in each bucket is modest (at most 35), a simple bucket data structure, even with bad asymptotic behaviour, may perform well. In related work, unsorted arrays of strings have been used as bucket data structures for burst tries as a basis for the *burstsort* algorithm [Sinha and Wirth 2008; Sinha et al. 2006; Sinha and Zobel 2005, 2004; Sinha 2004], which is a cache-efficient radix sorting algorithm.

We experimented with balanced binary trees as well as with unsorted arrays and sorted arrays. Over all, we found the arrays are far more efficient in practice than the search trees. It is likely that the arrays incur far fewer cache misses than the search trees. If unsorted arrays of c elements are used as the bucket data structures, searching in a bucket takes $\Theta(c)$ time (all time complexities in the current section are worst case). Insertion also requires $\Theta(c)$ time, because each key to be inserted must be searched for in the bucket before it can be inserted in order to avoid duplication. If sorted arrays are used, searching in a bucket takes $\Theta(\log c)$ time, while insertion takes $\Theta(c)$ time. In practice, the insertion into a sorted array is more expensive than the insertion into an unsorted array. This is because in unsorted arrays the elements need simply be scanned to check for the presence of the key to be inserted. In the sorted case the elements of the array must be rearranged to maintain sorted order. However, the logarithmic search time in sorted buckets gives much improved search times in practice. Since in many

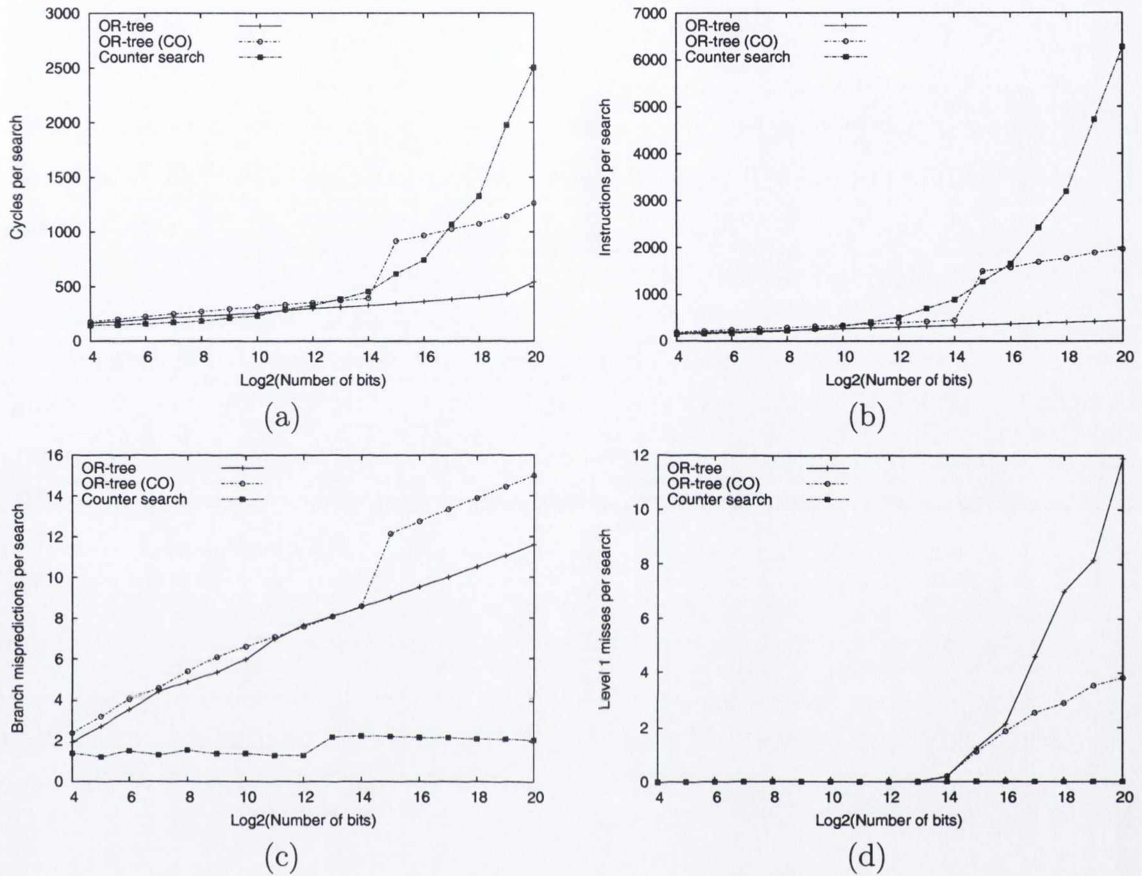


Figure 4.3: This figure shows a comparison of the in-node data structures described in Section 4.3.1. The results are averaged over several thousand successor search operations on a sparse bit-vector, and were gathered using PAPI [Dongarra et al. 2003] on an Intel Core 2 2.13GHz, having a 32KB level 1 data cache. All of the data structures fit within the level 1 data cache. We note the reduced branch mispredictions of the counter search, shown in (c) and its better spatial locality shown in (d), are not enough to compensate for its larger instruction count compared to the OR-tree, shown in (b). As a result, except for very small inputs, the OR-tree performs best, as is shown in (a). The cache oblivious OR-tree is labelled "OR-tree (CO)". Due to the extra expense of computing the cache oblivious indexing for this data structure, its performance is worse than the standard OR-tree, despite its better spatial locality, visible in (d). The transition points visible in the cache oblivious OR-tree's performance at 2^{15} bits are a result of that fact that at 2^{15} bits indexing reverts to breadth-first layout (without this reversion, the overhead of its cache oblivious indexing is even higher).

applications, searching is the most frequent operation executed on a data structure we have chosen the sorted arrays over the unsorted arrays. In addition, predecessor and successor operations are less efficiently supported by unsorted arrays.

In contrast to the array buckets of the burstsort algorithm, our buckets are sorted holding at most 128 keys. We investigate the effect of bucket size on our burst trie variant in Section 4.4.1. This is in contrast to the bucket sizes used in burstsort, where the buckets are allowed to grow until they reach the size of the processor's 2nd level cache which can be several megabytes in size. The buckets are implemented as growable arrays, and an insertion involves possibly doubling the size of the bucket followed by a linear scan to find the correct position for the key to be inserted.

As mentioned above, often the most frequent operation executed on a data structure is a search, and so searching buckets in particular should be efficient. We use a binary search that switches to a linear search when the number of keys that remain to be searched falls below a certain threshold. We found a threshold of between 16 and 32 keys gave a performance improvement over a simple binary search. Our burst trie implementation is designed to provide a mapping from a key to the satellite data associated with that key, which we refer to as the *value* for the key. To improve the spatial locality of searches the keys and values of a bucket should not be interleaved. Rather, all the keys should be stored sequentially, followed by all the values of that bucket. This ensures searching for a key makes better utilization of the processor's cache lines.

4.3.3 Level and Path Compressed Tries

The burst trie defined in Section 4.2, and introduced by Heinz *et al.* [2002] uses a simple trie to search for the bucket in which a key resides. In this section we describe a burst trie making using of both level and path compression. Level compression was proposed by Anderson and Nilsson [1993] (see also Nilsson [1996]). Dynamic level and path compressed tries, or LPC-tries were investigated experimentally by Nilsson and Tikkanen [2002], we discuss their work in more detail in Section 4.5. However, they concluded level compression gave rise to tries that offered slower insertions and deletions than comparison-based structures, while offering faster search operations. Moreover, their LPC-tries occupy a similar amount of space to comparison-based search trees. In contrast, the bucketed variations we describe in general provide all operations more efficiently than comparison-based search trees, while also occupying signficantly less space. We next give the definition of a path compressed burst trie (PCB-trie) and then of a level and path compressed burst trie ($LPCB$-trie), slightly amending the

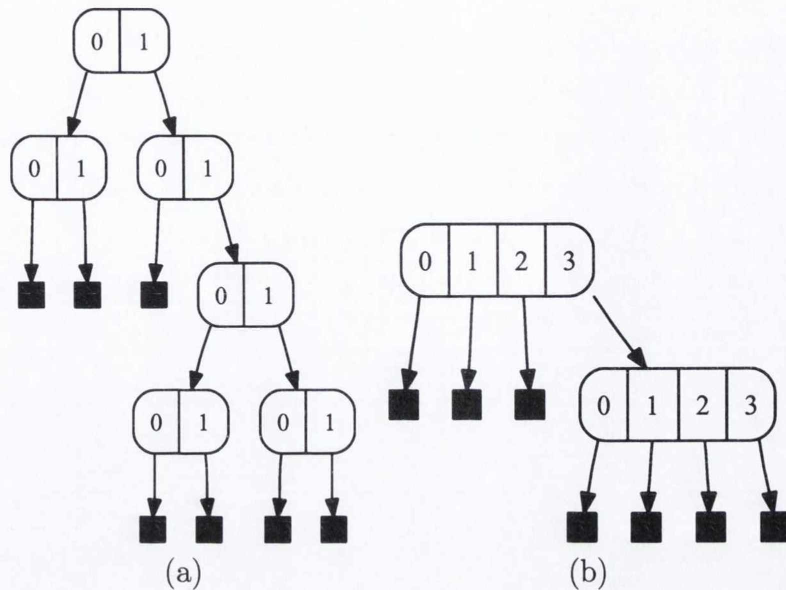


Figure 4.4: (a) Shows a binary trie, while (b) shows a level compressed trie. The level compressed trie in (b) is obtained from (a) by replacing the top-most i complete levels of the trie with a single node of branching factor 2^i, and then repeating the process on the children.

definitions of an LPC-trie given by Nilsson and Tikkanen [2002].

Definition A *path compressed burst trie* or *PCB-trie* with bucket capacity c containing n keys is a tree with the following properties:

1. If $n = 0$, the PCB-trie is empty.

2. If $n \leq c$, the PCB-trie is a bucket data structure containing the n keys and their associated values.

3. If $n > c$, the PCB-trie consists of an internal node with 2^i children, $i \geq 1$, and a binary string x. The string x is the longest common prefix of all the keys stored in the PCB-trie. For each binary string y of length i, there is a child trie containing all the xy-suffixes of the keys.

Definition A *level and path compressed burst trie* or *LPCB-trie*, with bucket capacity c, is a tree with the following properties:

1. The root is a PCB-trie with bucket capacity c having 2^i children, where $i \geq 1$ is chosen as large as possible such that the root has no empty child tries.

2. Each child trie of the root is an $LPCB$-trie.

Figure 4.4 shows a simple example illustrating the idea of level compression in a trie. Nilsson and Tikkanen [2002] maintain level compression dynamically through *doubling* and *halving* of trie nodes. Doubling a trie node yields a new trie node having twice the number of children of the original node, with the children shifted up (at most) one level of the trie accordingly. Similarly, halving a trie node reduces its branching factor by half, shifting down its children through the introduction of new binary nodes where necessary.

The level compression defined for the $LPCB$-trie is what Nilsson and Tikkanen [2002] refer to as *perfect*. In a dynamic setting, a more relaxed approach to level compression results in a more robust data structure. Nilsson and Tikkanen point out that maintaining perfect level compression dynamically can cause a simple sequence of alternating insertions and deletions to cause computationally expensive restructurings of the trie. Instead it is desirable to associate two parameters $\alpha, \beta \in [0,1], \alpha < \beta$ with the trie. A node of branching factor 2^i is halved if the number of empty sub-tries is greater than or equal to $\lfloor \alpha 2^i \rfloor$. A node is doubled if the result is a new node of branching factor 2^i, of which at least $\lfloor \beta 2^i \rfloor$ of its subtries are non-empty. In all experiments in this chapter, we have used $\alpha = 1/4$ and $\beta = 3/4$. Nilsson and Tikkanen noted experimentally that this more relaxed approach, somewhat surprisingly, could result in LPC-tries of smaller total depth than perfect level compression. Later, Janson and Szpankowski [2007] showed that using this so-called *partial fill-up* in a level compressed trie results in a *smaller* constant factor in the leading term of asymptotic search times in level compressed tries in the Bernoulli model.

We found that an $LPCB$-trie with suitably engineered level compression gave, in general, the best performance in both time and space compared to all the other trie data structures in our experimental comparison. In the experiments presented in this chapter the minimum branching factor of any node in the level compressed trie structures is 16. We found this to offer a reasonable trade-off between time and space in the trie. Allowing nodes with very small branching factors can lead to large amounts of memory allocation, note also that a substantial proportion of the storage of smaller nodes is simply overhead from the memory allocations required for the node and its internal structures.

We describe the engineering of level compression in the following section.

4.3.4 Engineering Level Compression

As described in the previous section, level compression can be maintained dynamically through doubling and halving of trie nodes introduced by Nilsson and Tikkanen [2002].

Their trie structure begins with binary trie nodes, which then expand when they have a sufficiently large number of non-empty subtries. We note that doubling and halving operations can be generalized to the case where the number of bits trie nodes branch on grows by j bits at a time, $j \geq 1$. We refer to 2^j as the *growth factor* of the trie. When nodes grow or shrink, the number of subtries respectively increases and decreases by the growth factor.

We investigated the case where the minimum branching factor of a trie node and the growth factor were both 2^j, for $j \in \{1, 2, 4, 8\}$. There are two reasons the minimum branching factor and the growth factor are kept equal, the first is that it simplifies the implementation. It also simplifies the implementation if j divides the number of bits in the keys in the trie, explaining the choices of j we have used. The second, less important reason is that when a trie node grows by j bits, only subtries branching on j or fewer bits have their depth reduced. So if the minimum branching factor is j, growing by fewer than j bits will never reduce the depth of any subtries. Of course, growing several times could eventually reduce the depth of the subtries, and even growing by j bits does not guarantee a reduction in depth of any subtrie – if the subtries have already grown. In general however, we found this aggressive growth of j bits at a time to perform well experimentally (see Section 4.4 and Section A.2).

With this more aggressive trie node growth, we use a different critera to Nilsson and Tikkanen [2002] in order to decide when to increase the branching factor of a node. The $LPCB$-trie keeps track of the total number of (distinct) keys it contains. A parameter γ is associated to the trie, γ is the maximum number of subtrie pointers per key that a node is allowed to contain. The invariant maintained at each trie node is that $B < \gamma n$, where B is the branching factor of the trie node and n is then number of distinct keys in the trie. In other words, γ is the space overhead per key allowed for trie nodes. After a key insertion, there is the possibility that certain trie nodes can be grown while maintaining the given invariant. We use a very simple local property similar to Nilsson and Tikkanen's mentioned above. Namely, if an insertion causes a trie node to contain at least $\lfloor \alpha B \rfloor$ non-empty subtries, then it is grown (assuming, as just mentioned doing so does not violate the space invariant). More complicated priority schemes could also be used to decide the best node to grow, for example, the node whose growth reduces the depth of the largest number of other nodes could be chosen. However, we have not examined the trade-offs in practice that result between maintaining these more complicated priority schemes and the possible performance improvements they yield. After a sequence of deletions it is possible that certain nodes in the trie should be shrunk. A node is shrunk (i.e. the number of bits it branches on reduced by j bits)

if it violates the space invariant $B > (\gamma + \epsilon)n$. The parameter $\epsilon > 0$ is included to keep the amortized cost per insertion and deletion operation reasonable, otherwise, it is possible that an alternating sequence of insertions and deletions could result in a node repeatedly growing and shrinking. We provide further details in Section 4.3.5, where we describe how the individual operations on the *LPCB*-trie function.

4.3.5 Operations

The preceding sections have described the data structures required for efficiently extending burst tries to an ordered data structure. The goals of this section are two-fold. Firstly, to show show how the bucket, in-node and trie data structures described above can be used efficiently to provide a burst trie with all the usual operations associated with a dynamic ordered data structure. Secondly, to give important details of how the operations are implemented, together with their time complexities. The time complexities are provided so that the dependencies on the various parameters of the *LPCB*-trie can be understood and summarized succinctly.

The operations described below are for a *LPCB*-trie with maximum branching factor b and bucket capacity c. Recall that we use an OR-tree as the in-node structure for the trie nodes, and growable sorted arrays as buckets in the trie. We denote the maximum height of the trie as h, the value of which is determined by the number of bits in the keys and the minimum branching factor allowed for nodes in the trie. In describing the time complexities given below, we refer to the worst-case time unless otherwise qualified. For clarity, we now give values for the parameters of our *LPCB*-trie implementation:

- The minimum bucket capacity (i.e. the capacity when a new bucket is created) is 2 keys, upon insertion, buckets double until a maximum size of 128 keys. This gives the parameter c above as 128.
- The minimum branching factor, m, of any trie node is 16. When nodes grow and shrink, they increase or decrease in size by 16 times. The maximum branching factor permitted is $b = 2^{20}$.
- Empirically, by examining space usage, the space overhead parameter, was chosen as $\gamma = 8 \times 10^{-3}$. Also experimentally, the slack parameter was chosen as $\epsilon = 3 \times 10^{-3}$.
- Finally, the word-size, w, was chosen as either 32 or 64 bits depending on the input. The maximum height of the trie, h, is $w/\lg m$ where m is the minimum

branching factor of any node. Since we chose $m = 16$, h is 8 in the 32 bit case, and 16 in the 64 bit case.

In describing the following operations on the $LPCB$-trie, we assume that its leaves (i.e. the buckets) are maintained in order in a doubly linked list.

Locate. We first describe the *locate* operation, which finds the value associated with the largest key less than or equal to a supplied key k (or NIL if there is no such key). Assuming the path in the burst trie determined by k leads to a bucket, then that bucket is searched for the largest key less than or equal to k, and its corresponding value is returned. If k is not found in this bucket, the result is the largest key in the bucket immediately before it in the linked list of buckets, which can be found in constant time. In this case, the locate operation takes $\Theta(h + \log c)$ time. In the case where k does not lead to a bucket, the in-node data structure can be used to find a bucket requiring $\Theta(h + \log b)$ time, thus *locate* requires $\Theta(h + \max\{\log b, \log c\})$ time.

Insert. Insertions to an $LPCB$-trie are one of three types.

1. An existing, non-full bucket is found for the key. In this case, insertion takes time $\Theta(h+c)$. Recall that the buckets are growable sorted arrays, requiring $\Theta(c)$ time to insert into.

2. Insertion of key k requires the creation of a new bucket. In this case, the in-node data structure and doubly linked list of buckets must be updated. This requires finding the two buckets whose keys are the immediate predecessors and successors of k, and can be accomplished in $\Theta(h + \log b)$ time. Note that the in-node data structures should be augmented with indices storing the minimum and maximum non-nil pointer at each node, which we refer to as the node's LOW and HIGH fields respectively. The LOW and HIGH fields are used to avoid the process of locating the predecessor and successor buckets requiring $\Theta(h \log b)$ time. When a new bucket is created, it may cause the growth of a trie node, requiring $\Theta(b)$ time. However, a simple argument shows that due to the use of relaxed level compression, the amortized time spent on node growth per insertion can be regarded as constant – as we describe below.

3. An existing, full bucket is found for k. In this case, the insertion can take time $\Theta(hc)$. The $\Theta(hc)$ time is required to find the longest common prefix of all keys in the bucket to be burst, so that path compression is maintained. Note that without path compression $\Theta(hc)$ time is also required, because all the keys may repeatedly enter the same new bucket and require it to burst. Figure 4.2(a)

Name	Architecture	Clock Speed	L2 Cache Size	Compiler
Knuth	Core 2 6400/32-bit	2.13 GHz	2 MB	gcc v3.4.6 -O3
Beaker	Xeon 5138/64-bit	2.13 GHz	4 MB	gcc v4.3.3 -O3
Stoker	Xeon 5140/32-bit	2.33 GHz	4 MB	icc v9.1 -O3
Melody	UltraSPARC IIIi/32-bit	1.28 GHz	1 MB	gcc v3.4.5 -O3

Table 4.1: The basic specifications of the machines used for gathering experimental results for the integer data structures.

illustrates the bursting of a bucket. Since a bucket can burst at most once every c insertions, a straightforward argument can be used to show at most amortized $\Theta(h)$ time is spent bursting buckets per insertion.

In total, insertion requires $\Theta(h + \max\{c, \log b\})$ amortized time.

Here we do not include a term of $A = \Theta(\gamma m/(m-1))$ that results in a sequence of insertions. This term results from the fact that growing a node to branching factor m^k, $k > 1$, requires at least m^k/γ insertions, due to the space invariant introduced in the previous section. Summing the geometric series for the total cost of node growth and amortizing over this number of insertions gives the A term as shown. For any realistic choice of γ and m the term A can be ignored (for instance we chose $\gamma = 8 \times 10^{-3}$ and $m = 16$ above) and so the time complexity for insertion can be regarded as amortized $\Theta(h + \max\{c, \log b\})$.

Other Operations. Deletion from an $LPCB$-trie is similar to insertion. When an empty bucket is deleted from a node, the in-node data structure and linked list of buckets should also be updated. In addition, a node may be shrunk as a result of a deletion. In general, deletion takes $O(h+\max\{\log b, c\})$ amortized time. Note that this amortized time for deletion does not include the small magnitude term, similar to A just mentioned for insertion. In a mixed sequence of insertions and deletions the amortized costs can easily calculated by considering the minimum number of operations that can occur between growing and shrinking a node. For example, having grown a node to branching factor m^k, requiring at least m^k/γ insertions, for it to shrink, requiring $\Theta(m^k)$ time, the number of keys must be less than $m^k/(\gamma + \epsilon)$, and so the amortized cost per insertion or deletion in this sequence is $\Theta(\gamma/\epsilon(\gamma + 1))$. Iterating over the keys in order can be easily implemented using the linked list of buckets. Since the buckets are sorted arrays, in-order iteration is likely to exhibit excellent locality of reference.

4.4 Experimental Comparison

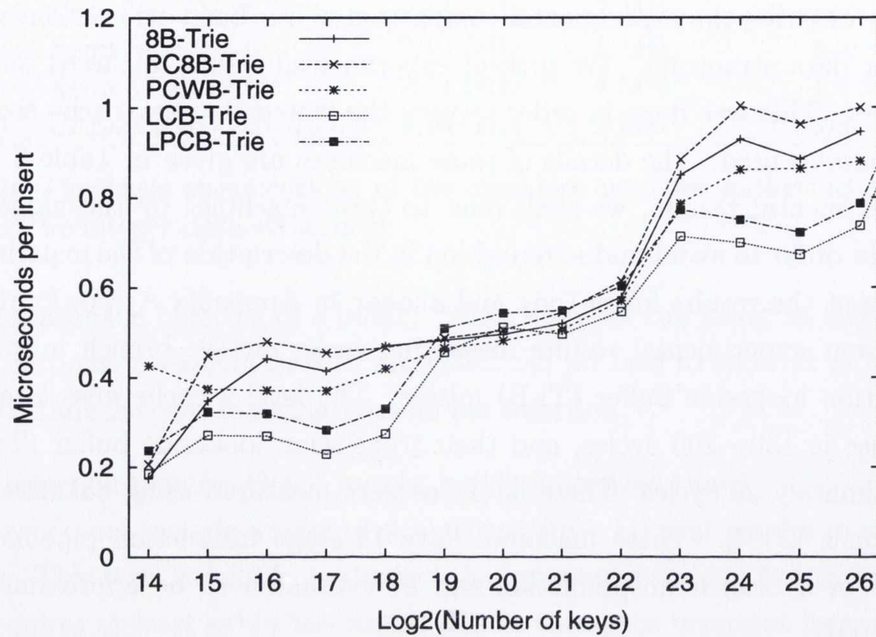
We now describe the experimental comparison of our burst trie variant with a number of other data structures. We present experimental results gathered on four separate machines. This was done in order to vary the instruction set, cache sizes, word sizes, and compiler used. The details of these machines are given in Table 4.1. In referring to experimental results, we shall refer to these machines by the name given in this table. In order to avoid undue repetition in the description of the experimental results, we present the results for `melody` and `stoker` in Appendix A. For `knuth` and `beaker` we present experimental results measuring cache misses, branch mispredictions and translation lookaside buffer (TLB) misses. The level 2 cache miss latencies on these machines is 130—200 cycles, and their translation lookaside buffer (TLB) latency is approximately 20 cycles. These latencies were measured using `calibrator` Manegold and Boncz [2004]. These machines have 14 stage instruction pipelines, and so the latency of a branch misprediction can be estimated to be approximately 14 cycles [Intel 2007].

Our experiments investigate the case where the entire data structure fits in main memory. We used the POSIX standard `gettimeofday` function for timing measurements, because it offered microsecond resolution timing and is portable to any POSIX system. Memory use was measured by augmenting the memory allocator to maintain counts of allocated memory, including the overhead of the allocator. Our memory allocation results report the number of bytes requested by the data structures themselves, not the entire program. Our memory measurements were conducted in experiments separate to our timing experiments, and all experiments were conducted on a lightly loaded machine.

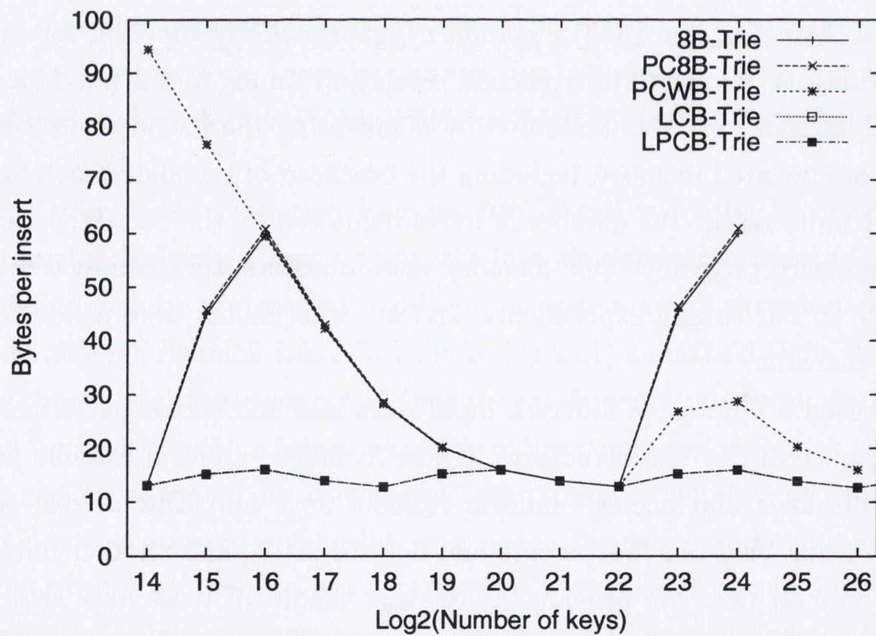
We used a number of different input data sets and access patterns to compare the performance of the data structures. These included: uniform random keys and access, biased-bit keys and access, uniform random keys and Zipf access, as well as data derived from *Valgrind* [Nethercote and Seward 2007], and another data set comprised of Genome strings. We provide the relevant details of these data sets in the sections below discussing the relative performance of the data structures upon them, Table 4.2 provides a brief summary of some these data sets.

4.4.1 Burst Trie Configuration

Our goal is for our burst trie variant to be efficient in both time *and space*. In particular, we believe that for our burst trie variant to be of wide applicability it should occupy

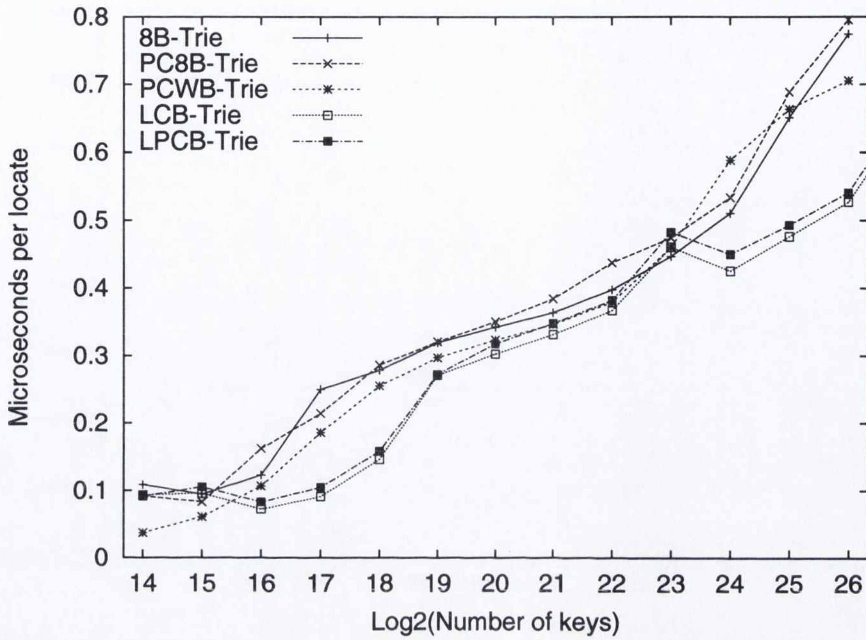


(a)

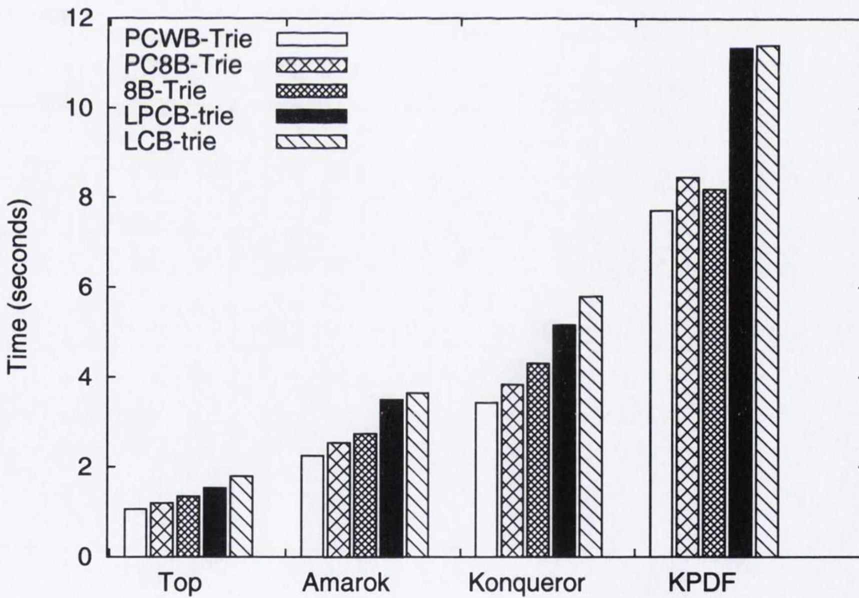


(b)

Figure 4.5: (a) Shows the average time per insertion operation, while (b) shows the space required by the data structures. This figure shows measurements gathered for uniform random 32-bit keys. The data structures are burst tries all sharing the same bucket structure, but with their trie structure varied, as described in Section 4.4.1. These results were gathered on `knuth`.

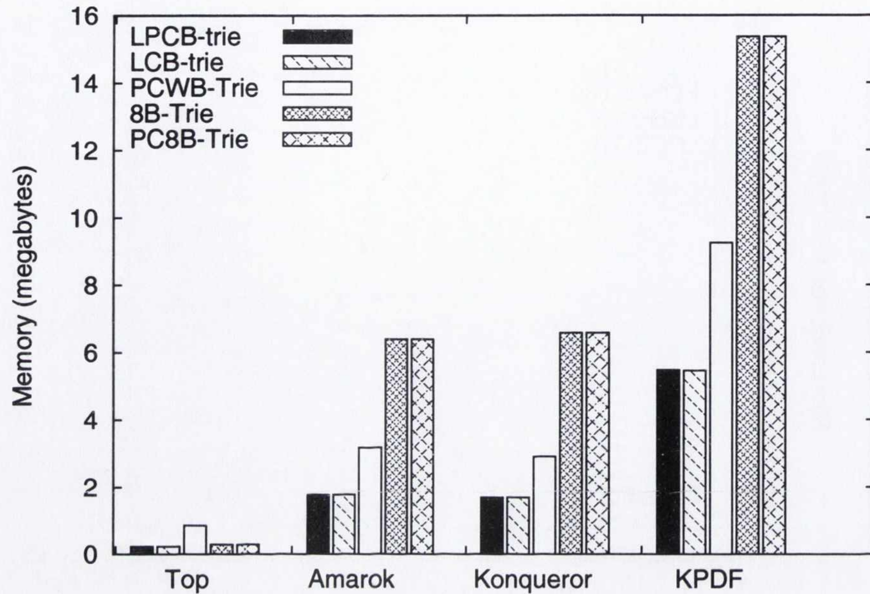


(a)

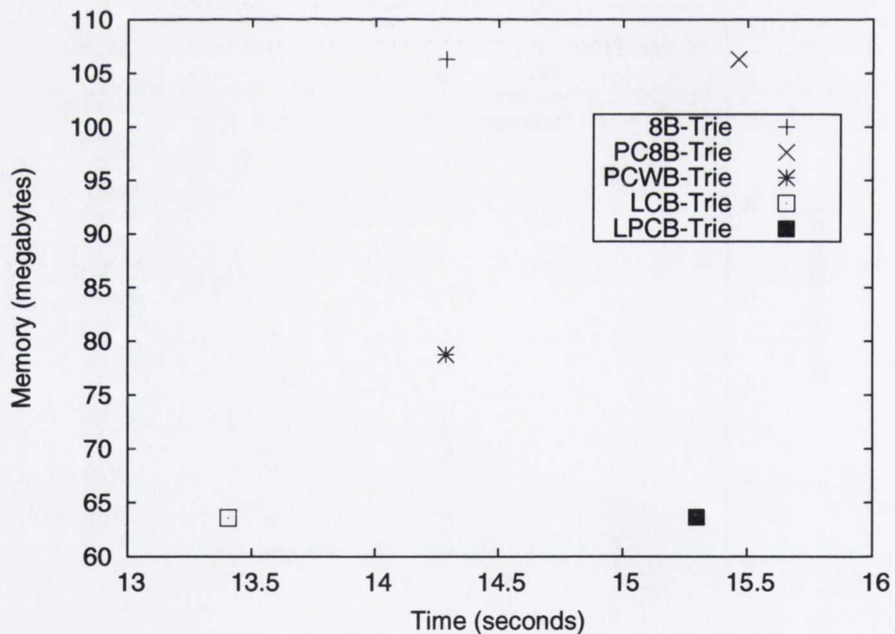


(b)

Figure 4.6: (a) Shows the average time per locate operation for uniform random 32-bit keys. (b) Shows the time for processing the 32-bit Valgrind data sets. The data structures are burst tries all sharing the same bucket structure, but with their trie structure varied, as described in Section 4.4.1. These results were gathered on `knuth`.



(a)



(b)

Figure 4.7: (a) Shows the space required by the data structures for processing the 32-bit Valgrind data sets. (b) Shows the time and space required by the data structures for processing the 32-bit Genome data set. The data structures are burst tries all sharing the same bucket structure, but with their trie structure varied, as described in Section 4.4.1. These results were gathered on `knuth`.

Name	Total keys		Distinct keys	
	32-bit	64-bit	32-bit	64-bit
Uniform random	1.3×10^8	6.7×10^7	1.3×10^8	6.7×10^7
Top (Valgrind)	1.7×10^7	2.2×10^7	2.0×10^4	4.0×10^4
Amarok (Valgrind)	3.3×10^7	4.0×10^7	1.3×10^5	2.4×10^5
Konqueror (Valgrind)	5.4×10^7	6.4×10^7	1.4×10^5	5.2×10^5
KPDF (Valgrind)	1.0×10^8	1.3×10^8	4.3×10^5	8.9×10^5
Genome	1.6×10^7	1.6×10^7	5.0×10^6	5.0×10^6

Table 4.2: This table shows characteristics of the data sets used in our experimental comparison. For the uniform random data, we used data sets ranging of at most 2^{27} insertions in the 32-bit case, and 2^{26} insertions in the 64-bit case. The characteristics given are for this maximum sized uniform random input. The four data sets: Top, Amarok, Konqueror and KPDF are derived from Valgrind. Further details of the data sets are given where we consider experimental results gathered using them.

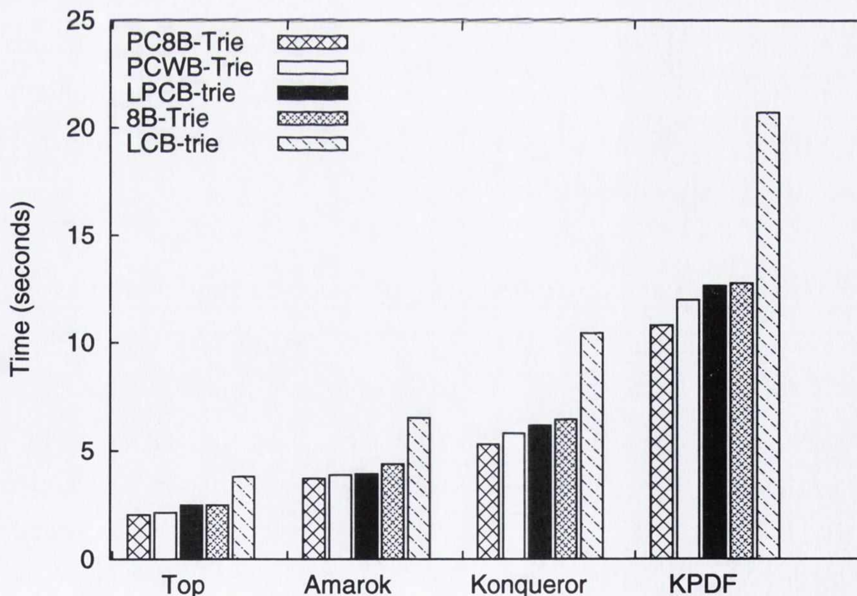


Figure 4.8: This figure shows the time required by the data structures for processing the 64-bit Valgrind data sets. The data structures are burst tries all sharing the same bucket structure, but with their trie structure varied, as described in Section 4.4.1. These results were gathered on `beaker`.

Name	Path	Level	Wide Root	Min. Branching Factor
8B-Trie	No	No	No	$2^8 = 256$
PC8B-Trie	Yes	No	No	$2^8 = 256$
PCWB-Trie	Yes	No	Yes	$2^6 = 64$
LCB-Trie	No	Yes	No	$2^4 = 16$
LPCB-Trie	Yes	Yes	No	$2^4 = 16$

Table 4.3: The burst trie organizations experimented with in Section 4.4.1. The column "Path" refers to whether the trie is path compressed or not while "Level" refers to whether the trie is level compressed. When a trie has a "Wide Root" its root nodes's branching factor is 2^{16}.

no more space than a well implemented comparison based data structure. In this section we examine experimentally two aspects of our burst trie variant that have a significant impact on its performance in time and space. Firstly, we examine the trie structure used to index the buckets, considering simple tries that do not make use of path and/or level compression as well as both level and path compressed tries. Secondly, we examine the maximum bucket size used by the burst trie. We do not intend the experimental results in this section to provide an exhaustive examination of all the parameters that influence the performance of our burst trie variant, instead, we have chosen to present two in detail that we believe are important. Experimental results for a third parameter, the growth factor and minimum branching factor of trie nodes are presented in an appendix (see Section A.2), and are not as extensive as the results for these preceding two factors.

Trie Organization

The choice of trie structure used to index the buckets of a burst trie has a significant influence over the performance in both time and space of the data structure. As mentioned above, we have found that an *LPCB*-trie offers excellent performance in time and space. In this section, we experimentally compare an *LPCB*-trie with several alternative burst trie structures: (1) A burst trie with level compression but without path compression, referred to as an *LCB*-trie. (2) A burst trie where each node has branching factor 2^8 and with path compression, referred to as a *PC8B*-trie. (3) A burst trie where each node has branching factor 2^8 and without either level or path compression, referred to as an *8B*-trie. Finally, (4) A burst trie with a wide root node of branching factor 2^{16} with all children nodes having branching factor 2^6, and with path compression. We refer to this latter burst trie as a *PCWB*-trie. In the level compressed tries, all nodes begin with branching factor 2^4. These burst trie

organizations are summarized in Table 4.3.

Figure 4.5(a) shows the time required by these burst trie variations to perform a sequence of insertions of 32-bit uniform random keys. Clearly the two level compressed tries, the LCB-trie and $LPCB$-trie, perform best on this data. The two level compressed tries also require the least space, shown in Figure 4.5(b). Due to its wide root the $PCWB$-trie is very inefficient in space for small input sizes. As we shall see in Section 4.4.2, a well-implemented B-tree requires approximately 20 bytes per insertion[†] for this data. Even for large sets of keys the $PCWB$-trie exceeds this (e.g. at 2^{24} insertions it requires approximately 27 bytes per insertion). Figure 4.5(b) also shows that the two simpler trie structures, the $8B$-trie and $PC8B$-trie are also highly inefficient in space. On this data only the LCB-trie and $LPCB$-trie have acceptable space usage, of consistently between 12 and 17 bytes per insertion. In addition, they are also the most efficient structures for the *locate* operation for 32-bit uniform random keys, shown in Figure 4.6(a). The locate operation for a key k returns the value associated with the largest $k' \leq k$. Note that this operation is more general than can be answered efficiently with a hash-table. On this uniform random data, it appears that the path compression used by the $LPCB$-trie constitutes an overhead in time compared to the LCB-trie. This can be seen in both Figure 4.5(a) and Figure 4.6(a), where the LCB-trie performs slightly better than the $LPCB$-trie.

Figure 4.6(b) shows the performance of the burst trie variations on the 32-bit Valgrind data sets. On this data the simpler burst trie structures out-perform the level compressed tries, over-all the $PCWB$-trie performs best in time. Figure 4.7(a) shows that the two level compressed tries offer the best performance in space, with the $8B$-trie and $PC8B$-trie trie structures again requiring space usage that exceeds what is required by a well implemented comparison based data structure (see e.g. the B-tree memory consumption results in Figure 4.19(b)). Figure 4.7(b) shows the time and space required by the data structures for the 32-bit Genome data set. Here, the level compressed trie structures perform best with the LCB-trie performing slightly better than the $LPCB$-trie. On all the data just presented, the LCB-trie and $LPCB$-trie both offer acceptable memory usage (that is, close to or less than a comparison based structure for the same data) and the LCB-trie generally performs slightly better in time than the $LPCB$-trie. That is, path compression appears to constitute an over-

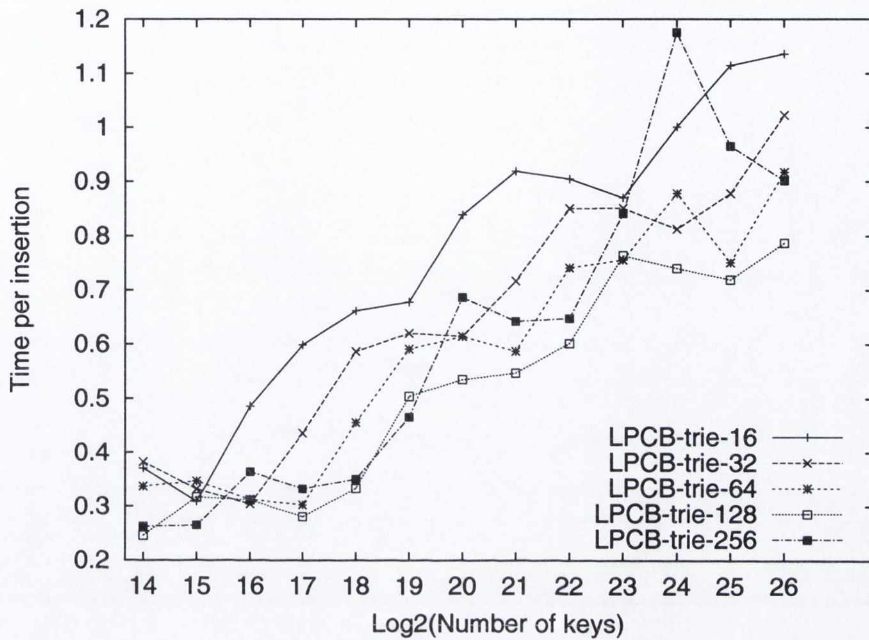
[†]We say "per insertion" rather than "per key" because we simply generate a new uniform random key for each insertion, thus the keys are not distinct – although the number of duplicates is very small compared to the total number of keys inserted, never exceeding 1.6% in the 32-bit random data. There are no duplicates in the 64-bit random data. Note also that the data structures contain values associated with the keys. In our experiments these values contain the same number of bits as the keys.

head. However, for longer keys, path compression can be important. Figure 4.8 shows the performance of the burst trie variations on the 64-bit Valgrind data sets. Note that the $LPCB$-trie performs substantially better than the LCB-trie. For these data sets, the use of path compression is important. For example, on the 64-bit top Valgrind data set, almost all keys have their upper 28 bits consisting of zeros (the keys involved are in fact memory addresses stored to and loaded from by a real program). This long common prefix of the keys gives rise to sparsely populated nodes towards the root of the trie which are never level compressed due to their sparsity. This results in a long path in the trie to distinguish the keys in the LCB-trie, whereas in the $LPCB$-trie the path compression can distinguish the keys with a single trie edge. Given the importance of path compression for longer keys, demonstrated on the 64-bit Valgrind data sets, we have chosen to use a level and path compressed burst trie, $LPCB$-trie in our experimental comparison with other data structures for 32 and 64-bit integer keys. We next experimentally examine the choice of maximum bucket size used by the $LPCB$-trie.

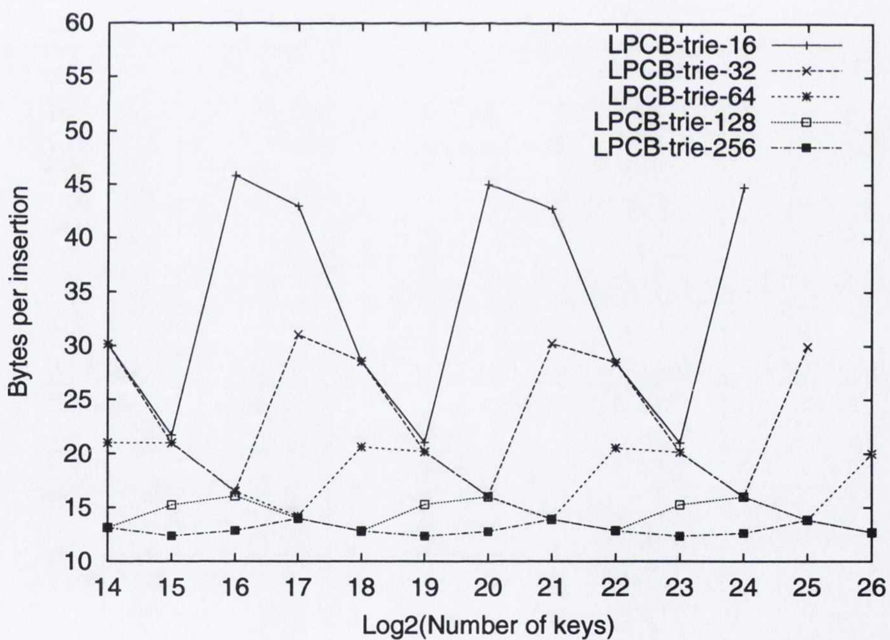
Bucket Size

Recall that the bucket data structure used by the $LPCB$-trie in question is a growable sorted array, described in Section 4.3.2. We experimented with bucket data structures of maximum size 16, 32, 64, 128 and 256 records. A very important consideration for choosing the maximum bucket size is the space usage of the resultant $LPCB$-trie. Intuitively, it seems a larger bucket size should give rise to a more compact data structure.

We begin by examining uniform random data, on `knuth`, a 32-bit Core 2 machine. Figure 4.9(a) shows the time required for insertions into the $LPCB$-trie, for these bucket sizes, generally speaking a bucket size of 128 records gives the best performance on this machine. Figure 4.9(b) shows that as the bucket size is increased, the space usage of the trie decreases. The periodic peaks in the memory usage of the data structure are likely a result of node growth due to level compression, possibly combined with bucket bursting. As we will see in Section 4.4.2, a compact comparison based data structure (such as a B-tree) requires about 20 bytes per insertion for this data. For a bucket size of 128 keys, the space required is slightly less than this: between 12 and 17 bytes per insertion. Figure 4.10(a) shows the time per locate operation for uniform random data as the bucket size varies. The difference in the time required by the data structures for the locate operation, is much smaller than is observed for the insert operation of Figure 4.9(a). Broadly speaking, the two larger bucket sizes of 128 keys

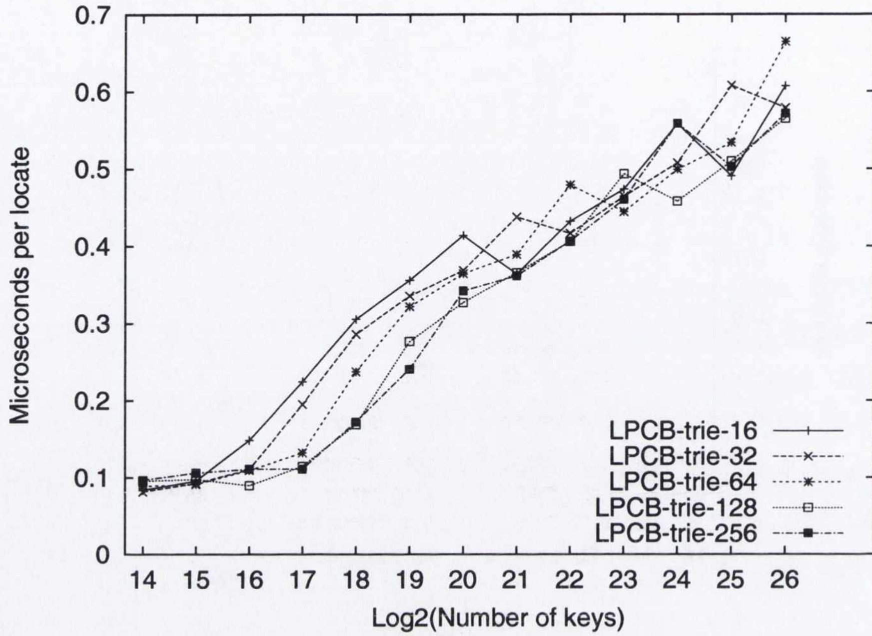


(a)

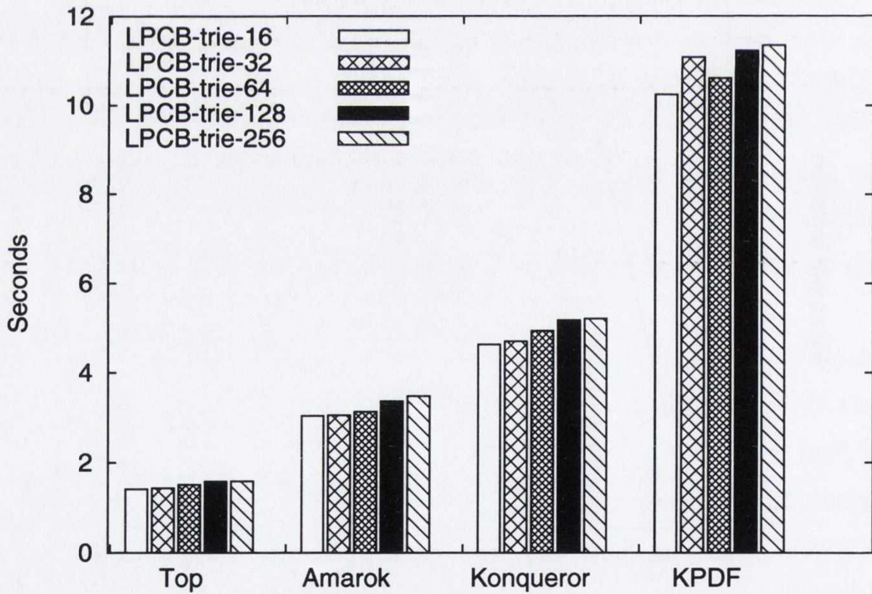


(b)

Figure 4.9: (a) Shows the time required for insertion operations on the data structures, while (b) shows the space required by the data structures. This figure shows measurements gathered for uniform random 32-bit keys. The data structures are level and path compressed burst tries with the maximum bucket size varied between 16 to 256. These results were gathered on the 32-bit machine, and are described in Section 4.4.1.

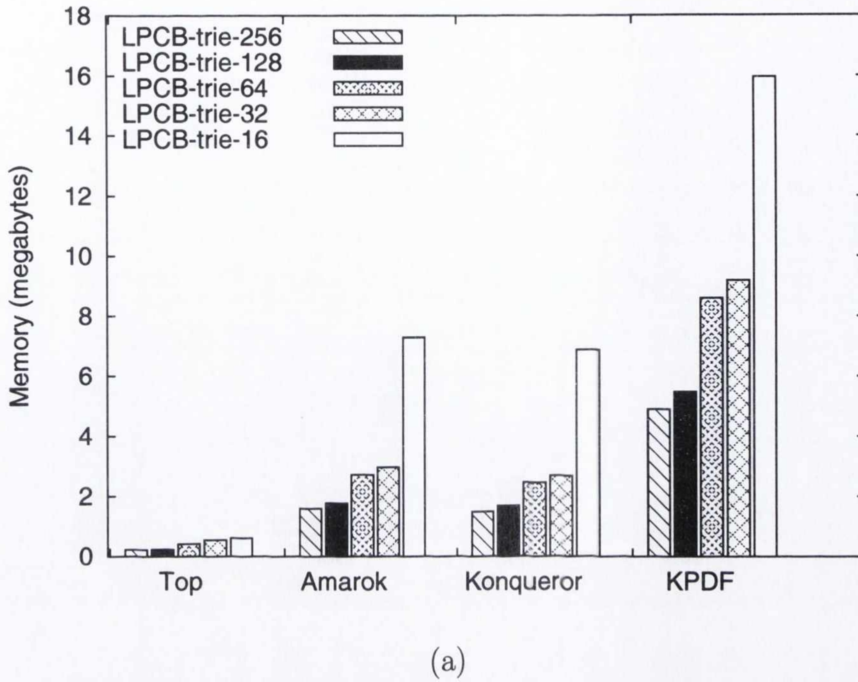


(a)

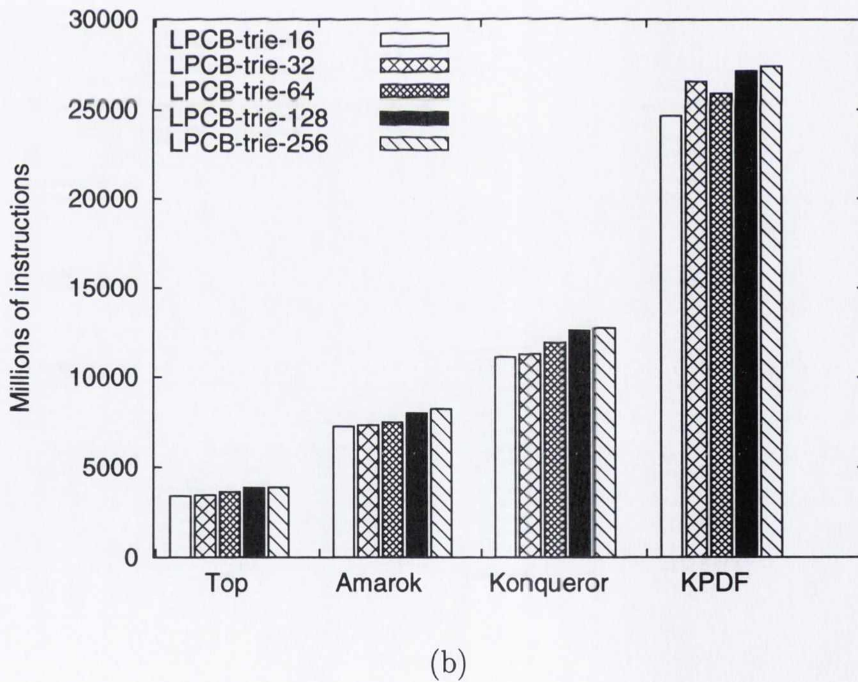


(b)

Figure 4.10: (a) Shows the time required for locate operations on the data structures for uniform random 32-bit keys, while (b) shows the time required by the data structures for processing the 32-bit Valgrind data sets. The data structures are level and path compressed burst tries with the maximum bucket size varied between 16 to 256. These results were gathered on `knuth` and are described in Section 4.4.1.

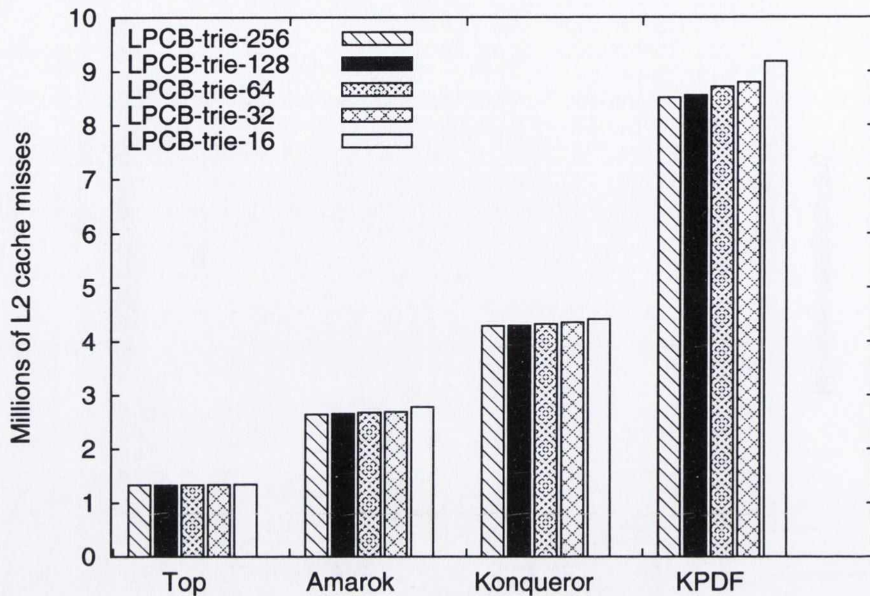


(a)

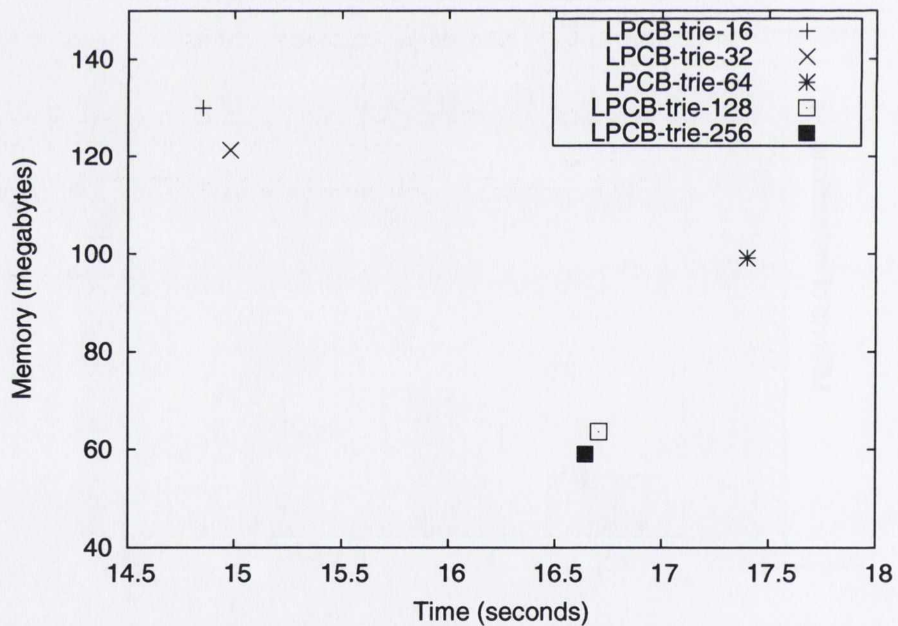


(b)

Figure 4.11: (a) Shows the space required by the data structures for processing the 32-bit Valgrind data sets, while (b) shows the total number of instructions executed for processing the 32-bit Valgrind data sets. These instructions counts were gathered using PAPI [Dongarra et al. 2003]. The data structures are level and path compressed burst tries with the maximum bucket size varied between 16 to 256. These results were gathered on `knuth`, and are described in Section 4.4.1.

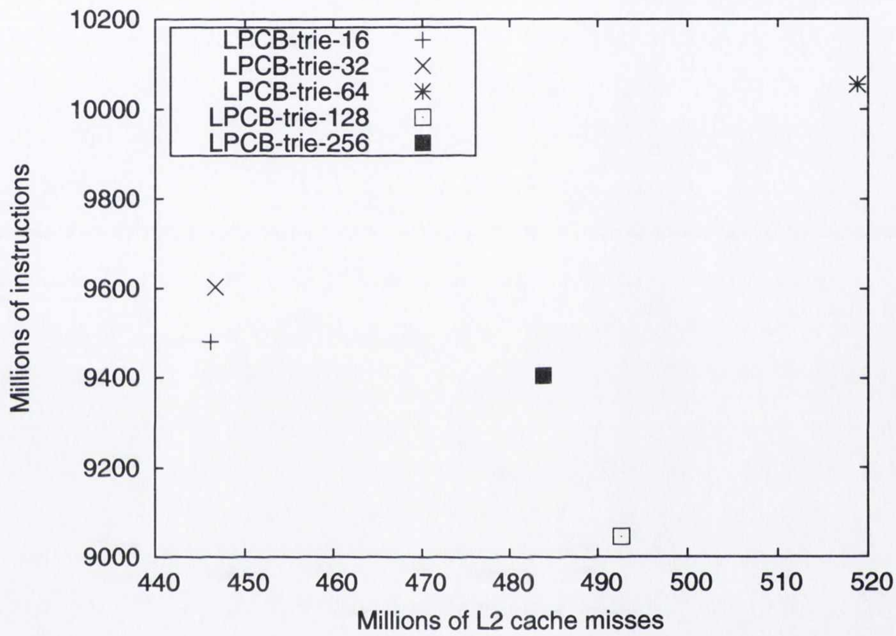


(a)



(b)

Figure 4.12: (a) Shows the level 2 data cache misses caused by the data structures when processing the 32-bit Valgrind data sets, while (b) shows the time and space required by the data structures when processing the 32-bit Genome data set. The data structures are level and path compressed burst tries with the maximum bucket size varied between 16 to 256. These results were gathered on the `knuth` using PAPI [Dongarra et al. 2003], and are described in Section 4.4.1.



(a)

Figure 4.13: This figure shows the level 2 data cache misses and total instructions executed when for processing the 32-bit Genome data set. The data structures are level and path compressed burst tries with the maximum bucket size varied between 16 to 256. These results were gathered on `knuth` using PAPI [Dongarra et al. 2003], and are described in Section 4.4.1.

and 256 keys perform better than the smaller bucket sizes.

We next examine the performance of the data structures on the 32-bit Valgrind data sets. Figure 4.10(b) shows the time required to process the Valgrind data set for the different bucket sizes. Figure 4.11(a) shows the space required for the different bucket sizes on the Valgrind data. As was observed for the uniform random data, the larger bucket sizes result in more compact data structures, with the burst trie of bucket size 16 being especially inefficient in space compared to the other data structures.

In contrast to the random data above, the smaller bucket sizes of 16 or 32 records give better performance in time than the larger bucket sizes of 128 or 256 records on the Valgrind data. Recall from Table 4.2 that the 32-bit Valgrind data sets contain relatively small numbers of distinct keys, resulting in relatively small data structures that miss the second level cache infrequently. Figure 4.11(b) shows the total number of instructions required to process the Valgrind data sets, while Figure 4.12(a) shows the total level 2 cache misses. Even allowing for the much greater cost of a level 2 cache miss compared to the average instruction (recall from Section 4.4, a level 2 cache miss, as we measured on `knuth`, costs approximately 200 cycles), the number of instructions executed clearly dominates. Indeed, comparing the times in Figure 4.10(b) to the instruction counts in Figure 4.11(b) provides strong evidence that the times are determined by the number of instructions executed. On the other hand, for the larger 32-bit Genome data set (Table 4.2 provides details), the relative performance of the data structures does appear to be determined by their level 2 cache misses. Figure 4.12(b) shows that the time and space required to process the Genome data set, while Figure 4.13 shows the level 2 cache misses and number of instructions executed. The level 2 cache misses appear to be dominant in the execution time of the data structures. For example, the $LPCB$-trie with a bucket size of 256 causes executes approximately 9,000 million instructions for this data set, and causes approximately 485 million level 2 cache misses. Recalling again from Section 4.4 that the cost of a level 2 cache miss costa approximately 200 cycles on this machine, they are clearly the dominant component of the execution time. Indeed, the ordering of the data structures along the horizontal axis in Figures 4.12(b) and Figure 4.13 is the same, indicating that the level 2 cache misses are dominant in the execution time. Finally, we note that as was observed for the uniform random data, for both the Valgrind and Genome data sets, as the maximum bucket size increases, the resulting burst trie variants are much more compact. In Figure 4.12(b) the burst trie variants with maximum bucket sizes of 128 and 256 keys are also seen to be much more compact than those with smaller maximum bucket sizes.

Based on the experimental results just presented, we have used a maximum bucket size of 128 records for our burst trie variant. We discount smaller bucket sizes of 16, 32 or 64 records because they result in burst tries that consume significantly more space than comparison based data structures. From the data presented above it appears that a bucket size of 128 or 256 keys results in data structures with similar performance. We note that a bucket size of 128 keys gives rise to a compact data structure, performing slightly better in time on the uniform random and Valgrind data sets than a bucket size of 256 records. Our $LPCB$-trie uses buckets of 128 keys.

4.4.2 Burst Trie Comparison

We now compare the $LPCB$-trie implementation derived from the experimental results above to a number of other data structures. These are:

1. An $LPCQ$-trie based on the data structure of Korda and Raman [1999], described in Section 4.5. Our version of their data structure makes use of the same level and path compressed trie, and bucket data structures as our $LPCB$-trie implementation. The resulting data structure is substantially more efficient and requires less space than the data structure originally described by Korda and Raman.

2. The S-tree data structure of Dementiev *et al.* [2004], described in Section 4.5. Note that the only implementation of this data structure available to us is dependant on the endianness of the underlying architecture, and so it cannot be executed on our big-endian SPARC machine, `melody`. Moreover, as described in Section 4.5 this data structure is inherently restricted to 32-bit keys.

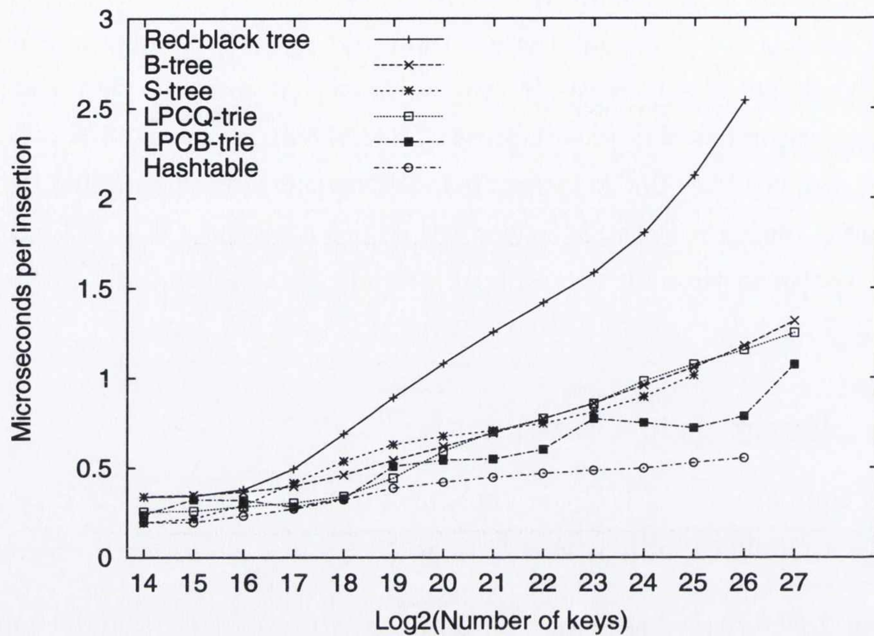
3. A B-tree implementation[‡].

4. A balanced tree implementation provided by the *C++* STL map implementation[§] [Stroustrup 1997]. Note that in all compilers used this data structure was a red-black tree [Cormen et al. 2001].

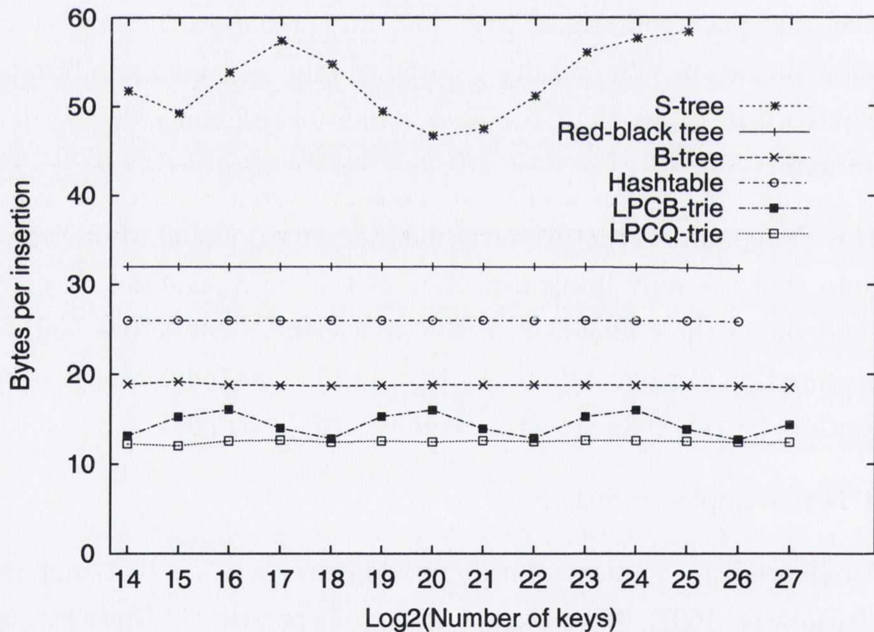
5. Finally, we include a comparison with a simple hash-table implementation. This implementation is the `hash_map` data structure obtained from an extension to

[‡] Obtained from `http://www.textelectric.net`

[§] See `http://gcc.gnu.org`. The STL map implementation used the same code on all compiler versions we present experimental results for (see Table 4.1) – including the Intel C++ Compiler `icc`, which shares the `gcc` STL implementation.

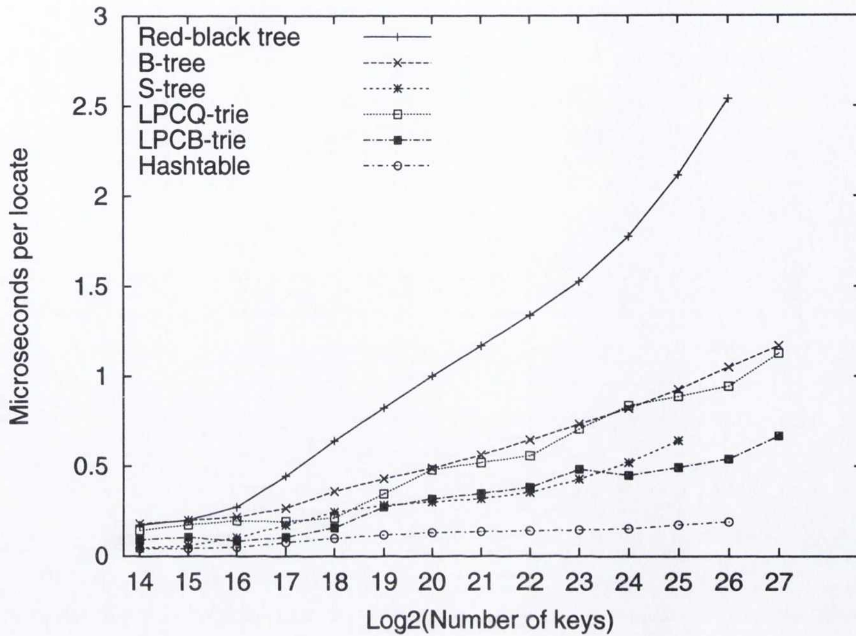


(a)

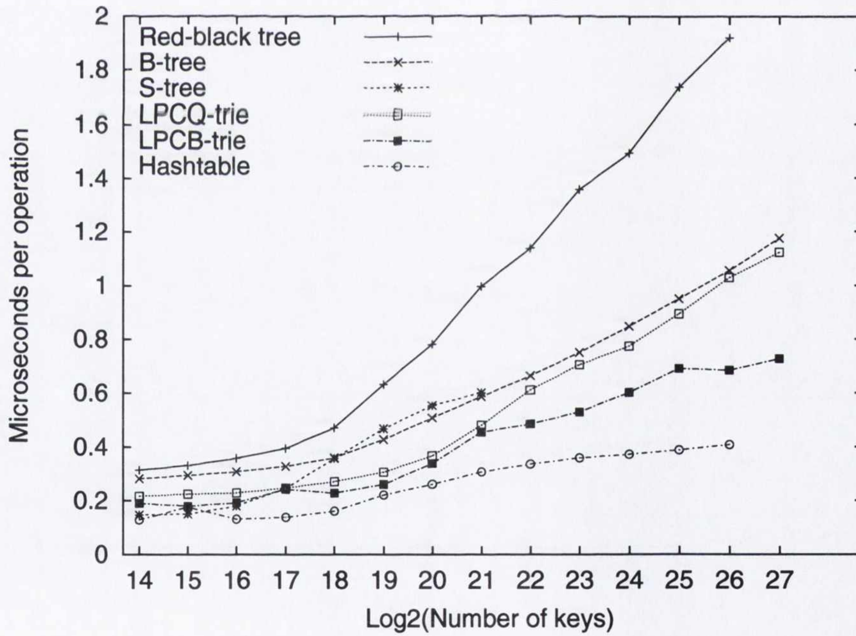


(b)

Figure 4.14: This figure shows measurements gathered for uniform random 32-bit keys. (a) Shows the average time per insertion operation. The *LPCB*-trie performs best, and better than the comparison-based structures at all input sizes, while the *LPCQ*-trie and *S*-tree also perform well. (b) Shows the space occupied by the data structures after a sequence of insertions. The *LPCB*-trie and *LPCQ*-trie require the least space, and the *B*-tree is also competitive. We note that the *S*-tree consistently requires a large amount of extra space compared to the other structures. These results are discussed in Section 4.4.2

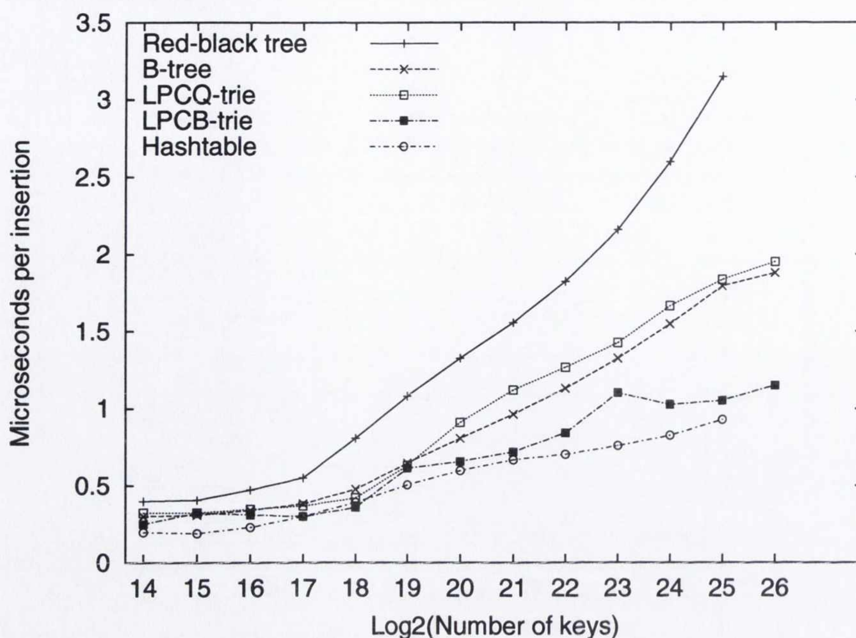


(a)

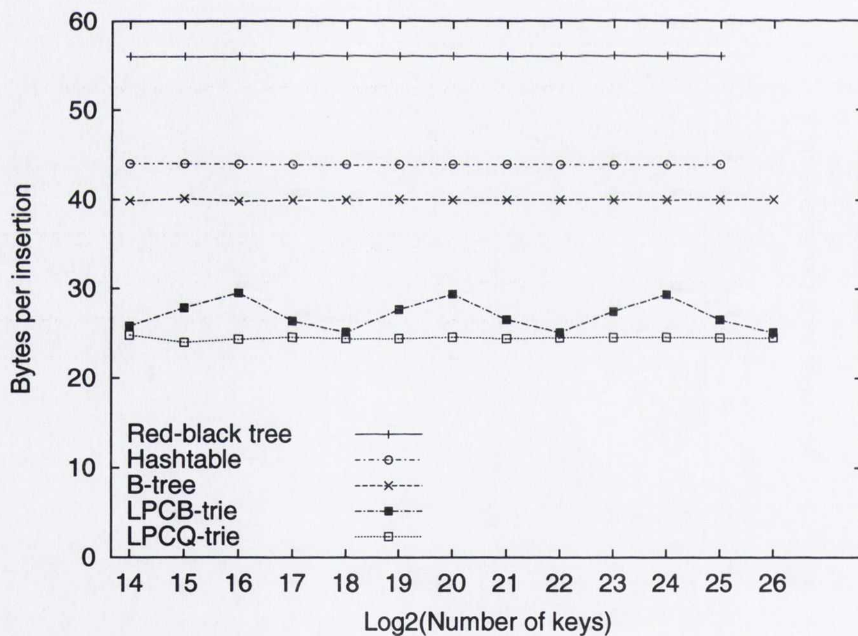


(b)

Figure 4.15: This figure shows measurements gathered for uniform random 32-bit keys. (a) Shows the average time per locate operation on the data structures as the number of keys in the structure increases. Note that results are shown until the data structure occupies all of main memory. The S-tree performs very well, as does the $LPCB$-trie. (b) Shows the average time per operation for a mixed sequence of equiprobable insertions and deletions. The deletions are of keys that have already been inserted to the data structure. These results are dicussed in Section 4.4.2.

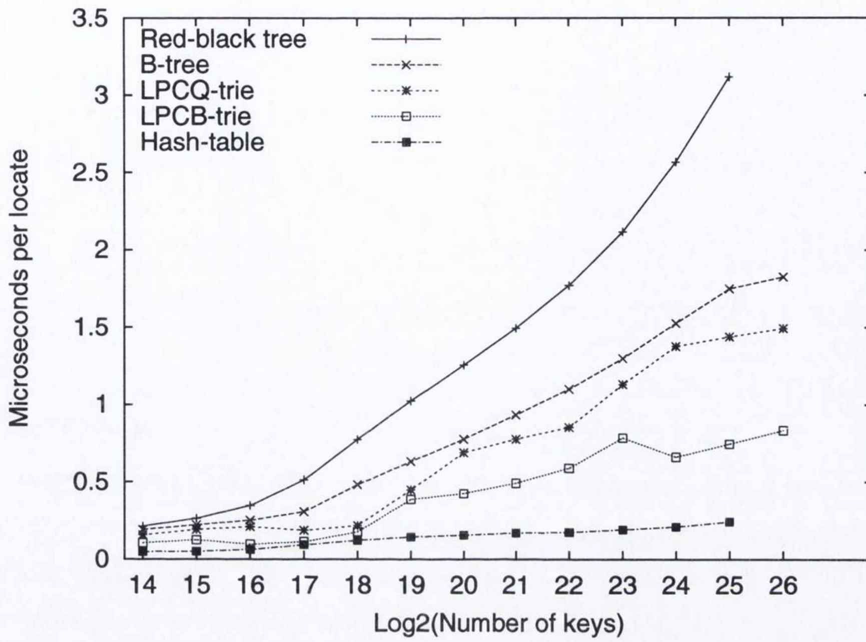


(a)

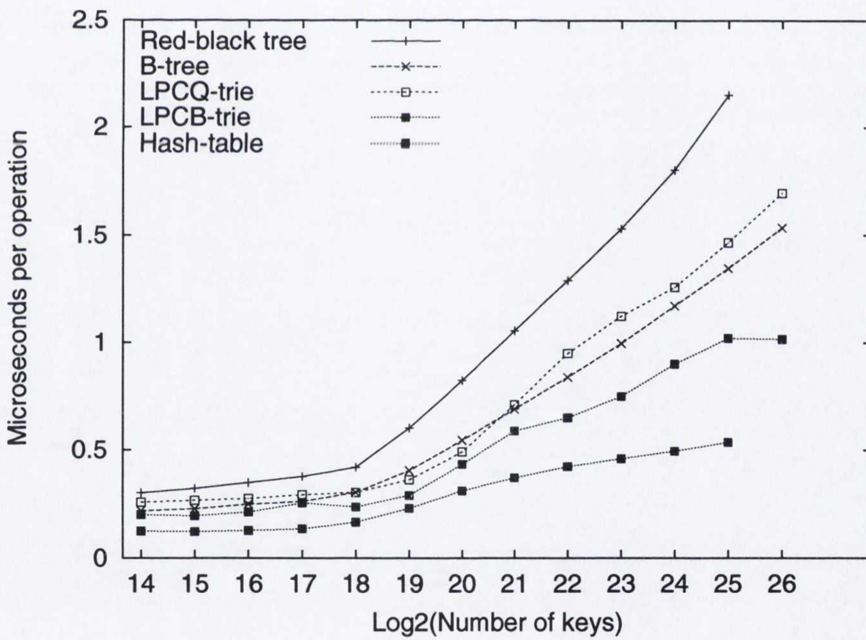


(b)

Figure 4.16: This figure shows measurements gathered for uniform random 64-bit keys, gathered on `beaker`. Note that the S-tree does not appear in this figure because it is restricted to use with 32-bit keys. Results are shown until the data structures occupies all of main memory. (a) Shows the average time per insertion operation. The $LPCB$-trie performs best, and better than the comparison-based structures at all input sizes, while the $LPCQ$-trie also performs well. (b) Shows the space occupied by the data structures after a sequence of insertions. Again the $LPCQ$-trie requires the least space, followed by the $LPCB$-trie. As expected, both use approximately double what is required in the 32-bit case, shown in Figure 4.14(b), as one expects. These results are discussed in Section 4.4.2.



(a)



(b)

Figure 4.17: This figure shows measurements gathered for uniform random 64-bit keys. Note that the S-tree does not appear in this figure because it is restricted to use with 32-bit keys. Results are shown until the data structures occupies all of main memory. (a) Shows the average time per locate operation. The $LPCB$-trie performs best, and better than the comparison-based structures at all input sizes, while the $LPCQ$-trie also performs well. (b) Shows the time required for a mixed sequence of equiprobable insertions and deletions. The deletions are of keys that are already present in the data structures. These results are discussed in Section 4.4.2.

the GNU C++ v3.4.6 STL implementation. This is a simple chaining hash-table. Although a hash-table is less general than the ordered data structures that this chapter focuses on, we include it because it nevertheless provides some insight into the overhead of maintaining order in integer data structures. We note that the hash-table implementation we have chosen is not likely to be the most efficient possible (although as we shall see below, it is more efficient in time than all the ordered data structures), Askitis [2009] provides an experimental comparison of hash-tables for integer keys.

Our implementations of the $LPCB$-trie and $LPCQ$-trie data structures do not feature low-level optimizations, and are not very carefully optimized at the language level. They are also generic, supporting a mapping from any key data type to any value data type. The only restriction is that the keys are all some fixed length concatenation of bits. To give an indication of the genericity of the implementations, the $LPCB$-trie and $LPCQ$-trie differ only in the manner in which buckets are indexed and split – the code implementing their bucket data structures and their level and path compressed trie is shared.

Uniform Random Data

In this section we describe the performance of the data structures over uniform random data. We used Brent's [2004] pseudorandom number generator implementation for generating both 32 and 64-bit random numbers. For this data, the results we present are averaged over several thousand executions of the data structure operations.

Figure 4.14(a) shows the time per insertion for the data structures over 32-bit uniform random keys, on the 32-bit Core 2 machine `knuth`. Note that the S-tree uses all available memory at 2^{25} keys, while the red-black tree and hash-table use all available memory at 2^{26} keys. Clearly, the red-black tree is much less efficient than the other data structures. The noticeable change in slope for the red-black tree's insertion time at 2^{17} keys is due to the fact that `knuth` has a 2 MB = 2^{21} byte level 2 cache. As we will see below, the red-black tree requires 32 bytes per insertion, and $2^{17} \times 32 = 2^{22}$, exceeding the level 2 cache size.

As is to be expected, the hash-table out-performs the data structures that maintain order. However, especially for up to approximately 2^{20} insertions, its performance is close to that of the $LPCB$-trie. Of the data structures that maintain order, the $LPCB$-trie generally performs best.

Even for small numbers of keys the ordered data structures tailored to integer keys

out-perform the comparison based structures. For example, for less than 2^{20} keys on average the $LPCB$-trie is approximately 23% faster than the B-tree.

Figure 4.14(b) shows the memory consumption of the data structures for the insertions of Figure 4.14(a). The S-tree is clearly a very memory hungry data structure, this, combined with its restriction to 32-bit keys limits its practicality. After the S-tree the red-black tree requires the second largest amount of memory per insertion. Note that the data structures store a mapping from a 32-bit key to a 32-bit value. The 32 bytes per insertion required by the red-black tree can be explained by considering the data stored in each node of the tree: 4 bytes for each of: the key, the value, the left child pointer, the right child pointer, the parent pointer, and the node colour (due to alignment). Finally, there is an 8 byte overhead from the memory allocator.

It is noteworthy that the hash-table is not particulary space efficient. The hash-table consists of m linked list chains. After m insertions it approximately doubles in size (the table does not precisely double because its size is always selected as a prime). The main space inefficiency is similar to the red-black tree's. It uses single nodes, in this case of a linked list, to store keys in the chains of the hash-table. A bucketed linked list Frias et al. [2009] would likely be more efficient in both time and space. Nevertheless these results emphasize that classical hash-table implementations are not particulary space efficient.

The B-tree, which uses nodes consisting of arrays of keys of up to 200 keys (this number was chosen because it gave the best performance), uses substantially less memory than the other comparison-based structure, the red-black tree. The $LPCQ$-trie requires approximately 12 bytes per insertion — the least memory of any the data structures. The $LPCB$-trie also has modest memory use, occupying between 12 and 17 bytes per insertion. This difference in memory consumption is likely a result of the different approaches taken to bucket creation by the two data structures. The $LPCQ$-trie splits full buckets into exactly two new half-full buckets. On the other hand, a full bucket in the $LPCB$-trie is burst into a potentially large number of new buckets. Naturally, when many small buckets are created the space overhead per bucket is emphasized. Furthermore, the oscillation in the memory usage of the $LPCB$-trie is likely due to this bursting of buckets too.

Figure 4.15(a) shows the time per locate operation on the data structures, recall from above that `locate`(k) is the value associated with the largest $k' \leq k$. Note that this operation is more general than can be answered efficiently with a hash table. The times reported here for the hash-table in this figure are for the simpler look-up operation, as is expected and was the case for insertions, the hash-table operates the

most rapidly, showing the overhead of even the most efficient ordered data structures compared to an unordered structure.

Typically, data structures are searched more often than they are updated, and so the performance of this searching operation is likely to be very important in practice. Amongst the ordered data structures, the integer-specific structures perform better than the comparison-based structures at all the input sizes shown in Figure 4.15(a). For up to a large number of keys, 2^{24}, the S-tree slightly out-performs the $LPCB$-trie. For the remainder of the input sizes, the $LPCB$-trie performs better. For 2^{25} insertions, the largest data set before the S-tree exhausts main memory, the $LPCB$-trie's locate is approximately 23% faster than the S-tree's. At 2^{27} insertions, the $LPCB$-trie's locate is approximately 41% faster than the locate of its nearest competitor, the $LPCQ$-trie.

Figure 4.15(b) shows the time per operation for a mixed sequence of equiprobable insertions and deletions. The deletions are of keys that have previously been inserted into the data structures. This explains why the average time per operation in this figure is considerable less than what is observed in Figure 4.14(a), since the data structures contain approximately half the number of keys for a given input size due to deletions. Unfortunately the only S-tree implementation available to us[¶] had bugs in its delete operation causing it to fail on inputs larger than 2^{21} operations. As with the results just described, with the hash-table aside, the integer-specific data structures out-perform the comparison-based structures, with the $LPCB$-trie peforming best in general, followed by the next integer-specific data structure, the $LPCQ$-trie.

The results for random 64-bit keys are shown in Figure 4.16 and Figure 4.17. These results were gathered on the 64-bit Core 2 machine `beaker`. For these data sets, memory usage, shown in Figure 4.16(b) is slightly under twice what is observed in the 32-bit case. This is because the keys and values stored in the data structures are now 64-bits in length, as well as their pointers, however, depending on the data structure, other fields remain 32-bits or less. The S-tree is excluded from these results because it is tailored specifically for 32-bit keys, and extending it efficiently to 64-bit keys requires an enormous amount of extra space. For the remaining data structures the results for the 64-bit case are broadly similar to those observed in the 32-bit case. The $LPCB$-trie and $LPCQ$-trie perform better than the comparison-based data structures, with the B-tree performing substantially better than the red-black tree.

The performance difference between the $LPCQ$-trie and $LPCB$-trie is noteworthy, and would likely be rather involved to determine analytically. We note that the $LPCQ$-trie and $LPCB$-trie use exactly the same implementations for their in-node

[¶]Obtained from `http://www.mpi-inf.mpg.de/~kettner/proj/veb/index.html`

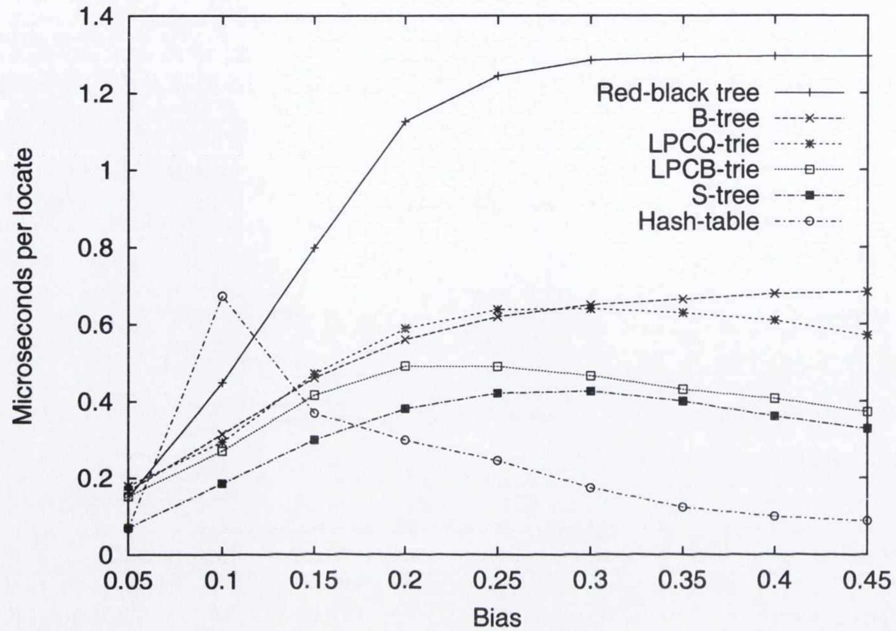
data structures, bucket data structures as well as for the level and path compressed tries. Their performance difference is a result of the fact that all operations in the *LPCQ*-trie begin with a predecessor search in the trie, which is slightly more complicated than a simple trie search. The other difference between the data structures is that the *LPCQ*-trie splits rather than bursts buckets (as described in Section 4.1). The more aggressive bursting of buckets employed by the *LPCB*-trie results in its slightly higher memory usage observed in Figures 4.14(b) and 4.16(b) compared to the *LPCQ*-trie. However, the resulting performance improvement for all operations is substantial.

In general, for both 32 and 64-bit uniform random keys, the *LPCB*-trie offers modest space usage and excellent performance in time compared to the alternative data structures.

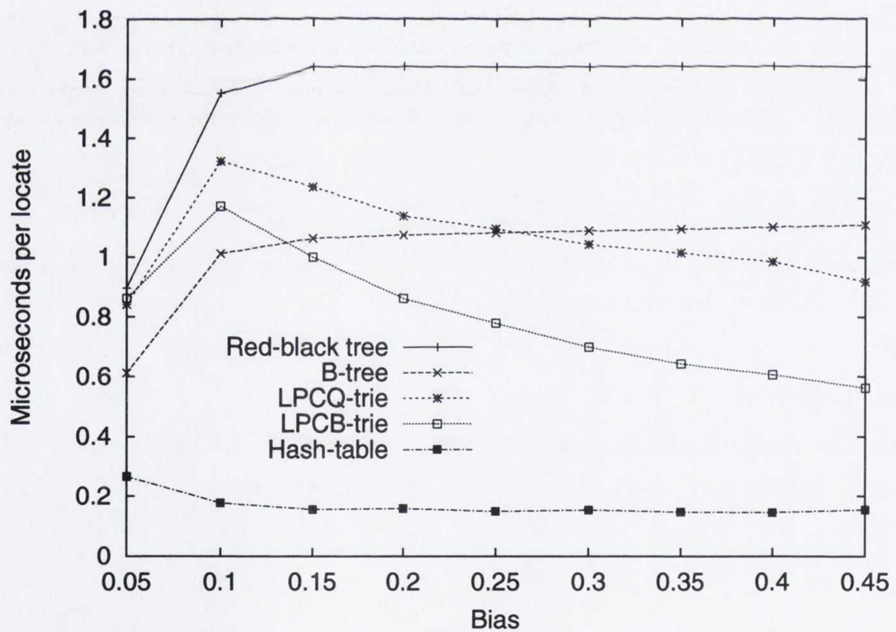
Non-uniform Random Data

The preceding section examined the performance of the data structures over uniform random data. In the current section, we examine *biased-bit* data. In these experiments, a series of $2^{22} \approx 4.2 \times 10^6$ insertions were performed where each bit of the keys is chosen to be a zero independently with probability p. Then a series of `locate` operations were performed using keys chosen in the same way.

Figure 4.18(a) shows the average time taken per `locate` operation as p varies (again for the hash-table, we are measuring the simpler look-up operation), on the machine `knuth`. In this case the S-tree and *LPCB*-trie both maintain a considerable advantage over the other ordered data structures, but only when p is not too small. For very small p, the performance of all the ordered data structures is similar, with the S-tree performing best. Note that as p increases, the number of distinct keys inserted into the data structure also increases. For $p = 0.05$, the first data point visible in Figure 4.18(a), there are approximately $100,000$ distinct keys inserted, for the final data point visible, $p = 0.45$, close to 2^{22} distinct keys are inserted (recall that in total 2^{22} insertions are performed). For the comparison based structures, as the number of distinct keys inserted increases, so too does the resultant search time. However, the time of the searches on each of the integer data structures is seen to increase and then decrease again as p increases. The performance of the hash-table observed in Figure 4.18(a) is quite unexpected. Although it out-performs the ordered data structures as one expects, one would also expect its performance to be independent of the distribution of input keys, but its performance improves as the distribution becomes more uniform. We have verified that this is simply due to reducing instruction count, and does not appear to



(a)



(b)

Figure 4.18: (a) and (b) shows the performance of locate operations on the data structures on the machines `knuth` and `beaker` respectively for biased-bit data, as the parameter, p, used to generate the 2^{22} keys inserted (prior to the locates shown in the figures here) and the query keys is varied. These results are discussed further in Section 4.4.2.

be due to cache or other effects.

Figure 4.18(b) shows the performance of searches for biased-bit data on the machine `beaker`. The keys are 64-bits in length and for $p = 0.05$ approximately 1.3×10^6 distinct keys are inserted into the data structures. There are fewer duplicate keys because the keys are longer in this case. The number of distinct keys increases rapidly to 2^{22} (the maximum) as p increases (e.g. for $p \geq 0.2$ all keys are unique). This explains the times peaking more rapidly in Figure 4.18(b) compared to Figure 4.18(a) for the comparison based data structures. For $p = 0.25$ there are already 2^{22} distinct keys in the data structures. Notably, the trie structures' performance improves as p increases, while the total number of distinct keys remains constant. This is likely due to the lower average depth of the trie structures when data is closer to uniformly distributed. We discuss the average depth of trie structures in more detail in Section 4.5.

The results in this section have shown that although the $LPCB$-trie is sensitive to the uniformity of its data, it nevertheless out-performs the alternative ordered data structures (except the S-tree on 32-bit keys) except for the most biased data ($p < 0.15$).

Valgrind Data

We now examine the performance of the data structures over data sets generated using Valgrind [Nethercote and Seward 2007]. Valgrind comprises a suite of very widely used tools for dynamic binary instrumentation. The tools include **`memcheck`**, used for detecting memory errors (e.g. leaked memory, wild pointers), **`helgrind`** for race detection, and **`cachegrind`** for cache performance profiling, as well as a number of other tools.

The data sets generated using Valgrind consist of the memory addresses of all the data memory accesses performed during the execution of a program. This reflects the use of the data structure to track every memory access performed by a program, as is done by some Valgrind-based tools. We generated data sets for the Linux program Top (a task viewer) as well as three applications from the K Desktop Environment: Amarok (a music player), Konqueror (a web browser and file manager) and KPDF (a PDF viewer). Each of these data sets contain between 10^7 and 10^8 operations in total, and 70-80% of the operations are loads. As one would expect for the memory accesses of a program, these data sets are highly repetitive. The 32-bit data sets contain between 20,000 and 500,000 distinct memory locations, while the 64-bit data sets contain between 40,000 and 900,000 distinct memory locations. A load in a data set generates a search operation on the appropriate data structure while a store generates an insert operation. Currently, Valgrind uses an AVL tree to perform these operations

[Seward 2007][‖]. Note that in-order iteration over the data structure is also required at certain times, removing the possibility of using a hash-table. However, we include a hash-table in our experimental results to show the margin between it and the fastest ordered data structure.

Figure 4.19(a) shows the time for processing 32-bit Valgrind data sets. Due to its simplicity, the hash-table is again the most efficient data structure. Amongst the ordered data structures, the S-tree is clearly the most efficient data structure in time, followed by the $LPCB$-trie. We note also that Figure 4.19(b) shows the $LPCB$-trie requires the least space of any of the data structures, and that the S-tree is especially inefficient in space.

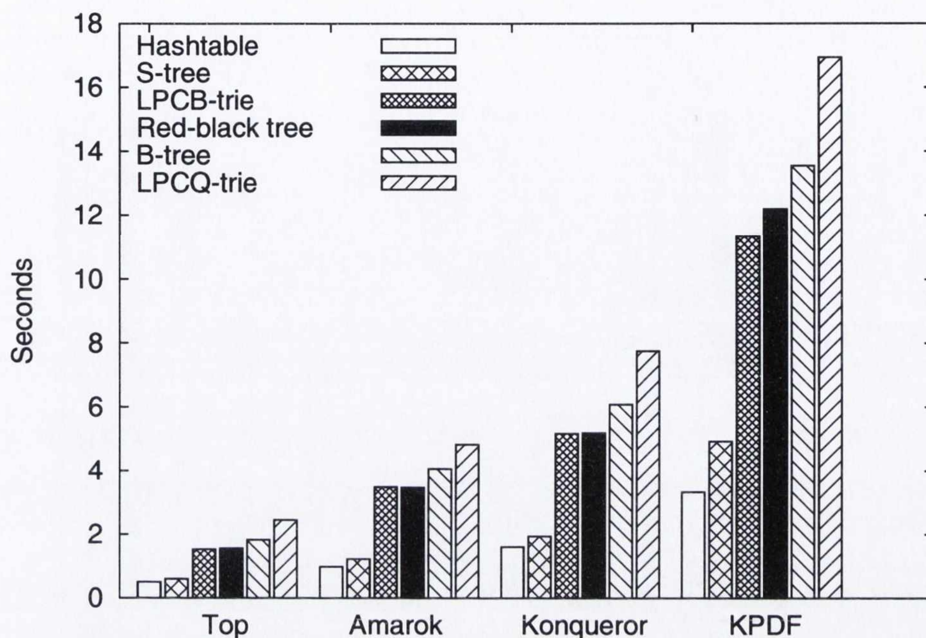
On the uniform random data in Section 4.4.2, the $LPCQ$-trie required less space than the $LPCB$-trie, whereas on this data this trend is reversed. Recall that the bucket data structure is a growable array, doubling in size when it is full. When the number of insertions to a bucket is uniform random, it is easy to show the expected fullness of the bucket is 75%. For the uniform random data we have observed that both the $LPCB$-trie and $LPCQ$-trie generally have approximately this fullness. For this Valgrind data however, we have noted the $LPCB$-trie has approximately 70% bucket fullness, while the $LPCQ$-trie has only 55% fullness – close to its worst case, since the $LPCQ$-trie splits buckets guaranteeing they are always at least 50% full. This difference in bucket fullness explains the space difference between the $LPCB$-trie and $LPCQ$-trie, it is interesting that the $LPCB$-trie's space usage appears less dependent on the randomness of the data than the $LPCQ$-trie.

It is notable that the B-tree performs worse than the red-black tree on these data sets. Moreover, the $LPCB$-trie performs only slightly better than the red-black tree, and the $LPCQ$-trie performs worst on this data – we analyse the behaviour of these structures in more detail for the 64-bit Valgrind data.

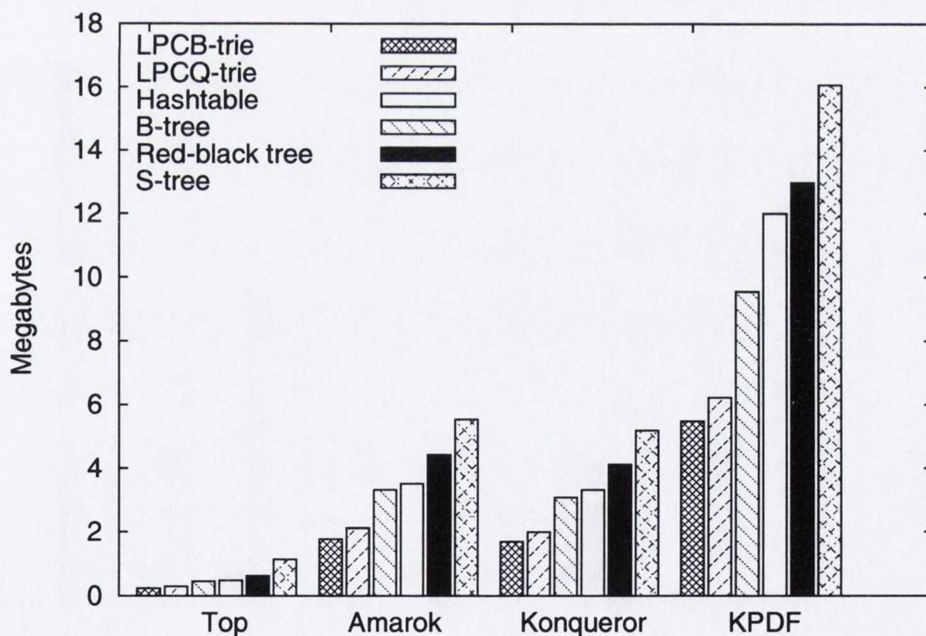
Figure 4.20(a) shows the time required by the data structures for the 64-bit Valgrind data sets, the S-tree is not shown here because it cannot operate on 64-bit keys. The $LPCB$-trie performs best in time of any of the ordered data structures. Figure 4.20(b) shows that the $LPCB$-trie also requires the least space of any of the data structures.

For these 64-bit Valgrind data sets, we have performed some additional experiments to examine their relative performance in more detail. Figure 4.21(a) shows the numbers of instructions executed by the data structures. As is expected, the hash-table has a very low instruction count compared to the ordered data structures. It is noteworthy

[‖]Although our information comes from personal communications with Julian Seward, Valgrind's lead developer, they can also be verified by examining Valgrind's source code. Specifically the `WordFM` data structure, see `http://valgrind.org/downloads/repository.html`.

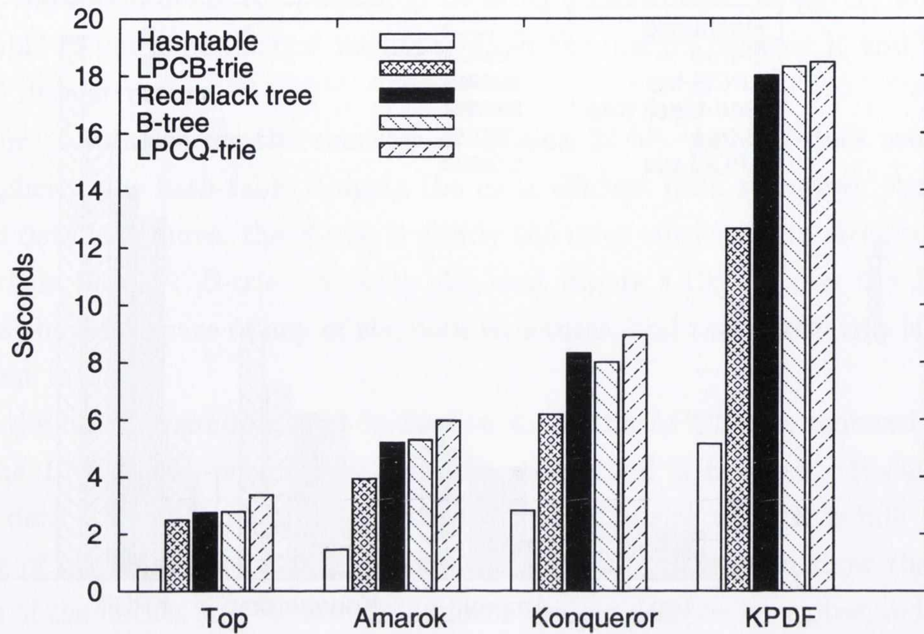


(a)

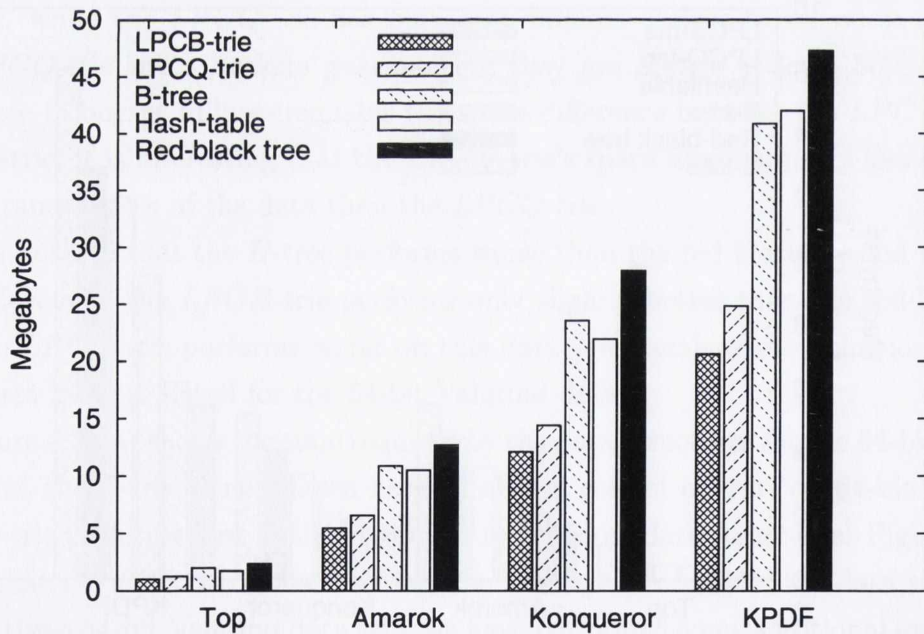


(b)

Figure 4.19: This figure shows the time, in (a) and space in (b) required by the data structures to process the 32-bit Valgrind data sets. These results were gathered on `knuth` and are discussed in Section 4.4.2.

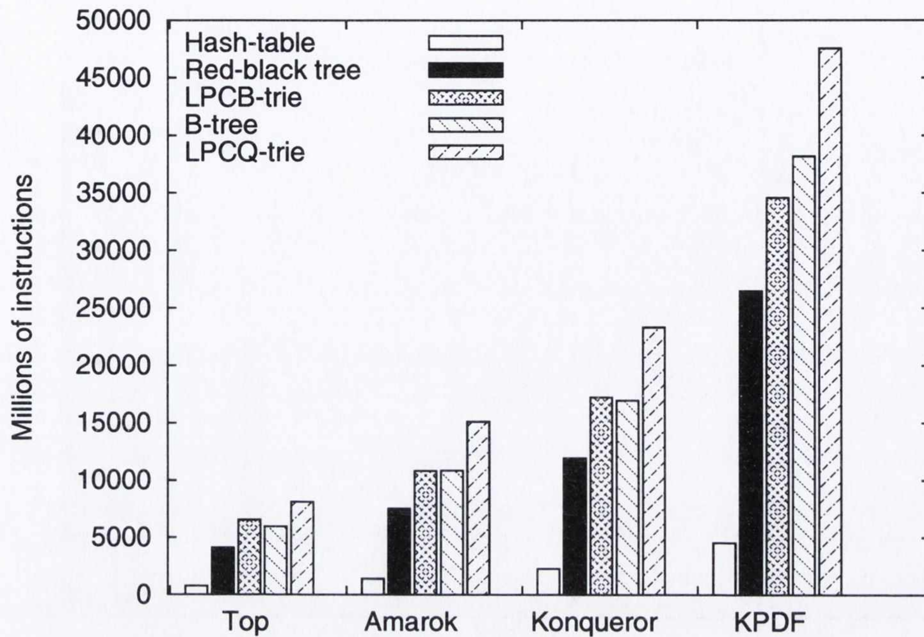


(a)

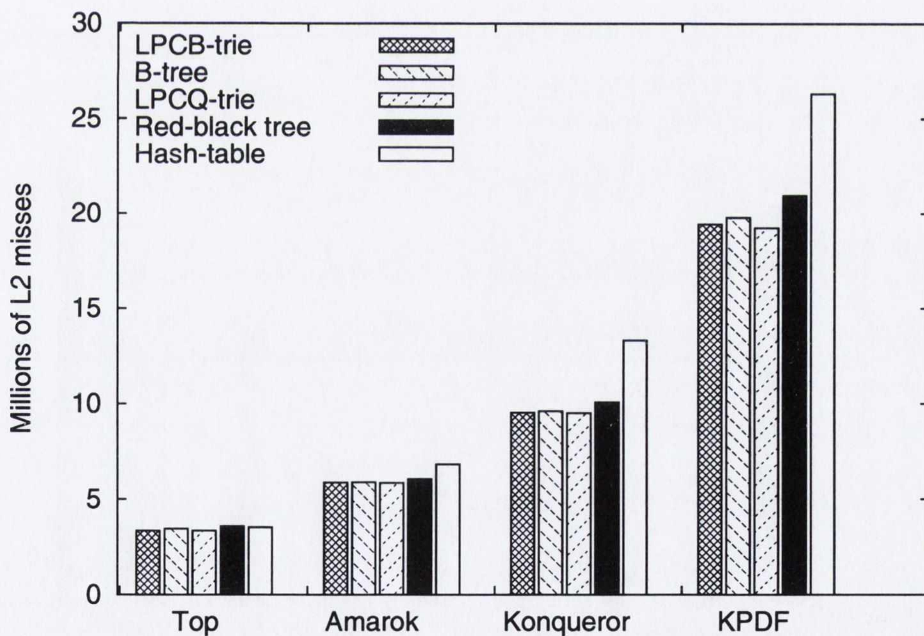


(b)

Figure 4.20: This figure shows the time, in (a) and space in (b) required by the data structures to process the 64-bit Valgrind data sets. These results were gathered on `beaker` and are discussed in Section 4.4.2.

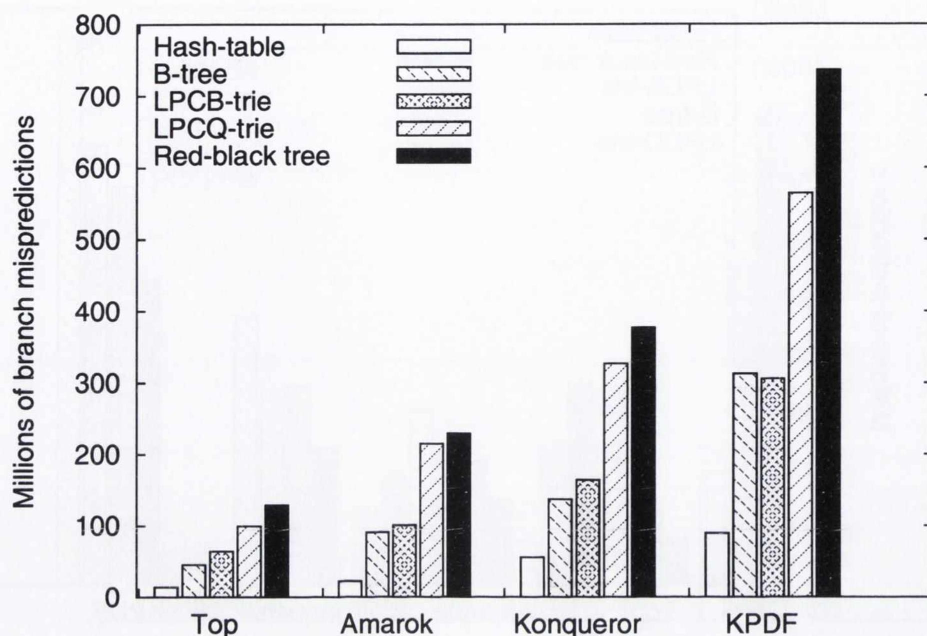


(a)

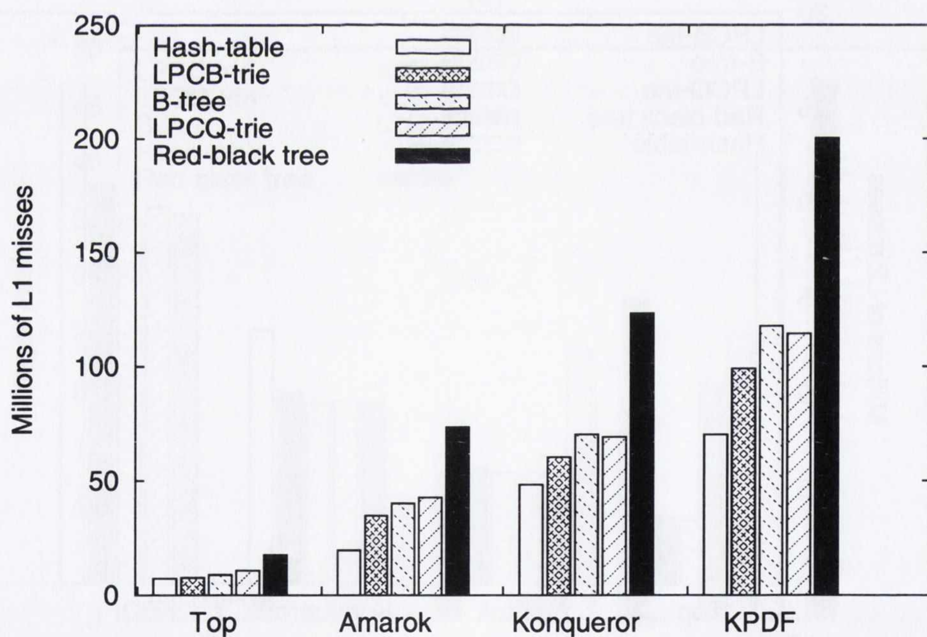


(b)

Figure 4.21: This figure shows the shows the total number of instructions, in (a) and the total number of level 2 cache misses in (b) when the data structures process the 64-bit Valgrind data sets. These results were gathered on `beaker` using PAPI [Dongarra et al. 2003], and are described in Section 4.4.2.



(a)



(b)

Figure 4.22: This figure shows the total number of branch mispredictions in (a) and the total number of level 1 cache misses, in (b) when the data structures process the 64-bit Valgrind data sets. These results were gathered on `beaker` using PAPI [Dongarra et al. 2003], and are described in Section 4.4.2.

that while the $LPCB$-trie executes more instructions than the red-black tree, it was seen to perform better in time in Figure 4.20(a). The observed performance cannot be explained by level 2 cache misses, since as Figure 4.21(b) shows, the number of level 2 cache misses is insignificantly small compared to the number of instructions executed – even allowing for the high cost of a level 2 cache miss – and moreover, the red-black tree and $LPCB$-trie (as well as the $LPCQ$-trie and B-tree) have very similar level 2 cache performance.

A possible explanation of the fact that the low instruction count of the red-black tree compared to the other ordered data structures does not translate into better performance is the red-black tree's high level of branch mispredictions compared to the $LPCB$-trie shown in Figure 4.22(a). For example, on the KPDF Valgrind data the red-black tree executes approximately 2.6×10^{10} instructions compared to 3.4×10^{10} instructions of the $LPCB$-trie. For the same data, the red-black tree causes approximately 7.3×10^8 branch mispredictions compared to the 3.0×10^8 branch mispredictions of the $LPCB$-trie. Recall from Section 4.4 that a branch misprediction costs approximately 14 cycles, making the difference between the number of branch mispredictions significant in magnitude compared to the difference in instruction counts for the red-black tree and $LPCB$-trie observed in Figure 4.21(a). As a result, branch mispredictions are a strong contributing factor to the red-black tree operating more slowly than the $LPCB$-trie, despite its lower instruction count for this data. Another contributing factor, shown in Figure 4.22(a) is that the red-black tree incurs approximately 10 times as many level 1 cache misses as level 2 cache misses, with the result that the level 1 cache misses are likely to have a bigger influence on the red-black tree's performance than its level 2 misses. We note that the red-black tree causes approximately double the number of level 1 cache misses of the $LPCB$-trie, on average.

We note that the poor relative performance of the $LPCQ$-trie on this data compared to its good relative performance on uniform random data in Section 4.4.2 can be explained by its high instruction count, seen in Figure 4.21(a), and its high level of branch mispredictions seen in Figure 4.22(a). For data where cache performance heavily dominates running time (such as the Genome data set we shall see below in Section 4.4.2) the $LPCQ$-trie performs well. As we have just seen, level 2 cache misses are not dominant for this data. Recall from Section 4.1 that in a search operation the $LPCQ$-trie must perform a predecessor search in its trie structure to find a bucket. This is a significantly more complicated operation than simply traversing the path defined in the trie by a key, as a burst trie does. This predecessor search results in the high instruction count and branch mispredictions for the $LPCQ$-trie observed on this

data, and in turn, its poor relative performance.

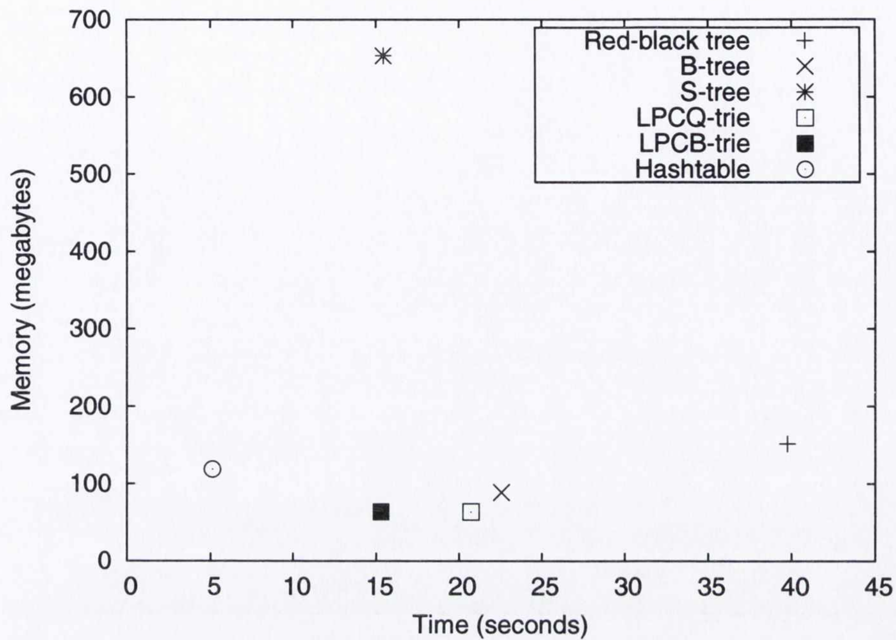
Amongst the ordered data structures, on the 32-bit Valgrind data sets, the S-tree performs better than the $LPCB$-trie, however, it requires more than twice as much memory. On the 64-bit Valgrind data sets, the $LPCB$-trie is the best performing data structure. For both the 32-bit and 64-bit Valgrind data sets the $LPCB$-trie requires the least space of any data structure.

Genome Data

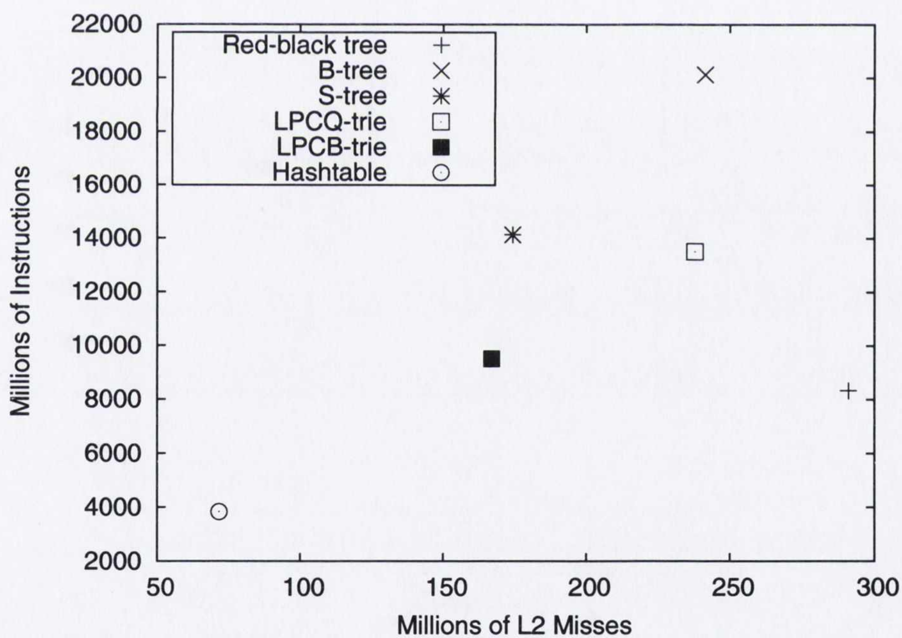
The final data set used in our experimental comparison is the Genome data set. This data comes from GenBank, and originally used by Sinha and Zobel [2004] in the evaluation of the burstsort algorithm from whom we obtained it. This data set consists of approximately 3.1×10^7 strings of 9 characters in length, with each character either 'a', 'c', 'g' or 't'. Thus each string can be encoded in 18 bits. We increased the lengths of the bit-strings to 36-bits by concatenating consecutive pairs in the data set and using them as a single 36-bit key. The resultant data-set contains approximately 1.5×10^7 keys, and approximately 5×10^6 distinct keys. Where we experiment with the Genome data set for 32-bit keys, we use the 32 least significant bits of the keys.

Figure 4.23(a) shows the time and space taken by the data structures for the 32-bit genome data set. The space usage of the data structures is similar to what was observed on the uniform random data, the $LPCQ$-trie is the most compact data structure, followed by the $LPCB$-trie, B-tree and hash-table. It is notable that the space usage of the S-tree on this data is particularly high, requiring more than a factor of 10 times the space of the $LPCB$-trie. We note that as above the hash-table is much more efficient than the ordered data structures. The most efficient ordered data structures are the $LPCB$-trie and S-tree. The hash-table is approximately a factor of 3 times more efficient in time than these data structures. Figure 4.23(b) shows the number of instructions executed and level 2 cache misses caused by the data structures. Of the ordered data structures, the $LPCB$-trie has the best cache performance on this data, coupled with the lowest instruction count. Noting again that the cost of a level two cache miss is more than 100 times that of the average instruction, it is likely they are the dominant factor in the execution times of the data structures. Note that the ordering of the data structures is the same along the horizontal axis of Figure 4.23(a) and (b).

Figure 4.24(a) shows the time and space taken by the data structures for the 64-bit genome data set. Here the S-tree is not shown because it is restricted to 32-bit keys. On this data, amongst the ordered data structures, the $LPCB$-trie is also

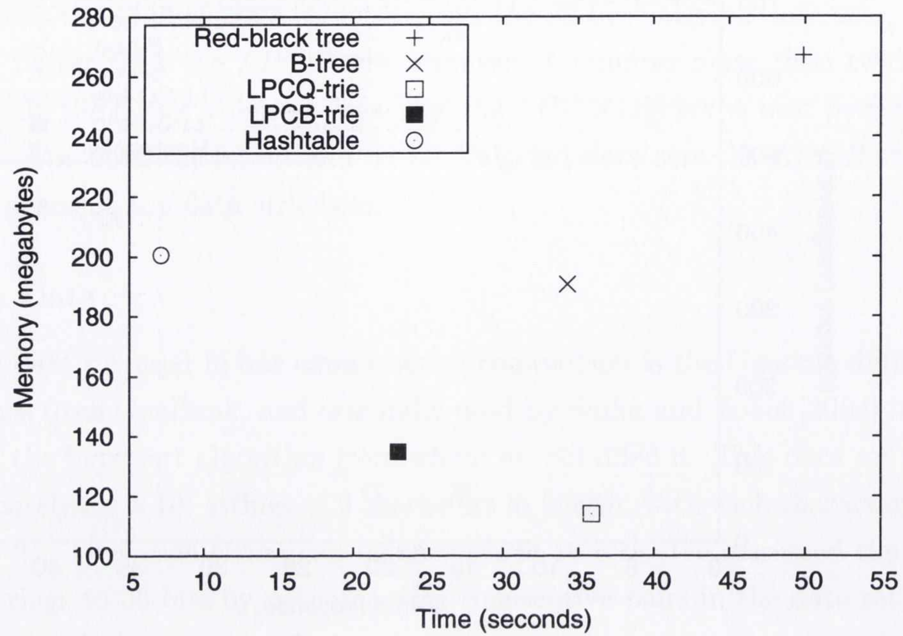


(a)

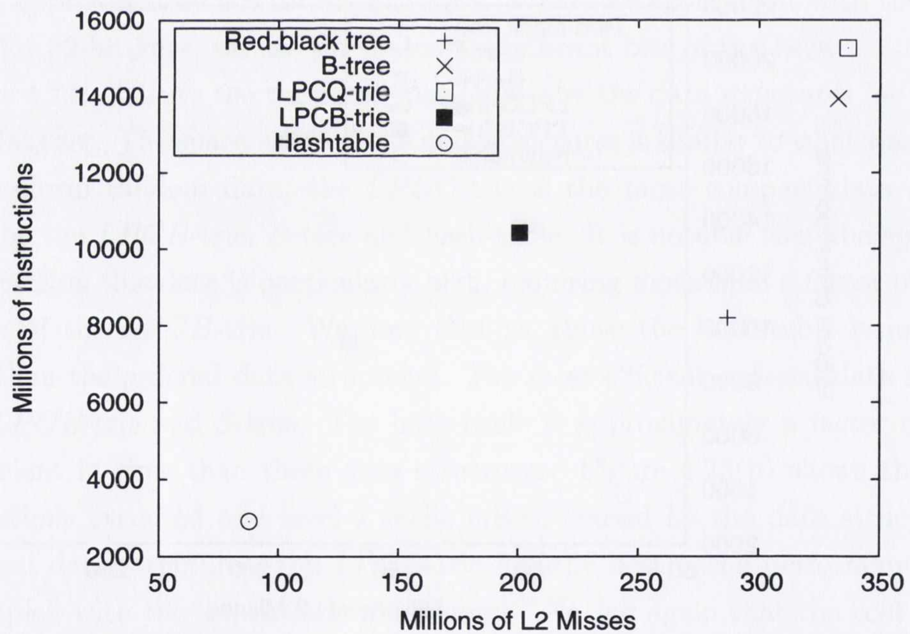


(b)

Figure 4.23: (a) Shows the time and space required by the data structures to process the 32-bit Genome data set. (b) Shows the total instructions executed and level 2 cache misses incurred for the Genome data set, these instruction counts and level 2 cache misses were gathered with Valgrind [Nethercote and Seward 2007] (technical reasons prevented us from using PAPI [Dongarra et al. 2003] in this case), on `knuth` and are described in Section 4.4.2.



(a)



(b)

Figure 4.24: (a) Shows the time and space required by the data structures to process the 64-bit Genome data set. (b) Shows the total instructions executed and level 2 cache misses incurred for the Genome data set, these instruction counts and level 2 cache misses were gathered with Valgrind [Nethercote and Seward 2007] (technical reasons prevented use from using PAPI [Dongarra et al. 2003] in this case), on `beaker` and are described in Section 4.4.2.

the best performing data structure. Figure 4.24(b) shows the level 2 cache misses and instruction counts of the data structures. Aside from the red-black tree, the performance of the data structures in time is ordered by their level 2 cache misses (as can be seen by examining the ordering of these data structures along the x-axis of Figure 4.24(a) and (b)). However, the red-black tree executes fewer instructions than the B-tree and $LPCQ$-trie (e.g. approximately 40% fewer than the B-tree), and causes fewer level 2 cache misses (approximately 14% fewer than the B-tree), nevertheless, the red-black tree requires approximately 45% more time than the B-tree, as Figure 4.24(a) shows.

This poor performance of the red-black tree can be explained by examining the branch mispredictions and TLB misses it incurs. As was observed for the Valgrind data set in Section 4.4.2, the red-black tree again incurs a large number of branch mispredictions compared to the B-tree. On this data it causes approximately 4.8×10^8 branch mispredictions, compared to 1.1×10^8 branch mispredictions of the B-tree. In addition, on this data, the red-black tree incurs approximately 5.5×10^8 TLB misses, compared to the 1.2×10^8 of the B-tree.

In summary, amongst the ordered data structures, the $LPCB$-trie is the most efficient in both time and space for these data sets. On the 32-bit genome data, the S-tree is almost as efficient in time, but requries more than 10 times as much space as the $LPCB$-trie. For both the 32-bit and 64-bit genome data sets, the hash-table is more efficient in time (although not in space) than the ordered data structures, as is to be expected, and once again showing the overhead of maintaining order in data structures.

4.5 Related Work

Being a classic problem, maintaining a dynamic ordered data structure has a large and rich literature, including many data structures that are theoretical as well as practical. The most commonly implemented solution to this problem is a balanced search tree – often binary. For example, the GNU *C++* [Stroustrup 1997] STL implementation** of its `map` type is a red-black tree [Guibas and Sedgewick 1978; Cormen et al. 2001]. The collection of algorithms and data structures LEDA [Mehlhorn and Näher 1998] offers very similar functionality in its `sortseq` data structure, which can be realized using a BB[α] tree [Blum and Mehlhorn 1980], B-tree [Bayer and McCreight 1972], red-black tree, randomized search tree [Seidel and Aragon 1996] or skip-list [Pugh 1990].

**See `http://gcc.gnu.org`

These data structures have two very general features in common: They implement their operations only by comparing the input keys, and they offer their operations in $O(\log n)$ time (variously amortized, expected or worst-case), when storing n keys.

This $O(\log n)$ time bound for searching in a data structure is the best possible when only comparisons are made between keys. This is easy to see via the usual decision tree argument: A search proceeds based on the out-come of a series of comparison operations organized into a decision tree. The decision tree has $n+1$ leaves (one extra for when the key is not found), giving a depth of at least $\lceil \lg(n+1) \rceil$.

In many applications, the keys in a data structure are strings of bits, such as integers or floating point numbers. Moreover, real computers have a much more powerful set of operations than just comparisons, supporting operations such as indirect addressing, bitwise operations, multiplication and division, all in constant time. A model of such computers is the unit-cost word RAM, which supports all C-like operations in unit time, or some subset of them. This model of a computer also includes the restriction that memory is divided into fixed sized cells, called words. The number of bits, w, in a word is called the word-size of the machine. In data structure problems over n keys, it is assumed in this model that $w > \lg n$, so that all the keys of the data structure can be represented in single words.

In this context, different asymptotic results can be obtained by data structures that do not rely solely on key comparisons. These results can be in terms of w, n or in terms of w and n. We begin our discussion of these results by mainly dealing with theoretical results first, and describing practical instantiations of them, and practical data structures afterwards. Hagerup [1998] provides an excellent survey of some of the results mentioned in this section. A classic early theoretical data structure is the stratified tree data structure of van Emde Boas [1977] which supports all operations in $O(\log w)$ worst-case time, when operating on w-bit keys. This data structure introduced a subsequently very influential technique for data structures called *exponential range reduction*, which we briefly describe. Given an ordered data structure, D, that operates on b bit keys, this technique extends it to a new data structure, also ordered, that operates on $2b$ bit keys whose operations take constant time plus the query time of D. The new data structure is composed of two parts a *top* and *substructures*, the latter being instances of D. The top consists of an instance of D called T together with an array S of 2^b pointers to substructures. The top also contains a second array M of 2^b entries holding pairs (min, max) which are the minimum and maximum element in each substructure. We now describe how a predecessor query (i.e. the largest key less than a given key) can be performed given a $2b$ bit key, k. The upper b bits of k are

used to form an index i. If $S[i] = \text{NIL}$ or $k < min$ where $M[i] = (min, max)$, then a predecessor query for i is performed on T. If there is no predecessor, NIL is returned. If there is a predecessor, at index j, the value of max retrieved from $M[j]$ is returned as the b low-order bits of the answer – the b high-order bits are i.In the other case, where $S[i] \neq \text{NIL}$ and $k \geq min$ (as before retrieved from $M[i]$), then a predecessor query for the b lowest bits of k is performed on the substructure pointed to by $S[i]$. The result of this query is the b lowest bits of the result to the original predecessor query, the higher bits are again simply i. Note that in both cases constant time is spent (indexing the arrays S or M) followed by a single query in an instance of D. Recursively building the instances of D in the same manner repeatedly halves the length in bits of the keys that needs to be considered in constant time, giving rise to the $O(\log w)$ worst case time of van Emde Boas's stratified trees.

Another important data structure is the fusion tree of Fredman and Willard [1990] which supports all operations in $O(\log_w n)$ amortized time, later improved by Willard [1992; 2000] to worst-case time, and simplified by Andersson [1995]. Combined with stratified trees, their data structure takes expected $O(\min\{\log_w n, \log w\})$ time, giving a time that is always $O(\sqrt{\log n})$. A more recent example was described by Andersson and Thorup [2007]. They gave a dynamic data structure supporting all operations in $O(\sqrt{\log n / \log\log n})$ worst-case time. This data structure is optimal in terms of n, since Beame and Fich [2002] showed a matching lower bound for the static case, that is, when the data structure is built once and for all and then only queried.

Such data structures are attractive in theory, and potentially in practice too because of their superior worst-case times compared to comparison-based data structures. An important consideration in both theory and practice however is the amount of space required by these data structures. For example, as described by van Emde Boas [1977], stratified trees require $\Omega(2^w)$ space. Willard's y-fast tries [Willard 1983] showed how to reduce the space to $O(n)$ in the static case. Mehlhorn and Näher [1990] showed how the space could be reduced to $O(n)$ in the dynamic case, via the use of dynamic perfect hashing [Dietzfelbinger et al. 1988]. Data structures using more than $O(n)$ space are unlikely to be acceptable in many practical situations, for example, in a library where the data structure is provided without a specific application in mind. With linear space Pătraşcu and Thorup [2006] have shown a lower bound of $\Omega(\log w)$, and so stratified trees are optimal in terms of the word size with linear space. With more space, better results can be obtained however. For example, in $O(n^2 \log n / \log\log n)$ space, Beame and Fich [2002] described a data structure in the static case supporting predecessor queries in $O(\min(\log w / \log\log w, \sqrt{\log n / \log\log n}))$ time.

Aside from the space required by the data structures, other considerations include whether the data structure's time bounds are amortized or worst-case and additionally whether they are deterministic or randomized. For example, the $O(\log w)$ bound achieved by stratified trees in $O(n)$ space is achieved via hashing, and so is not deterministic. Yet another consideration is the exact set of instructions made use of in constant time by the data structure. Often just integer multiplication and division are excluded [Hagerup 1998], or instructions are restricted to those in the class called AC^0 [Vollmer 1999], that is, instructions that can be realized by a constant depth circuit consisting of unbounded fan-in AND and OR gates (NOT gates are not counted), where the total number of gates is polynomial in the word-size w. This includes familiar C like bitwise instructions as well as addition and subtraction, but notably excludes multiplication and division. It also permits other "non-standard" AC^0 instructions (i.e. those that are not currently realized by any real-world computer, or that are realized, but are not directly available in a standard manner through languages such as C), which are variously used or unused in designing data structures [Thorup 2003]. In practice, a realistic assumption is probably to assume all standard C instructions are available, with a caveat that multiplication is usually more expensive than bit-wise instructions, addition or subtraction, and so data structures firmly based upon it (e.g. the optimal data structure of Andersson and Thorup [2007] mentioned above) may be more difficult to construct a practical realization of.

Many of the word-based data structures described above have never been implemented (a notable exception is the work of Andersson [1995], which includes C source code). In translating theoretical data structures to practice, the principle ideas can be preserved while making simplifications that may worsen the time bounds, but still offer improved performance over comparison based structures. A good example is the Q-trie described by Korda and Raman [1999]. As mentioned in Section 4.1, this data structure is based on the q-fast trie of Willard [1984]. The latter data structure offers all operations in $O(\sqrt{w})$ worst-case time, and requires linear space. Korda and Raman's structure can be regarded as a q-fast trie where the trie's in-node search trees are disgarded, and the bucket binary search trees are replaced with sorted arrays. The resulting data structure no longer has $O(\sqrt{w})$ time for its operations, but is nonetheless more efficient in practice than both traditional tries and binary search trees on the random data they examine. A second example of beginning with a theoretical structure is that of Dementiev *et al.* [2004], described in Section 4.1. This data structure makes use of exponential range reduction recursively, described above, inspired by the structure of van Emde Boas [1977].

Many data structures for integer keys can be constructed using tries, including some of the data structures mentioned above – most obviously Willard's y-fast tries [Willard 1983] and his p-fast and q-fast tries [Willard 1984]. Tries are often used in string processing problems, a context more general than the subject of the present chapter, since the keys are variable length. A particularly popular string based trie structure is the *suffix trie* [Weiner 1973]. The expected height of trie structures and related quentities has been the subject of intensive study and precise analysis (that is, constant factors and lower order terms that would normally be hidden by a "big-Oh" analysis have been analysed), Flajolet [2006] provides a survey. A basic result is that when n independent random strings are inserted into a trie, the average depth is $O(\log n)$ [Devroye 1982]. Andersson and Nilsson [1993] described a (static) trie data structure called a *level compressed trie*, or LC-trie that achieved a much smaller average depth of just $O(\log^* n)$ on uniformly distributed data, or, on Bernoulli distributed data (where the probability of a zero is not equal to the probability of a one), average depth $O(\log \log n)$. These small depths for random data show the importance of not using it as the sole basis for evaluating the performance of trie structures.

When all nodes in a trie having b or fewer descendant leaves are replaced with an alternative data structure containing the keys associated with the leaves, a bucketed trie, or b-trie is obtained. The first detailed experimental analysis of a dynamic bucketed trie structure where the bucket data structure was varied was given by Heinz *et al.* [2002] in the context of string keys and string processing. They refer to the structure as a *burst trie*, and we have chosen to adopt their terminology in this chapter. This work, together with the practical variations of level compression described by Nilsson and Tikkanen [2002] gave rise to the $LPCB$-trie described in this chapter, which is a level and path compressed burst trie for integer keys. Apart from the two main data structures, the S-tree and Q-trie compared to the $LPCB$-trie structure in this chapter, there are a number of other data structures that have been described for integer keys that have also been the subject of experimental analyses. We note a number of these practical data structures here, the general point here is that a comparison between them and our $LPCB$-trie structure cannot be direct since they do not fully fulfill the requirements of an ordered data structure (i.e. supporting predecessor and successor queries). For example, the cache efficient tries of Acharyra *et al.* [1999] were the subject of implementation and experimental examination, but it does not appear that they can be extended to an ordered data structure simply, since they make use of hash tables inside the trie nodes. Despite being based on a hash-table data structure (and hence, unordered), the work of Askitis [2009] is notable in the present context since it

is an experimental examination of a hash table specialized to integer keys.

Another trie based data structure that has been the subject of an experimental analysis is the *HAT*-trie of Askitis and Sinha [2007]. Although designed for string keys, it is potentially relevant due to the fact that it may be possible to specialize the structure to integer keys. Although the authors describe their data structure as ordered, it makes use of hashing in its buckets, and so certain alternating sequences of insertions and successor operations are likely to be inefficient as is in-order iteration over the keys. A final example is the work of Nilsson and Tikkanen [2002], their data structure is a dynamic level compressed trie, and has been described in this chapter.

We now briefly summarize the context of the work in this chapter, and in particular the *LPCB*-trie that we argue is likely to be useful in practice. We note that there is a rich literature on theoretical word-based data structures, that is, those that take advantage of the fact that the representation of each key is a (usually fixed length) bit-string. Although scarcely implemented, the best two practical instantiations we are aware of are the *S*-tree and *Q*-trie, included in the experiments of this chapter. Aside from theoretical data structures, there are also many practical word-based data structures, in particular those based on trie structures. The *LPCB*-trie of this chapter was principally inspired by the combination of a practical variation of *LC*-tries and burst tries, specialized to the case of integer keys, and generalized to the case of an ordered data structure.

4.6 Future Work

There are several avenues open to future work related to the work described in this chapter. Although the *LPCB*-trie implementation measured in this chapter is fully general, supporting insert, search, predecessor, successor operations and in-order iteration over keys, it would be interesting to construct an implementation the conformed to a standard interface. For example, that of the *C++* STL's `map` data structure [Stroustrup 1997]. This is a significant challenge, mainly due to *iterator guarantees.* On a *C++* `map`, an insertion to the data structure is guaranteed not to invalidate any iterators currently pointing to other keys in the data structure. When keys are stored in growable arrays, how can this be maintained efficiently? Frias *et al.* [2009] describe related issues that arise when implementing a bucketed *C++* `list` implementation.

A second avenue of exploration could attempt to further improve the performance in time and space of the *LPCB*-trie. Given a fixed burst trie, it need only store key suffixes in its buckets, improving space usage as well as spatial locality (note that the

same cannot be applied to Q-tries). However, when the branching factors of trie nodes can vary (i.e. via level compression), the length of the suffixes that need to be stored in the buckets also varies. For fixed organization burst tries, we provided some results of limited scope [Nash and Gregg 2008], however it would be interesting to see if there is a solution that can be engineered for the case of a level compressed burst trie. A caveat is that, in anecdotal experience, the storage of key suffixes only in buckets results in a highly complex implementation in order for generality over key lengths and efficiency to be maintained.

Lastly, of the many theoretical data structures described in Section 4.5, few have ever been implemented. While direct implementations of many of the data structures are likely to be infeasible in practice, the literature on integer data structures includes many techniques for achieving parallelism at the bit-level and some of these techniques could likely be used in practice. A simple starting point is to take the principle ideas of a theoretical data structure but replace its components with those that are efficient in practice, even if not as asymptotically efficient as those used in the theoretical constructions. Andersson [1995] provides an excellent example of this, giving C source code for a modified version of his $O(\sqrt{\log n})$ time search tree (the time complexity of operations in his implementation is increased to $O(\log n / \log\log n)$). He however only provides anecdotal experimental experience, and does not concentrate on engineering effort, and moreover his implementation is likely to be impractical because of its space usage (in fact, his principle motivation for providing an implementation is not to demonstrate good performance, but to concretely establish a sublogarithmic time data structure in a real programming language without the use of multiplication). His implementation does demonstrate that theoretically appealing data structures may have realizations that are not impractically complex. An important tool in this context could be the multi-media instructions now available on almost all modern desktop processors, some of these instructions operate on very wide operands of up to 256 bits, and could be used to increase word-level parallelism. A project of particular interest here would be to create a generic, portable implementation of a data structure such as the one of Andersson [1995], that can have its "back-end" replaced. This back-end could then be specialized to use the multi-media instructions available on different architectures, such as SSE and XOP [AMD 2009; Intel 2009] in order to extract the maximum word-level parrallelism from the architecture on which the data structure is operating. Recent work has explored the use of 64-bit architectures to provide additional parralelism (compared to 32-bit architectures) although targeted towards succinct data structures, and without making use of multi-media instructions [Vigna 2008; Gog 2009].

4.7 Conclusion

This chapter has provided a thorough experimental comparison of efficient data structures operating over 32 and 64-bit integer keys. In particular we have shown that extending burst tries to an ordered data structure for integer keys provides a data structure that is very efficient in both time and space.

In comparisons using uniform and biased random data with red-black trees and B-trees we have shown that our level and path compressed variant of burst tries, $LPCB$-tries, provide all operations more efficiently in both time and space. We have also compared our $LPCB$-trie to Dementiev *et al.*'s [2004] S-tree data structure based on stratified trees, and found that while Dementiev *et al.*'s data structure is competitive in time, it requires far more memory than an $LPCB$-trie and is less general, being restricted to 32-bit keys. We have also compared $LPCB$-trie to Q-tries, a data structure based on Korda and Raman's [1999] modification of Willard's q-fast tries [1984]. We carefully engineered an implementation of $LPCQ$-tries, using the same bucket and node data structures as our burst trie as well as incorporating level compression, and found that they are generally slightly less efficient in time than $LPCB$-tries. The $LPCQ$-tries requires slightly less space however. On the uniform random data, we found the $LPCB$-trie to require 12 and 17 bytes per insertion, whereas the $LPCQ$-trie required approximately 12 bytes per insertion.

We have also presented results for an application of dynamic, ordered integer data structures in Valgrind where the keys are 32 and 64-bit integers. Our results show that in the 32-bit case only the S-tree data structure operates faster than the $LPCB$-trie, but the S-tree requires almost twice as much space as the $LPCB$-trie. In the 64-bit case, the $LPCB$-trie requires less space and operates more rapidly than any of the alternative data structures. We also provided experimental results over a data set derived from a Genome data set, where the keys are 36-bit integers. In this case, the $LPCB$-trie also operated substantially faster and required less space than the alternative comparison-based structures.

Our experimental results have shown that bucketed tries often achieve good performance compared to alternative data structures by incurring low numbers of cache misses. However, our burst trie variant has not been explicitly designed to be cache aware. If cache awareness was introducted, it would be necessary to compare the performance of the trie structures with other cache aware and cache oblivious search trees, a structure of particular relevance is the cache aware B+-tree of Rao and Ross [2000].

This chapter demonstrates, through a detailed experimental comparison on both

naturally occuring and synthetic data sets, that $LPCB$-tries should be considered as one of the many alternative data structures for applications requiring a general purpose dynamic ordered data structure over keys such as integers or floating point numbers.

Chapter 5

Maximum Independent Sets in Circle Graphs

5.1 Introduction

Chapters 3 and 4 of this dissertation have presented experimental and algorithm engineering contributions for two closely related problems: sorting and searching. In this chapter, we explore the engineering of a problem arising in a somewhat different area: graph algorithms. We again take an experimental and algorithm engineering approach, culminating in the proposal of new, output sensitive algorithms for the problem considered, whose asymptotic efficiency we also justify experimentally.

Finding a maximum independent set in a weighted circle graph is an important problem that arises in a number of areas. Our interest in the problem arose from an application in the field of compiler optimization, where one wishes to rapidly compute a maximum independent set of a circle graph when allocating registers in software pipelined loops [Eisenbeis et al. 1995; de Werra et al. 1999, 2002]. The problem also arises in other fields, including VLSI design [Supowit 1987; Cong and Liu 1990] computational geometry [Asano et al. 1986; Liu and Ntafos 1988], and computational biology [Zou et al. 2009].

We begin this chapter by providing a detailed experimental evaluation of the two most efficient algorithms for computing a maximum independent set of a *weighted* circle graph. We provide details of our implementations of the algorithms of Apostolico *et al.* [1992; 1993] and Valiente [2003], together with time, space and other measurements. These algorithms operate in a very different manner, but have the same asymptotic time. In implementing the algorithm algorithm of Apostolico *et al.* we describe optimizations that significantly reduce both its running time and its memory consumption.

We also correct an error in this algorithm. In addition, we show how to restructure and simplify Valiente's algorithm, allowing us to remove redundant computations from the algorithm resulting in much improved performance. Moreover, in the case of both algorithms, when the density of the circle graph's associated interval representation is increased beyond a certain point the efficacy of the optimizations we apply increases dramatically and as a function of the density. We provide experimental results over dense and sparse random circle graphs, as well as for circle graphs that arise when performing register allocation of software pipelined loops in a compiler.

Following this, we compare the best variation of these algorithms to new *output sensitive* algorithms that we describe. Compared to our very efficient variations of previous approaches, we demonstrate experimentally that our output sensitive algorithms are even more efficient in practice.

5.2 Background

The *intersection graph* of a finite family of sets $S_1, \ldots, S_n$, is an undirected graph of n vertices $v_1, \ldots, v_n$ with an edge connecting v_i and v_j only if $S_i \cap S_j \neq \phi$.

A *circle graph* is an undirected graph isomorphic to the intersection graph of a finite set of chords in a circle. A circle graph can be represented either by chords in a circle or intervals on the real line, and the two representations may easily be obtained from one another [Gavril 1973]. Figure 5.1 shows an example of these representations together with the circle graph they give rise to. We can assume without loss of generality that no two chords (or intervals) share an end-point, since if two chords share an end-point we can slightly move the end-point of one of the chords without changing the circle graph [Gavril 1973].

An interval representation of an n vertex circle graph can be encoded as a permutation σ of $\{1, ..., 2n\}$ where the end-points of the intervals are formed by pairs $(\sigma_{2k-1}, \sigma_{2k})$, for $1 \leq k \leq n$. Note that this makes use of the assumption that no intervals share an end-point. An arbitrary set of intervals can be transformed into this form in $\Theta(n \log n)$ time. Further background on circle graphs and related graph families is provided by Golumbic [2004].

In this chapter we make use solely of this interval representation. The *density* of an interval representation is the maximum number of intervals crossing any point on the real line. Two intervals are said to *overlap* if neither contains the other and they are not disjoint. If two intervals overlap then their corresponding chords in the circle intersect. A maximum weight independent set is then a set of non-overlapping intervals

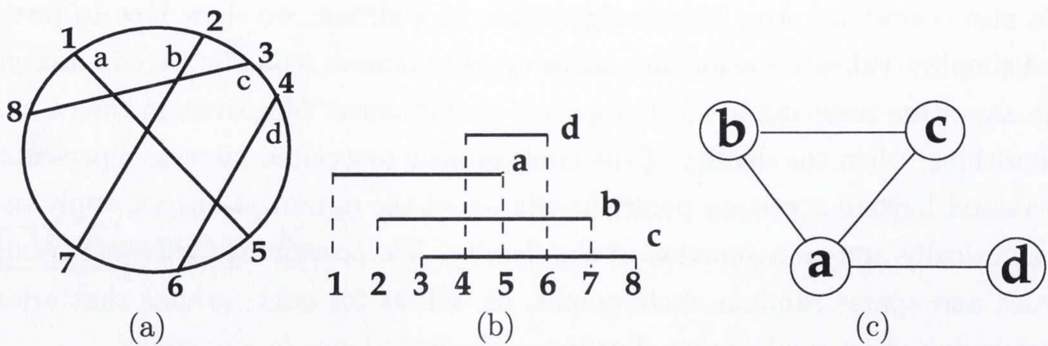


Figure 5.1: The chords in the circle of (a) give rise to the circle graph in (c). The intervals in (b) also give rise to the circle graph of (c). The interval representation in (b) has density 4.

with largest total weight.

Finding a maximum independent set of a circle graph can be solved in time polynomial in the number of vertices of the graph, Gavril [1973] presented an $\Theta(n^3)$, algorithm while others developed $\Theta(n^2)$ algorithms [Supowit 1987; Asano et al. 1991; Goldschmidt and Takvorian 1994]. Other problems that are NP-complete for general graphs can also be solved in polynomial time for circle graphs, for example Tiskin [2009] has shown an $O(n \log^2 d)$ time algorithm for the maximum clique problem, and there are also problems that are NP-complete for both circle graphs and general graphs [Keil 1993].

In this chapter we begin by presenting highly efficient implementations of the two efficient algorithms for computing a maximum independent set of a circle graph. These are the algorithms of Apostolico *et al.* [1992; 1993] and Valiente [2003]. Apostolico *et al.* solve the problem in $\Theta(dn)$ time and space, given the interval representation of an n vertex circle graph with density d. Valiente's algorithm solves the problem in $\Theta(\ell)$ time and $\Theta(n)$ space, where ℓ is sum of the lengths of the intervals in the interval representation. Note that $\ell = \Theta(dn)$ [Valiente 2003]. We also compare the performance of these two algorithms to that of Supowit's simple $\Theta(n^2)$ time and space algorithm.

Following this, we describe an algorithm requiring $\Theta(n\alpha)$ time for an n vertex unweighted circle graph with independence number α. Such an algorithm is inferior to the $\Theta(dn)$ time algorithms mentioned above when $\alpha = o(d)$. As a result, we then show how this output sensitive algorithm can be modified to give an algorithm operating in time $\Theta(n \min\{d, \alpha \log n\})$. In comparison to the best of the variations we develop of the $\Theta(dn)$ time algorithms, we show experimentally that this algorithm performs excellently in practice. Finally, we sketch how a more complicated algorithm operating in time $\Theta(n \min\{d, \alpha\})$ can be achieved, although we do not present experiments determining the practicality of this algorithm.

5.3 Experimental Setup

In this section we briefly describe how our time, space and other measurements were gathered, as well as the machine configuration used. Our machine had a pair of dual-core Intel Xeon 5140 processors, with each core running at 2.33 GHz (note that our implementations are not parallelized and run only on a single processor), and 4 GiB of RAM.

We gathered measurements over two types of random interval representation. Recall that an interval representation of an n vertex circle graph can be encoded as a permutation of $\{1, \ldots, 2n\}$. Thus, to generate a random interval representation it is sufficient to generate a random permutation of $\{1, \ldots, 2n\}$. Such a generator clearly generates each interval representation with equal probability, since each interval representation is described by the same number of permutations (in particular, each interval representation corresponds to exactly $n! \times 2^n$ distinct permutations).

These random interval representations were generated with sizes beginning at 2000 intervals and growing in increments of 2000 intervals to 50000 intervals. Results gathered over these interval representation were averaged over 20 runs. These random interval representations are equivalent to the ones studied by Scheinerman [1988]. It should be noted that the density of interval representations generated in this manner is almost always very close to $n/2$. Figure 5.2(a) gives an idea of the distribution of the density of these random interval representations. We refer to these random interval representations as Type-I interval representations.

In order to study the behaviour of the algorithms with interval representations of more varied densities, we generated random interval representations corresponding to another model described by Scheinerman [1990]. In this model, n random centre points $X_1, \ldots, X_n$, uniformly distributed in $[0, 1]$ are chosen. In addition, n random so-called radii $R_1, \ldots, R_n$, uniformly distributed in $[0, r]$ are also chosen. The n intervals are then formed by $[X_i - R_i, X_i + R_i]$. These interval representations are then converted to the usual permutation representation described above in $\Theta(n \log n)$ time. We refer to these random interval representations as Type-II interval representations.

The expected interval length in this model is r, and by varying this parameter interval representations in a wide range of densities can be randomly generated. Figure 5.2(b) shows how the average density of these interval representation varies as the radius parameter is increased. We examine the behaviour of the maximum independent set algorithms firstly by fixing r and generating interval representations with sizes beginning at 2000 intervals and growing in increments of 2000 intervals to 50000 intervals.

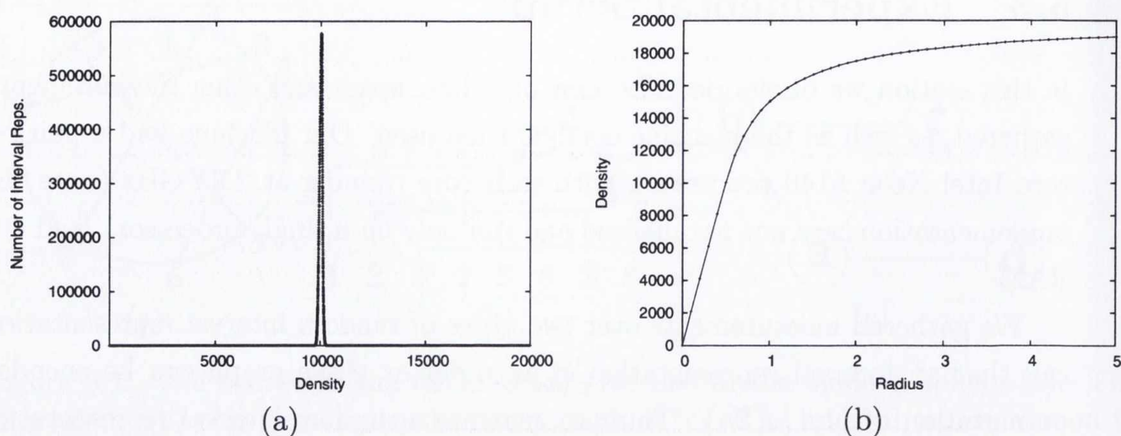


Figure 5.2: For (a) We generated 10^8 random Type-I interval representation on 20000 chords and recorded their density. This figure shows that the vast majority of interval representations have density close to $n/2$. In (b) we generated random Type-II interval representations of 20000 chords while varying the radius parameter linearly between 0 and 5 in 50 steps, we did this 20 times and averaged the results.

As with the other results, results over these interval representation were averaged over 20 runs. We examined the behavior of the algorithms for radii of 0.1, 0.5, 1 and 5. These radii give on average densities of 10%, 50%, 75% and 95% of the number of intervals in the interval representation. We also generated results by fixing the number of intervals at 20000, and then allowing r to vary. To do this, we varied r linearly between 0 and 5 in 25 steps.

We used the *C* library function `gettimeofday` for timing measurements. For memory usage we used counters to keep track of the memory allocated and freed by calls to `malloc` and `free`, including the overheads of the memory allocator. Below we also present results of the number of times particular loops in our implementations iterate, these iteration counts were measured by inserting counters at the appropriate points of the program.

5.4 Overview of Algorithms

5.4.1 Supowit's Algorithm

We begin by briefly describing Supowit's [1987] simple dynamic programming algorithm for computing a maximum independent set of a circle graph, requiring $\Theta(n^2)$ time and space.

Given a circle graph represented by a permutation σ of $\{1, \ldots, 2n\}$ as described

in Section 5.2, let $MIS[a, b]$ denote the weight of a maximum independent set of all intervals with both end-points contained in $[a, b]$. Here $[a, b]$ denotes the set $\{x \in R : a \leq x \leq b\}$. For an interval i with left end-point a and right end-point b we write $i = [a, b]$, and denote its weight by w_i.

Clearly $MIS[a, b] = 0$ for $a \geq b$. The entries of MIS are evaluated such that $MIS[a_1, b]$ is evaluated before $MIS[a_2, b]$ for $a_1 > a_2$, and by decreasing b from $2n$ down to 1. If a is the right end-point of some interval then $MIS[a, b] = MIS[a+1, b]$. Otherwise, if a is the left end-point of some interval $i = [a, r]$ with $r \leq b$ then

$$MIS[a, b] = \max(MIS[a+1, b], w_i + MIS[a+1, r-1] + MIS[r+1, b])$$

The weight of the circle graph's maximum independent sets is given by $MIS[1, 2n]$. A maximum independent set itself (and not just its weight) can easily be maintained during the evaluation of this recurrence. Although very simple, the $\Theta(n^2)$ space requirement of this algorithm limits its practicality (see Sections 5.7 and 5.8).

5.4.2 More Efficient Approaches

The problem of computing a maximum independent set of a weighted circle graph can be reduced to computing a maximum independent set of an appropriately weighted *interval* graph. Both of the algorithms we focus on in this chapter [Apostolico et al. 1992; Valiente 2003] make use of this reduction.

An interval graph is an undirected graph isomorphic to the intersection graph of a finite set of intervals on the real line. Given an interval graph represented by a permutation σ a maximum independent set of that interval graph corresponds to a maximum weight set of disjoint intervals in σ. This maximum independent set can be found easily in $\Theta(n)$ time and space using a simple dynamic programming algorithm [Gupta et al. 1982] that we now review.

Given the σ representation of an n vertex interval graph, let $T[q]$ denote the weight of a maximum independent set of all intervals with both end-points in the range $[q, 2n]$. Clearly $T[2n] = 0$. If q is the right end-point of some interval, then $T[q] = T[q+1]$. Otherwise, if q is the left end-point of some interval $i = [q, r]$ with weight w_i then

$$T[q] = \max(T[q+1], T[r+1] + w_i) \tag{5.1}$$

The weight of a maximum independent set is given by $T[1]$ and a maximum independent set itself is easily maintained while evaluating the recurrence.

Before continuing our discussion of a maximum independent set algorithms, we introduce a small amount of notation that is used throughout this chapter. We denote by $MIS[a,b]$ the weight of a maximum independent set of all intervals with both end-points in $[a,b]$. For an interval $i = [a,b]$, the maximum weight independent set contained in i (i.e. a maximum weight independent set of all intervals with both end-points in $[a,b]$) will often be of interest, and we use the notation $CMIS[i]$ to denote $MIS[a,b]$.

If the interval representation of a circle graph has interval weights given by $w_i = CMIS[i]$, then we can use the preceding recurrence on the circle graph's interval representation to compute the weight of the circle graph's maximum independent set. If a maximum independent set of this weighted interval graph is $S = \{i_1, i_2, \ldots, i_m\}$, then a maximum independent set of the underlying circle graph is composed of S, together with all the intervals in a maximum weight independent set contained in each of the intervals $i_1, \ldots, i_m$ in S.

We next describe two efficient algorithms for computing these $CMIS$ values for a weighted circle graph, showing optimizations that greatly improve their run-time in practice.

5.5 Apostolico et al's Algorithm

Given an interval representation with density d of an n vertex circle graph, Apostolico *et al*'s [1992; 1993] algorithm computes the $CMIS$ values described in the previous section in $\Theta(dn)$ time and space. We begin by describing this algorithm and then describe optimizations that significantly improve its running time and decrease its memory consumption.

For an interval representation with $2n$ end-points, the algorithm works as a left-to-right scan, with a counter m beginning at 1 and counting towards $2n$. As the scan moves from left-to-right the $CMIS$ values are updated progressively. If the scan is at position m, then all intervals $i = [a,b]$ with $a < b \leq m$ have $CMIS[i] = MIS[a,b]$. We refer to these intervals as *closed*, and their $CMIS$ values require no further updating. On the other hand, all intervals i with $a \leq m < b$ are referred to as *open* and have $CMIS[i] = MIS[a,m]$. We next describe the two data structures required to maintain these $CMIS$ values as the left-to-right scan progresses.

Firstly, a linked list $OPEN$ is maintained that contains all open intervals in increasing order of left end-point. Secondly, for each interval $i = [a,b]$ a linked list L_b of all open intervals containing i is maintained. These intervals are stored in decreasing

order of left end-point. For each of these open interval $j = [c, d]$, containing i, we also store $MIS[c, b]$ in the linked list node for j in L_b. We now describe how these data structures are used by the algorithm.

If the scan is at position m and m is the left end-point of some interval i, then i is added to the end of $OPEN$ and $CMIS[i] \leftarrow 0$. Otherwise, if m is the right end-point of some interval $i = [a, m]$, i is removed from $OPEN$ and $CMIS[i] \leftarrow CMIS[i] + w_i$. The algorithm must now consider the affect of this new $CMIS$ value on the $CMIS$ values of all intervals containing i, note that the set of such intervals are a subset of those in $OPEN$. Assuming there are intervals in $OPEN$ containing i, we denote by P_{begin} the first interval in $OPEN$ and by P_{end} the interval in $OPEN$ containing i that has the largest left end-point.

The $CMIS$ values of these intervals are updated by performing a right-to-left scan, with a counter q counting downwards from $a - 1$ (recall that $i = [a, m]$). We refer to this right-to-left scan as the *local* scan. At each position q of the local scan the actions taken depend on whether q is the right or left end-point of an interval:

1. If the current position q of the local scan corresponds to the right end-point of some interval $j = [c, q]$ then the list L_q is scanned until we pass the interval P_{begin}. If any interval encountered in this scan is closed, it is removed from L_q. For any open interval $k = [d, e]$ encountered the following update is performed:

$$CMIS[k] \leftarrow \max(CMIS[k], MIS[d, q] + CMIS[i])$$

   Here $MIS[d, q]$ is retrieved from L_q. P_{begin} is now advanced to the interval in $OPEN$ that immediately succeeds the first open interval encountered during the scan of list L_q, if there is no such interval, the local scan terminates. In addition, if the left end-point P_{begin} is to the right of the left end-point of P_{end} the local scan also terminates. This new value of P_{begin} removes from consideration all intervals that have just had their $CMIS$ values updated. It is important that P_{begin} is computed in constant time. This can be done during the scan of L_q by noting the first open interval encountered in L_q.

2. On the other hand, if q corresponds to the left end-point of some interval $j = [q, d]$ containing i, then there are no intervals contained in $[q, a]$ (since otherwise j would have been removed from consideration when the right end-point of any such interval contained in $[q, a]$ was encountered in step 1) and so the following update is performed:

$$CMIS[j] \leftarrow \max(CMIS[j], CMIS[i])$$

Finally, P_{end} is set to the interval immediately preceding j in $OPEN$, since j requires no further updating during this local scan. The local scan terminates if there is no such interval, or if the left end-point of this new P_{end} is to the left of the left end-point of P_{begin}.

After the local scan has finished, the final step to be taken is to construct the list L_m (assuming m is a right end-point of interval i, and a local scan actually occurred at all). Creating L_m requires traversing all intervals in $OPEN$ from start to end and for each interval $j = [c, d]$ containing i adding it to the front of L_m with its $MIS[c, m]$ value set to $CMIS[j]$.

This completes the description of computing the $CMIS$ values for an interval representation of a circle graph. We now give the space and time analysis of the algorithm just described. The space required by the algorithm is dominated by the space required by the L lists. For each of the n intervals we store an L list holding entries for all intervals this interval is contained in. If the density of the interval representation is d then there can be no more than $d-1$ entries in each L list. Thus the space complexity of the algorithm is $\Theta(dn)$.

The left-to-right scan iterates $2n$ times. When the left-to-right scan encounters the left end-point of an interval, a constant time update is incurred. When the right end-point of an interval is encountered, a local right-to-left scan begins. It is simple to show that the local scan need never iterate more than $2d$ times [Apostolico et al. 1992, 1993]. At any point during a local scan when a right end-point is encountered (i.e. step 1 described in the previous section) up to $\Theta(d)$ time can be spent removing intervals from the L lists. However, the total time spent removing intervals from the L lists can be at most $\Theta(dn)$ since they contain in total fewer than dn entries. Thus the algorithm takes $\Theta(dn)$ time in total.

We next describe how to construct a maximum independent set itself and correct an error in the description provided by Apostolico *et al.*

5.5.1 Set Construction

Having constructed the $CMIS$ values in $\Theta(dn)$ time and space, they are used to weight the interval representation of the circle graph and then find a maximum weight set of

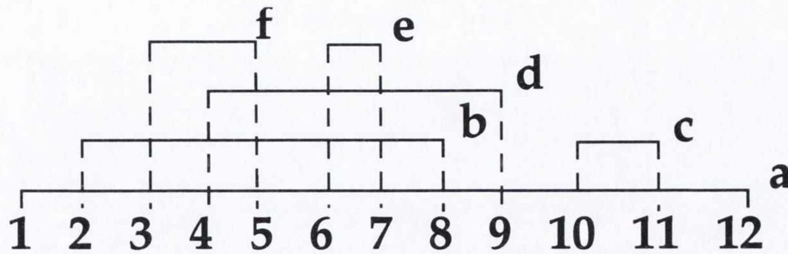


Figure 5.3: This figure demonstrates an error in Apostolico *et al*'s algorithm [1992; 1993]. When the algorithm's left-to-right scan is at position 11, we have (assuming all intervals have unit weights) $CMIS[f] = 1, CMIS[e] = 1, CMIS[b] = 3, CMIS[d] = 2$ and $CMIS[c] = 1$. Since position 11 is the right end-point of c a local right-to-left scan begins from position 9. Now, 9 is the right end-point of d, and so an update of the type described in step 1 in Section 5.5 occurs. The update has $P_{begin} = P_{end} = a$, list L_9 is scanned and the value $MIS[1,9] = 3$ is retrieved. This causes $CMIS[a]$ to be (correctly) updated to 4 from 3. Moreover the pair $\langle c, d\rangle$ is added to V_a. This latter action is incorrect however, instead the pair $\langle c, b\rangle$ should have been added to V_a.

disjoint intervals, which takes $\Theta(n)$ time and space. As described in Section 5.4, a maximum independent set of the circle graph is then formed by these disjoint intervals, as well as maximum independent sets contained in these intervals.

Therefore, in order to construct a maximum weight independent set itself, a linked list V_i is associated with each interval i. The list V_i is used for obtaining a maximum independent set contained in interval i. We first describe the incorrect updation strategy for these V lists described by Apostolico *et al**. If $CMIS[k]$ is successfully updated in step 1 of the local scan described in the previous section the pair $\langle i, j\rangle$ is added to V_k. On the other hand, if $CMIS[j]$ is successfully updated in step 2 of the local scan, the pair $\langle i, \text{NIL}\rangle$ is added to V_j.

The maximum independent set contained in interval i can then be constructed by examining V_i. If V_i is empty, then there are no intervals in a maximum independent set contained in i. Otherwise, V_i is traversed from start to end. The first pair $\langle j_1, j_2\rangle$ encountered in this traversal implies j_1 is contained in the set. If $j_2 = \text{NIL}$, then the traversal terminates. Otherwise, j_2 is also included in the set and the traversal continues until it encounters another pair $\langle j_2, j_3\rangle$, or the end of the list is reached. Assuming such a pair is found, and $j_3 \neq \text{NIL}$ then j_3 is included in the set and the traversal searches for a pair with first element j_3. The traversal continues in this manner until a pair with a second element equal to NIL is encountered or it reaches the end of the list. Figure 5.3 shows an example that demonstrates that this updation scheme is incorrect.

*Actually we describe a slight simplification of their scheme, eliminating an array without changing their algorithm's (in)correctness.

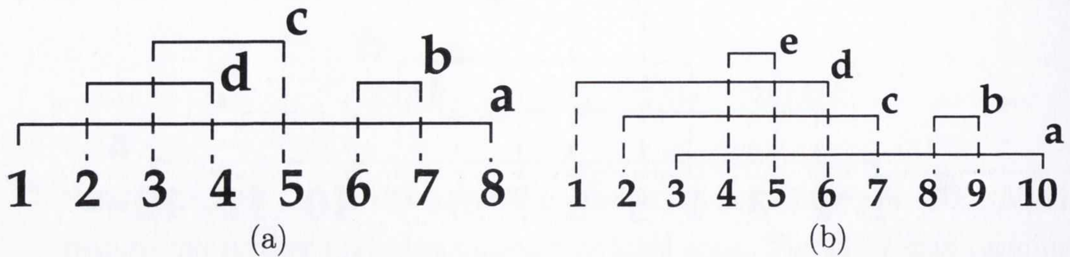


Figure 5.4: In (a), the right end-points of c and d occur contiguously. We have $limit[4] = 3$, and since the left end-point of d is 2, and $2 < 3$, we require no entries in L_4 because any interval (like a) containing d must contain c too. In (b) the right end-points of c, d and e occur contiguously, and we have $limit[5] = 2$. Since the left end-point of e is 4 and $4 > 2$, L_5 will have an entry at least for c. L_5 does not require an entry for d because the left end-point of d is 1 and $1 < 2$, i.e. an interval containing d must also contain c. It does require an entry for a however, because $3 > 2$.

The incorrect update to the V lists occurs in step 1 of the local scan described in the previous section. Recall that this is an update to an interval $k = [d, e]$ containing intervals $i = [a, m]$ and $j = [d, q]$, which has the form

$$CMIS[k] \leftarrow \max(CMIS[k], MIS[d, q] + CMIS[i])$$

Where $MIS[d, q]$ is retrieved from L_q. In general adding $\langle i, j \rangle$ to V_k is incorrect whenever j did not update $MIS[d, q]$ to its current value. What is required to make this correct is to store in each L list node the interval h that caused the most recent update to the MIS value also stored in that node. The pair $\langle i, h \rangle$ is then added to V_k, where h is retrieved from L_q. The L lists can easily be augmented with this extra data during their updation after the local scan terminates. When updating an L list for the right end-point of some interval $k = [a, b]$, and the first entry in V_k is $\langle i, j \rangle$ the pair $\langle i, CMIS[k] \rangle$ should be added to L_b.

Referring again to Figure 5.3 the error is fixed because when constructing L_9 we add $\langle b, CMIS[a] \rangle$ to L_9, here b is obtained from the front of V_a. Now when the left-to-right scan reaches position 11 and the local right-to-left scan is triggered from position 9, we add the pair $\langle c, b \rangle$ to V_a, with b retrieved from L_9.

5.5.2 Optimizations

The running time of Apostolico *et al*'s [1992; 1993] algorithm is dominated by the time spent adding and removing nodes from the L lists. A simple implementation of the algorithm is likely to call on the operating system in order to dynamically

Input: An interval representation σ of an n vertex weighted circle graph.
Output: The *limit* table described in Section 5.5.2.

```
l ← 0
for m ← 2n downto 1 do
    limit[m] ← l
    if m is the right end-point of some i = [a, m] then
        if a > l then
            l ← a
    else
        l ← 0
```

Figure 5.5: Pseudo-code to build the *limit* table described in Section 5.5.2, which we use to avoid adding nodes to the L lists of Apostolico *et al*'s [1992; 1993] algorithm.

allocate the nodes in these lists (e.g. in C using `malloc` and `free`). Doing so results in a very inefficient implementation however. This is because the dynamic memory management operations are often very expensive in time, especially to free memory. Moreover, when many small units of memory are requested dynamically, as is often the case when allocating linked list nodes, the overhead of the memory allocator can be unacceptably high.

As a result, for our implementation linked list nodes are allocated in large pools and added and returned to those pools by the application. This results in a significant speed-up to the implementation.

The second optimization we apply is also to the L lists. Consider two intervals $i = [a, b]$, $j = [c, b + 1]$. In this case, L_b need only contain nodes for those intervals that i is contained in but in which j is not contained. This is because in any local scan (described in Section 5.5) the right end-point of j will be encountered before the right end-point of i, and all intervals in which i and j are both contained will thus be removed from consideration before the local scan reaches the right end-point of i. Note that if $c > a$ then no entries are required in L_b. This technique can be applied to any number of intervals whose right end-points occur continguously.

Given the interval representation of an n vertex circle graph, this optimization is implemented by first preforming a pre-processing pass taking $\Theta(n)$ time and space. A table *limit* is constructed and *limit*$[m]$, where m is the right end-point of some interval $i = [a, m]$, gives the left end-point of the interval with right end-point at $m + 1$, or zero if there is no such interval. Now if $a <$ *limit*$[m]$, L_m requires no entries, Figure 5.4(a) shows an example of this situation. On the other hand, if $a >$ *limit*$[m]$, then

L_m should contain entries for all open intervals $j = [c, d]$ with $c \geq limit[m]$. If at a right end-point m, there are also right end-points of intervals at $m+1, m+2, \ldots, m+p$ then $limit[m]$ holds the largest left end-point of those p intervals. Figure 5.4(b) shows another example of how this *limit* table is used. Figure 5.5 shows pseudo-code for constructing this *limit* table.

Figure 5.6 shows pseudo-code for the entire, optimized version of the algorithm. A small amount of extra notation is introduced in the pseudo-code. We denote by $left(i)$ and $right(i)$ the left and right end-points of an interval i. The functions $first$, $prev$, $next$, add_front, add_end, and $remove$ are all constant time linked-list operations. For example, $first(X)$ returns the first node in linked list X, while $remove(p, X)$ removes node p from list X. As a final example, $next(p, X)$ returns the node immediately following p in X. Both *next* and *prev* return NIL if the requested node does not exist. In the pseudo-code we write $X \leftarrow remove(p, X)$ simply as a reminder that *remove* modifies X, and similarly for add_front and add_end. Given a node of a linked list containing data pertaining to an interval i, we use i to denote both that interval and the node itself. The context always makes the distinction clear however.

Finally, recall that for an interval $i = [a, q]$ the list L_q holds $MIS[c, q]$ for each interval $j = [c, d]$ containing i. In the pseudo-code we write $w(j, L_q)$ to denote this MIS value. L_q also stores, for each interval j containing interval i, the interval that updated $MIS[c, q]$ to its current value. We denote this interval by $h(j, L_q)$ in the pseudo-code.

5.5.3 Type-I Results

We now present results for Apostolico *et al*'s [1992; 1993] algorithm operating on randomly generated Type-I interval representations. Figure 5.7(a) shows the run-times of a basic implementation of the algorithm, together with implementations using a custom memory allocator, and the fully optimized version, which uses the *limit* table as well as a custom memory allocator. The fully optimized version is more than twice as fast as the basic implementation.

Figure 5.7(b) shows the iteration counts of the loop that builds the L lists, in both the basic and optimized implementations (note that the the loop that builds the L list iterates the same number of times in both the basic implementation and the implementation using a custom memory allocator). In the basic implementation, the table *limit* is not employed to reduce the number of nodes added to the L lists and consequently its iteration counts are greater than in the optimized implementation. On average, the optimized implementation's L list building loop iterates about 33% fewer

```
Input: An interval representation σ of an n vertex weighted circle graph.
Output: The maximum weight independent set contained in each interval i of σ.
for m ← 1 to 2n do
    if m is the left end-point of some i then
        CMIS[i] ← 0
        OPEN ← add_end(i, OPEN)
    else m is the right end-point of some i = [a, m]
        P_end ← prev(i, OPEN)
        CMIS[i] ← CMIS[i] + w_i
        OPEN ← remove(i, OPEN)
        P_begin ← first(OPEN)
        q ← a − 1
        while P_begin ≠ NIL and P_end ≠ NIL and left(P_begin) ≤ left(P_end) do
            if q is the right end-point of some j = [c, q] then
                k ← first(L_q)
                new ← NIL
                while k ≠ NIL and left(k) ≥ left(P_begin) do
                    tmp ← next(k, L_q)
                    if right(k) > m then
                        if new = NIL then
                            new ← next(k, OPEN)
                        if CMIS[k] < w(k, L_q) + CMIS[i] then
                            CMIS[k] ← w(k, L_q) + CMIS[i]
                            V_k ← add_front(⟨i, h(k, L_q)⟩, V_k)
                    else
                        L_q ← remove(k, L_q)
                    k ← tmp
                P_begin ← new
            else q is the left end-point of some j = [q, d]
                if d > m then
                    if CMIS[j] < CMIS[i]
                        CMIS[j] ← CMIS[i]
                        V_j ← add_front(⟨i, NIL⟩, V_j)
                    P_end ← prev(j, OPEN)
            q ← q − 1
        if limit[m] < a then
            j ← first(OPEN)
            while j ≠ NIL and a > left(j) do
                if left(j) ≥ limit[m] then
                    ⟨k, g⟩ ← first(V_j)
                    L_m ← add_front(⟨k, CMIS[j]⟩, L_m)
                j ← next(j, OPEN)
```

Figure 5.6: Pseudo-code for a corrected, optimized version of Apostolico *et al*'s [1992; 1993] algorithm, described in Section 5.5. The *limit* table is pre-computed using the algorithm of Figure 5.5.

times than in the basic implementation.

Figure 5.8 shows the memory consumption of our implementations. Using a custom memory allocator reduces the dynamic memory allocator's overhead resulting in a significantly lower memory consumption. The fully optimized version avoids adding many nodes to the L lists using the *limit* table and so has an even lower memory consumption. On average, the memory fully optimized version requires less than half the memory of the basic implementation.

5.5.4 Type-II Results

We now describe the behaviour of Apostolico *et al*'s [1992; 1993] algorithm on randomly generated Type-II interval representations. Figure 5.9(a) shows the run-time of a basic implementation of the algorithm as the radius parameter is varied. As the radius is increased, the run-time increases too. This is to be expected since the density of the interval representations increase as the radius is increased, and the algorithm operates in $\Theta(dn)$ time. Figure 5.9(b) shows the run-time of our optimized implementation of the algorithm. The performance of the optimized implementation of the algorithm is significantly better in all cases. What is especially noteworthy however is that the running time of the optimized implementation does not simply increase as the density of the supplied interval representation increases. The slowest case for the basic implementation, that is, when the radius is 5, is the second fastest case for the optimized implementation. This speed-up when the density is increased is a result of the optimization using the *limit* table. Recall that the *limit* table is used to avoid adding nodes to the L lists (saving both time and space) when two or more right end-points occur contiguously in the interval representation. As the density increases, the probability of consecutive right end-points increases. Thus this optimization results in progressively greater gains.

Figure 5.10(a) shows the memory consumption of the basic implementation of the algorithm. As the radius is increased the memory consumption of the algorithm increases. As with the timing measurements, this is to be expected since the algorithm operates in $\Theta(dn)$ space. Figure 5.10(b) shows the memory consumption of the optimized implementation. Again, like the timing results just described, the memory consumption is better in all cases, but also does not simply increase as the radius is increased. Figure 5.11(a) and Figure 5.11(b) show the number of times the loop that builds the L lists iterates in the basic and optimized implementations respectively. This gives a more machine independent view of the optimizations applied to the algorithm. Note that the speed-up observed is not solely due to these loops iterating a smaller

number of times, since the memory allocator has also been replaced.

5.6 Valiente's Algorithm

Valiente [2003] describes a simple algorithm for constructing a maximum independent set of a weighted circle graph, taking $\Theta(\ell)$ time and only $\Theta(n)$ space, where ℓ is the total interval length of an interval representation of the circle graph. We first describe this algorithm and then give optimizations that dramatically improve its running time.

The algorithm applies the reduction of Section 5.4 recursively, resulting in a simple bottom-up dynamic programming algorithm. The reduction of Section 5.4 provided an $\Theta(n)$ time and space algorithm for finding a maximum independent set contained in the interval $[1, 2n]$ given the $CMIS$ values of all intervals contained in $[1, 2n]$. The idea of Valiente's algorithm is, given any interval $i = [a, b]$, if the $CMIS$ values are known for all intervals contained in i then we can compute $CMIS[i]$ in $\Theta(l_i)$ time and space, where $l_i = b - a$ is the length of interval i.

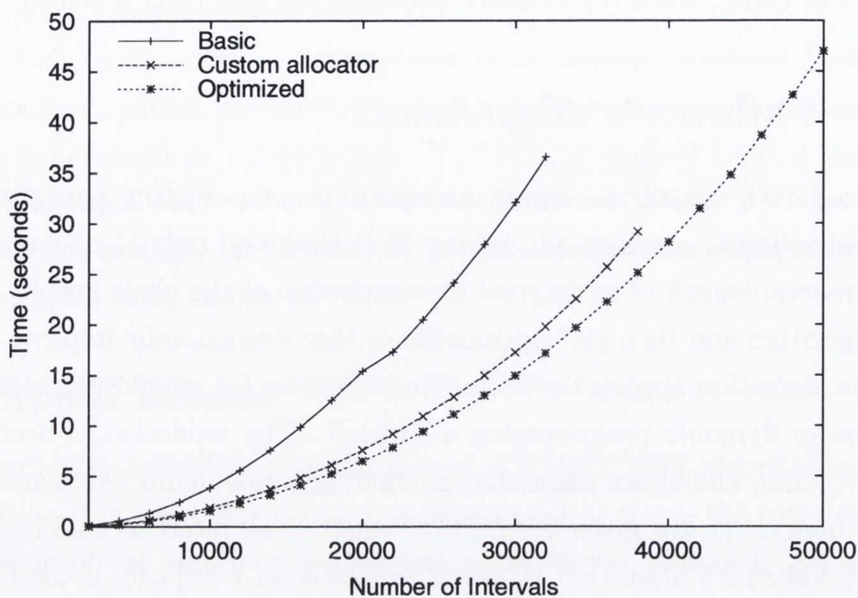
To ensure all $CMIS$ values of intervals contained in i are known when computing $CMIS[i]$, Valiente suggests sorting the intervals in nondecreasing order of interval length. A bucket sort takes $\Theta(n)$ time and space to sort the intervals, and then the $CMIS$ values are computed in order for each interval.

For an interval $i = [a, b]$ we define $MIS[q]$ for $a < q \leq b$ as the weight of a maximum independent set of all intervals contained in $[q, b]$. Clearly $MIS[b] = 0$. If q is the right end-point of some interval then $MIS[q] = MIS[q+1]$. Otherwise, if $j = [q, c]$ with $c < b$ (i.e. q is the left end-point of an interval j contained in i), we have

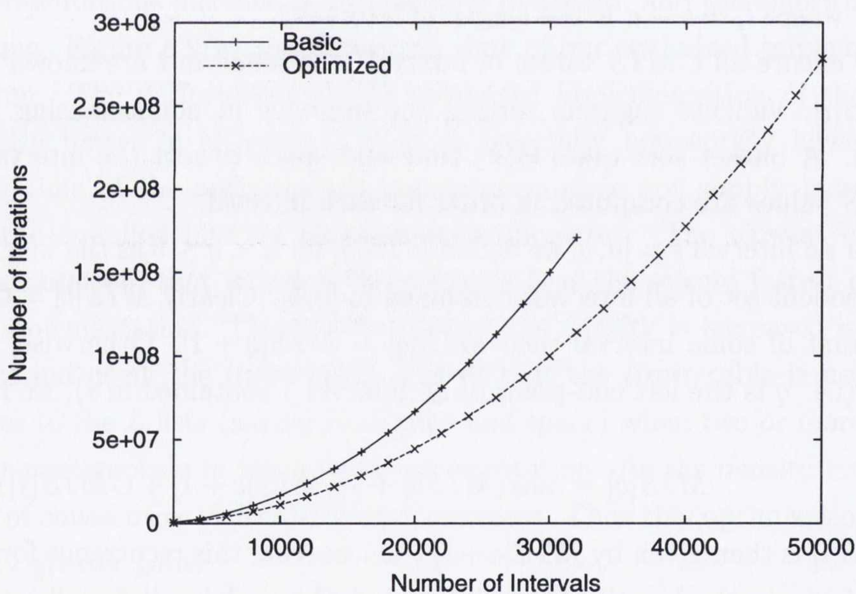
$$MIS[q] = \max(MIS[q+1], MIS[c+1] + CMIS[j]) \tag{5.2}$$

$CMIS[i]$ is then given by $MIS[a+1] + w_i$. Solving this recurrence for an interval takes time linear in the length of that interval, and so solving it for all intervals takes $\Theta(\ell)$ time where ℓ is the sum of the lengths of the intervals. Note that the space required by the recurrence can be re-used for each interval, and so computing the $CMIS$ values takes $\Theta(l_{max})$ space, where l_{max} is the maximum length of any interval. In fact, a simple strategy is to use a single table to evaluate both Recurrences 5.1 and 5.2, requiring $\Theta(n)$ space. Storing all the $CMIS$ values requires an additional $\Theta(n)$ space.

In order to construct a maximum independent set itself, a linked list is maintained for each interval while computing Recurrence 5.2. For an interval i, a list C_i is maintained which stores a reference to all intervals directly contained in a maximum independent set of i (that is, C_i contains all j in a maximum weight independent set



(a)



(b)

Figure 5.7: This figure shows the results for our implementations of Apostolico *et al*'s [1992; 1993] algorithm operating on Type-I interval representations. (a) Shows the running times of our implementations. (b) Shows the the iteration counts for the loop that builds the L lists in the basic and optimized implementations. The optimized implementation's L list building loop iterates on average about 33% fewer times than the loop that builds the L list in the basic implementation. In both figures, the optimized version is shown operating on larger interval representations than the other two, because they allocate all available memory on larger inputs. For the same reason, the implementation using a custom memory allocator is shown operating on larger interval representations than the basic implementation.

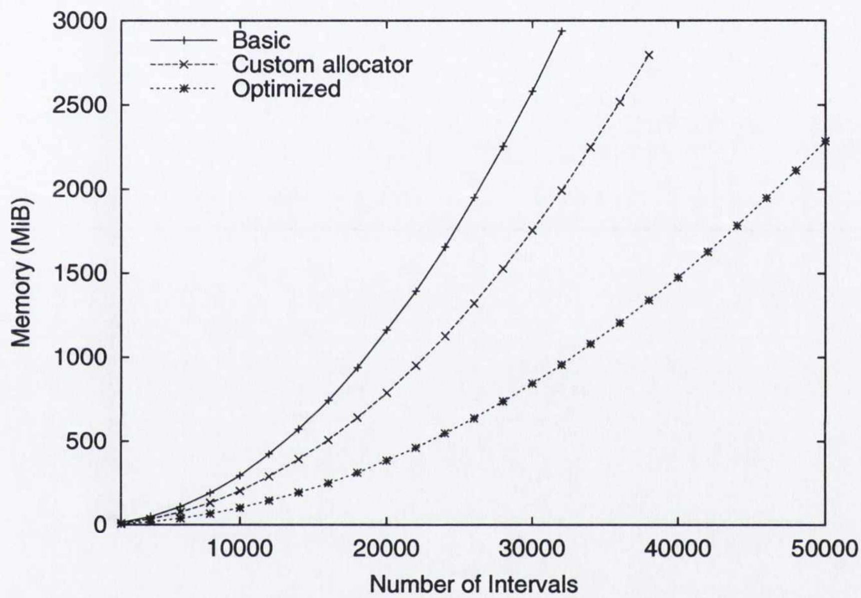
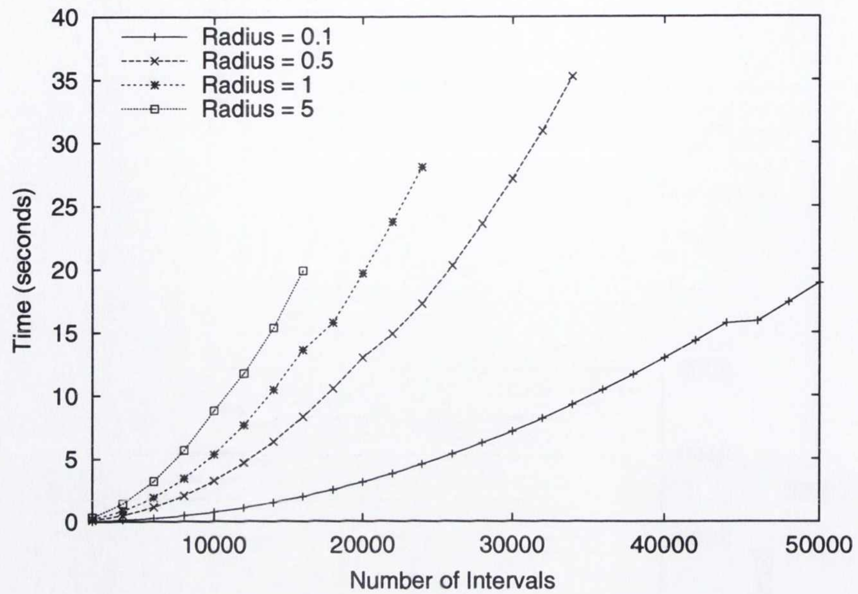
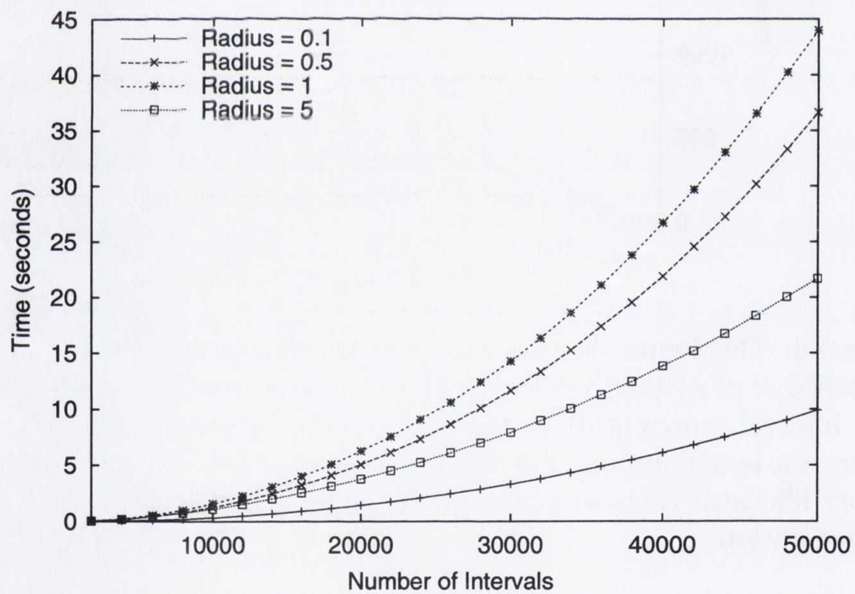


Figure 5.8: This figure shows the memory consumption of the our implementations of Apostolico *et al*'s [1992; 1993] algorithm. The optimized version is shown operating on larger interval representations than the other two, because they allocate all available memory on larger inputs. For the same reason, the implementation using a custom memory allocator is shown operating on larger interval representations than the basic implementation.

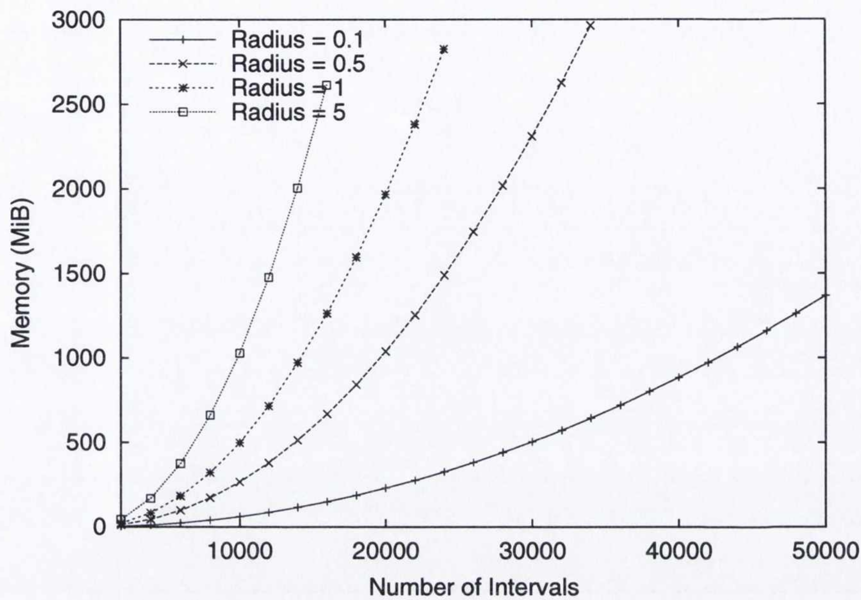


(a)

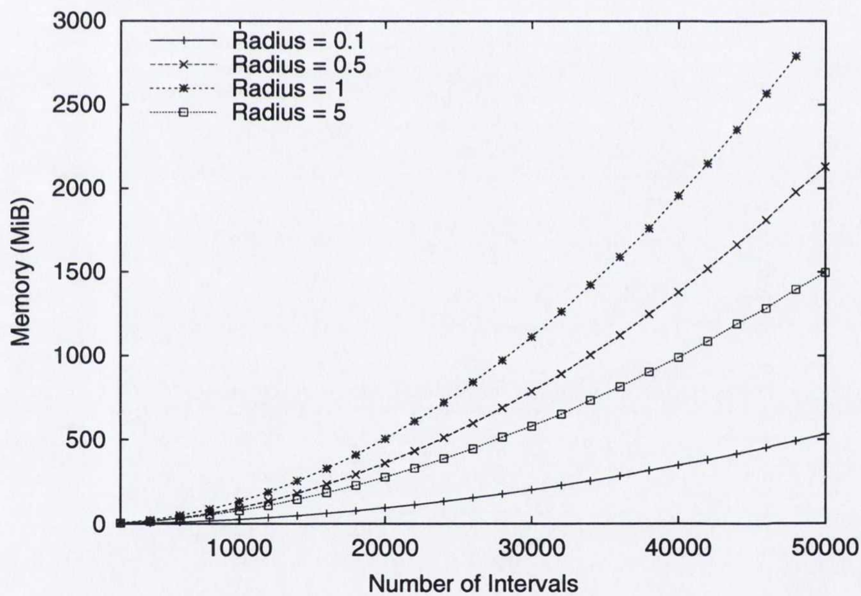


(b)

Figure 5.9: This figure shows the results for our basic implementation of Apostolico *et al's* [1992; 1993] algorithm operating on Type-II interval representations. (a) Shows the running times of our basic implementations of the algorithm. As the radius parameter is increased the density of the interval representations increases and so does the running time. In (b) we see that the optimizations to the algorithm allow it to operate faster in all cases. However, the algorithm also operates much more rapidly on *high* density interval representations, the worst case for the basic algorithm, shown in (a).

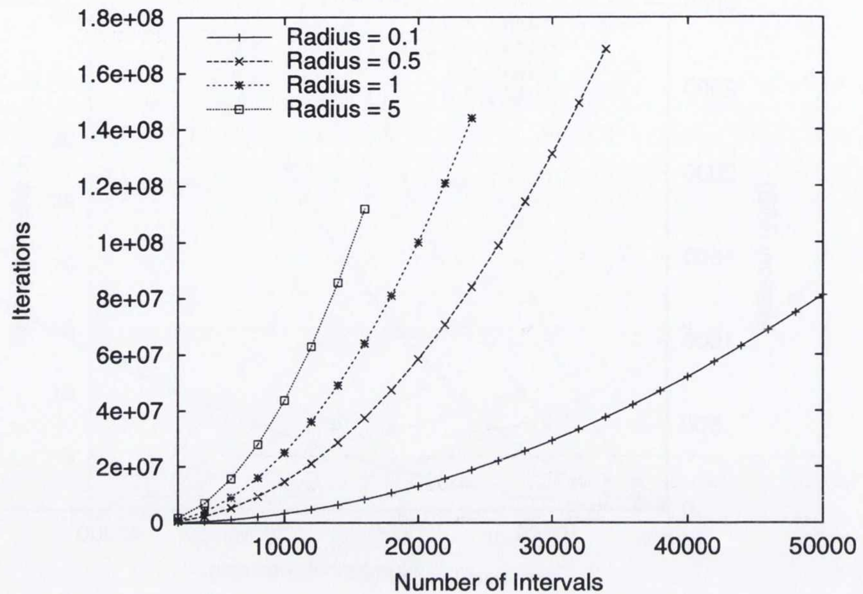


(a)

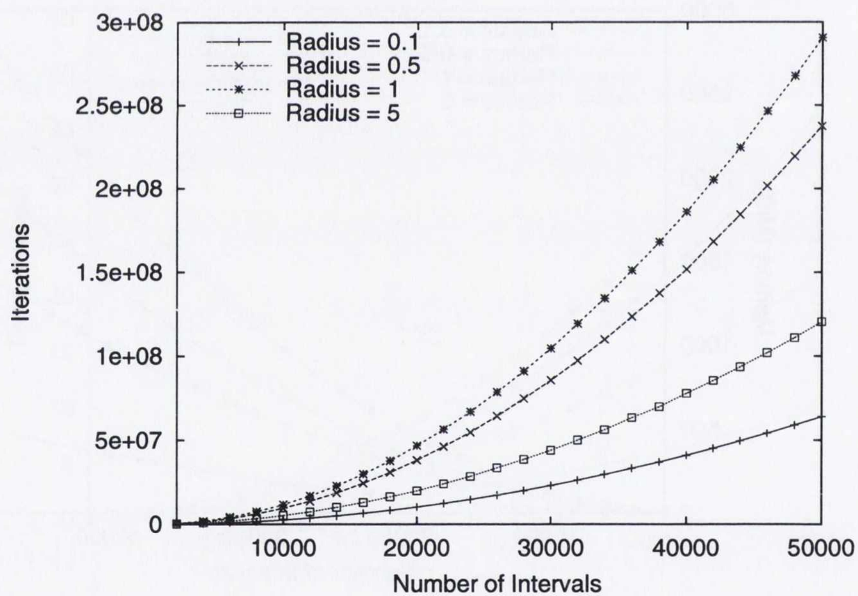


(b)

Figure 5.10: This figure shows the results for our basic and optimized implementations of Apostolico *et al's* [1992; 1993] algorithm operating on Type-II interval representations. In (a) we see that as the radius is increased the memory required by the basic implementation of the algorithm also increases. The results for the optimized version are shown in (b). Due to the optimizations described in Section 5.5.2, memory usage is substantially smaller in all cases, and does not simply increase as the radius is increased.



(a)



(b)

Figure 5.11: This figure shows the number of iterations of the L list creation loop for our basic and optimized implementations of Apostolico *et al's* [1992; 1993] algorithm operating on Type-II interval representations. These results show that the speed-up observed when comparing Figure 5.9(a) to Figure 5.10(a) is due to a reduction in the number of iterations of this loop. This reduction is a consequence of the observations described in Section 5.5.2

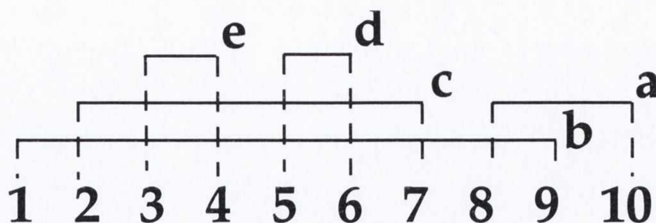


Figure 5.12: Shows an example of our optimization to the right-to-left scan of Valiente's [2003] algorithm. The right-to-left scan for interval b would ordinarily begin at position 8, with our optimization however, it begins at position 2. Similarly, the right-to-left scan for interval c would ordinarly begin at position 6, but using our optimization it begins at position 5 instead.

of i such that $j \notin C_k$ for any other $k \in C_i$). Thus the C lists require $\Theta(n)$ space.

When Recurrence 5.1 is evaluated using the $CMIS$ values, it stores a linked list of the disjoint intervals contained in a maximum independent set. This linked list requires $\Theta(n)$ space. Thus the total space complexity of Valiente's algorithm is $\Theta(n)$. Using this linked list and the C lists it is then a simple matter to retrieve a maximum independent set itself.

5.6.1 Optimizations

Recall from the previous section that to compute $CMIS[i]$, we must have previously computed $CMIS[j]$ for all intervals j contained in i. While sorting the intervals in nondecreasing order of length is a sufficient condition for ensuring that this is the case, it is not necessary. Instead, the algorithm can operate in manner similar to that of the algorithm of Apostolico *et al* [1992; 1993]. That is, as a left-to-right scan with a counter m beginning at 1 and counting towards $2n$. If m is the right end-point of some interval i, we use Recurrence 5.2 to compute $CMIS[i]$ via a right-to-left scan. We are guaranteed to have already computed $CMIS[j]$ for all intervals j contained in i since their right end-points must have occured to the left of m. Thus, there is no need to sort the intervals by length.

Sorting the intervals generally takes only a very small fraction of the algorithm's total execution time, and so simply removing the sorting does not significantly improve the algorithm's performance. However, when the algorithm operates as a left-to-right scan with local right-to-left scans as described in the previous paragraph there is potential for a more significant optimization. Consider a right-to-left scan for an interval $i = [a, q]$, and suppose that the interval $j = [c, d]$ is the interval with d chosen as large as possible while $d < q$. The right-to-left scan will begin by setting $MIS[q] = 0$, and

then, since there are no intervals with both end-points in $[d, q]$ we will have $MIS[q] = 0$ for all $q \geq d$. When the right-to-left scan reaches d and we have $MIS[d] = 0$, it is at this point clear that we are going to re-evaluate Recurrence 5.2 for interval j as the right-to-left scan continues.

To avoid such re-evaluations we evaluate Recurrence 5.2 in a single table M of $2n+1$ elements. Initially M is initialized to all zeros. We also introduce an additional variable *last* to the algorithm which is initially zero. During the left-to-right scan *last* holds the left end-point of the interval whose right end-point was most recently encountered. When scanning from left-to-right if m corresponds to the right end-point of some interval $[a, m]$ we compute only the values $M[last], M[last-1], \ldots, M[a]$ (using Recurrence 5.2) in the right-to-left scan. The variable *last* is then set to a and the left-to-right scan continues. Figure 5.12 shows an example of this optimization at work.

Pseudo-code for an optimized implementation of Valiente's algorithm can be seen in Figure 5.13. The pseudo-code introduces one new function *interval*, this is a constant time operation and $interval(x)$ returns the interval with (left or right) end-point x. The remainder of the notation used in the pseudo-code was already described when the pseudo-code for Apostolico *et al*'s [1992; 1993] algorithm was described in Section 5.5. The pseudo-code also shows the details of how the C lists are constructed using a table P.

5.6.2 Type-I Results

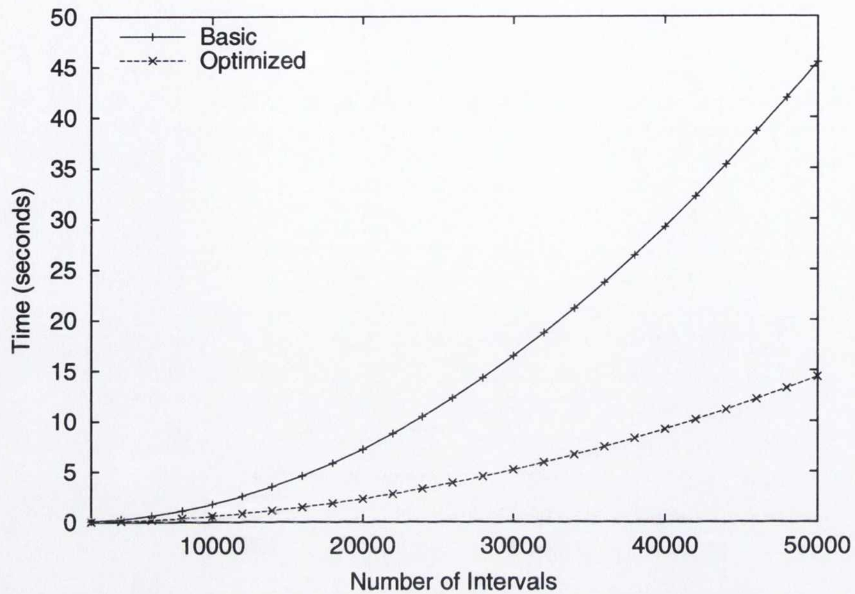
We now describe our measurements of Valiente's [2003] algorithm operating on Type-I interval representations. Figure 5.14(a) shows the running times of our implementation of Valiente's algorithm. The optimized version is more than three times faster on average than the basic implementation. Figure 5.14(b) shows the number of times the right-to-left scan of Valiente's algorithm (see the pseudo-code in Figure 5.13) iterates in the basic implementation versus the optimized implementation. Note that, in the basic implementation, the right-to-left scan begins at $m-1$ instead of the *last* variable which we introduced. These iteration counts give a more machine independent account of the performance of the algorithms. In this case it is clear that the factor of 3 speed-up comes directly from our optimization reducing the number of iterations of the inner-loop also by a factor of 3.

Finally, Figure 5.15 shows the memory consumption of our optimized implementation of the algorithm. The basic and optimized implementations have nearly identical memory usage (in the optimized implementation two buffers associated with bucket sorting the intervals are elided), and so only the memory consumption of the opti-

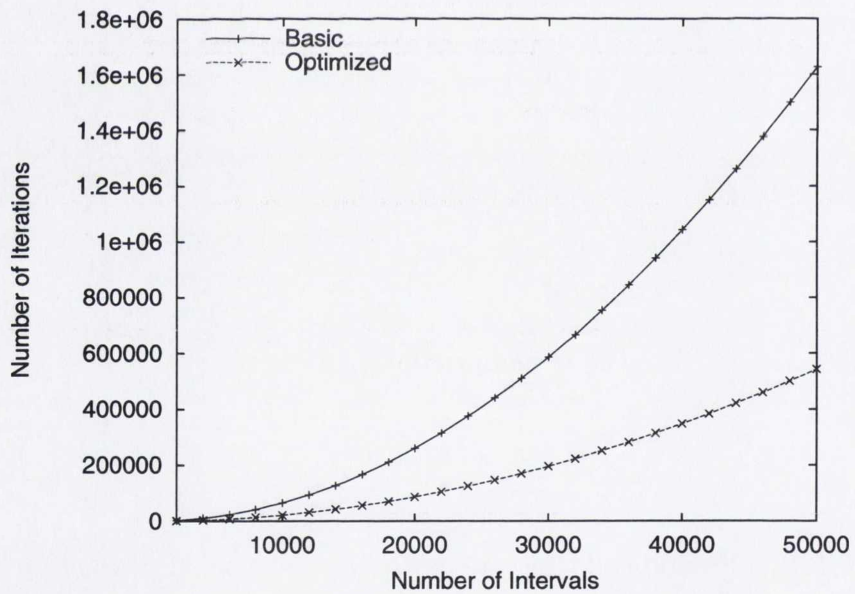
Input: An interval representation σ of an n vertex weighted circle graph.
Output: The maximum weight independent set contained in each interval i of σ.

```
M[1..2n + 1] ← 0
P[1..2n + 1] ← 0
last ← 1
for m ← 1 to 2n do
    if m is the right end-point of some interval i = [a, b] then
        for q ← last downto a + 1 do
            if q is the left end-point of some interval j = [q, c]
                and M[c + 1] + CMIS[j] > M[q + 1] then
                M[q] ← M[c + 1] + CMIS[j]
                P[q] ← c
            else
                M[q] ← M[q + 1]
                P[q] ← P[q + 1]
        CMIS[i] ← M[a + 1] + w_i
        x ← P[a + 1]
        while x ≠ 0
            j ← interval(x)
            C_i ← add_front(j, C_i)
            x ← P[x]
        last ← a
```

Figure 5.13: Pseudo-code for our optimized version of Valiente's [2003] algorithm, described in Section 5.6.



(a)



(b)

Figure 5.14: Shows the running time and memory usage of our implementations of Valiente's algorithm [2003] operating on Type-I interval representations. For the timing measurements in (a), two implementations are shown: A basic implementation, and an implementation using the optimizations we describe in Section 5.6.1, these optimizations provide more than a factor of 3 speed-up. (b) Shows the number of times the right-to-left scans of the implementations iterate, as is to be expected from the timing results, the number of iterations of the optimized implementation is approximately one third that of the basic implementation.

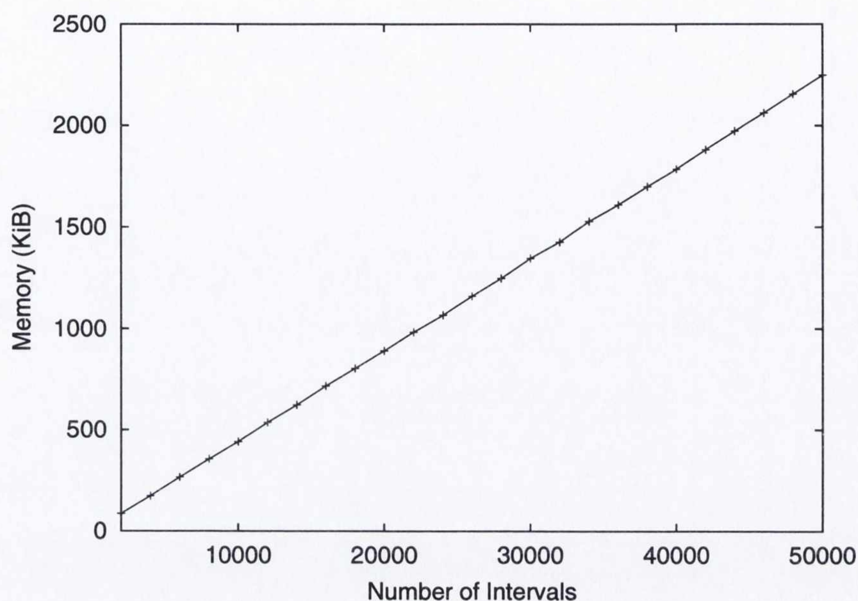
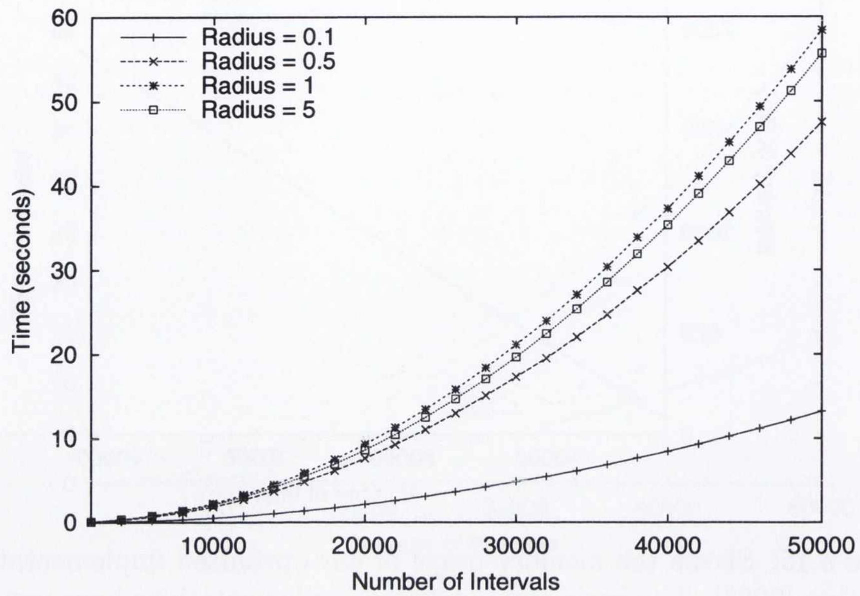


Figure 5.15: Shows the memory usage of our optimized implementation of Valiente's algorithm [2003]. The basic and optimized implementations have very similar memory usage. Compared to the memory usage of Apostolico *et al*'s shown in Figure 5.8 the memory usage is very modest.

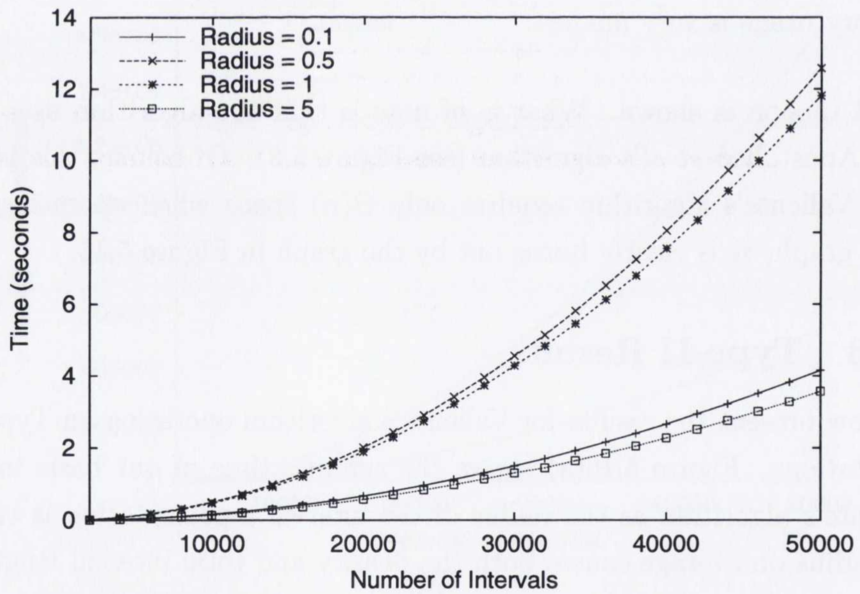
mized version is shown. What is of note is that the algorithm uses far less memory than Apostolico *et al*'s algorithm (see Figure 5.8). Of course, this is to be expected, since Valiente's algorithm requires only $\Theta(n)$ space when operating on an n vertex circle graph, as is clearly borne out by the graph in Figure 5.15.

5.6.3 Type-II Results

We now present the results for Valiente's algorithm operating on Type-II interval representations. Figure 5.16(a) shows the running time of our basic implementation of Valiente's algorithm as the radius of the interval representation is varied. Increasing the radius on average causes both the density and total interval length of the interval representations to increase. As expected, since Valiente's algorithm operates in $\Theta(\ell)$ time the running time of our basic implementation of the algorithm also increases. Figure 5.16(b) shows the running time of our optimized version of Valiente's algorithm as the radius of the interval representation is varied. The optimizations cause the algorithm to operate significantly faster in all cases. The optimizations we have applied to Valiente's algorithm have a similar affect to those we applied to Apostolico *et al*'s [1992; 1993]. That is, as the radius (i.e. density and total interval length) is increased, the run-time of the optimized algorithm at first increases, but then decreases again.

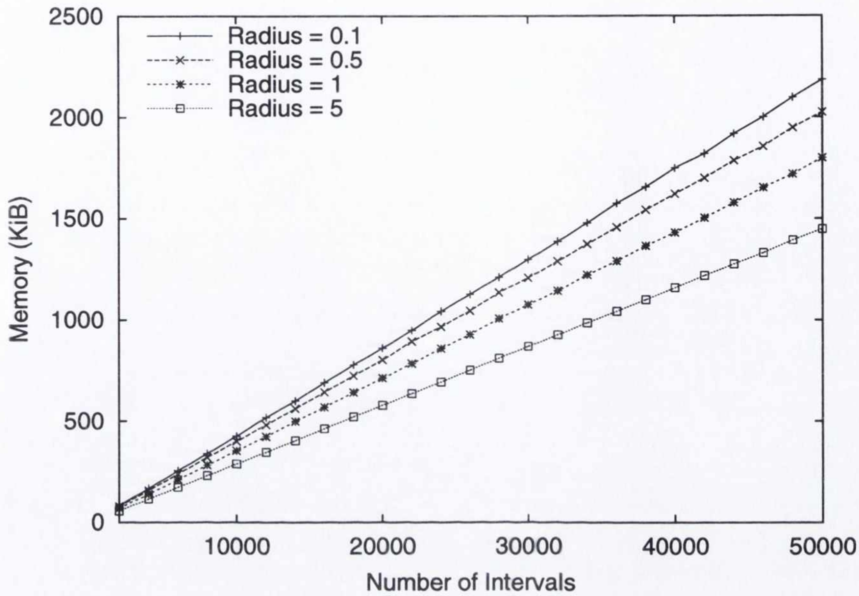


(a)

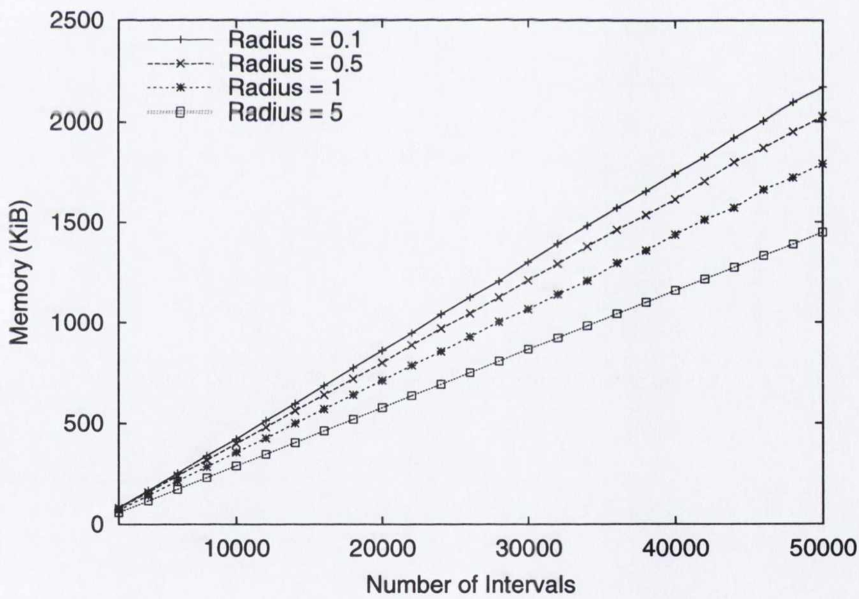


(b)

Figure 5.16: This figure shows the results of Valiente's [2003] algorithm operating on Type-II interval representations. In (a) we see that as the radius is increased (which causes both the average density and average total chord length to increase too) the running time of the basic implementation of the algorithm also increases. The performance of the optimized algorithm is significantly better in all cases. We also observe that the run-time no longer increases as the radius increases.

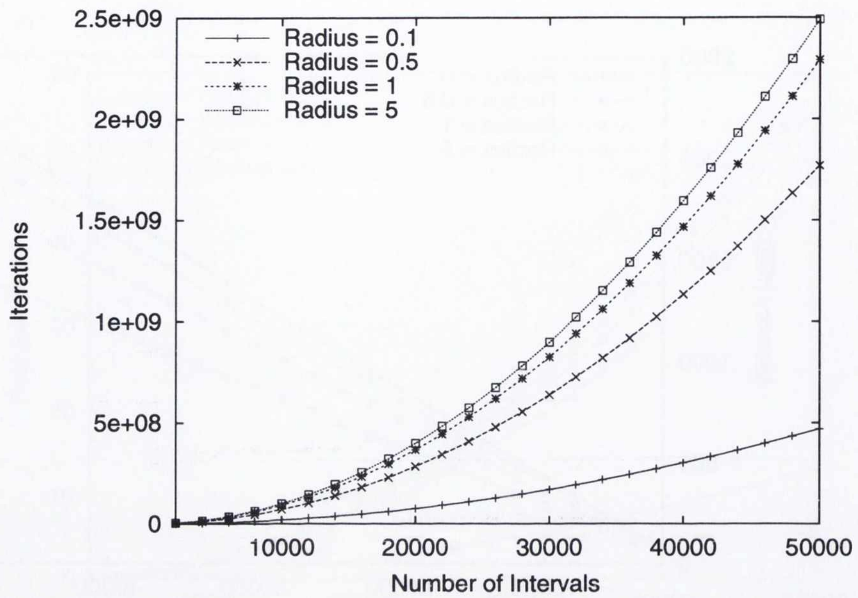


(a)

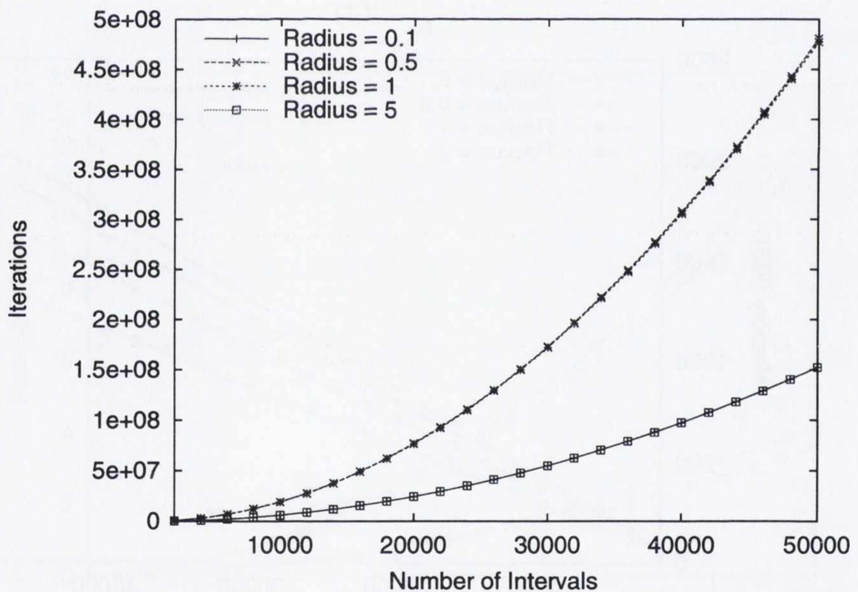


(b)

Figure 5.17: This figure shows the results of Valiente's [2003] algorithm operating on Type-II interval representations. In both (a) and (b) we observe the (modest) memory consumption of the basic and optimized versions of the algorithm respectively. As the radius increases, the memory consumption descreases, this is because the C lists (see Figure 5.13) require less space at higher radii. The memory consumption, as expected, is essentially identical for both implementations.



(a)



(b)

Figure 5.18: This figure shows the results of Valiente's [2003] algorithm operating on Type-II interval representations. Figures (a) and (b) show the respective number of inner loop iterations for the basic and optimized versions of the algorithm. For the basic implementation the number of inner loop iterations increases as the radius increases. However, in the optimized version, the number of inner-loop iterations for radii of 0.1 and 0.5 are close to identical, as are the number of inner-loop iterations for radii of 1 and 5. This figure demonstrates the speed-up observed in Figure 5.16 is due to a reduction in the number of inner-loop iterations as a consequence of the observations made in Section 5.6.1.

Figure 5.17(a) and Figure 5.17(b) show the memory consumption of the basic and optimized versions of the algorithm respectively. Our optimizations have only a tiny affect on the memory consumption of the algorithm. As a result, the figures are very close to identical. The fact that at lower radii the memory consumption is higher is a result of the fact that the total number of disjoint intervals in the interval representation falls as the radius increases, and consequently fewer C list entries are required on average.

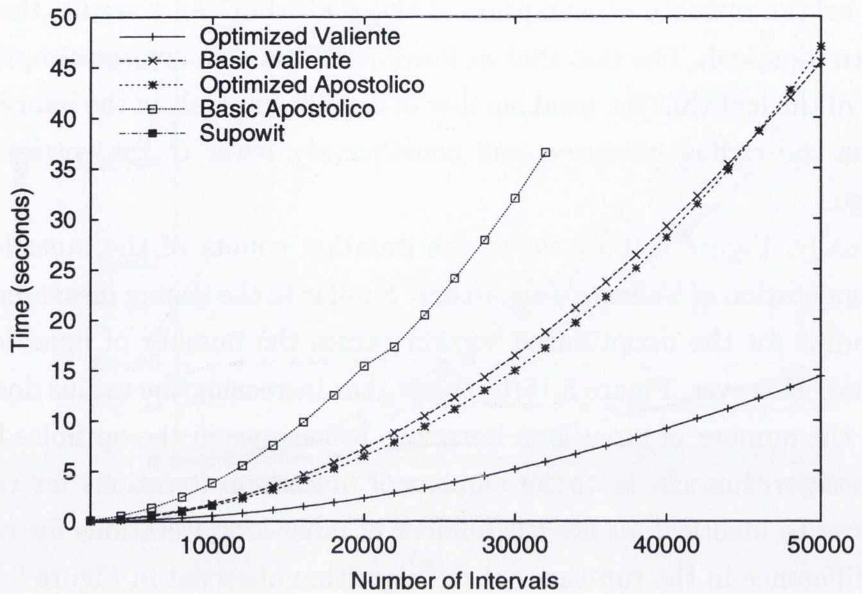
Finally, Figure 5.18(a) shows the iteration counts of the inner-loop of our basic implementation of Valiente's algorithm. Similar to the timing measurements, increasing the radius for the unoptimized version causes the number of inner-loop iterations to increase. However, Figure 5.18(b) shows that increasing the radius does not necessarily cause the number of inner-loop iterations to increase in the optimized implementation of the algorithm. In fact, the number of inner-loop iterations for radii of 0.1 and 5 are close to identical, as are the number of inner-loop iterations for radii of 0.5 and 1. The difference in the run-times of the algorithm observed in Figure 5.16(a) and Figure 5.16(b) can be attributed to the extra time spent constructing the C lists (see Figure 5.13).

5.7 Comparison

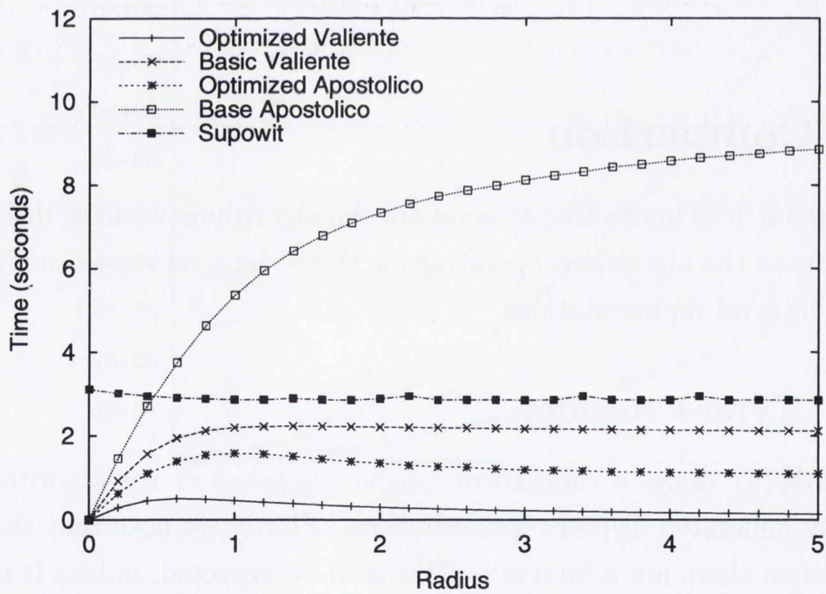
At this point it is interesting to compare the algorithms we have discussed above. We first compare the algorithms operating on Type-I interval representations, and then on Type-II interval representations.

5.7.1 Type-I Results

Figure 5.19(a) shows a comparison of the run-times of the algorithms operating on randomly generated Type-I representations. Firstly we note that the run times grow like n^2 when there are n intervals. This is to be expected, indeed it is straightforward to show that the average total interval length, ℓ_{avg} of a randomly generated Type-I interval representation is given by $\ell_{avg} = \frac{1}{3}n(2n + 1)$. To see this, note that there are $C(n) = (2n)!/(n! \times 2^n)$ distinct interval representations of n vertex circle graphs. Moreover, the total length of all such interval representations is $T(n) = C(n-1)[(2n-1) + 2(2n-2) + 3(2n-3) + \cdots + (2n-1)] = C(n)\frac{1}{3}n(2n+1)$, which gives the average length as claimed. Moreover, as we saw above in Figure 5.2 that d_{avg}, the average density is given by $d_{avg} \approx n/2$.

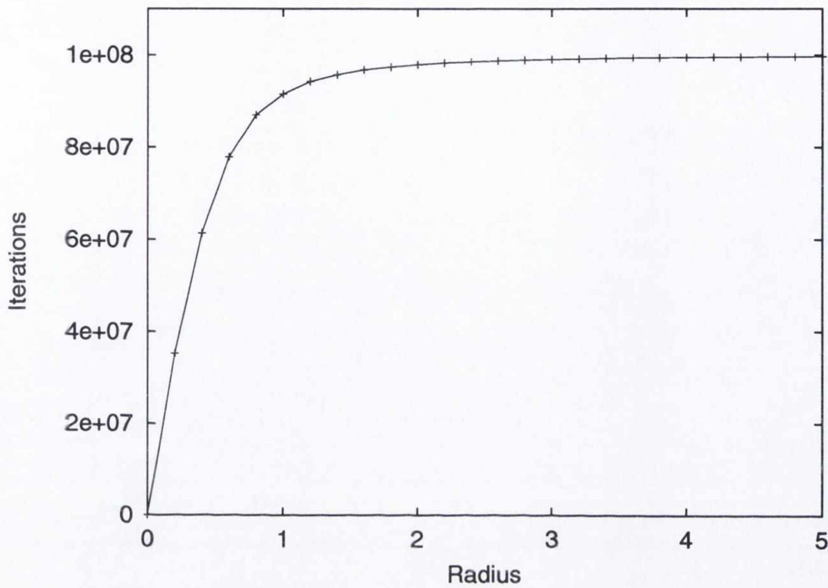


(a)

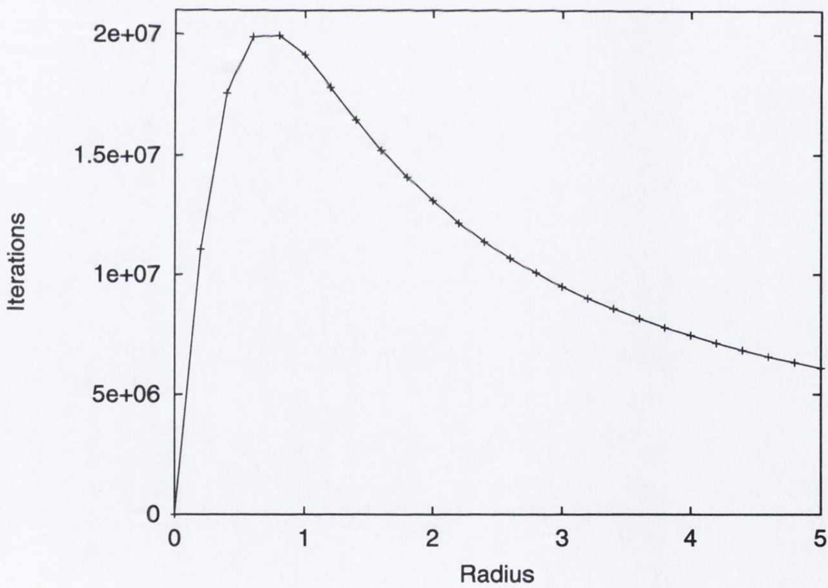


(b)

Figure 5.19: (a) Shows a comparison of the run-times of our optimized and unoptimized implementations operating on randomly generated Type-I interval representations. (b) Shows the results of the algorithms operating on randomly generated Type-II interval representations of 20,000 intervals as the radius parameter is varied. As the radius is increased the density and total chord length of the circle graphs' interval representations increases.

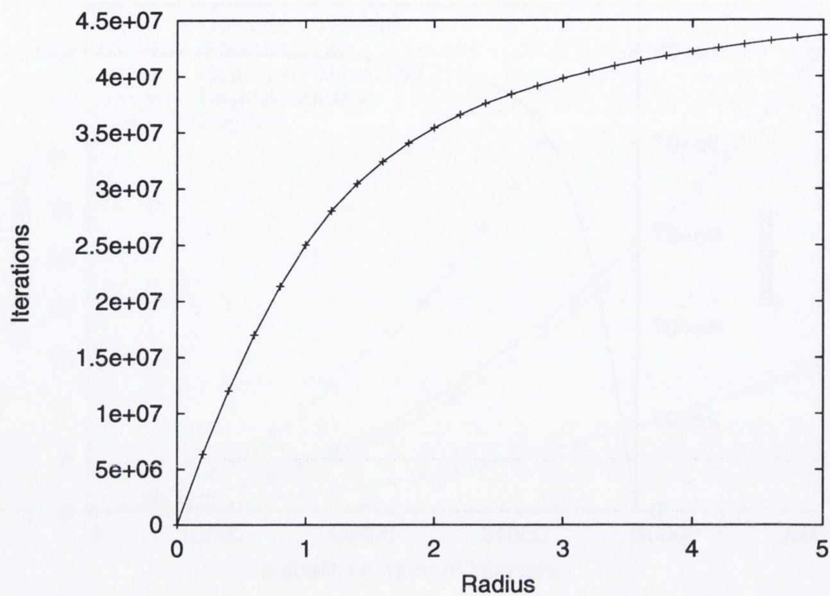


(a)

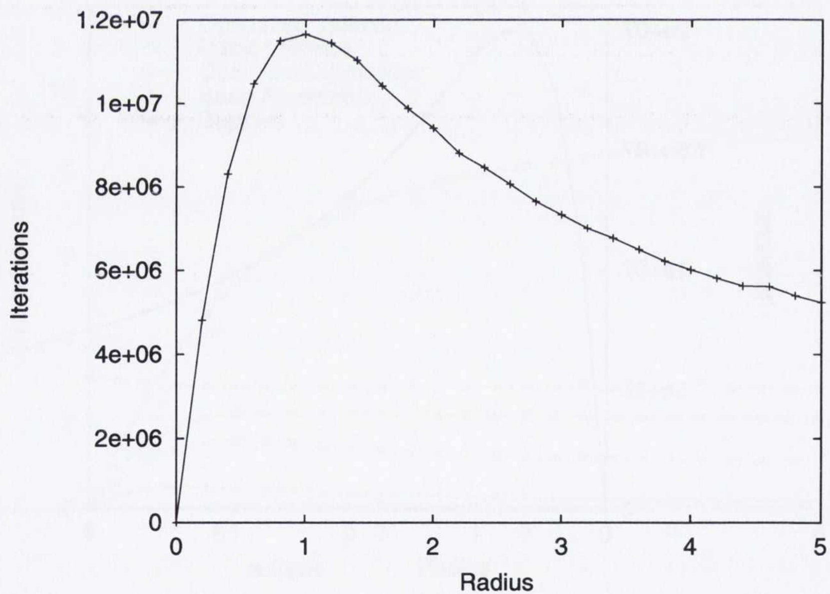


(b)

Figure 5.20: (a) Shows the inner-loop iteration counts of a direct implementation of Valiente's [2003] algorithm operating on randomly generated Type-II interval representations of 20,000 intervals as the radius parameter is varied. (b) Shows the inner-loop iteration counts of our optimized implementation of Valiente's algorithm (see Figure 5.13) on the same data.



(c)



(d)

Figure 5.21: (a) Shows the number of iterations of the L list building loop in a direct implementation of Apostolico *et al*'s [1992; 1993] algorithm, operating on randomly generated Type-II interval representations of 20,000 intervals as the radius parameter is varied. (b) Shows the number of iterations of the L list building loop in an optimized implementation of Apostolico *et al*'s algorithm (see Figure 5.6) on the same data.

It is noteworthy that when our optimizations are applied to Apostolico *et al*'s [1992; 1993] algorithm it actually performs slightly better than a simple implementation of Valiente's algorithm on inputs of up to about 45000 intervals. This small gain however comes at the price of the much larger memory usage of Apostolico *et al*'s algorithm. Note that Supowit's [1987] simple $\Theta(n^2)$ time and space algorithm is the most space hungry — allocating all available memory on circle graphs with more than 10,000 vertices, and so its timing measurements are only visible in the bottom left of Figure 5.19(a).

The most efficient of the algorithms in both time and space is the optimized version of Valiente's algorithm, which dramatically out-performs the other implementations.

5.7.2 Type-II Results

Figure 5.19(b) shows a comparison of the run-times of the algorithms, on randomly generated Type-II interval representations having 20,000 intervals. As the radius (i.e. density and total interval length) is increased, the run-time of the unoptimized implementations increases. There is a very slight decrease in the run-time of the unoptimized implementation as the radius grows beyond about 1, because of the reducing number of entries required in the C lists. In the case of the optimized implementations however, the run-times increase with the radius, and then begin to rapidly decrease. The run-time of Supowit's [1987] algorithm, whose time complexity is not dependent on the density of the interval representation, does not vary as the radius parameter is adjusted. Although this algorithm is more efficient than a direct implementation of Apostolico *et al.*'s algorithm, it is much less efficient than the optimized implementations.

Figure 5.20(a) shows the inner-loop iteration counts of a basic implementation of Valiente's [2003] algorithm as the radius is increased, while Figure 5.20(b) shows the inner-loop iteration counts of our optimized implementation of Valiente's algorithm. The optimized algorithm's inner-loop always iterates fewer times, but the gains become most pronounced as the radius gets large. Similarly, Figure 5.21(a) shows the iteration counts of the L list building loop in a direct implementation of Apostolico *et al*'s algorithm [1992; 1993], and Figure 5.21(b) shows the iteration counts of the L list building loop in our optimized implementation of Apostolico *et al*'s algorithm. As with our implementation of Valiente's algorithm, the loop always iterates far fewer times in the optimized implementation, but the improvement is most significant as the radius becomes large. The memory consumption of Apostolico *et al*'s algorithm is dominated by these L lists, and so the difference in memory consumption of the unoptimized versus the optimized implementations progresses in a similar way to the

difference between the iteration counts shown.

We do not show the inner-loop iteration counts for Supowit's [1987] algorithm, because they are independent of the radius parameter (i.e. the density of the circle graph's interval representation).

5.8 Register Allocation Results

Circle graphs arise naturally in the context of register allocation during the compilation of software pipelined loops using the so-called *meeting graph* [Eisenbeis et al. 1995; de Werra et al. 1999, 2002]. In this section we provide experimental results for the algorithms described above operating on circle graphs arising during register allocation.

The circle graphs are derived from loops in the standard SPEC Integer and Floating point benchmarks, Livermore loops and Linpack and Whetstone benchmarks. We evaluated the algorithms using approximately 2,000 such circle graphs in total. The number of vertices, n, of these graphs ranges from only 4 vertices to approximately 12,000. The circle graphs arising in this application are on average quite dense, with average density about $0.6n$. The minimum density of any graph is $0.4n$, while the maximum density is n.

Figure 5.22(a) shows the performance of the algorithms on these circle graphs. As observed in Section 5.7, Supowit's very simple algorithm out-performs a direct implementation of the more complex algorithm of Apostolico *et al.* However, the competetiveness of our improved version of Apostolico *et al*'s algorithm is notable, performing better than a direct implementation of Valiente's algorithm. Note that each data-point in Figure 5.22(a) corresponds to the register allocation of a single loop in a program being compiled, and so very fast processing is essential. The best performance on these circle graphs is achieved by our improved version of Valiente's algorithm.

The great majority of circle graphs arising in our tests are reasonably small, having less than 2,000 vertices. Figure 5.22(b) shows that the relative performance of the algorithms holds also for these smaller graphs. The two most practical algorithms are our improved versions of Apostolico *et al*'s algorithm and Valiente's algorithm. It is again notable that the observations of Section 5.5.2 lead to an implementation of Apostolico *et al*'s algorithm that is more efficient than a direct implementation of Valiente's algorithm. This is despite the apparently substantial complication of Apostolico *et al*'s algorithm.

Finally, comparing Figure 5.23(a) and Figure 5.23(b) shows, as expected, the much

more efficient use of space made by Valiente's algorithm compared to the other algorithms.

5.9 Faster Algorithms

Our optimized version of Valiente's algorithm described above is appealing because of its combination of simplicity and efficiency. In this section, we describe output sensitive algorithms for the maximum independent set problem in a circle graph. That is, their time complexities are bounded in terms of the independence number (i.e. the cardinality of the maximum independent sets) of the circle graph.

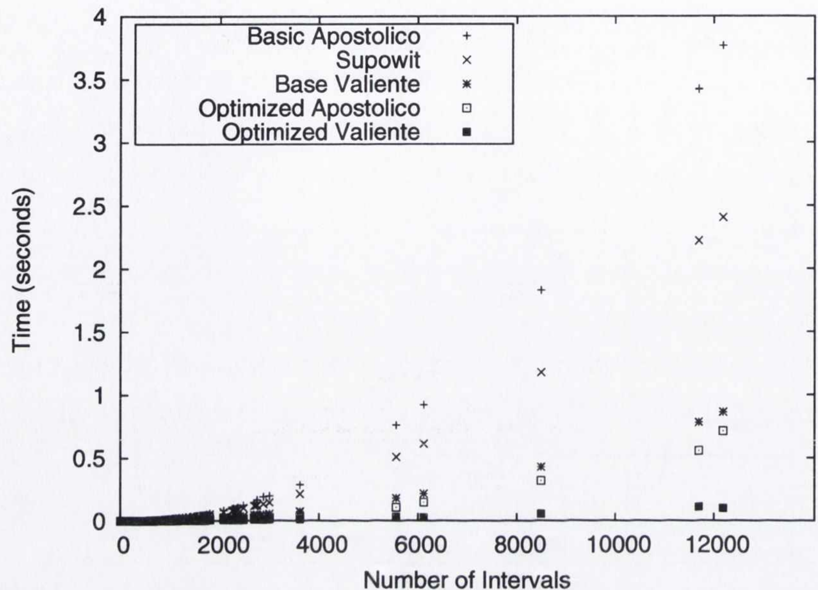
The development of an output sensitive algorithm is motivated by the experimental results presented in Figure 5.24, in both (a) and (b), it can easily be seen that the independence number of the circle graphs is substantially smaller than their density. These experiments indicate that an output sensitive approach may be fruitful in designing an efficient maximum independent set algorithm for circle graphs.

We begin by describing a simple output sensitive algorithm requiring $\Theta(n\alpha)$ time, for an n vertex unweighted circle graph with independence number α. As we note via an example in Section 5.9.1, there are simple low density circle graphs with high independence number, making this simple output sensitive algorithm inferior to the $\Theta(dn)$ time algorithms described above for certain circle graphs. We then show how this can be modified to achieve an algorithm operating in time $\Theta(n \min\{d, \alpha \log n\})$. We experimentally evaluate both of these algorithms showing that on both the random circle graphs this latter algorithm performs better than all the algorithms described above by a substantial factor – between 5 and 7 in our experiments.

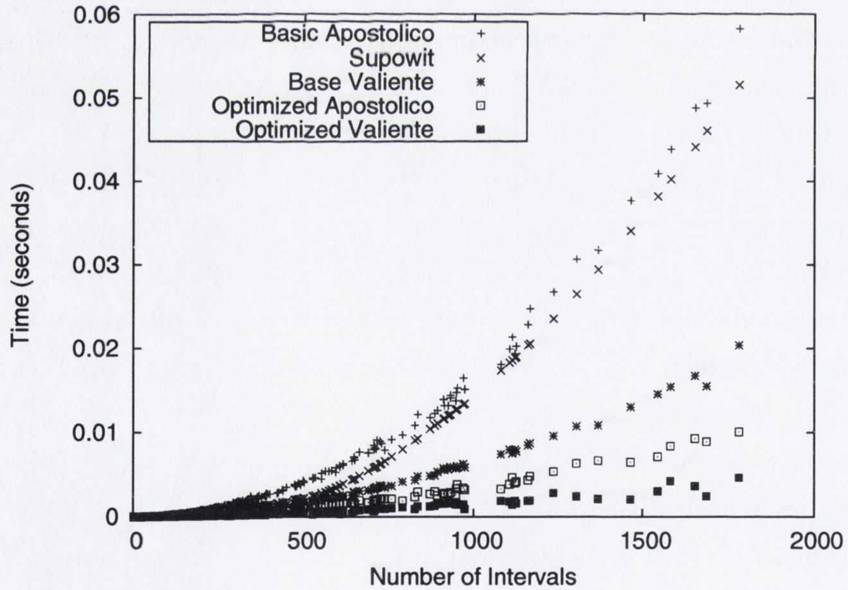
Finally we finish by sketching how an improved algorithm can be constructed, which we have we leave an experimental evaluation of to future work. This algorithm operates in time $\Theta(n \min\{d, \alpha\})$. Note that, in work completed shortly after the submission of this dissertation [Nash and Gregg 2010], an alternative approach to achieving an algorithm with the aforementioned time complexity has been noted. That article [Nash and Gregg 2010] also includes a more formal treatment of the output sensitivity.

5.9.1 Introducing Ouput Sensitivity

Output sensitivity can be introduced to Valiente's algorithm (see Section 5.6 and Figure 5.13), by tracking *changes* to the values of the cells of M. Note that Recurrence 5.2 is repeatedly evaluated within M. When a cell $M[j]$ changes (i.e. increases in value),

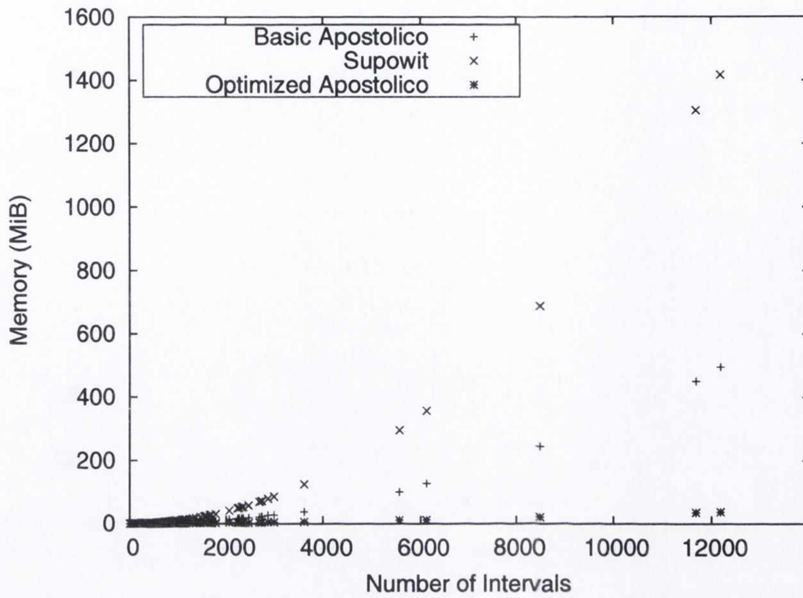


(a)

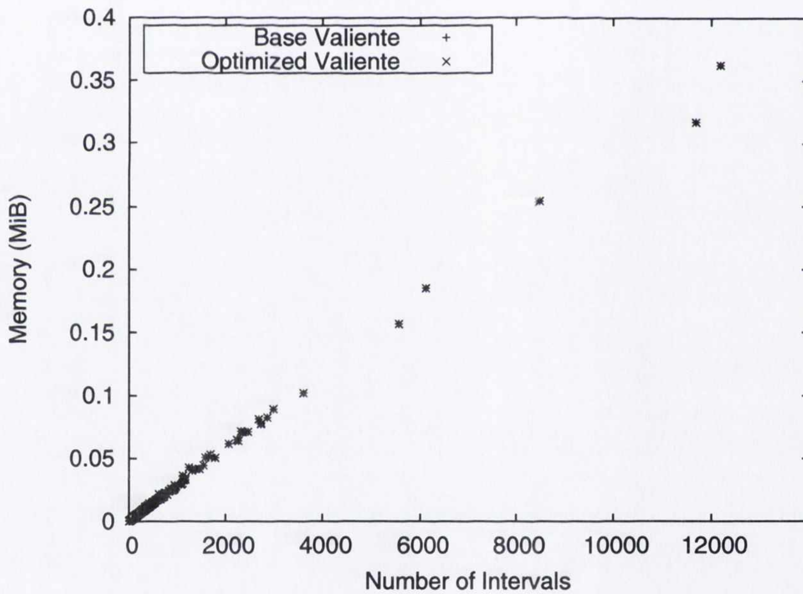


(b)

Figure 5.22: (a) Shows the times taken by the algorithms on circle graphs arising from performing register allocation on software pipelined loops. (b) Provides a closer look at the times taken on the smaller of these circle graphs, in particular those with at most 2,000 vertices.



(a)



(b)

Figure 5.23: This figure shows the space used by the algorithms when processing circle graphs arising from performing register allocation of software pipelined loops. (a) Shows the peak memory occupied by Supowit's algorithm, which requires $\Theta(n^2)$ space, as well as Apostolico *et al*'s algorithm and our improved version of it, which both require $\Theta(dn)$ space. (b) Shows the peak space required by Valiente's algorithm and our improved version. The space requirements are very close to identical, with the improved version requiring very slightly less space in practice, although this is not visible in the figure. The space usage is very modest compared to that of Supowit and Apostolico *et al*'s algorithms, shown in (a).

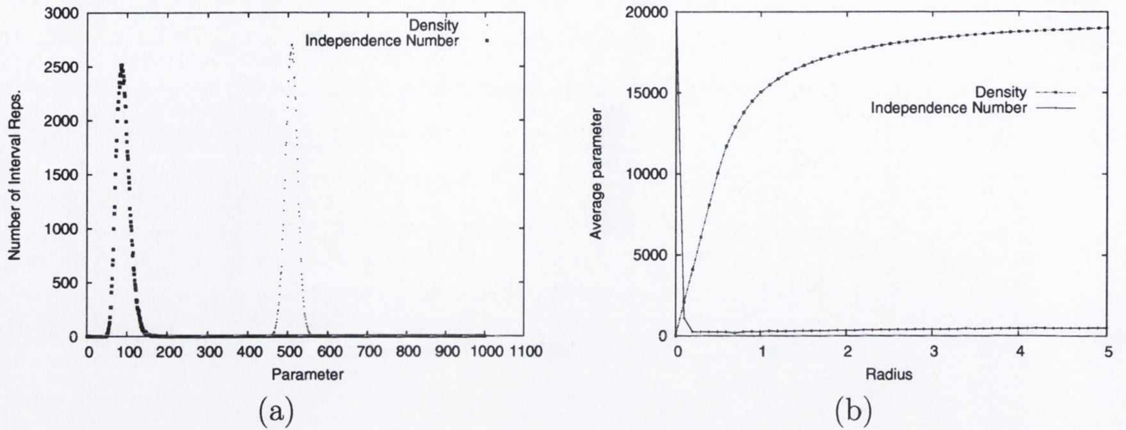


Figure 5.24: For (a) We generated 10^5 random Type-I interval representation of 1000 intervals and recorded their density and independence number. This figure shows that the vast majority of interval representations have density close to $n/2$, while the independence number is substantially smaller. In (b) we generated random Type-II interval representations of 20000 intervals while varying the radius parameter linearly between 0 and 5 in 50 steps, we did this 20 times and averaged the results.

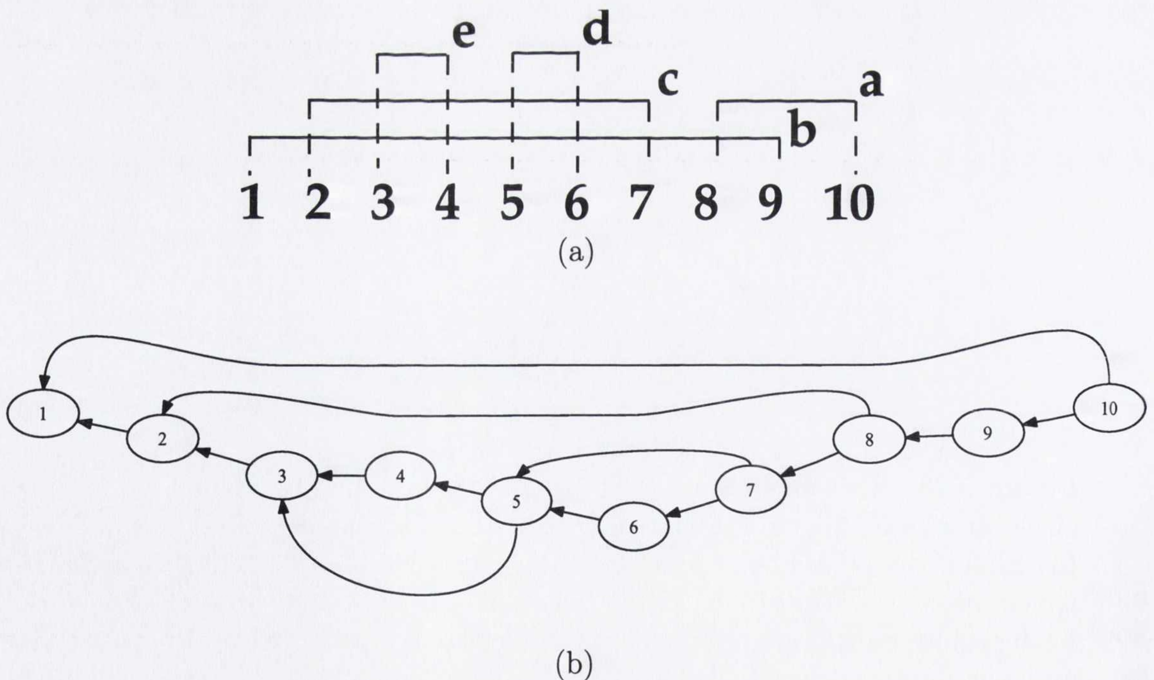


Figure 5.25: (a) Shows an interval representation of a circle graph while (b) shows the flow graph associated to this interval representation.

then, according to Recurrence 5.2, at most two other cells can also increase in value as a result. Namely, $M[j-1]$, and if there is an interval $[i, j-1]$, then $M[i]$ can also increase. If these values do indeed increase, they will result in the need for further cells to be processed, again determined by Recurrence 5.2.

Figure 5.25 shows an interval representation in (a), and in (b), a directed graph associated to that interval representation. We refer to this directed graph as the *flow graph* of the interval representation. Each end-point in the interval representation in (a) has a corresponding node in (b). There is an edge from node number x to node number y if a change in the value of $M[x]$ can cause a change in the value of $M[y]$, as defined by Recurrence 5.2.

We now outline how a maximum independent set of a circle graph can be computed using this flow graph. Note that the flow graph itself need not be constructed, since it is implicit in the interval representation.

We use the same notation as in Figure 5.13. For an n interval σ representation of a circle graph, the algorithm proceeds as a left-to-right scan with a counter m beginning at 1 and counting towards $2n$. A linked list W is maintained, which contains pairs (t, v). The presence of (t, v) in W indicates that if the current value of $M[t]$ is less than v, then $M[t]$ should be updated to v.

When the left-to-right scan reaches position m and m is the right end-point of an interval, then the pair $(last, M[r+1] + CMIS[i])$ is added to W. Here $last$, is the left end-point of the interval $i = [last, r]$ whose right end-point is largest that is to the left of m. As in Figure 5.13, $CMIS[i]$ is the weight of a maximum independent set contained in i.

Following this possible addition to W, a loop begins that terminates only when W is empty. In each iteration of the loop an update (t, v) is removed from W and applied to M as described above. If the value of $M[t]$ changes, it add pairs to W for all out-going edges in the associated flow graph at node number t, provided there is at least one open interval at the destination node of those edges (an interval is open if its left end-point is to the left of m, and its right end-point is to the right of m.). Determining whether there is an open interval containing any point can be determined easily in constant time. If the value of $M[t]$ does not change, no new pair is added to W.

When this process terminates, if m is the right end-point of an interval $j = [a, m]$ $CMIS[j]$ is assigned the value $w_j + M[a+1]$. The left-to-right scan then advances by increasing m by one.

The appeal of this algorithm is that it attempts to update M the minimal amount

required. Indeed, for an unweighted circle graph each of the n cells in M can be updated at most α times, and so the algorithm requires at worst $\Theta(n\alpha)$ time. Such an algorithm is of course desirable in a situation when $\alpha = o(d)$.

However, there are simple low-density graphs with $\alpha = n$, for which the algorithm requires $\Theta(n^2)$ – inferior to the $\Theta(dn)$ algorithms described in Sections 5.5 and 5.6. This quadratic time behaviour can be seen when the circle graph has the following interval representation: a single interval $[1, 2n]$, together with the intervals $[2a, 2a+1]$ for $1 \leq a < n$. In this case, it is easily seen that at each right end-point $2a+1$ of one of the unit-length intervals, the inner-loop of the algorithm (i.e. the loop that terminates when W is empty) will in fact update every value of M from $a-1$ down to 1, clearly causing quadratic time performance.

Quadratic time performance in the worst case can be avoided by ensuring all updates for each interval are processed in strictly right-to-left order. This involves maintaining order in the list W, and results in an algorithm requiring $\Theta(n \log n \min\{d, \alpha\})$ time[†], which is still asymptotically worse than the $\Theta(dn)$ algorithms we saw above when α dominates d.

As we shall see in the next section, a more efficient output sensitive algorithm can be obtained by dividing M into *blocks* of contiguous cells, and tracking changes in-order only between blocks. For sufficiently large blocks this approach will be seen to provide an algorithm operating in time $\Theta(n \min\{d, \alpha \log n\})$.

5.9.2 A Combined Algorithm

In this section we describe an algorithm that avoids the quadratic time behaviour just described, giving an algorithm operating in time $\Theta(n \min\{d, \alpha \log n\})$.

Referring again to Figure 5.13 we regard M as being partitioned into $\lceil (2n+1)/B \rceil$ *blocks* of size B, where n is the number of vertices in the circle graph. The ith $1 \leq i \leq \lceil (2n+1)/B \rceil$ block is a contiguous region of M comprised of the B cells at indices $1 + (i-1)B, \ldots, iB$.

The output sensitive algorithm for computing a maximum independent set of an n vertex circle graph proceeds as a left-to-right scan over the array M. The algorithm also maintains a priority queue X of blocks ordered increasingly by left end-point.

If at position m there is an interval $i = [a, m]$, then a loop iterates until X is empty or the highest priority block in X is disjoint from the interval $i = [a, m]$ being processed.

[†]Of course, a log-logarithmic factor could also be achieved with the data structure of van Emde Boas [1977].

On each iteration, the highest priority block is removed from X. Let the left and right end-points of this block be u and $u+B-1$. Recurrence 5.2 is directly evaluated in the array M for this block, with a counter q counting down from $\min\{u+B-1, m-1\}$ to $\max\{a+1, u\}$. We refer to this as *processing* a block. During this processing, if the value of any entry $M[q]$ increases, and there is an interval $j = [t, q-1]$ with $t < u$ and $M[t] < M[u] + CMIS[j]$, then the block containing t is added to X provided there is at least one open interval containing t In addition, after processing the block, if $M[u]$ has increased in value and $M[u-1] < M[u]$, then block containing $u-1$ is added to X provided there is at least one open interval at $u-1$.

Assuming this iteration over X occured, $CMIS[i]$ is assigned the value $w_i + M[a+1]$, and if $M[a] < M[m+1] + CMIS[i]$ the block containing a is added to X. Then, unless $m = 2n$, the left-to-right scan advances, increasing m by 1.

When this algorithm terminates, the value $CMIS[i]$ will have been computed for each interval i. To compute the weight of a maximum independent set of the entire circle graph, a single invocation of the interval processing described for Valiente's algorithm in Section 5.6 is invoked with end-points $[1, 2n]$. This completes the description of our output sensitive algorithm.

In the worst case, for each interval $i = [l, r]$, each of the $\lceil (r-l)/B \rceil$ blocks of that interval will require evaluation and be processed in $\Theta(B)$ time. Thus, in total for all intervals, $\Theta(\ell)$ time is spent processing blocks. Recall from Section 5.2 that in terms of the density parameter d this is $\Theta(dn)$ time.

Note that when the algorithm terminates, X is empty. This is because the processing of an interval is guaranteed to remove all blocks from X that the interval intersects, moreover, as was stipulated in the condition for adding a block b to X, there must be at least one open interval intersecting b, implying that any block b added to X will also be removed from X. As a result, the total number of additions to X is equal to the number of removals from X.

For each removal from X the algorithm spends $\Theta(B)$ time processing the removed block, thus, the algorithm can perform at most $\Theta(dn/B)$ removals from X because as we just noted at most $\Theta(dn)$ time can be spent processing blocks at worst. Thus, assuming X is implemented as a heap that can contain $\Theta(n/B)$ blocks, $\Theta(dn/B \log(n/B))$ time is spent in total performing removals from X. It follows that choosing $B = \Omega(\log n)$ implies at most $\Theta(dn)$ time is spent performing additions to and removals from X.

The time complexity of this algorithm can also be characterised in terms of the independence number, α, if the circle graph is unweighted. Note that, in processing

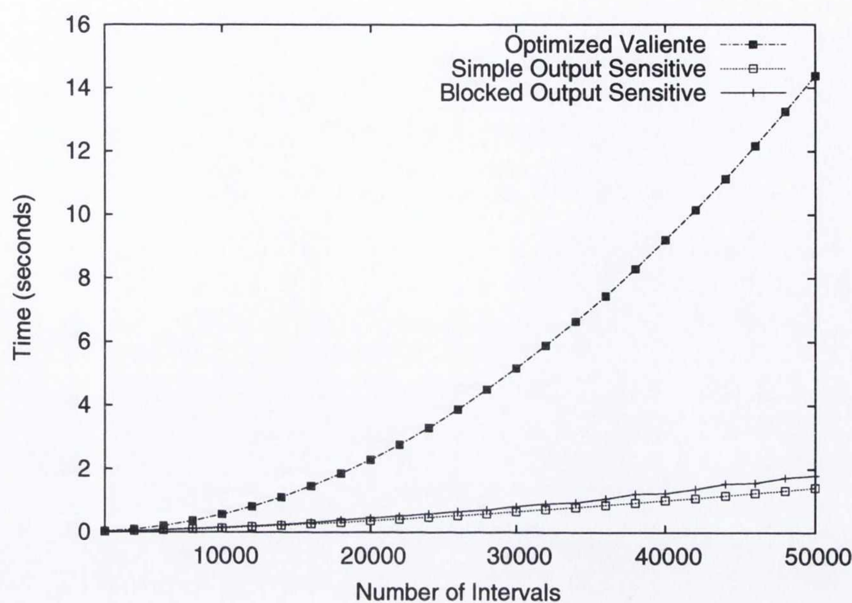
any block, at least one value in the table M contained in that block increases. Thus, since no value in M can increase more than α times, we spend in total $\Theta(B^2\alpha)$ time processing the $\Theta(B\alpha)$ updates to a particular block. Since there are $\Theta(n/B)$ blocks in total, $\Theta(n\alpha B)$ time in total is spent processing blocks. Recall from above that to guarantee $\Theta(dn)$ worst-case time we must have $B = \Omega(\log n)$, and thus, choosing $B = \Theta(\log n)$ at most $\Theta(n\alpha \log n)$ time will be spent processing blocks. In addition, each block can be added to X in total $\Theta(B\alpha)$ times and so in total for the $\Theta(n/B)$ blocks, at most $\Theta(n\alpha \log(n/B))$ time is spent adding and removing blocks from X. Thus the algorithm runs in time $\Theta(n \min\{d, \alpha \log n\})$ in the worst case. As we shall see in the following section, this algorithm also performs well in practice.

5.9.3 Experimental Comparison

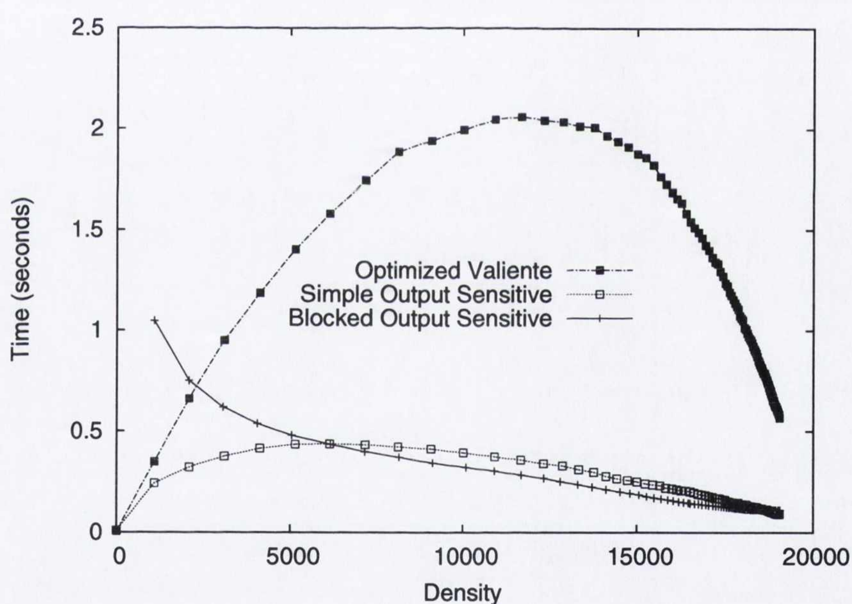
Random Graphs

Figure 5.26(a) shows the performance of the maximum independent set algorithms on random Type-I interval representations. Clearly the two output sensitive algorithms outperform the efficient implementation of Valiente's algorithm described in Section 5.6.1. We have measured experimentally that the these graphs have independence number about $2.47\sqrt{n}$ on average, as Figure 5.28(a) shows. Moreover, as we noted in Figure 5.2, they have density about $n/2$ on average. In this situation, the efficient implementation of Valiente's algorithm is effectively operating in $\Theta(n^2)$ time. While the simple output sensitive algorithm is operating in time $\Theta(n^{1.5})$ and the blocked output sensitive algorithm is operating in time $\Theta(n^{1.5} \log n)$. This explains the relative performance of the algorithms for these random circle graphs, in particular why the simple output sensitive algorithm performs better than the blocked output sensitive algorithm for these random circle graphs.

Figure 5.26(b) shows the performance of the maximum independent set algorithms on random Type-II interval representations of 20,000 intervals. Here, as the radius parameter is varied the density of the interval representation increases. We have excluded the first data point for the simple output sensitive algorithm from this plot, since it requires approximately 6 seconds to complete and makes the plot difficult to read. Here it is notable that the simple output sensitive algorithm performs worst until the interval representations become dense enough. Figure 5.28(b) shows how the independence number of these graphs varies (somewhat unusually) with their density, excluding the first data point where the density is 1 and the independence number is 20,000 making the plot difficult to read. The improvement to Valiente's algorithm in-

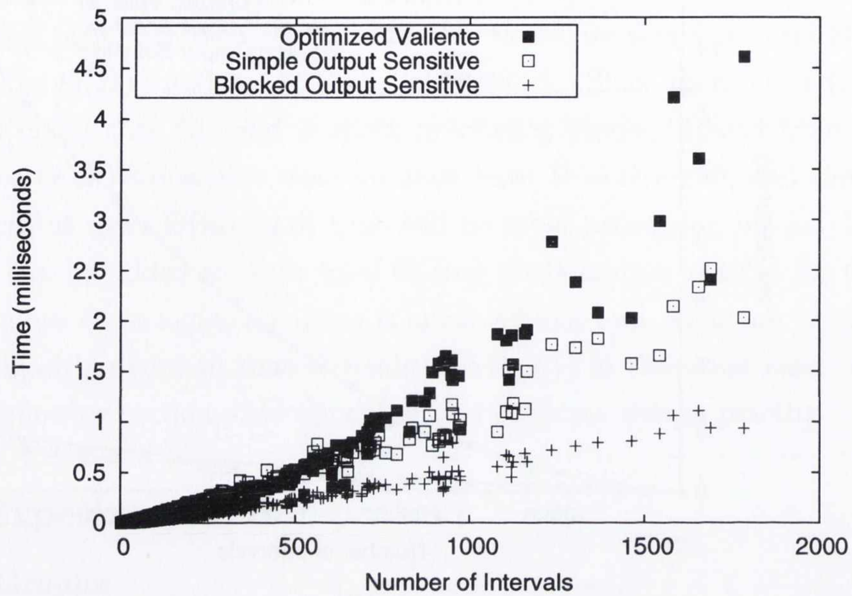


(a)

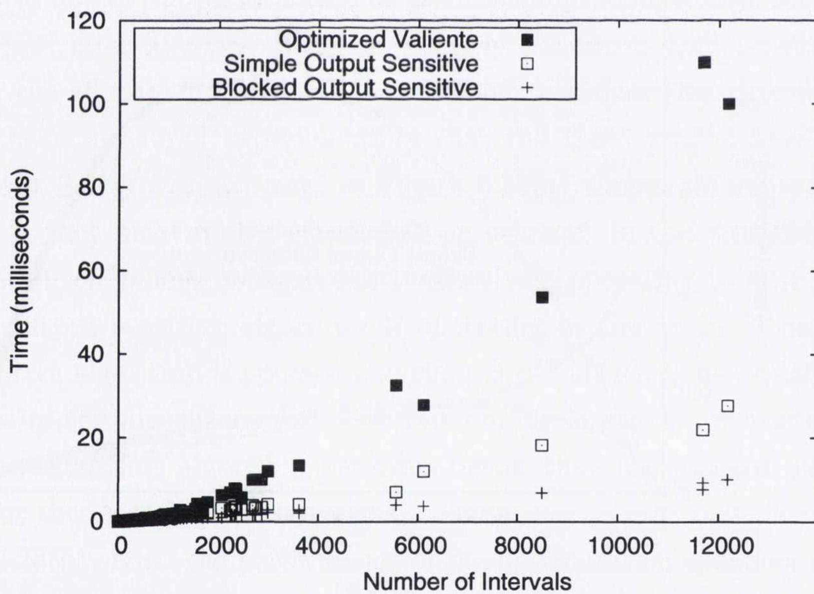


(b)

Figure 5.26: This figure shows the performance of the maximum independent set algorithms on randomly generated interval representations of circle graphs as described in Section 5.3. In (a) the performance of the algorithms is shown on what we refer to as Type-I interval representations as the number of intervals is increased. While (b) shows the performance of the algorithms for Type-II interval representations of 20,000 vertices as the radius parameter (and consequently the density) is varied. The generation of these Type-I and Type-II interval representations is described in Section 5.3. In both plots, "Optimized Valiente" is the variation of Valiente's algorithm described in Section 5.6.1. "Simple Output Sensitive" is the $\Theta(n\alpha)$ time algorithm described in Section 5.9.1, and "Blocked Output Sensitive" is the $\Theta(n \min\{d, \alpha \log n\})$ time algorithm described in Section 5.9.2. These results are discussed in Section 5.9.3.



(a)



(b)

Figure 5.27: This figure shows the performance of the maximum independent set algorithms on the interval representations of circle graphs that arise when performing the register allocation of software pipelined loops, as described in Section 5.3. In (a), all interval representations are shown while in (b) we show only interval representations having at most 2,000 intervals. In both plots, "Optimized Valiente" is the variation of Valiente's algorithm described in Section 5.6.1. "Simple Output Sensitive" is the $\Theta(n\alpha)$ time algorithm described in Section 5.9.1, and "Blocked Output Sensitive" is the $\Theta(n \min\{d, \alpha \log n\})$ time algorithm described in Section 5.9.2. These results are discussed in Section 5.9.3.

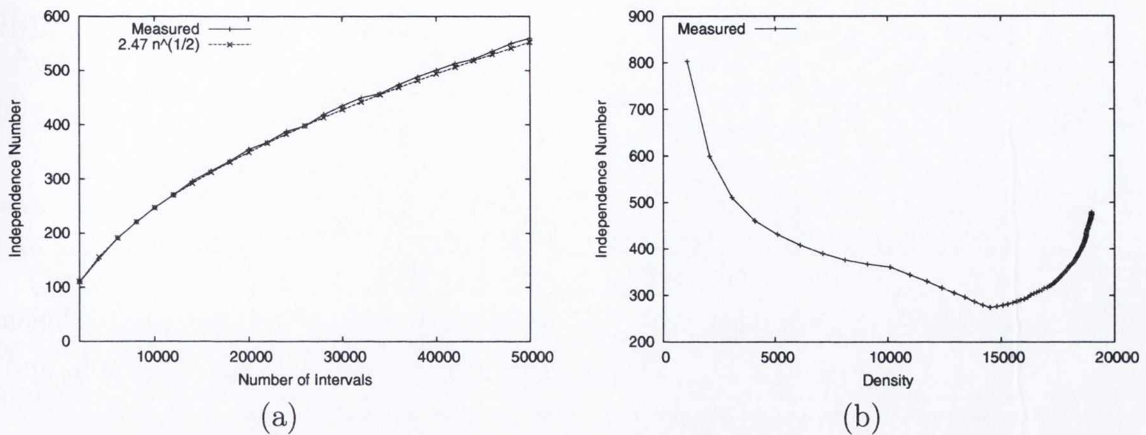


Figure 5.28: This figure shows, in (a), how the average independence number of the Type-I interval representations varies as the number of intervals, n, increases. We also show the function $2.47\sqrt{n}$, which closely matches the measured independence number. In (b) the independence number of Type-II interval representations of 20,000 intervals varies as the radius parameter (and consequently their density, see Figure 5.2) is varied.

volving the introduction of the variable *last* (see Figure 5.13 and Section 5.6.1) causes the efficient implementation of this algorithm to operate more rapidly for dense Type-II interval representations. The simple output sensitive algorithm also incorporates this improvement, and so despite the increase in independence number visible to the right in Figure 5.28(b), its performance nevertheless improves. The blocked output sensitive algorithm seems the most desirable on these Type-II interval representations, it performs almost as well as the simple output sensitive algorithm for the majority of the interval representations, and enormously better for the low density interval representations with high independence number (for example, as mentioned above, the first data point that we have excluded in order to make the plot readable).

Register Allocation Graphs

Figure 5.27(a) shows the performance of the maximum independent set algorithms on the interval representations of circle graphs that arise while performing register allocation, as described in Section 5.3. Here it is again clear that output sensitive algorithms offer substantially improved performance – up to a factor of 10 for the largest graphs, with the blocked output sensitive algorithm performing best. Figure 5.27(b) shows the same data as Figure 5.27(a) but shows the results only for interval representations of up to 2,000 vertices. Here it is clear that the output sensitive algorithms are also much more efficient, with the blocked output sensitive algorithm performing up to 5 times better than the efficient implementation of Valiente's algorithm. These results

demonstrate that on naturally occuring circle graphs the output sensitive algorithms also substantially better than existing algorithms.

5.10 Conclusion and Future Work

This chapter has provided an experimental evaluation of the two most efficient algorithms for computing a maximum weight independent set of a circle graph, and shown how to construct highly efficient implementations of these algorithms.

We have described optimizations for the algorithm of Apostolico *et al* [1992; 1993] that dramatically improve its running time and decrease its memory consumption. For example, on randomly generated Type-I interval representations, we observed on average a halving of both memory consumption and running time, although the gains are even greater when the density of the interval representations is high for Type-II interval representations. In addition we have pointed out and described how to correct an error in this algorithm.

Although Apostolico *et al*'s algorithm has worse memory consumption than Valiente's [2003] algorithm, both operate in $\Theta(dn)$ time. Therefore it was important to compare the running times of both algorithms in practice. When implemented efficiently the running time of Apostolico *et al*'s algorithm is lower than that of Valiente's algorithm for moderate to large numbers of intervals while operating on Type-I interval representations. In addition, when randomly generating Type-II interval representations of 2,000 to 50,000 intervals, at a range of radii we observed that our optimized implementation of Apostolico *et al*'s algorithm significantly out-performs our basic implementation of Valiente's algorithm (compare Figure 5.9(b) and Figure 5.16(a)).

We also described how to adjust Valiente's algorithm so that it uses a single left-to-right scan with many local right-to-left scans similar to that of Apostolico *et al*'s algorithm. Doing this allows an unnecessary sorting step to be removed from the algorithm. This change to Valiente's algorithm also enabled us to remove unnecessary recomputations from the algorithm and describe an implementation that is more than 3 times faster on average than a direct implementation of the algorithm, while operating on randomly generated Type-I interval representations. Like our optimizations to Apostolico *et al*'s algorithm, the optimizations we describe are even more effective at high densities.

In addition to these results on random circle graphs, we have also demonstrated the practicality of our improvements on circle graphs that arise in practice. We have

provided results for circle graphs that arise when performing register allocation of software pipelined loops. In this application area, efficiency is crucial since many thousands of loops may need to be compiled.

In addition, after this detailed experimental evaluation of previous approaches, we have compared the best resulting variant with our output sensitive algorithm, showing that our algorithm is the also the most efficient in practice.

Recall that our output sensitive algorithm operates in time $\Theta(n \min\{d, \alpha \log n\})$ for an unweighted circle graph. An immediate avenue for future work is to extend the analysis of our algorithm to the case of weighted circle graphs, or at least to examine its performance on weighted circle graphs. At present, we do not have naturally occuring data sets that include weighted circle graphs, although we could include random weights in the generation of our Type-I and Type-II random interval models.

Of course, with the use of a more efficient priority queue data structure [van Emde Boas 1977] the time complexity of the output sensitive algorithm can be improved to $\Theta(n \min\{d, \alpha \log\log n\})$, although it is unclear if the resulting algorithm would be practical. We discuss this issue in more detail in the following chapter (see Section 6.3).

We note that a there is also an algorithm operating in $\Theta(n \min\{d, \alpha\})$ time. We first sketch an algorithm operating in time $\Theta(n \min\{d, \alpha\} + n \log(n/d))$ and then show how its time can be improved to the one above. This algorithm is a combination of the algorithms described in Sections 5.9.1 and 5.9.2, where the block size, B is chosen as $\Theta(d)$. Within each block, Recurrence 5.2 is only evaluated directly, in $\Theta(d)$ time, if processing the block in the simple output sensitive manner described in Section 5.9.1 causes more than $\Theta(d)$ updates to the array M. It can then be seen that there can be at most $\Theta(n)$ additions and removals from the priority queue, which can contain at most $\Theta(n/d)$ entries, this results in the $\Theta(n \log(n/d))$ term in the time complexity above. Moreover, we are guaranteed that at most $\Theta(d)$ time is spent processing each of the $\Theta(n)$ blocks, since as mentioned above we explicitly switch to direct evaluation of Recurrence 5.2 (requiring $\Theta(d)$ time) when the simple output sensitive updation described in Section 5.9.1 performs more than $\Theta(d)$ updates. This gives a $\Theta(dn)$ bound on the algorithm's time. Finally, we can be sure the simple output sensitive evaluation performs no more than $\Theta(n\alpha)$ updates. The $\Theta(n \min\{d, \alpha\} + n \log(n/d))$ time bound above on this algorithm then follows.

As a final improvement, it is straightforward to show that $\alpha \geq \lceil n/d \rceil$ (since there must be at least this many *disjoint* intervals). As a result we see immediately that the time above can be written $\Theta(n \min\{d + \log(n/d), \alpha)$. Moreover, for $d = O(\log n)$

we can simply use Valiente's algorithm without affecting the output sensitivity of the algorithm, while for larger d, we use the $\Theta(d)$ sized block output sensitive algorithm just described, giving time $\Theta(n \min\{d, \alpha\})$, since d dominates the logarithmic term. We leave the full description and experimental evaluation this somewhat complicated algorithm to future work.

Another interesting item for future work is the measurement of the independence number of circle graphs that occur in applications, and a comparison of this with random graph models. Knowledge of the independence number allows simple comparisons with $\Theta(dn)$ time algorithms presented in this chapter. For example, for a randomly generated Type-I interval representation, the number of disjoint intervals converges to $2\sqrt{n/\pi}$ in probability [Boucheron and de la Vega 2001]. It seems that such results could be extended to independence number of circle graphs and then compared to graphs that occur in practice. As we noted in Figure 5.28 random Type-I unweighted circle graphs, appear to have independence number on average approximately $2.47\sqrt{n}$. We have not measured the independence number of circle graphs that occur in applications such as register allocation and RNA analysis, and further investigation of both these random and application derived circle graphs is the subject of future work.

Chapter 6

Final Thoughts

This dissertation has presented experimental results for, and improvements to some important algorithms and data structures. This chapter briefly presents some concluding reflections on these results, drawing conclusions both about the results in this dissertation and the experimental study of algorithms and data structures in computer science in general. Finally, we describe general directions for future work.

6.1 Why Experiment?

Algorithms and data structures are indisputably the building blocks of Computer Science. It is our opinion that a fruitful point of view is to study algorithms and data structures in an experimental manner reminiscent of the way in which natural phenomenona are studied in the natural sciences. In this section, we outline the importance of experimentation in developing the contributions of this dissertation.

Chapter 3 of this dissertation studied the interaction between a ubiquitous feature of almost every modern computer – instruction pipelines – and a sorting algorithms, an extremely common computational task. These experiments naturally lead to a deepened *understanding* of sorting algorithms. We believe understanding all aspects of the basic components of our science is of paramount importance. This dissertation provided the first full account of what is likely to cause variations of bubble sort such as shaker sort to perform so badly in practice: their surprisingly bad branch prediction characteristics, something previously over-looked in explaining their poor performance in practice. This dissertation also, for the first time, pointed out and analysed another factor contributing to quicksort's tight inner-loop compared to mergesort or heapsort: it has better branch prediction properties. We emphasize again that we believe it is very important to understand these basic algorithms of Computer Science.

Chapter 4 of this dissertation illustrates another way in which experimentation can be fruitful. This time in the context of data structures. This chapter provides a demonstration of an important outcome of experimentation with algorithms and data structures, often referred to as *algorithm engineering*. By adapting a simple data structure from the string sorting literature – the burst trie – we were able to engineer a general purpose integer data structure that we believe should be of use in many applications. By experimenting extensively we were able to understand the behaviour of many data structures in practice. We noted that cache performance is very often dominant; where it is not, branch mispredictions can be. We note that this latter observation provides further motivation for making use of the rich (and in our view, largely neglected in terms of implementation) theoretical literature on data structures that do *not* use only comparison instructions to order data.

Chapter 5 of this dissertation again illustrates algorithm engineering. It also illustrates what we view as another important aspect of experimentation with algorithms: *practice can lead to good theory*. By noting experimentally that two random circle graph models lead to circle graphs with small independence number, we developed output sensitive algorithms for this problem. We then validated their theoretical performance both on the random circle graph models *and* on naturally occuring circle graphs.

6.2 Future Directions

In this section we briefly outline future directions for research arising from the work conducted for this dissertation. Although the opportunities for future work were noted throughout this dissertation, we pause here to point out more general directions.

The work in Chapter 3 of this dissertation, concerning sorting and branch prediction, was focused on the average case performance, and used only uniform random data to evaluate the branch prediction properties of the algorithms. An interesting experimental study could examine the behaviour of the algorithms with more realistic data sets. Unfortunately, no such standard data sets are available, but such a study could also attempt to devise such data sets. Failing the definition of new data sets, at least different input distributions (and not just uniform random) could be used. This is potentially quite a large undertaking, a lot of experimental data may result, and the analysis may not be simple.

An immediate direction for future work arising from the work on integer data structures in Chapter 4 is the engineering of a theoretically efficient word-parallel data structure. The performance of the resulting data structure could then be compared

against the engineered variants of the Q-trie and burst trie introduced in Chapter 4. Although the burst trie variant developed in this chapter is efficient in both time and space in practice, it does not take advantage of many of the techniques of word-level parallelism. As we noted in that chapter, a suitable candidate for implementation of a theoretical data structure might be Andersson's [1995] simple, multiplication free integer search tree, providing all operations in $O(\sqrt{\log n})$ time. If an implementation of this, or another word-parallel data structure proved successful (and we emphasize that such implementations may not achieve the exact time bounds offered by the theoretical version, but simply operate more efficiently than comparison-based data structures), a more general programme for research could then attempt to develop a library of word-parallel data structures with well-known interfaces, for example, similar to the *C++* STL where possible.

Finally, Chapter 5 also offers immediate directions for future work. Finding the maximum independent set of a circle graph arises in a number of areas in addition to compiler optimization, such as VLSI design, RNA analysis and computational geometry. An important piece of experimental work could examine the use of the algorithm in these areas, gather data sets, and assess the relative performance of the algorithms presented in Chapter 5. In particular, the importance of output sensitivity in these application areas could be assessed. As noted in Chapter 5, a simple, immediate piece of future work is the complete description, implementation and experimental evaluation of the $\Theta(n \min\{d, \alpha\})$ time algorithm for the maximum independent set of an unweighted circle graph.

6.3 Theory and Practice

Here we pause for a moment to make a general point about the relationship between theoretical algorithms and data structures, and their practical realizations. The goal of this section is certainly not to suggest that the theoretical development of algorithms should carefully avoid constructions that may not be practical – the ideas involved in the development of a new algorithm or data structure are often simply important in themself. Rather, our goal in this section is to identify a class of gap that we believe is important to address, and that it may be possible to narrow or close. We are referring to development of efficient implementations of integer data structures mentioned in the previous section.

Recall that in the development of the output sensitive algorithms in Chapter 5 we assumed a priority queue with logarithmic time operations was available. This

lead to the time complexities of our blocked output sensitive algorithms (see Section 5.9.2) having logarithmic factors. As was briefly mentioned in Chapter 4, assuming the availability of a stratified (van Emde Boas) tree [1977], these times can be improved to having log-logarithmic factors. For example, the resulting time complexity for our block output sensitive algorithm is then $\Theta(n \min\{d, \alpha \log\log n\})$. In Chapter 3, we expressed the time complexity of a stratified tree in terms of the word-length, w, of the underlying machine as $O(\log w)$, since we were interested in a general purpose data structure for integer keys. These time complexities can also be expressed in terms of the *universe size*, u. In Chapter 3, $u = 2^w$ and so we have the operations on the stratified tree in time $O(\log\log u)$ while requiring $O(u)$ space. Of course, in the case of computing a maximum independent set of a circle graph, $u = n$. However, there is an important *practical* reason that we did *not* quote the log-logarithmic bound.

The point is that, for a *fixed size* bounded universe, a stratified tree provides $O(\log\log u)$ time operations, however, in almost every algorithmic application involving the data structure (such as the maximum independent set algorithm mentioned) an implementation is required that works for a range of *different* sizes of universe. Despite this, we are unaware of any practical implementation of the data structure that provides such flexibility. For instance, Mehlhorn and Näher [1990] do provide such an implementation, but it is far from efficient. Indeed Dementiev *et al.* [2004] show that it is less efficient than even simple comparison based data structures. The situtation is different for a *fixed* universe size, indeed, as Dementiev *et al.* demonstrated, and as our experimental results of Chapter 3 showed, for 32-bit keys (a universe of size 2^{32}), an efficient (in terms of time) stratified tree implementation is possible.

It might be noted that in the case at hand, all our experiments for the maximum indepedent set algorithms of Chapter 4 involved circle graphs of no more than $50,000$ vertices, and so a stratified tree over a universe of size 2^{16} would have sufficed. However, the resulting time complexity should be quoted as $\Theta(n \min\{d, \alpha \log\log u\})$ where u is the universe size of the implementation. Without further modifications, it must then be accepted that the implementation will not function for circle graphs of $n > u$ vertices.

Despite the fact that, as far as we are aware, *no* sufficiently general implementation of a stratified tree exists to make the claim of an algorithm having a $O(\log\log n)$ factor realistic, it is common to see such bounds quoted. For example, Apostolico *et al.* [1992] present a maximum clique algorithm for an m edge n vertex circle graph operating in time $O(\min\{n^2, m \log\log n\})$, via the use of a stratified tree. Such a bound is neither incorrect nor dishonest, but it is definitely not the bound that will currently arise in a practical realization. This is an unfortunate gap between theory

and practice. This gap is one that we hope to address in future work, as mentioned in the previous section. Note also that a *negative* result in this engineering work would also be important. If integer data structures, including stratified trees as well as word-parallel data structures, resisted substantial experimental and engineering effort at their implementation, it could still, in our view, help to drive theory closer to practice, since bounds achieved using such data structures might at least be accompanied by a caveat.

We are not the first to point out such gaps between theory and practice, Moret [2001] provides several examples. For example, He points out that the Fredman and Tarjan's [1987] $O(|E|\beta(|E|,|V|))$ time* minimum spanning tree algorithm is less efficient than Prim's simple $O(|E|\log|V|)$ time algorithm, except for dense graphs with billions of vertices — well beyond the size of those that occur in any application. However, we have not seen the importance of engineering integer data structures highlighted before, we emphasize that practical implementations of such data structures have the potential to improve the practical performance of the many algorithms relying on such data structures.

6.4 Conclusion

Sorting and searching are two fundamental, closely related problems in Computer Science. This dissertation has presented experimental insights and their analysis for classic, comparison-based sorting algorithms. A detailed experimental study of data structures for searching integer keys has been presented, based around the engineering of a data structure, the $LPCB$-trie, for integer keys that is efficient in both time and space. We believe this data structure could fruitfully be used in many applications. Finally, in an additional contribution, we have studied the maximum independent set problem on circle graphs. As well as carefully implementing and evaluating previous techniques, we have presented the first output sensitive algorithms for this problem, and experimentally demonstrated their efficiency.

We believe these contributions advance the understanding of some basic algorithms and data structures in Computer Science, as well as improving their efficiency *in practice.*

*Here $\beta(m,n) = \min\{k : \log^{(k)}(n) \leq m/n\}$, and so $\beta(n,n) = \log^* n$

Bibliography

Acharya, A., Zhu, H., and Shen, K. (1999). Adaptive Algorithms for Cache-Efficient Trie Search. In *ALENEX '99: 1st International Workshop on Algorithm Engineering and Experimentation*, pages 296–311, London, UK. Springer-Verlag.

Agarwal, R. C. (1996). A super scalar sort algorithm for RISC processors. pages 240–246.

Aggarwal, A. and Vitter, Jeffrey, S. (1988). The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127.

AMD (2009). *AMD64 Architecture Programmer's Manual Volume 6: 128-Bit and 256-Bit XOP, FMA4 and CVT16 Instructions.*

Andersson, A. (1995). Sublogarithmic searching without multiplications. In *FOCS '95: Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, page 655, Washington, DC, USA. IEEE Computer Society.

Andersson, A. and Nilsson, S. (1993). Improved behaviour of tries by adaptive branching. *Inf. Process. Lett.*, 46(6):295–300.

Andersson, A. and Thorup, M. (2007). Dynamic ordered sets with exponential search trees. *J. ACM*, 54(3):13.

Apostolico, A., Atallah, M. J., and Hambrusch, S. E. (1992). New clique and independent set algorithms for circle graphs. *Discrete Applied Mathematics*, 36(1):1–24.

Apostolico, A., Atallah, M. J., and Hambrusch, S. E. (1993). Erratum: New Clique and Independent Set Algorithms for Circle Graphs (Discrete Applied Mathematics 36 (1992) 1-24). *Discrete Applied Mathematics*, 41(2):179–180.

Asano, T., Asano, T., and Imai, H. (1986). Partitioning a polygonal region into trapezoids. *J. ACM*, 33(2):290–312.

Asano, T., Imai, H., and Mukaiyama, A. (1991). Finding a maximum weight independent set of a circle graph. *IEICE Transactions*, E74(4):681–683.

Askitis, N. (2009). Fast and compact hash tables for integer keys. In Mans, B., editor, *Thirty-Second Australasian Computer Science Conference (ACSC 2009)*, volume 91 of *CRPIT*, pages 101–110, Wellington, New Zealand. ACS.

Askitis, N. and Sinha, R. (2007). Hat-trie: A cache-conscious trie-based data structure for strings. In Dobbie, G., editor, *Thirtieth Australasian Computer Science Conference (ACSC2007)*, volume 62 of *CRPIT*, pages 97–105, Ballarat Australia. ACS.

Astrachan, O. (2003). Bubble sort: an archaeological algorithmic analysis. In *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 1–5, New York, NY, USA. ACM.

Bayer, R. and McCreight, E. M. (1972). Organization and Maintenance of Large Ordered Indices. *Acta Inf.*, 1:173–189.

Beame, P. and Fich, F. E. (2002). Optimal bounds for the predecessor problem and related problems. *J. Comput. Syst. Sci.*, 65(1):38–72.

Belady, L. (1966). A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5(2):78–101.

Bell, T. C., Cleary, J. G., and Witten, I. H. (1990). *Text Compression (Prentice Hall Advanced Reference Series)*. Prentice Hall.

Bender, M. A., Demaine, E. D., and Farach-Colton, M. (2000). Cache-oblivious b-trees. In *FOCS '00: Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, page 399, Washington, DC, USA. IEEE Computer Society.

Bentley, J. L. and McIlroy, M. D. (1993). Engineering a sort function. *Softw. Pract. Exper.*, 23(11):1249–1265.

Bentley, J. L. and Ottmann, T. (1979). Algorithms for Reporting and Counting Geometric Intersections. *IEEE Trans. Computers*, 28(9):643–647.

Bida, E. and Toledo, S. (2007). An automatically-tuned sorting library. *Softw. Pract. Exper.*, 37(11):1161–1192.

Biggar, P. and Gregg, D. (2005). Sorting in the presence of branch prediction and caches. Technical Report TCD-CS-05-57, University of Dublin, Trinity College.

Biggar, P., Nash, N., Williams, K., and Gregg, D. (2008). An Experimental Study of Sorting and Branch Prediction. *J. Exp. Algorithmics*, 12:1–39.

Blum, N. and Mehlhorn, K. (1980). On the average number of rebalancing operations in weight-balanced trees. *Theoretical Computer Science*, 11(3):303–320.

Boucheron, S. and de la Vega, F. (2001). On the independence number of random interval graphs. *Combinatorics, Probability and Computing*, 10(05):385–396.

Brent, R. P. (2004). Note on marsaglia's xorshift random number generators. *Journal of Statistical Software*, 11(5):1–4.

Brodal, G. S., Fagerberg, R., and Jacob, R. (2002). Cache oblivious search trees via binary trees of small height. In *SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 39–48, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

Brodal, G. S., Fagerberg, R., and Moruz, G. (2005). On the Adaptiveness of Quicksort. In *Proc. 7th Workshop on Algorithm Engineering and Experiments*, pages 130–140.

Brodal, G. S., Fagerberg, R., and Vinther, K. (2007). Engineering a cache-oblivious sorting algorithm. *ACM Journal of Experimental Algorithmics*, 12.

Brodal, G. S. and Moruz, G. (2005). Tradeoffs Between Branch Mispredictions and Comparisons for Sorting Algorithms. In *WADS*, pages 385–395.

Brodal, G. S. and Moruz, G. (2006). Skewed binary search trees. In *ESA'06: Proceedings of the 14th conference on Annual European Symposium*, pages 708–719, London, UK. Springer-Verlag.

Burks, A. W. (1989). *Perspectives on the computer revolution.* Ablex Publishing Corp., Norwood, NJ, USA.

Burks, A. W., Goldstine, H. H., and von Neumann, J. (1946). Preliminary discussion of the logical design of an electronic computing instrument. . Report to the U.S. Army Ordnance Department.

Case, R. P. and Padegs, A. (1978). Architecture of the IBM system/370. *Commun. ACM*, 21(1):73–96.

Cong, J. and Liu, C. L. (1990). Over-the-cell channel routing. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 9(4):408–418.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to Algorithms.* MIT Press, Cambridge, MA, USA, 2nd edition.

de Werra, D., Eisenbeis, C., Lelait, S., and Marmol, B. (1999). On a graph-theoretical model for cyclic register allocation. *Discrete Applied Mathematics*, 93(2-3):191–203.

de Werra, D., Eisenbeis, C., Lelait, S., and Stöhr, E. (2002). Circular-arc graph coloring: on chords and circuits in the meeting graph. *European Journal of Oper. Research*, 136:483–500.

Demaine, E. (2003). Fixed Universe Successor Problem . *Advanced Data Structures, MIT Course 6.897.*

Dementiev, R., Kettner, L., Mehnert, J., and Sanders, P. (2004). Engineering a Sorted List Data Structure for 32 Bit Keys. In *Proc. of the Sixth SIAM Workshop on Algorithm Engineering and Experiments, New Orleans, LA, USA*, pages 142–151.

Denning, P. J. (1968). Thrashing: its causes and prevention. In *AFIPS '68 (Fall, part I): Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pages 915–922, New York, NY, USA. ACM.

Denning, P. J. (2005). The locality principle. *Commun. ACM*, 48(7):19–24.

Devroye, L. (1982). A note on the average depth in tries. *Computing*, 28:367–371.

Dietzfelbinger, M., Karlin, A., Mehlhorn, K., auf der Heide, F. M., Rohnert, H., and Tarjan, R. E. (1988). Dynamic perfect hashing: upper and lower bounds. In *SFCS '88: Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 524–531, Washington, DC, USA. IEEE Computer Society.

Dongarra, J., London, K., Moore, S., Mucci, P., Terpstra, D., You, H., and Zhou, M. (2003). Experiences and lessons learned with a portable interface to hardware performance counters. In *IPDPS '03: Proc. of the 17th International Symposium on Parallel and Distributed Processing*, page 289.2, Washington, DC, USA. IEEE Computer Society.

Eichenberger, A. E., O'Brien, J. K., O'Brien, K. M., Wu, P., Chen, T., Oden, P. H., Prener, D. A., Shepherd, J. C., So, B., Sura, Z., Wang, A., Zhang, T., Zhao, P.,

Gschwind, M. K., Archambault, R., Gao, Y., and Koo, R. (2006). Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture. *IBM Systems Journal*, 45(1):59–84.

Eisenbeis, C., Lelait, S., and Marmol, B. (1995). The meeting graph: a new model for loop cyclic register allocation. In *PACT '95: Proceedings of the IFIP WG10.3 working conference on Parallel architectures and compilation techniques*, pages 264–267, Manchester, UK. IFIP Working Group on Algol.

Feller, W. (1968). *An Introduction to Probability Theory and its Applications.* Wiley.

Flajolet, P. (2006). The ubiquitous digital tree. In Durand, B. and Thomas, W., editors, *STACS 2006*, volume 3884 of *Lecture Notes in Computer Science*, pages 1–22. Proceedings of 23rd Annual Symposium on Theoretical Aspects of Computer Science, Marseille, February 2006.

Floyd, R. W. (1964). Treesort 3: Algorithm 245. *Communications of the ACM*, 7(12):701.

Fredman, M. L. and Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615.

Fredman, M. L. and Willard, D. E. (1990). Blasting through the information theoretic barrier with fusion trees. In *STOC '90: Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 1–7, New York, NY, USA. ACM.

Frias, L., Petit, J., and Roura, S. (2009). Lists Revisited: Cache Conscious STL Lists. *Journal of Experimental Algorithmics (JEA).* To appear.

Frigo, M., Leiserson, C. E., Prokop, H., and Ramachandran, S. (1999). Cache-Oblivious Algorithms. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 285, Washington, DC, USA. IEEE Computer Society.

Gavril, F. (1973). Algorithms for a maximum clique and a maximum independent set of a circle graph. *Networks*, 3(3):261–273.

Gog, S. (2009). Broadword computing and fibonacci code speed up compressed suffix arrays. In Vahrenhold, J., editor, *Experimental Algorithms, 8th International Symposium, SEA 2009, Dortmund, Germany, June 4-6, 2009. Proceedings*, pages 161–172.

Goldschmidt, O. and Takvorian, A. (1994). An efficient algorithm for finding a maximum weight independent set of a circle graph. *IEICE Transactions*, E77-A(10):1672–1674.

Golumbic, M. C. (2004). *Algorithmic Graph Theory and Perfect Graphs (Annals of Discrete Mathematics, Vol 57)*. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands.

Graham, R. L., Knuth, D. E., and Patashnik, O. (1994). *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Guibas, L. J. and Sedgewick, R. (1978). A dichromatic framework for balanced trees. In *SFCS '78: Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21, Washington, DC, USA. IEEE Computer Society.

Gupta, U. I., Lee, D. T., and Leung, J. Y.-T. (1982). Efficient algorithms for interval graphs and circular-arc graphs. *Networks*, 12(4):459–467.

Hagerup, T. (1998). Sorting and searching on the word ram. In *STACS '98: Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science*, pages 366–398, London, UK. Springer-Verlag.

Heinz, S., Zobel, J., and Williams, H. E. (2002). Burst tries: a fast, efficient data structure for string keys. *ACM Trans. Inf. Syst.*, 20(2):192–223.

Hennessy, J. L. and Patterson, D. A. (2006). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 4th edition.

Hinton, G., Sager, D., Upton, M., Carmean, D., Kyker, A., , and Roussel, P. (2001). The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, Q1.

Hoare, C. A. R. (1962). Quicksort. *Computer Journal*, 5(1):10–15.

Ibbett, R. N. (1971). The MU5 Instruction Pipeline . *The Computer Journal*, 15(1):42 – 50.

IEEE (2008). Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–58.

Intel (2007). *Intel 64 and IA-32 Architectures Optimization Reference Manual*.

Intel (2009). *Intel Reference Manuals, see http://www.intel.com/products/processor/manuals*

Janson, S. and Szpankowski, W. (2007). Partial fillup and search time in lc tries. *ACM Trans. Algorithms*, 3(4):44.

Kaligosi, K. and Sanders, P. (2006). How Branch Mispredictions Affect Quicksort. In *Proceedings of The 14th Annual European Symposium on Algorithms (ESA 2006)*. (to appear).

Kasheff, Z. (2004). Cache-Oblivious Dynamic Search Trees. M.eng., Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

Keil, J. M. (1993). The complexity of domination problems in circle graphs. *Discrete Applied Mathematics*, 42(1):51–63.

Kessler, R. E., Mclellan, E. J., and Webb, D. A. (1998). The Alpha 21264 microprocessor architecture. In *In Proceedings of the 1998 International Conference on Computer Design*, pages 90–95.

Knessl, C. and Szpankowski, W. (2000a). A Note on the Asymptotic Behavior of the Heights in b-Tries for b Large. *Electr. J. Comb.*, 7.

Knessl, C. and Szpankowski, W. (2000b). Heights in Generalized Tries and PATRICIA Tries. In *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay*, pages 298–307. Springer. LNCS 1776.

Knuth, D. E. (1997a). *The Art Of Computer Programming, Volume 1 (3rd ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

Knuth, D. E. (1997b). *The Art Of Computer Programming, Volume 1 (3rd ed.): Fundamental Algorithms*, pages pp120–160, pp73–78. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

Knuth, D. E. (1998a). *The Art Of Computer Programming, Volume 3 (2nd ed.): Sorting And Searching*, pages 458–478, 482–491, 506. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

Knuth, D. E. (1998b). *The Art Of Computer Programming, Volume 3 (2nd ed.): Sorting And Searching*, pages p110, p175, pp73–180, pp153–155, pp158–168. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

Korda, M. and Raman, R. (1999). An Experimental Evaluation of Hybrid Data Structures for Searching. In *Proc. of the 3rd International Workshop on Algorithm Engineering (WAE), London, UK*, pages 213–227.

LaMarca, A. and Ladner, R. (1996). The influence of caches on the performance of heaps. *J. Exp. Algorithmics*, 1:4.

LaMarca, A. and Ladner, R. E. (1997). The influence of caches on the performance of sorting. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 370–379. Society for Industrial and Applied Mathematics.

Leighton, F. T. and Rivest, R. L. (1983). Estimating a Probability Using Finite Memory (Extended Abstract). In *Proceedings of the 1983 International FCT-Conference on Fundamentals of Computation Theory*, pages 255–269, London, UK. Springer-Verlag.

Li, X., Garzaran, M. J., and Padua, D. (2005). Optimizing Sorting with Genetic Algorithms. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 99–110, Washington, DC, USA. IEEE Computer Society.

Liptay, J. S. (1968). Structural aspects of the System/360 Model 85: II The cache . *IBM Systems Journal*, 7(1):373–379.

Liu, R. and Ntafos, S. C. (1988). On decomposing polygons into uniformly monotone parts. *Inf. Process. Lett.*, 27(2):85–89.

Manegold, S. and Boncz, P. (2004). Cache-memory and tlb calibration tool.

Martínez, C. and Roura, S. (2001). Optimal Sampling Strategies in Quicksort and Quickselect. *SIAM J. Comput.*, 31(3):683–705.

McFarling, S. and Hennessey, J. (1986). Reducing the cost of branches. *SIGARCH Comput. Archit. News*, 14(2):396–403.

McGeoch, C. C. (2007). Experimental algorithmics. *Commun. ACM*, 50(11):27–31.

McGeoch, C. C. (2008). Experimental Methods for Algorithm Analysis. In Kao, M.-Y., editor, *Encyclopedia of Algorithms*. Springer.

Mehlhorn, K. and Näher, S. (1990). Bounded ordered dictionaries in O(loglogN) time and O(n) space. *Inf. Process. Lett.*, 35(4):183–189.

Mehlhorn, K. and Näher, S. (1998). *LEDA*. Cambridge University Press.

Michaud, P. (2004). Analysis of a tag-based branch predictor . Technical report, INRIA.

Moret, B. (2001). Towards a discipline of experimental algorithmics. In *Proc. 5th DIMACS Challenge*.

Mudge, T., Chen, I.-C., and Coffey, J. (1996). Limits to Branch Prediction. Technical Report CSE-TR-282-96.

Nash, N. and Gregg, D. (2008). Comparing integer data structures for 32 and 64 bit keys. In McGeoch, C., editor, *Proceedings of the Seventh International Workshop on Experimental Algorithms*, pages 28–42, Provincetown, Cape Cod, MA, USA. LNCS 5038.

Nash, N. and Gregg, D. (2010). An output sensitive algorithm for computing a maximum independent set of a circle graph. *(submitted)*.

Nethercote, N. and Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100.

Nilsson, S. (1996). *Radix Sorting and Searching*. PhD thesis, Lund University.

Nilsson, S. and Tikkanen, M. (2002). An Experimental Study of Compression Methods for Dynamic Tries. *Algorithmica*, 33(1):19–33.

Nyberg, C., Barclay, T., Cvetanovic, Z., Gray, J., and Lomet, D. (1994). AlphaSort: a RISC machine sort. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 233–242, New York, NY, USA. ACM Press.

Pan, S.-T., So, K., and Rahmeh, J. T. (1992). Improving the accuracy of dynamic branch prediction using branch correlation. In *ASPLOS-V: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pages 76–84, New York, NY, USA. ACM.

Pătraşcu, M. and Thorup, M. (2006). Time-space trade-offs for predecessor search. In *Proc. 38th ACM Symposium on Theory of Computing (STOC)*, pages 232–240. See also arXiv:0603043.

Pugh, W. (1990). Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676.

Rahman, N. and Raman, R. (2001). Adapting Radix Sort to the Memory Hierarchy. *J. Exp. Algorithmics*, 6:7.

Rao, J. and Ross, K. A. (2000). Making B+- trees cache conscious in main memory. *SIGMOD Rec.*, 29(2):475–486.

Sanders, P. (2009). Algorithm Engineering - An Attempt at a Definition. In Albers, S., Alt, H., and Näher, S., editors, *Efficient Algorithms*, volume 5760 of *Lecture Notes in Computer Science*, pages 321–340. Springer.

Sanders, P. and Winkel, S. (2004). Super Scalar Sample Sort. In Albers, S. and Radzik, T., editors, *Algorithms . ESA 2004: 12th Annual European Symposium*, volume 3221 of *Lecture Notes in Computer Science*, pages 784–796, Bergen, Norway. Springer.

Scheinerman, E. R. (1988). Random interval graphs. *Combinatorica*, 8(4):357–371.

Scheinerman, E. R. (1990). An evolution of interval graphs. *Discrete Math.*, 82(3):287–302.

Sedgewick, R. and Flajolet, P. (1996). *An introduction to the analysis of algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Seidel, R. and Aragon, C. R. (1996). Randomized Search Trees. *Algorithmica*, pages 540–545.

Seward, J. (2007). Personal Communication.

Singleton, R. C. (1969). Algorithm 347: an efficient algorithm for sorting with minimal storage [M1]. *Commun. ACM*, 12(3):185–186.

Sinha, R. (2004). Using Compact Tries for Cache-Efficient Sorting of Integers. In Ribeiro, C. C., editor, *Proc. of the Third International Workshop on Efficient and Experimental Algorithms (WEA 2004)*, pages 513–528, Angra dos Reis, Rio de Janeiro, Brazil. LNCS 3059.

Sinha, R., Ring, D., and Zobel, J. (2006). Cache-efficient String Sorting using Copying. *J. Exp. Algorithmics*, 11:1.2.

Sinha, R. and Wirth, A. (2008). Engineering Burstsort: Towards Fast In-Place String Sorting. In *WEA*, pages 14–27.

Sinha, R. and Zobel, J. (2004). Cache-conscious Sorting of Large Sets of Strings with Dynamic Tries. *J. Exp. Algorithmics*, 9:1.5.

Sinha, R. and Zobel, J. (2005). Using Random Sampling to Build Approximate Tries for Efficient String Sorting. *J. Exp. Algorithmics*, 10:2.10.

Sleator, D. D. and Tarjan, R. E. (1985). Self-adjusting binary search trees. *J. ACM*, 32(3):652–686.

Smith, J. E. (1981). A study of branch prediction strategies. In *ISCA '81: Proceedings of the 8th annual symposium on Computer Architecture*, pages 135–148, Los Alamitos, CA, USA. IEEE Computer Society Press.

Srinivasan, R., Frachtenberg, E., Lubeck, O., Pakin, S., and Cook, J. (2007). An Idealistic Neuro-PPM Branch Predictor . *Journal of Instruction Level Parallelism*, 9:1 - 13.

Stroustrup, B. (1997). *The C++ Programming Language, Third Edition.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Supowit, K. J. (1987). Finding a maximum planar subset of a set of nets in a channel. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 6(1):93–94.

Sussenguth, E. H. (1963). Use of Tree Structures for Processing Files. *Commun. ACM*, 6(5):272–279.

Thorup, M. (2003). On AC0 implementations of fusion trees and atomic heaps. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 699–707, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

Tiskin, A. (2009). Semi-local string comparison: algorithmic techniques and applications. *CoRR*, abs/0707.3619.

Uht, A. K., Sindagi, V., and Somanathan, S. (1997). Branch Effect Reduction Techniques. *Computer*, 30(5):71–81.

Valiente, G. (2003). A new simple algorithm for the maximum-weight independent set problem on circle graphs. In Ibaraki, T., Katoh, N., and Ono, H., editors, *ISAAC*, volume 2906 of *Lecture Notes in Computer Science*, pages 129–137. Springer.

van Emde Boas, P. (1977). Preserving Order in a Forest in Less Than Logarithmic Time and Linear Space. *Inf. Process. Lett.*, 6(3):80–82.

Vigna, S. (2008). Broadword implementation of rank/select queries. In McGeoch, C., editor, *Proceedings of the Seventh International Workshop on Experimental Algorithms*, pages 154–168, Provincetown, Cape Cod, MA, USA.

Vollmer, H. (1999). *Introduction to Circuit Complexity: A Uniform Approach.* Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Weiner, P. (1973). Linear pattern matching algorithms. In *SWAT '73: Proceedings of the 14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11, Washington, DC, USA. IEEE Computer Society.

Wickremesinghe, R., Arge, L., Chase, J. S., and Vitter, J. S. (2002). Efficient sorting using registers and caches. *J. Exp. Algorithmics*, 7:9.

Wilkes, M. (1965). Abstracts of Current Computer Literature. *Electronic Computers, IEEE Transactions on*, EC-14(2):281–293.

Wilkes, M. V. (2000). *Slave memories and dynamic storage allocation*, pages 371–372. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Willard, D. E. (1983). Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84.

Willard, D. E. (1984). New trie data structures which support very fast search operations. *J. Comput. Syst. Sci.*, 28(3):379–394.

Willard, D. E. (1992). Applications of the fusion tree method to computational geometry and searching. In *SODA '92: Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms*, pages 286–295, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

Willard, D. E. (2000). Examining computational geometry, van emde boas trees, and hashing from the perspective of the fusion tree. *SIAM J. Comput.*, 29(3):1030–1049.

Willems, F. M. J., Shtarkov, Y. M., and Tjalkens, T. J. (1995). The Context Tree Weighting Method: Basic Properties. *IEEE Transactions on Information Theory*, 41:653–664.

Williams, J. W. J. (1964). Heapsort: Algorithm 232. *Communications of the ACM*, 7(6):347–348.

Xiao, L., Zhang, X., and Kubricht, S. A. (2000). Improving memory performance of sorting algorithms. *J. Exp. Algorithmics*, 5:3.

Zargham, M. R. (1996). *Computer architecture: single and parallel systems.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Ziv, J. and Lempel, A. (1977). A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23:337–343.

Zou, Q., Zhao, T., Liu, Y., and Guo, M. (2009). Predicting rna secondary structure based on the class information and hopfield network. *Computers in Biology and Medicine*, 39(3):206 - 214.

Appendix A

Additional Experimental Results for Integer Data Structures

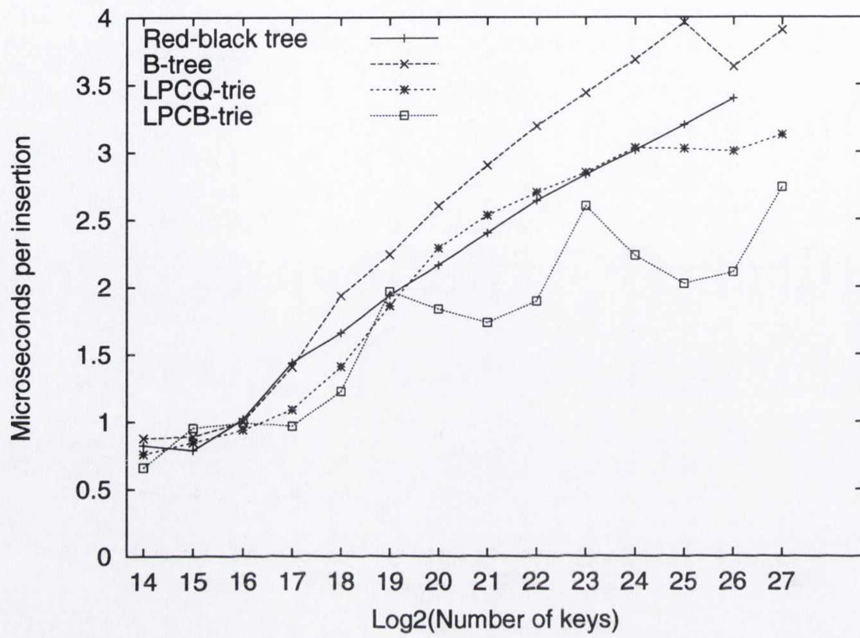
A.1 Alternative Machine Configurations

In this section we present additional experimental results for the data structures described in this Chapter 4. As was done in the main experimental comparison of Section 4.4, we refer to the machine configurations by the names (`beaker`, `knuth`, `melody`, `stoker`) given in Table 4.1. The descriptions of the experimental results in this section are brief, the naming of the data structures and other details can be found in Section 4.4. Note also that we do not provide space measurements for every machine, this is because they are identical for machines of the same word-size. Thus we provide space measurements for one 32-bit and one 64-bit machine.

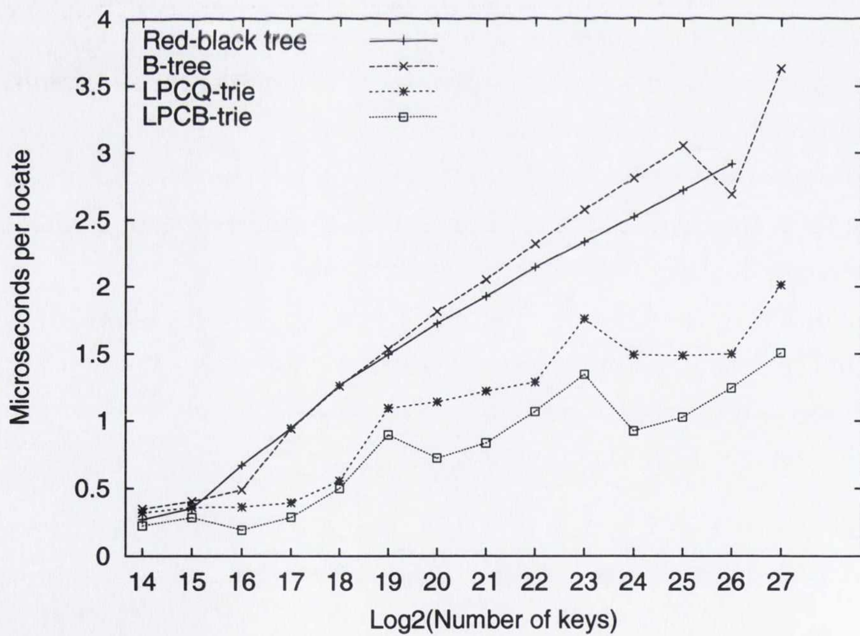
In addition to the data distributions described above, we also consider the performance of the data structures where queries are drawn from the Zipf distribution. That is, we insert a sequence of uniform random keys $k_1, \ldots, k_n$ and then perform a series of queries, where the probability of searching for k_m is

$$Z(m) = \frac{1/m^s}{\sum_{i=1}^{n} 1/i^s}$$

Here s is the parameter of the distribution. Note that $s = 0$ implies uniform random selection of query indices. Increasing s results in a distribution where the lower indices occur with higher probability than higher indices. For example, in general the first index is 2^s times more likely to occur than the second. For $s = 1$, k_1 is queried half the time, k_2 one third of the time, et cetera. We have used $s = 0.8$ in our experiments.

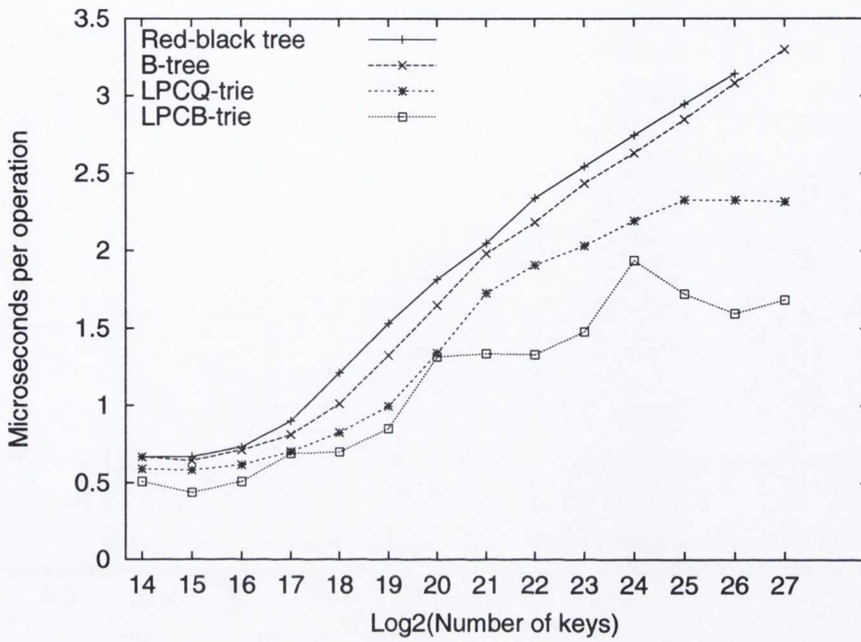


(a)

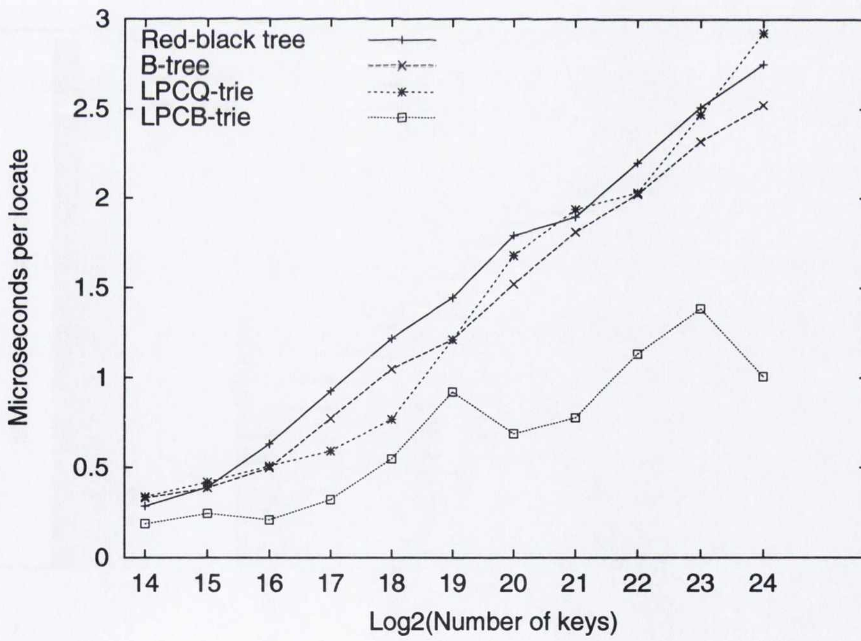


(b)

Figure A.1: These results were gathered on `melody`. (a) Shows the insertion times for the data structures, (b) shows the locate times for the data structures. The data is uniform random 32-bit keys.

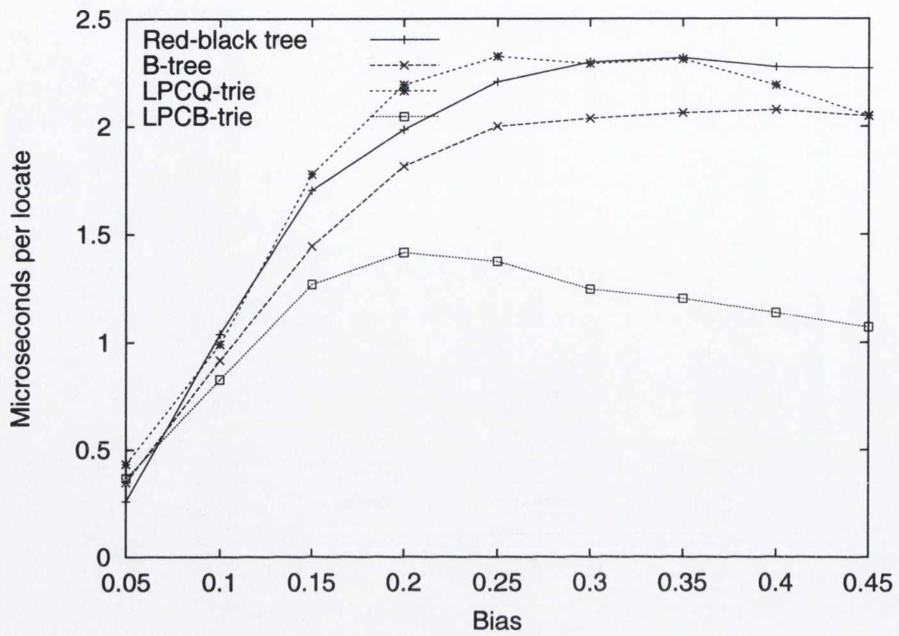


(a)

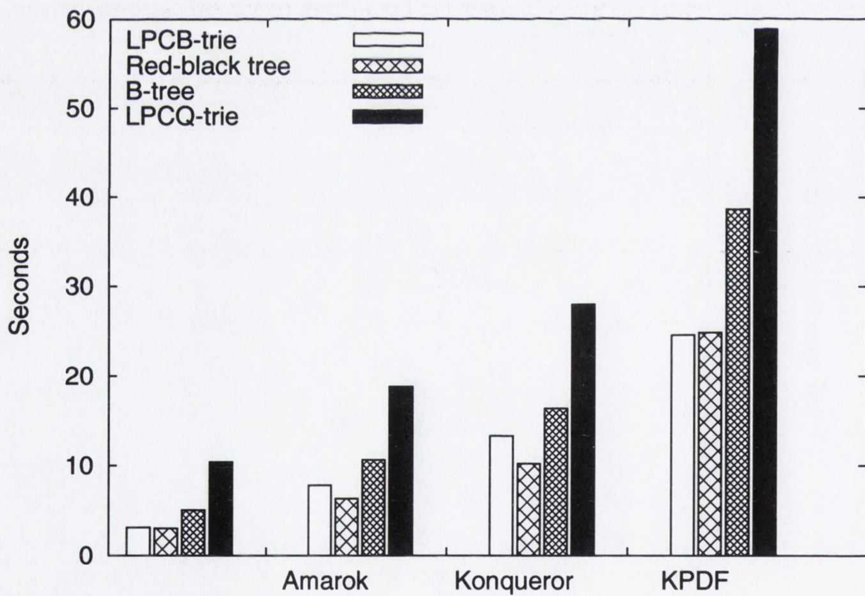


(b)

Figure A.2: These results were gathered on `melody`. (a) Shows the time for a mixed sequence of insertions and deletions for the data structures, (b) shows the locate times. The data is uniform random 32-bit keys. (b) Shows the time required for Zipf distributed locate operations, described above.

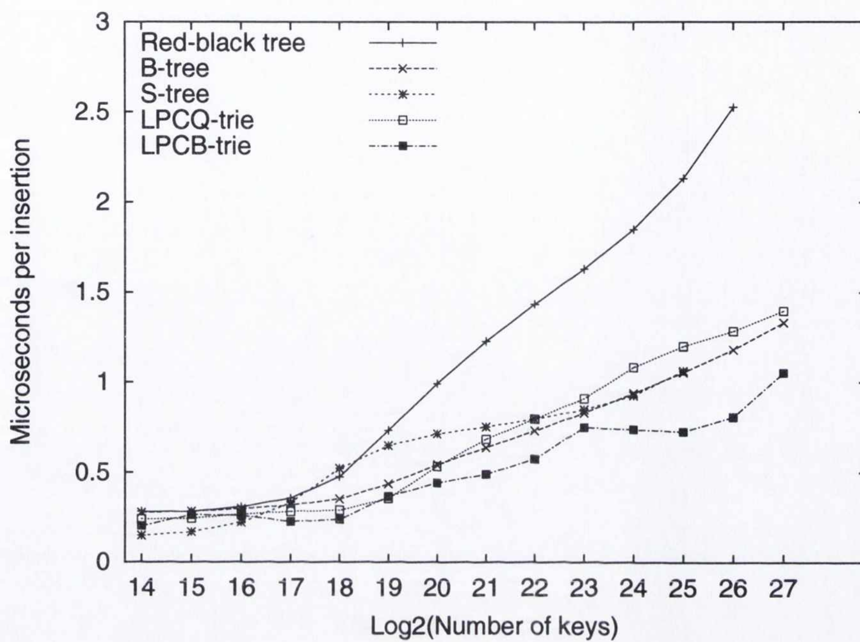


(a)

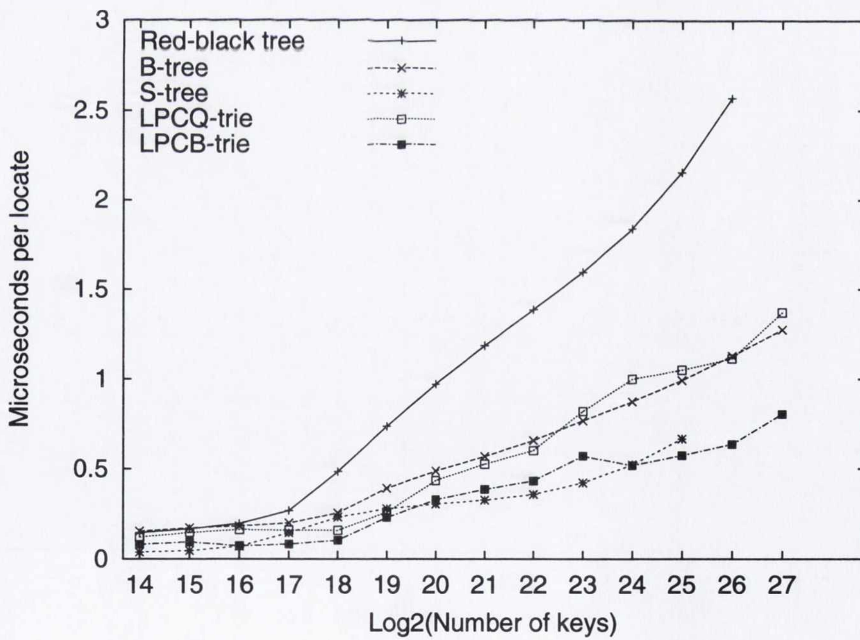


(b)

Figure A.3: These results were gathered on `melody`. (a) Shows the time for a sequence of locate operations for biased bit data, as p varies. Prior to the locates, 2^{22} insertions also of biased bit data are performed. The keys are 32-bits in length. (b) Shows the time for the data structures to process the 32-bit Valgrind traces.

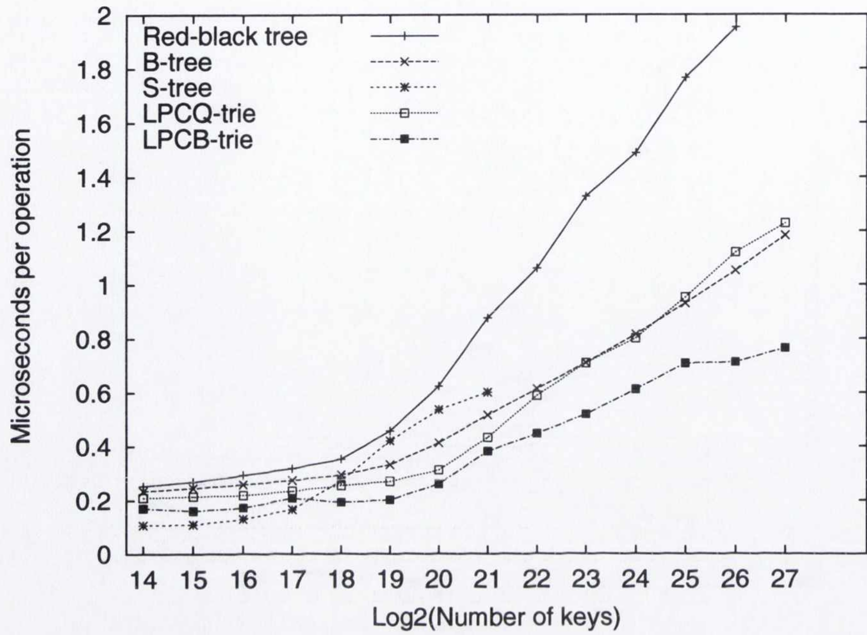


(a)

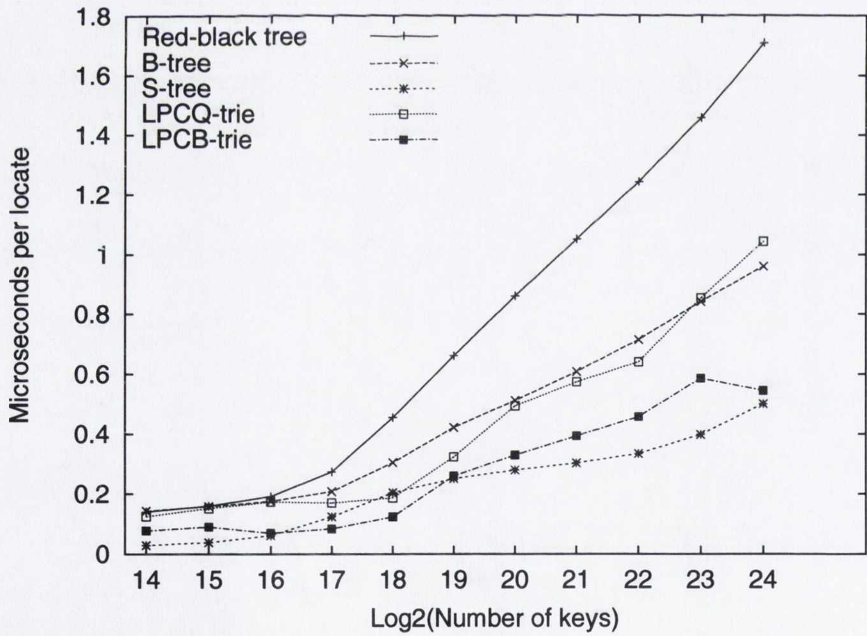


(b)

Figure A.4: These results were gathered on `stoker`. (a) Shows the insertion times for the data structures, (b) shows the locate times for the data structures. The data is uniform random 32-bit keys.

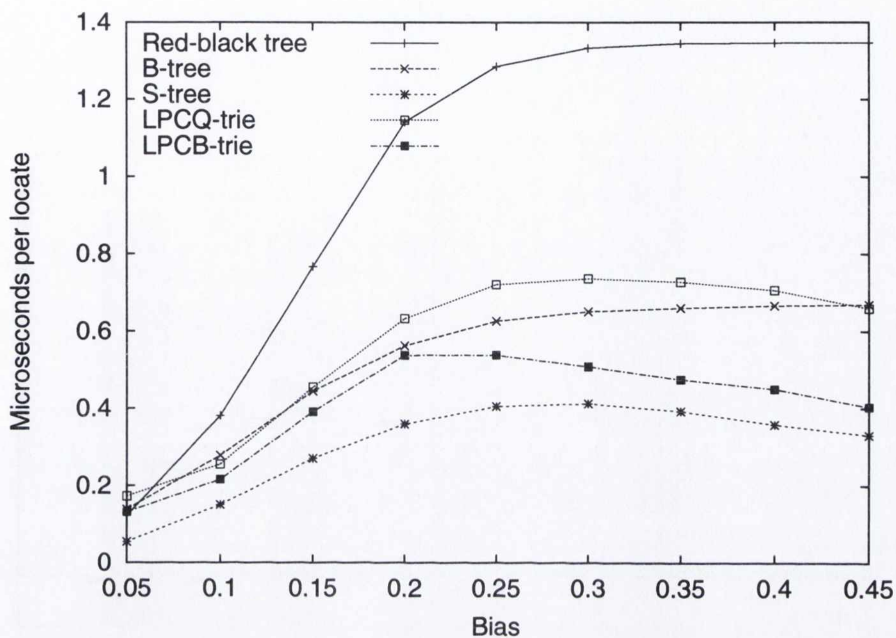


(a)

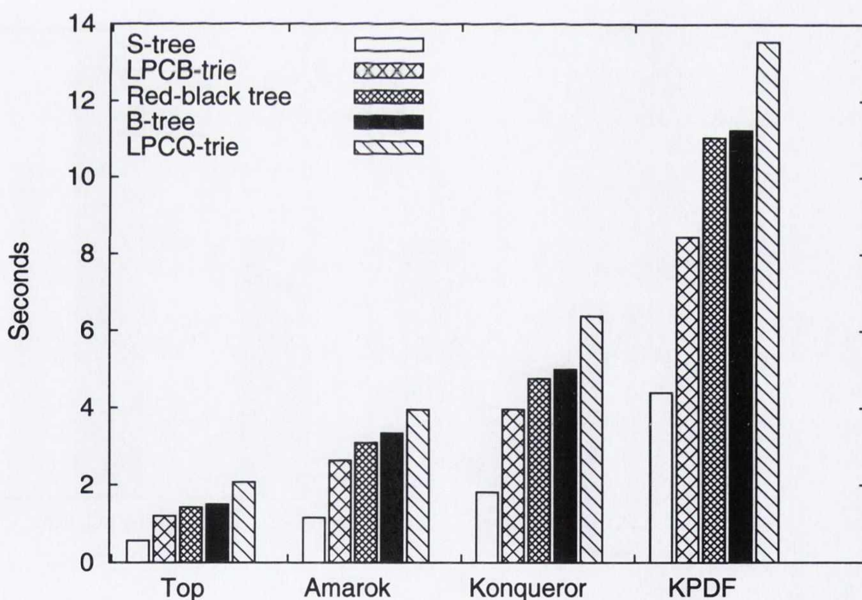


(b)

Figure A.5: These results were gathered on `stoker`. (a) Shows the time for a mixed sequence of insertions and deletions for the data structures, (b) shows the locate times. The data is uniform random 32-bit keys. (b) Shows the time required for Zipf distributed locate operations, described above.

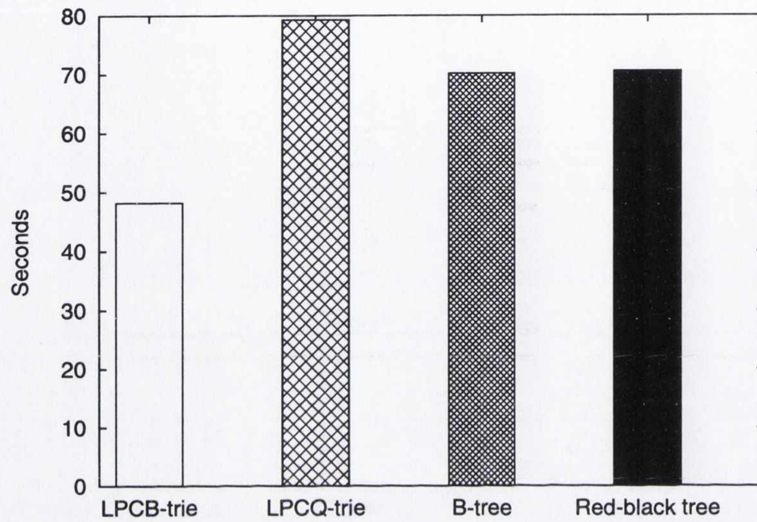


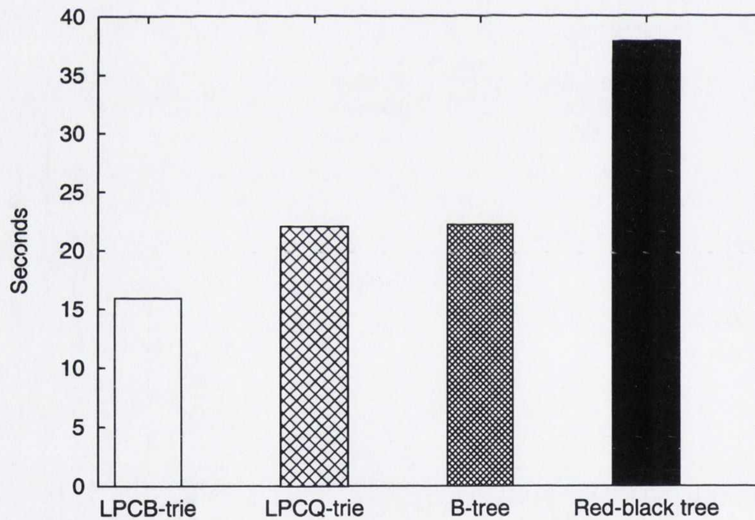
(a)



(b)

Figure A.6: These results were gathered on `stoker`. (a) Shows the time for a sequence of locate operations for biased bit data, as p varies. Prior to the locates, 2^{22} insertions also of biased bit data are performed. The keys are 32-bits in length. (b) Shows the time for the data structures to process the 32-bit Valgrind traces.

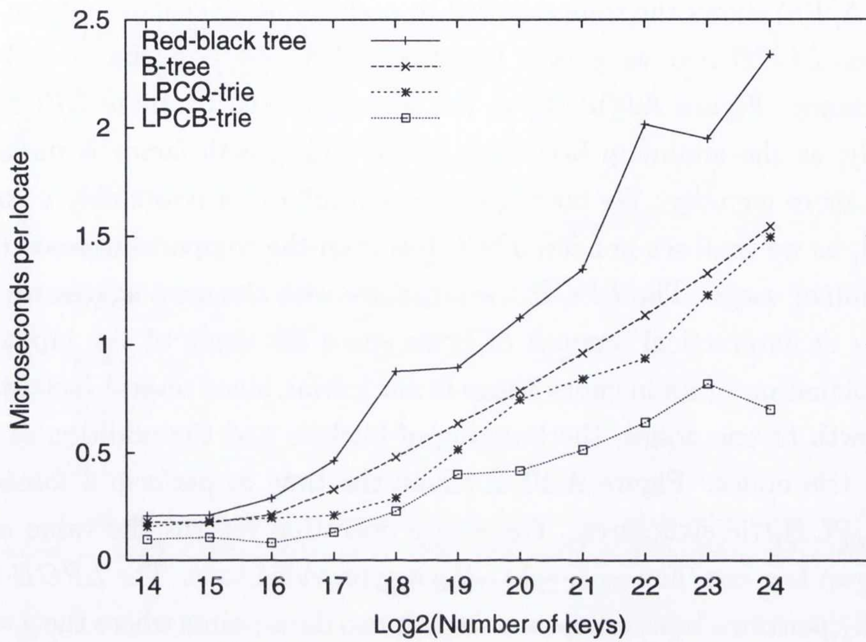




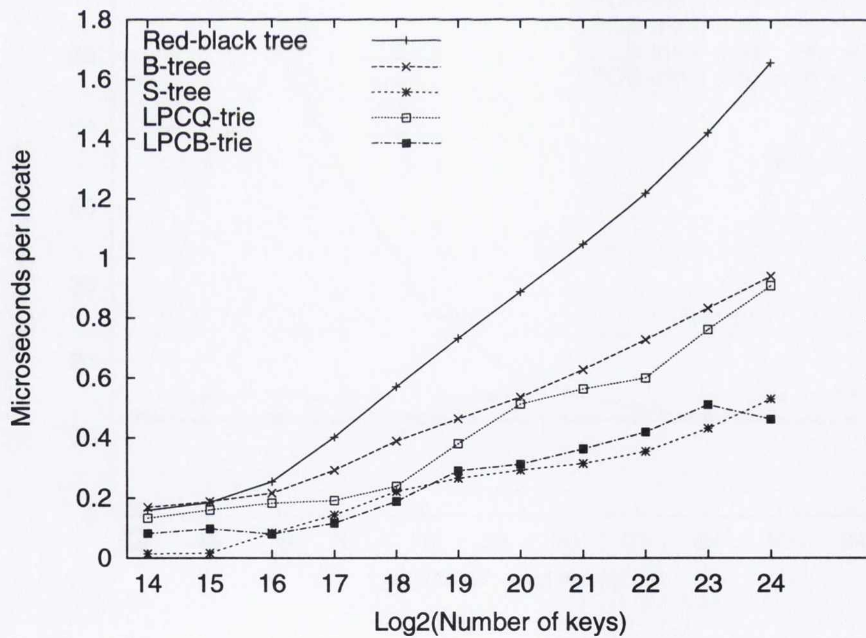
(a)

(b)

Figure A.7: These results show the time required to process the 32-bit Genome data set by the data structures. (a) Shows the time on `melody`, while (b) shows the time on `stoker`.



(a)

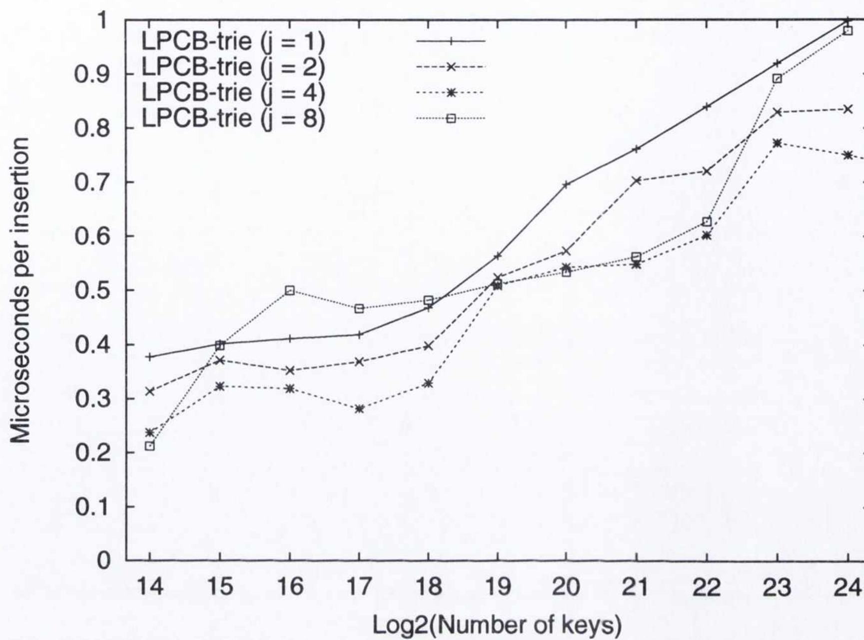


(b)

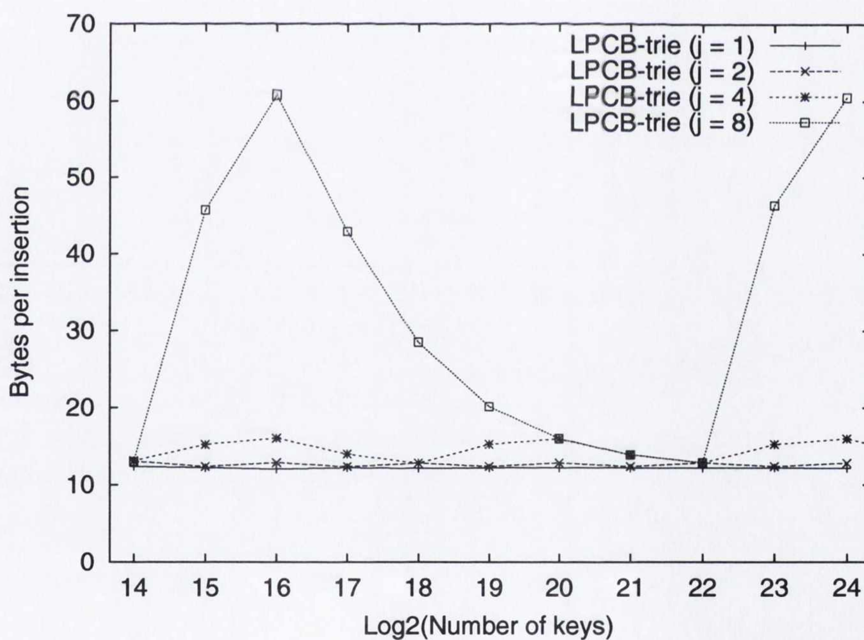
Figure A.8: (a) Shows the time to process a sequence of Zipf distributed locate operations on `beaker`. (b) Shows the time to process a sequence of Zipf distributed locate operations on `knuth`.

A.2 Growth Factor Results

Figure A.9(a) shows the time required to perform insertions of uniform random 32-bit keys into *LPCB*-tries as growth factor is varied. We note that $j = 4$ gives the best performance. Figure A.9(b) shows the space required by these *LPCB*-tries. Unsurprisingly, as the minimum branching factor and growth factor is increased, the tries require more memory. We note that $j = 4$ results in a reasonably compact structure (indeed, as we shall see in Section 4.4, less than the comparison based data structures we examine) usage. The *LPCB*-trie structure with the most aggressive growth, $j = 8$, requires an impractical amount of extra space for many of the input sizes. A precise explanation of its memory usage is not trivial, since several factors influence this: the growth of trie nodes, the bursting of buckets and the addition of (possibly very sparse) trie nodes. Figure A.10(a) shows the time to perform a *locate* operation on these *LPCB*-trie structures. The locate operation returns the value associated with the largest key less than or equal to the key provided to it. The *LPCB*-trie with $j = 4$ generally performs best, except for the last two data-points where the $j = 8$ *LPCB*-trie performs better.



(a)



(b)

Figure A.9: (a) Shows the time required for insertion operations on *LPCB*-tries with different minimum node degree and growth factors. (b) Shows the space required by the data structures for the sequence of insertions in (a). The keys inserted are uniform random 32-bit integers. These results were gathered on the machine `knuth`, we provide the details of our experimental setup in Section 4.4.

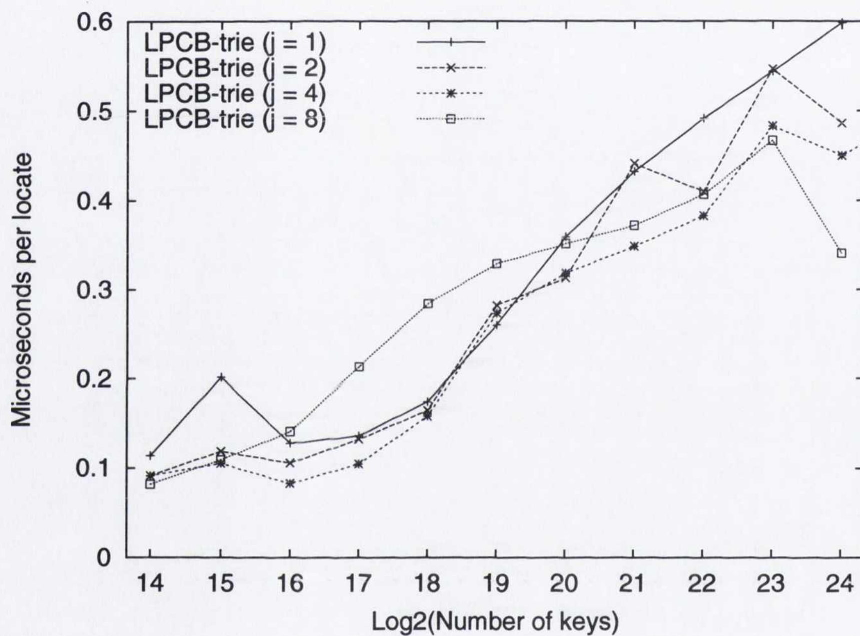


Figure A.10: This figure shows the time required for locate operations on *LPCB*-tries with different minimum node degree and growth factors. The keys are uniform random 32-bit integers. These results were gathered on the machine `knuth`. Details of our experimental setup can be found in Section 4.4.