Internal memory sorting and searching

By

Naila Yasmeen Rahman

A dissertation submitted in partial fulfilment of

the requirements for the degree of

Doctor of Philosophy in Computer Science

2002

King's College University of London

Abstract

In this thesis we consider methods of designing fast, practical algorithms for sorting and searching data in the main memory of a computer system. Algorithms are traditionally analysed on the Random Access Machine (RAM) model, but such analyses often give inaccurate predictions of performance. This is partially due to the fact that the RAM model assumes the same time for arithmetic operations and memory accesses, while on today's computer systems a main memory access may take more than 100 times as long as an arithmetic operation, and this difference is likely to increase in the future. To minimise main memory accesses computer systems use one or more fast, small memories called caches, which store copies of recently accessed data. The RAM model also does not capture the translation of virtual memory addresses into physical memory addresses. Computer systems use a specialised, fast, small memory called the translation-look-aside buffer (TLB) to facilitate such translations.

Main memory, caches and the TLB constitute the internal memory hierarchy, while main memory, fast external memory (disks) and slow external memories (such as tapes) can be referred to as the external memory hierarchy. The main challenge in designing fast algorithms for the internal memory hierarchy is to minimise the number of cache and TLB misses, that is the number of accesses to data which are not in the cache or TLB. We show how algorithms and their analyses for the external memory hierarchy can be converted to corresponding algorithms and analyses for the internal memory hierarchy. We are the first to systematically consider TLB misses in this context.

We then consider the problems of sorting independent random floating-point numbers using distribution sorting and sorting integers using radix sort. We analyse the number of cache and TLB misses for these problems and design algorithms which comprehensively outperform Quicksort. We also analyse the number of cache and TLB misses for search tree data structures and design new search tree data structures that outperform existing ones.

We include extensive experimental results, which support the presented analyses.

Declaration

I declare that this doctoral thesis was composed by myself and the work contained herein is my own, except as stated now. Some parts of the thesis contain work from the following jointly-authored papers:

[70] N. Rahman, R. Cole and R. Raman. Optimising Predecessor Data Structures for Internal Memory. In *Proc. 5th International Workshop on Algorithm Engineering*, LNCS 2141, pp. 67–78, 2001.

[71] N. Rahman and R. Raman. Analysing Cache Effects in Distribution Sorting. *ACM Journal of Experimental Algorithmics* **5**, Article 14, 2001. Preliminary version in *Proc. 3rd Workshop on Algorithm Engineering*, LNCS 1668, pp. 184–198, 1999.

[72] N. Rahman and R. Raman. Adapting radix sort to the memory hierarchy. *ACM Journal of Experimental Algorithmics*, (to appear). Preliminary version in *Proc. 2nd Workshop on Algorithm Engineering and Experiments*, 2000.

[73] N. Rahman and R. Raman. Analysing the Cache Behaviour of Non-uniform Distribution Sorting Algorithms. In *Proc. 8th Annual European Symposium on Algorithms*, LNCS 1879, pp. 380–391, 2000.

Acknowledgements

I am extremely grateful to my supervisor, Prof. Rajeev Raman. Rajeev got me started in this area of research and has given me constant support and encouragement throughout my PhD studies. He has taught me how to do the highest quality research.

I would especially like to thank my husband, Tomasz Radzik, for his love and support. I would also like to thank Tomek for reading and commenting on the various drafts of the thesis.

I would like to thank the EPSRC for the PhD studentship. Without their financial support, completing my research in any reasonable timeframe would have been very difficult.

I owe many thanks to my friends and colleagues at Avaya Communications for their support over the years. I would also like to thank the management at Avaya for always allowing me to work there when other sources of funding were unavailable.

I would also like to thank my friends and family for their encouragement. Most importantly, I would like to thank my parents for their love and guidance.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Application programs, dealing with increasingly large data sets, require a large memory to hold data and the results of computation. In many cases they also require the computation to be done as fast as possible. Increases in CPU speeds go some way towards meeting the need for speed. However the time to access data is usually a significant proportion of the overall computation time. The designers of computer systems need to meet the demand for a large storage capacity which can be accessed at high speeds, but at the same time they must minimise overall system costs. The cost of memory varies with the technology used. A tradeoff is made, where computers instead use a hierarchy of memories, where the size of each memory varies inversely with its speed and unit storage cost. The size of the memory also varies with its distance from the CPU. Such that, memories which are increasingly closer to the CPU are smaller, faster and more expensive. The hierarchy of memories exploits the fact that many computer programs have good *locality of reference*, where nearby memory locations are, often repeatedly, accessed around the same time. The faster, smaller, memories are used to keep subsets of data likely to be referenced in the near future. This allows data to be supplied at high speed to the CPU, as most memory accesses take place at the same speed as the faster memories, only when the subsets of data held in the faster memories do not contain the required data are the slower memories accessed.

Until the mid 1980s most CPUs were only a few times faster than main memory and the memory hierarchy in most computers consisted of the CPU registers, main memory and disk, see Figure 1.1. If all the data used by a program fitted in main memory then, until the mid 1980s using the random access machine (RAM) model of

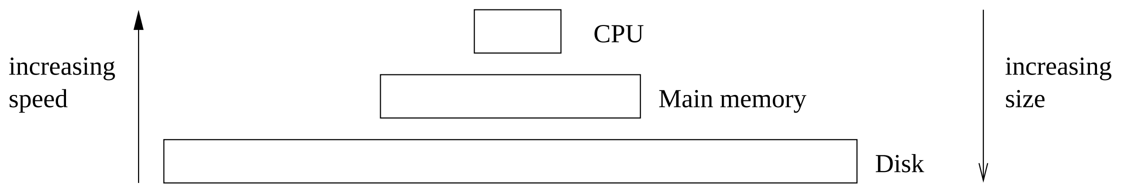

Figure 1.1: The typical memory hierarchy in computer systems before the mid 1980's.

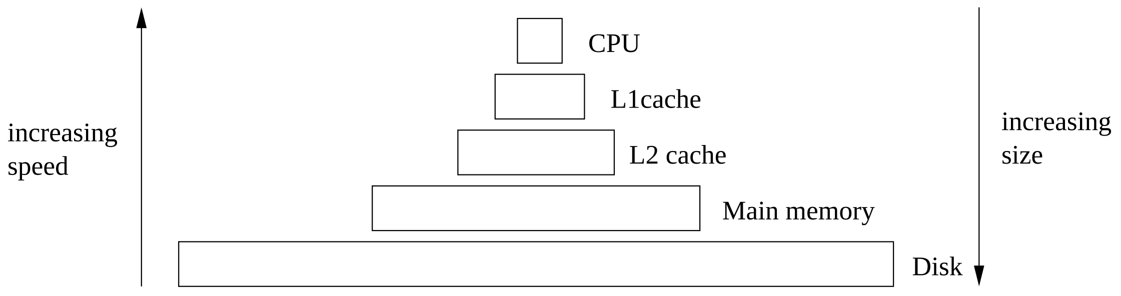

Figure 1.2: The typical memory hierarchy in recent computer systems.

computation [9] gave a fairly good prediction of performance. In the RAM model it is assumed that accessing a location in memory costs the same as a built-in arithmetic operation, such as adding two word-sized operands. Unfortunately this is no longer true. Consider the following example, which compares the performance of two sorting algorithms on a more recent computer system. Figure 1.3 shows the time per key to sort n uniformly-random single-precision floating-point numbers using a distribution sorting algorithm, *Flashsort1*, with an expected $O(n)$ running time and using Quicksort with an expected $\Theta(n \log n)$ running time. Both the algorithms have small constant factors. We consider quite large values of n, up to 67M. The asymptotic analysis of the running times tell us that, for sufficiently large n, as n grows the time per key for Flashort1 should remain constant and for Quicksort it should grow proportional to $\log n$. This suggests that for sufficiently large n, once Flashsort1 becomes faster than Quicksort, it should remain faster as n grows. However this is not true for n up to 67M. We see that for small n Flashsort1 is fast but for large n it is considerably slower than Quicksort. The algorithms are 'in-place', in that they use little additional memory, Flashsort1 uses $\approx 4.4n$ bytes and Quicksort uses $\approx 4n$ bytes. The running times are reported on a computer with 1GB of main memory, so there is almost no possibility of swapping code or data to disk during these experiments. In order to understand these running times we have to consider more recent developments in the memory hierarchy

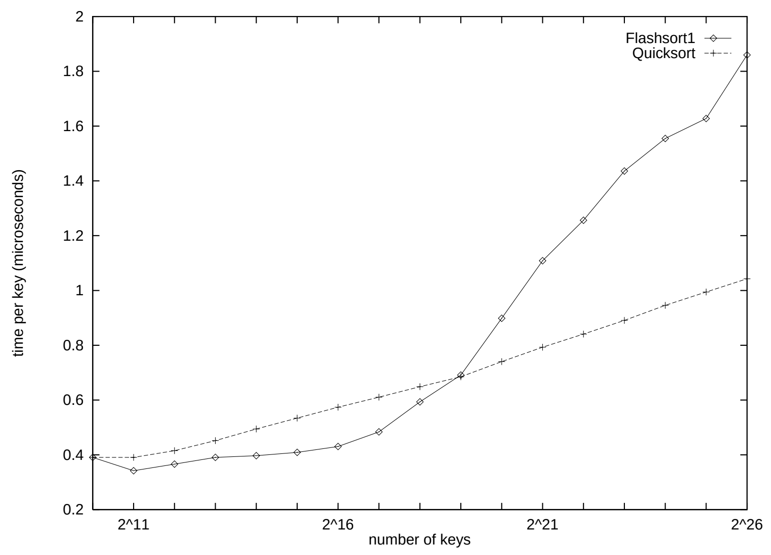

Figure 1.3: Time per key to sort single-precision floating-point keys using Flashsort1 and Quicksort on a Sun UltraSparc-II. For $n = 2^i, i = 10, \ldots, 26$. Note that the x-axis is on a log scale.

of modern computer systems.

Over the last 20 years or so, CPU clock rates have grown explosively, and CPUs with clock rates exceeding 2GHz are now available in the mass market. Unfortunately, the speed of main memory has not increased as rapidly: today's main memory typically has a latency of about 60ns. This implies that the cost of accessing main memory can be 120 times greater than the cost of performing an operation on data which are in the CPU's registers. Since the driving force behind CPU technology is speed and the primary driving force behind memory technology is storage capacity, this trend is likely to continue. In order to bridge this ever increasing performance gap, since the mid 1980s computer systems intended for the mass market have used one or more levels of *cache* memory between the CPU registers and main memory, see Figure 1.2.

Due to costs, caches are much smaller than main memory. In the example in Figure 1.3 the computer has a 512KB cache which is about 10-15 times faster than main memory. Since each key is four bytes, in both the algorithms, for $n \leq 2^{16}$ all the keys and additional data fit in cache. We see that for $n \leq 2^{16}$, the time per key grows more slowly for Flashsort1 than for Quicksort, however when the problem no longer fits in cache we see that the rate of increase in the time per key is much more for Flashsort1. This suggests that the performance of Flashsort1 is affected quite dramatically by the cache. It is clear that the RAM model may fail spectacularly to predict performance on modern computers. We need a model that considers both computation and memory access costs.

It is only fairly recently that improving cache usage has been identified as an important issue. For example, for many years the database community has been aware of the importance of improving algorithm and data structure design in order to reduce disk accesses, however it has only fairly recently been reported that in many standard database applications the bottleneck is main memory computation, caused by poor cache usage [10].

Movement of data between the disk and main memory is under software control, this may be through system or application software. This approach gives more flexibility than with hardware control but with relatively small overheads. The cost of an access to disk may be equivalent to a million CPU clock cycles (referred to as *cycles* in the future) and the cost of software control is amortised in the overall costs of the access. In contrast, movement of data between the main memory and cache(s) is always under hardware control. There are several reasons for this:

- Controlling the efficient movement of data between multiple levels of memory is a complex task, instead programmers like to develop software on a relatively simple model of the underlying machine architecture. Most programs are developed on the von Neumann architecture, which is modelled by the RAM model. This architecture assumes a single, sequentially addressed, memory. The programmer assumes that the operating system and compiler are responsible for the messy details of providing this virtualisation.

- By the time caches became widespread, a large number of applications had already been developed assuming the von Neumann architecture. If caches required programmer control then these applications would have to be rewritten in order to take advantage of the cache.

- The constant factors between main memory and CPU speeds are much less than those between disk and CPU speeds, so it is much more important to keep the overheads of using a cache low. Hardware control is much faster than any software control would be.

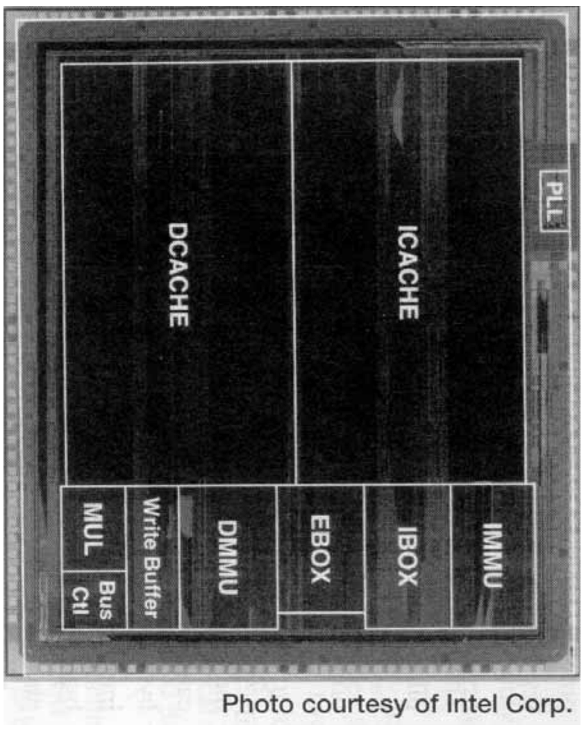

Figure 1.4: Photograph of the Intel StrongArm chip.

A large number of algorithmic techniques have been developed to improve locality in main memory accesses and so reduce accesses to disk. It seems reasonable to exploit these techniques to improve locality in caches and reduce accesses to main memory. However, since placement of data in a cache is hardware rather than software controlled, these techniques can not always be applied directly.

Caches not only improve speed of computation, but can also help with reducing power consumption, an important issue for handheld computing devices. The importance of the cache in the design of modern computer systems is demonstrated by the design of the Intel StrongArm chip. Figure 1.4 shows a photograph of this chip. This unit is used in many handheld computing devices such as PDA, and we see that a large proportion of the chip is occupied by two 16KB caches, the `DCACHE` and the `ICACHE`. Off-chip accesses to main memory consume more power than accesses to on-chip locations, and the large caches are used to improve locality and reduce off-chip accesses.

1.1 Internal memory hierarchy

We now give a brief overview of the memory hierarchy between the CPU and main memory.

1.1.1 Caches

A cache is a fast memory which holds the contents of some main memory locations. If the CPU requests the contents of a main memory location, and the contents of that location is held in some level of cache, the CPU's request is answered by the cache itself (a cache *hit*); otherwise it is answered by consulting the main memory (a cache *miss*). A cache hit has small or no cost (penalty), 1-3 CPU cycles is fairly typical, but a cache miss requires a main memory access, and is therefore very expensive. To amortise the cost of a main memory access in case of a cache miss, an entire *block* of consecutive main memory locations which contains the location accessed is brought into cache on a miss. Programs which have good *spatial locality*, where an access to a memory location is followed by an access to a nearby memory location, benefit from the fact that data are transferred from main memory to cache in blocks. While programs which have good *temporal locality*, where a memory location once accessed is soon accessed again, benefit from the fact that caches hold several blocks of data. Such programs make fewer cache misses and consequently run faster.

1.1.2 TLB

There is an important optimisation which can contribute as much or more to performance as minimising cache misses, namely minimising misses in the *translation-look-aside buffer (TLB)*. Some papers from the early and mid 90's (see e.g. [5, 66]) have noted the importance of minimising TLB misses when implementing algorithms, but there has been no systematic study of this optimisation, even though TLB misses are often at least as expensive as cache misses. A major contribution of this thesis is to study TLB misses systematically.

The TLB is used to support *virtual memory* in multi-processing operating systems [45]. Virtual memory means that the memory addresses accessed by a process refer to its own unique logical address space. This logical address space contains as

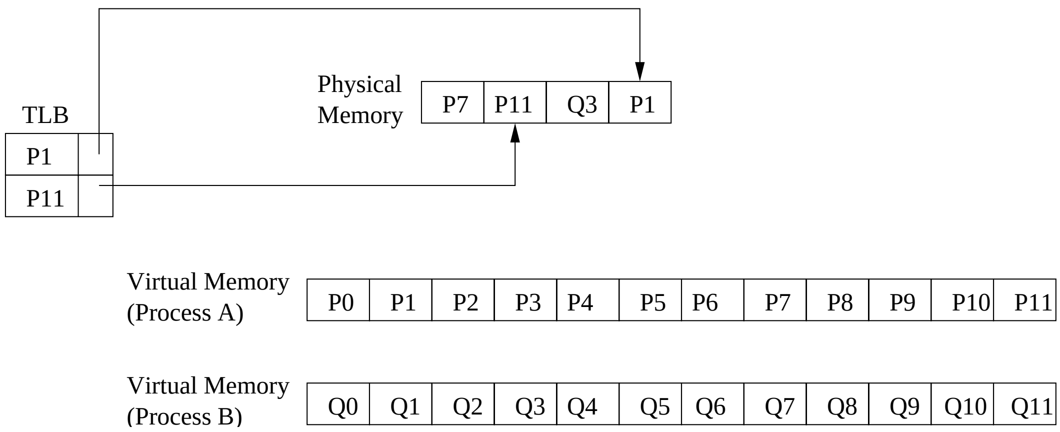

Figure 1.5: Virtual memory for processes A and B, each with 12 pages. Physical memory with 4 pages. TLB holds virtual (logical) page to physical page translations for 2 pages.

many locations as can be addressed on the underlying architecture, which far exceeds the number of physical main memory locations in a typical system. Furthermore, there may be several active processes in a system, each with its own logical address space. To allow this, most operating systems partition main memory and the logical address space of each process into contiguous fixed-size *pages*, and store only some pages from the logical address space of each active process in main memory at a time. Owing to its myriad benefits, virtual memory is considered to be "essential to current computer systems" [45]. The disadvantage of virtual memory is that every time a process accesses a memory location, the reference to the corresponding logical page must be *translated* to a physical page reference. This is done by looking up the *page table*, a data structure in main memory and would lead to unacceptable slowdown. In case the logical page is not present in main memory at all, it is brought in from disk. The time for this is generally not counted in the CPU times, as some other task is allowed to execute on the CPU while the I/O is taking place.

The TLB is used to speed up address translation. It is a fast memory which holds the translations of recently accessed logical pages. If a memory access results in a TLB hit, there is no delay, but a TLB miss can be significantly more expensive than a cache miss; hence locality at the page level is also very desirable. In most computer systems, a memory access can result in a TLB miss alone, a cache miss alone, neither, or both: algorithms which make few cache misses can nevertheless have poor performance if they make many TLB misses. Figure 1.5 shows large virtual memories for two processes, a smaller physical memory and a TLB which holds translations for a subset of the pages in physical memory.

The sizes of cache and TLB are limited by several factors including cost and speed [43]. Cache capacities are typically 256KB to 4MB, which is considerably smaller than the size of main memory. The size of TLBs are also very limited, with 64 to 128 entries being typical. Hence *simultaneous* locality at the cache and page level is needed for internal memory computation.

We refer to the caches, TLB and main memory of a computer system as the *internal memory hierarchy.* In this thesis we deal systematically with the internal memory hierarchy.

1.2 Sorting and searching

Caches and TLBs have been designed to exploit the locality exhibited in "typical" computer programs. In this thesis we study how some algorithms for fundamental problems such as sorting and searching behave with respect to the internal memory hierarchy. We show that algorithms that have good performance in the RAM model are "atypical", i.e. they naturally have poor TLB and/or cache performance. In this thesis we design new algorithms with improved performance.

1.2.1 Sorting random numbers

For many real-world problems we find that the data to be sorted is randomly distributed. We consider the problem of sorting random numbers independently drawn from a known distribution. The algorithms we use to solve this problem are 'in-place', in that they use relatively little auxiliary space in addition to the array holding the keys.

Many sorting algorithms are *comparison-based*, in that they compare keys in order to determine their relative order. Several cache-efficient comparison-based sorting algorithms already exist, examples are cache-tuned versions of Quicksort and Mergesort. Distribution sorting is a popular alternative to comparison-based sorting for independent random numbers. It involves placing n input keys into $k \leq n$ *classes* based on their value [53]. The classes are chosen so that all the keys in the ith class are smaller than all the keys in the $(i+1)$st class, for $i = 0, \ldots, k-2$, and furthermore, the class to which a key belongs can be computed in $O(1)$ time (e.g. if the keys are floats in the range $[a, b)$, we can calculate the class of a key x as $\lfloor \frac{x-a}{b-a} \cdot k \rfloor$). Thus, the original sorting problem is reduced in linear time to the problem of sorting the keys in each class. A number of distribution sorting algorithms have been developed which run in linear (expected) time under some assumptions about the input keys.

We analyse the cache misses in distribution sorting algorithms and use the analyses to design fast new algorithms. We compare these distribution sorting algorithms to 'in-place' comparison-based sorting algorithms.

When we refer to distribution sorting, we are interested in the following:

- the keys are independent random numbers,
- the algorithm is 'in-place'.

1.2.2 Sorting integers

We also consider the problem of sorting w-bit integers in the range $0, \ldots, 2^w - 1$. The input may be from any distribution including the worst-case. The algorithms that we consider are *stable*, in that equal value keys have the same relative order at the start and end of sorting. These algorithms are 'out-of-place', in that they use $O(n)$ additional space.

Radix sorting, applied to integers, consists in viewing w-bit integer keys as $\lceil w/r \rceil$ consecutive r-bit *digits*. The records are sorted in $\lceil w/r \rceil$ passes: in the i-th pass, for $i = 1, \ldots, \lceil w/r \rceil$ we sort the records according to the i-th least significant digit. There are a number of ways of implementing a single pass in $O(n + 2^r)$ time, the best one in practice being *counting sort* [32]. We follow current usage and refer to radix sort plus counting sort as 'LSB radix sort'.

LSB radix sort has an overall running time of $O(\lceil w/r \rceil (n + 2^r))$ for w-bit keys. For medium-to-large input sizes, LSB radix sort may be considered to be a linear-time algorithm for practical purposes. However, experimental work has shown that LSB radix sort, even when tuned for cache performance, fails to outperform good implementations of $O(n \log n)$ comparison-based algorithms on modern architectures [59].

We analyse the cache and TLB misses in LSB radix sort and design fast new algorithms. We compare the new algorithms to distribution and comparison-based sorting algorithms.

When we refer to the problem of integer sorting, we are interested in the following:

- the keys are integers,
- the keys may be from a non-random distribution,
- the algorithm is stable.

1.2.3 Searching

In the context of searching we consider the *dynamic predecessor* problem, which involves maintaining a set S of keys drawn from a totally ordered universe. The keys are maintained along with associated satellite data. We allow search, insert and delete operations on the set. A search operation takes a key q and returns the key x, along with its satellite data, such that x is the largest key in S satisfying $x \leq q$.

Search tree data structures are usually used to solve this problem. A binary search tree is known to lead to too many cache misses and the standard solution for two levels of hierarchical memory is to use the B-tree data structure [31, 53], which has nodes of higher degree and is consequently shallower than a binary search tree. The nodes are stored such that visiting each node incurs a small constant number of access to slower memory. With appropriately selected parameters, the B-tree has optimal cache misses, however its performance is still poor with respect to the TLB. We design new search tree data structures which simultaneously optimise cache and TLB misses and we compare these to B-trees.

1.3 Methodology

In this thesis we analyse algorithms on reasonably realistic models for the internal memory hierarchy. We use the analyses to obtain algorithms which are fast on modern computer systems. Our algorithms improve TLB and/or cache performance without an excessive increase in the instruction count. We show several instances where algorithms simultaneously optimised for the TLB and cache outperform algorithms optimised just for the cache. We are the first to systematically consider the TLB and cache together to improve the performance of internal memory algorithms.

Our methodology can be summarised as follows:

- Caches have complex architectures and are varied in their details, we ignore many of these details and give fairly simple and abstract models which capture the essential features of most caches and allow the memory behaviour of algorithms and data structures to be studied. We treat the TLB in the same abstract manner. The models we use are reasonably realistic, but simple and practicable.
- Assembly code and the CPU can make even very simple algorithms quite complex. So we again give simple abstract models of the behaviour of algorithms. One reason for having a simple model for algorithm behaviour is that a compiler will modify the details of the code and it is not always easy to determine what these modifications will be.
- We use a model which considers the TLB, cache and main memory to obtain better predictions of performance and to design faster algorithms than would be possible using a model which considers just the cache and main memory.
- Although many algorithms and techniques already exist for hierarchical memory, consisting of the disk and main memory, they are not directly applicable to the internal memory hierarchy. We adapt these algorithms and techniques for the internal memory hierarchy.
- We carefully balance the often conflicting demands of keeping instruction counts low, reducing cache misses and reducing TLB misses.
- We use the models to analyse sorting and searching algorithms. A precise analysis can often be quite complex and we demonstrate the usefulness of simple but accurate approximate analyses. Where appropriate, we use simulations to validate our analyses.
- We use our analysis to design algorithms which are fast in the models. We measure the running time of our algorithms to show that they are fast in practice and to validate the models and analyses.

1.4 Contributions of the thesis

The contributions of the thesis are as follows:

1. We give a model for the internal memory hierarchy of modern computer systems. Our model incorporates the cache, TLB and main memory.

2. We give emulation theorems to convert algorithms and analysis from other memory models to our model for the internal memory hierarchy.

3. We give two general techniques to obtain algorithms and data structures which are simultaneously optimal for the cache and TLB.

4. We give upper and lower bounds on the number of cache misses during the most time consuming phase of distribution sorting when the keys are drawn independently and randomly from a uniform distribution and when they are drawn independently and randomly from non-uniform distributions. We also give upper and lower bounds on the number of cache misses when accessing multiple sequences of data 'concurrently'.

5. We give a simple approximate analysis for the probability of a cache miss during the most time consuming phase of distribution sorting when the keys are drawn independently and randomly from a uniform distribution.

6. We demonstrate how to use our cache analysis, the cache miss penalty and an analysis of the instruction count in a distribution sorting algorithm to obtain a multi-pass variant of the algorithm which outperforms the original algorithm and highly optimised Quicksort and Mergesort.

7. We use our cache analysis to design an algorithm for distribution sorting when the keys are drawn independently and randomly from a non-uniform distribution. Our new algorithm outperforms highly optimised Quicksort and Mergesort.

8. We demonstrate the importance of reducing TLB misses as well as cache misses in the context of integer sorting. We give three techniques to simultaneously reduce cache and TLB misses in integer sort. Using these techniques we obtain several new integer sorting algorithms which all outperform highly optimised

Quicksort and Mergesort when tested on random integers independently drawn from a uniform distribution. The fastest of these algorithms is over twice as fast as highly optimised Quicksort and 2.8 times as fast as highly optimised Mergesort.

9. We show that standard integer sorting can have a very large number of cache and TLB misses in the worst-case. We give examples of input key sequences which cause this worst-case. We show that one of our fastest new integer sorting algorithms does not suffer this worst-case.

10. We further demonstrate the importance of reducing TLB misses and cache misses in the context of search trees. We show that a cache-tuned search tree is outperformed by cache and TLB tuned search trees, even when the latter are quite complex.

11. We also discuss the limitations of the models used in this thesis. We believe this will lead to future work in obtaining yet more precise but still practical models.

1.5 Organisation of the thesis

In Chapter 2 we review some important existing models for hierarchical memory, including one for cache memory. We also survey existing research in improving the performance of algorithms and data structures designed for the internal memory of computer systems. In Chapter 3 we introduce a model for the cache, TLB and main memory. We also give emulation theorems which allow algorithms designed on other memory models to be exploited on our memory models. In Chapter 4 we describe the problems of sorting and searching and we discuss in some detail existing results for these problems in internal memory and for external memory. We discuss distribution sort, integer sort, Mergesort and Quicksort. In the context of searching, we describe the B-tree data structure.

In Chapter 5 we give a precise cache analysis of a process which models the most time consuming phase in distribution sorting. The precise analysis is quite complex and we give an approximate analysis of the same phase in Chapter 6. In Chapter 7 we demonstrate how to design two distribution sorting algorithms, one for keys independently and randomly drawn from a uniform distribution and the other for keys independently and randomly drawn from a non-uniform distribution. We discuss an important tradeoff for internal memory algorithms, where we allow the number of cache misses to increase beyond the optimal in order to decrease the overall execution time.

In Chapter 8 we systematically analyse the TLB and cache misses in integer sorting. Again we demonstrate the tradeoff between minimising instruction count, cache misses and TLB misses in order to decrease the overall execution time. In Chapter 9 we design and implement several predecessor search data structures that are simultaneously optimal for the cache and TLB, and we show that all these outperform predecessor search data structures tuned just for the cache.

In Chapter 10 we consider the limitations of the models that we use for the internal memory hierarchy and in Chapter 11 we draw conclusions from the research presented in this thesis and suggest directions for future research.

Chapter 2

Previous work

There are several ways of measuring the number of cache or TLB misses that a program makes. However any measurement is for a particular execution of an algorithm and on a particular platform, and it may not tell us how an algorithm might behave on a different platform or with different parameters. Simple and precise analytical models allow us to study factors in an algorithm or in a machine platform that affect the number of cache and TLB misses and so allow us to predict behaviour as parameters vary.

Several models have been introduced to capture the memory hierarchy in a computer system and have been used to analyse algorithms and design new algorithms with improved performance with respect to the memory hierarchy. The models address the following issues:

- there may be two or more levels of memory,
- memories closer to the CPU are smaller and faster than memories further from the CPU,
- each memory has a limited capacity,
- in order to amortise the cost of access to slower memory, computer systems often transfer blocks of data between memory levels.

Algorithms designed on most of these models aim to reduce the number of accesses to the slower memories. Memory closer to the CPU is said to be at a higher level than memory further from the CPU. Since memories have limited capacities, when data is transferred to a higher level of the memory, some existing data in the higher level memory may have to be *evicted* in order to accommodate the new data. The strategy used to determine where to place the new data, and hence which existing data to evict, is referred to as the *replacement policy.* Some memories, such as main memory or disk, allow the programmer to implement a replacement policy, others enforce an *automatic* replacement policy that is controlled by the system, usually by the memory hardware. The replacement policy in a cache or TLB is always hardware controlled.

In this chapter we introduce several existing models for hierarchical memory and discuss their advantages and limitations, especially with regards to using them to model the internal memory hierarchy. We then introduce the *Cache Memory Model (CMM)* which has been used by several researchers to design and analyse algorithms for cache and main memory. This is one of the models that we will use extensively in this thesis. We discuss the advantages and limitations of this model.

We also review existing techniques and algorithms that reduce the number of accesses to main memory by utilising the cache, and in some cases the TLB, more effectively. Finally we review some research that demonstrates the affect of cache utilisation on power consumption.

2.1 Memory Models

In this section we introduce several existing models for hierarchical memory and discuss their advantages and limitations, especially with regards to using them to model the internal memory hierarchy. These models are of two broad types. In the first type of model, running times are expressed in terms of the parameters in the model, we refer to these as *discrete hierarchical memory models.* In the second type of model, running times are expressed in terms of the input size, we refer to these as *smooth hierarchical memory models.* In this section we also introduce a model specifically for cache memory.

2.1.1 Discrete hierarchical memory models

In this section we describe the *External Memory Model (EMM)*, also referred to as the *Single Disk Model*, and the *Parallel Disk Model (PDM).* These models are for a 2-level memory hierarchy with a fast main memory and a slow disk. In both these models blocks of data at consecutive memory location are moved between main memory and disk using explicit I/O operations. These models make the following assumptions:

- at the start and end of a program all data resides on disk,
- all computation is assumed to take place on data in the main memory,
- a block of data from disk can be placed at any block in main memory,
- blocks on disk can be accessed randomly.

The performance measures in these models are:

- the number of I/Os between main memory and disk,
- the number of disk blocks used.

Computation cost is usually not a performance measure, the assumption being that cost of I/O is much greater. Algorithms and data structures designed for these models aim to minimise the number of I/Os. An algorithm or data structures for a given problem is said to be *optimal* if it meets the lower bound for the number of I/O operations for that problem.

Both the models have the following parameters:

N = number of items in the problem,

M = number of items that can fit in internal memory,

B = number of items that can be transferred in a single block,

where $2 \leq B \leq M < N$. It is assumed that the disk size in unbounded.

For batched query processing problems, the size of the query and the size of the result are also performance measures, see for example [17]. So the following parameters are added to the models:

Q = number of items in an offline query,

Z = number of items in the query result.

The analytical approach in these models is to determine the number of I/Os for four fundamental problems: scanning N items, sorting N items, searching online through N data items and outputting Z items as the results for a query operation. The I/O performance of many algorithms and data structures can then be expressed in terms of the I/O bounds for these fundamental problems. A large number of algorithms and data structures have been designed on these models, see [17, 89].

External Memory Model (EMM)

In the EMM, introduced by Aggarwal and Vitter [8], there is a single disk which allows read or write operations to multiple blocks during one I/O operation. In this model the parameters above are augmented with:

D = number of blocks that can be transferred concurrently,

where $1 \leq D \leq \lfloor M/B \rfloor$.

Aggarwal and Vitter [8] give upper and lower bounds on the number of I/Os for the problems of sorting, matrix multiplication, computing FFT graphs, permutation networks and permuting in this model.

The model is generally considered to be unrealistic for $D \geq 2$ as most systems do not support multiple concurrent accesses to a single disk. We will occasionally use this model, but in our case $D = 1$.

Parallel Disk Model (PDM)

In the PDM, introduced by Vitter and Shriver [91], there are multiple disks which are accessed by multiple processors. The model is a more restrictive and realistic version of the EMM. The parameters above are augmented with:

D' = number of disks that can be accessed concurrently,

P = number of processors,

where $1 \leq D'B \leq M/2$. If $P < D'$ then there are D'/P disks per processor, if $D' < P$ then each disk is shared by about P/D' processors. The main memory size for each processor is M/P and the processors are connected by an interconnection network.

Since this model is a more restrictive version of the EMM, any lower bound results from the EMM apply in this model. The EMM can be simulated randomly on the PDM with an expected constant factor more I/Os [78], and deterministically with a factor of $O(\log(N/D')/\log\log(N/D'))$ more I/Os [18].

3-level External Memory Model (3-EMM)

The 3-EMM is a three-level external memory model, where there is a main memory and two levels of disk memories. It is assumed that there is a single disk in each level of the disk memories and concurrent accesses to the same disk are not allowed. The parameters for this model are as follows:

N = number of items in the problem,

M_1 = number of items that can fit in internal memory,

B_1 = number of items that can be transferred in a single block between internal memory and the first-level of disk memory,

M_2 = number of items that can fit in the first-level of disk memory,

B_2 = number of items that can be transferred in a single block between the first and second levels of disk memories,

where $2 \leq B_1 \leq B_2 \leq M_1 \leq M_2 < N$ and B_1, B_2, M_1 and M_2 are all powers of two. It is assumed that the second-level disk size is unbounded.

A k-EMM is a generalisation of the model for multiple levels of memory.

Criticisms of these models

The advantage of the EMM and PDM is their simplicity, which eases analysis and design of algorithms. Though the parameters can get quite unwieldy in a discrete hierarchical memory model with three or more memory levels. There are three main reasons why these models can not easily be used directly to analyse and design algorithms for internal memory. The first problem is the assumption that the algorithm can implement its own policy for replacing data in the faster memories, whereas in a cache or TLB this is strictly under hardware control. The second reason is that if an algorithm is tuned for both the cache and TLB then it needs to optimise for more than two levels of a memory hierarchy, whereas these models deal with strictly two levels. The final reason is that it is generally assumed that the cost of accessing the disk is so high that operation counts can essentially be ignored. This does not lead to the design of algorithms which minimise running time by taking both I/O and computation costs into account.

2.1.2 Smooth hierarchical memory models

In this section we describe various models that have been proposed for multiple levels of the memory hierarchy. Algorithms for fundamental problems such as sorting, computing FFT graphs and matrix multiplication have been described for these models.

Whereas in the EMM and PDM an algorithm or data structure is analysed in order to determine the number of I/Os between main memory and disk, these hierarchical memory models analyse the time for solving problems by using cost functions associated with accessing data at various levels of the memory hierarchy.

Hierarchical Memory Model (HMM)

The *Hierarchical Memory Model (HMM)* of Aggarwal et al. [6] is a RAM model where access to memory address x requires time $f(x)$, where typically $f(x) = \lceil \log x \rceil$ or $f(x) = x^\alpha$, for some $\alpha > 0$. Aggarwal et al. note that the cost function $f(x) = \lceil \log x \rceil$ seems suitable for semiconductor memories [64, pp. 316]. In the model there are an unlimited number of memory locations $x_1, x_2, \ldots$, each of which can hold a unit of data. The model allows standard RAM operations. A RAM operation involving addresses x and y takes time $f(x) + f(y) + O(1)$, i.e. the cost of accessing x and y and unit time for the operation. Efficient programming in this model requires data items to be copied from high addresses to low addresses during periods of high use.

Aggarwal at el. give upper and lower bounds on the time to solve the problems of sorting, computing a FFT graph, $n \times n$ matrix multiplication and searching in their model [6].

Hierarchical Memory Model with Block Transfer (HMM-BT)

The *Hierarchical Memory Model with Block Transfer (HMM-BT)* of Aggarwal et al. [7] is like the HMM but allows block transfers. This model is intended to capture the fact that accesses to a slower memory have a high latency followed by high throughput.

In this model the contents of l consecutive locations starting at x can be copied to l consecutive locations starting at y at a cost $\max(f(x), f(y)) + l$. Aggarwal et al. give upper and lower bounds on the time for scanning data, computing the dot product of two vectors, merging two sorted lists, performing a shuffle permutation, odd-even exchange permutation, rational permutations, arbitrary permutations, matrix transposition, computing FFT graphs, sorting and matrix multiplication [7]. For the cost function $f(x) = x^{\alpha}$, for some constant $0 < \alpha < 1$, they show that algorithms and their analyses on the Concurrent Read Concurrent Write PRAM model can be converted to equivalent algorithms and analyses on the HMM-BT model.

Vitter and Shriver [92] describe parallel versions of the HMM and HMM-BT models, where there are P memory hierarchies which are interconnected via a network and operate simultaneously. They describe parallel algorithms which are optimal in the hierarchical memory model for the problems of sorting, computing FFT graphs and matrix multiplication.

Uniform Memory Hierarchy Model (UMH)

In the uniform memory hierarchy model (UMH) of Alpern et al. [11] a memory at each level holds several blocks of data and the block size and number of blocks at each level grow at a uniform rate. Formally, a memory at level l has $\alpha\rho^l$ blocks each of size ρ^l, for integer constants $\alpha, \rho \geq 2$. The memory at level l is connected to memories at level $l-1$ and $l+1$ by a bus. A block at level l can be randomly accessed and transferred to or from level $l+1$ at a bandwidth of $b(l)$, so each transfer takes time $\rho^l/b(l)$. Typical values for the bandwidth at level l are 1, l or ρ^l. Computation takes place at level 0. A program is said to be *communication efficient* if it spends more time in computation at level 0 than in moving data between the other levels. Alpern et al. describe algorithms for matrix transpose, matrix multiplication and FFT on the UMH. For the above problems, they give *threshold* functions which separate bandwidth cost functions for which an algorithm is communication-efficient from those that are too costly.

Vitter and Nodine [90] describe a parallel version of the UMH, where there are P memory hierarchies which are interconnected via a network and operate simultaneously. They describe several optimal and near-optimal sorting algorithms on their model for a wide range of bandwidth functions, where the optimal time for sorting n items is $O(n \log n)$.

Criticisms of these models

Since these hierarchical memory models analyse the time for solving problems by using cost functions associated with accessing data at multiple levels of the memory hierarchy, analysis and design of algorithms is considerably more complex in these models than in the EMM or PDM, so few researchers have designed algorithms for these models.

As for the EMM and PDM, these models can not be used directly for caches and TLBs as they assume that the replacement policy is under programmer control, rather than hardware control.

Another criticism of these models is that they use 'fixed' costs functions which are not expressive enough to capture the true cost of a memory access in a system with hardware controlled replacement policies.

2.1.3 Cache-Oblivious Model

Algorithms designed for the models introduced in the previous sections need to know parameters of the memory hierarchy in order to tune performance for specific memories. Frigo et al. [38] introduced the notion of *cache-oblivious* algorithms, where algorithms do not have any parameters that tune it for a specific memory. They introduced a two-level hierarchical memory model called the *ideal cache model* or the *cache-oblivious model* for the analyses of these algorithms. In this model algorithms access an infinitely large slow memory, where some locations from the slow memory are stored in a fast memory. The fast memory is called an *ideal cache* and has the following characteristics:

- data is transferred between the slow memory and the ideal cache in blocks of B data items, where algorithms have no knowledge of the value of B,
- the ideal cache can hold C blocks, where algorithms have no knowledge of the value of C,
- the replacement policy in the ideal cache is *automatic*, this means that decisions about where to place a block of data from the slow memory is not under the

control of the algorithm, instead it is under the control of hardware or system software,

- the ideal cache is *fully-associative*, meaning that a block of data from the slow memory can be placed in any block in the ideal cache,
- the ideal cache uses an *optimal offline* replacement strategy of replacing the cache block whose next access is furthest in the future.

It is generally assumed that the ideal cache is *tall*, meaning $C = \Omega(B)$. The performance measures in this model are:

- *cache complexity*, the number of data blocks transferred from the slow memory to the ideal cache, expressed as a function of C and B,
- *work complexity*, the number of instructions.

The assumptions in the ideal cache may seem far removed from reality. For example, in real computing systems there may be multiple levels of cache between the CPU and main memory; clearly the cache can not use an optimal offline policy to decide which block to evict in order to accommodate a new block of data, in reality it may select the least-recently used (LRU) block or a random block for replacement; a main memory block can usually only be placed in one of a few cache blocks rather than any cache block. However Frigo et al. show how each restriction can be removed in practice. They prove that an algorithm that is optimal for two levels of the memory hierarchy has optimal data transfers between each level of a multiple level memory hierarchy. Using a result by Sleator and Tarjan [81], see Theorem 3.1 in Chapter 3, they show that an algorithm executed on a cache using an LRU replacement policy has no more than a constant factor more data transfers between cache and main memory than if it was executed on a cache with an optimal offline replacement strategy. They also show that, using a hash table and doubly linked lists, a fully-associative cache with an LRU replacement policy can be implemented using $O(C)$ main memory locations such that the cache can be accessed in $O(1)$ expected time.

Frigo et al. describe algorithms for FFT, rectangular matrix transpose and sorting which have optimal work and cache complexities when analysed on the ideal cache. Bender et al. describe optimal cache-oblivious B-trees [21]. Brodal et al. describe optimal cache-oblivious search trees [25]. Bender et al. describe a locality-preserving cache-oblivious dynamic dictionary [22]. Arge et al. describe optimal cache-oblivious priority queues with applications to graph algorithms [14]. Bender et al. describe cache-oblivious exponential search trees [20]. We describe the design and implementation of a cache-oblivious exponential search tree in this thesis.

The advantages of the ideal cache model are that it is portable, elegant and avoids explicit I/O. Another advantage is the fact that an optimal algorithm designed on a two-level memory is also optimal for a multi-level memory.

Criticisms of the model

The cache-oblivious model could be considered the most suitable for the design and analysis of algorithms for internal memory. However this model has the following disadvantages:

- The use of a hash table, and hash function, to maintain a fully-associative cache, means that the algorithms are randomised rather than deterministic. Though in practice this may not be a major issue as most systems have virtual memory which uses hashing to access page tables.
- Algorithms designed for this model seem to be considerably more complex than algorithms designed on most other memory models. This could lead to a high operation count, which may lead to algorithms which are slower than those which are carefully tuned for a given internal memory hierarchy.

Some practical implications of algorithms designed on the cache-oblivious model are demonstrated by the results of a study by Chatterjee and Sen [28]. They show that for large matrices, the cache-oblivious matrix transpose algorithm applied to a matrix with a canonical layout (such as column-major or row-major) has up to 5.7 times as many cache misses and up to 60 times as many TLB misses as a recursive architecture-aware matrix transposition algorithm performed on a matrix with a non-linear layout. They state that the cache-oblivious algorithm's poorer performance can partly be attributed to the facts that it is highly recursive and to the fact that the algorithm is analysed and designed for a fully-associative cache. As another example, we show that highly optimised cache-oblivious search trees outperform B-trees, but they are in turn outperformed by cache and TLB aware search trees. This is in part due to the complexity of the cache-oblivious search algorithm.

2.1.4 Cache Memory Model (CMM)

Several researchers have used a model where there is a random access machine, consisting of a CPU and main memory, augmented with a cache [24, 56, 59, 65, 77, 80]. We will use this model fairly extensively in this thesis.

The model has the following parameters:

n = the number of items in the problems,

B = the number of data items in a cache block,

C = the number of blocks in the cache,

S = the number of *sets* in the cache,

a = the *associativity* of the cache,

and the performance measures in this model are:

- the number of cache misses,
- the number of instructions.

For the cache, we view main memory as being partitioned into equal-sized *blocks* of locations, each consisting of B memory words. Blocks are *aligned*, that is, they begin at addresses which are congruent to 0 modulo B. The model assumes a single cache consisting of S *sets* each consisting of a *lines*. Each line can hold a memory block, and memory block i can be stored in any of the lines in set $(i \bmod S)$. We denote by $C = aS$ the capacity of the cache, and a is called the *associativity* of the cache. When $a = 1$, the cache is said to be *direct-mapped*, and when $a > 1$, we say the cache is *a-way* associative. If the program accesses a location in block i, and block i is not in the cache, then one of the blocks in the set $(i \bmod S)$ is *evicted* or copied back to main memory, and block i is copied into the set in its place. We assume that blocks are evicted from a set on a *least recently used (LRU)* basis. Note that for a direct-mapped cache, there is only one block in each set, and the replacement policy becomes trivial.

The advantage of counting instructions and cache misses separately is that it allows the use of the coarse O-notation for simple operations and also to analyse the number of cache misses more carefully, if necessary.

Criticisms of the model

The CMM is useful for *virtual* caches, where a virtual memory address can be used to search the cache. However most caches are *physical*, meaning they can only be searched using a physical address, and these caches require an address translation, which accesses the TLB, before checking for data. The model does not address hits and misses for TLB accesses.

Another criticism is that the above model simplifies the architecture of real machines considerably. For instance, the experiments in this thesis are performed on a Sun UltraSparc-II, which supports two levels of cache, as do most other current architectures. However, in the UltraSparc-II, the L1 (faster, smaller) cache is *write-through*, i.e., in case of a cache hit during a write, the value of the memory location is simultaneously updated in the L1 cache and in the L2 (slower, larger) cache. Since most cache misses in the algorithms we consider occur during writes to memory locations, this simplification is reasonable in our case. In Chapter 10 we discussion several other features of caches that the model fails to capture.

Classifying cache misses

Hill and Smith [49] classified cache misses as *compulsory* misses, *capacity* misses and *conflict* misses. These are as follows:

- A compulsory miss occurs on the very first access to a memory block, since the block could not have been in cache.
- A capacity miss occurs on an access to a memory block that was previously evicted because the cache could not hold all the blocks being actively accessed by the CPU.
- If all a ways in a cache set are occupied then an access to a memory block that maps to that set will cause a block in the set to be evicted, even though there

may be unused blocks in other cache sets. The next access to the evicted block will cause a conflict miss.

As an example, suppose we have an empty direct-mapped cache with C blocks, where each block holds B items and we have an array DATA with $n = 2CB$ items. In the direct-mapped cache memory address x is mapped to cache block $(x \text{ div } B) \bmod C$. If we sequentially read all items in DATA then we have n/B compulsory misses. The first access to DATA causes B items to be loaded into a cache block, then after B accesses another B items are brought into the cache. If we sequentially read all items in DATA again, then we have n/B capacity misses. Now suppose the cache is empty and we read DATA[0] and then DATA[CB], repeatedly $n/2$ times. DATA[0] and DATA[CB] map to the same cache block, so we have 2 compulsory misses, for the first accesses to DATA[0] and DATA[CB], and $n - 2$ conflict misses.

Clearly a fully-associative cache will not have conflict misses, as all blocks in the cache are in a single set, so if a block is evicted then we have a capacity miss. Similarly, conflict misses do not occur in the EMM.

2.2 Algorithms for internal memory

We now review existing techniques and algorithms to improve cache and TLB efficiency. Most of these are for caches, though a few researchers have noted the importance of the TLB. We first review general techniques and then algorithms for specific problems.

As stated earlier, a program is said to have good *temporal locality* if memory addresses once accessed are accessed again in the near future. A program is said to have good *spatial locality* if accesses to memory addresses are followed by accesses to nearby memory addresses. Programs that exhibit good temporal and spatial locality generally have fewer misses. Most of the techniques and algorithms we discuss aim to improve temporal or spatial locality in a program. A few address the problem of reducing conflict misses, which may occur even if a program has good locality.

2.2.1 System support

There are many hardware techniques for improving the cache performance of computer systems. These techniques aim to either reduce the cache miss rate or to reduce the penalty of a cache miss or to reduce the time for processing a cache hit. See Appendix B or the books [45] and [43] for further details.

Several techniques have been used by compilers to improve the cache performance of algorithms [45]. Examples are:

- Pre-fetching is used to load data into cache before the data is needed, thus reducing CPU stalls.
- If multiple arrays are accessed in the same dimension with the same indices at the same time then this can cause conflict misses. By merging elements from each array into an individual structure, which resides in one cache block, the conflict misses are avoided. This improves spatial locality.
- If nested loops access data in a non-sequential order then spatial locality may be improved by reordering the nesting of the loops such that data is accessed sequentially.
- If the same array elements are accessed in multiple loops, then temporal locality can be improved and cache misses can be reduced by fusing the loops into one.
- In algorithms which manipulate multi-dimensional arrays, such as matrix multiplication, the technique of operating on tiles of the arrays, rather than rows or columns, can be used to reduce cache misses.

Most of these compiler techniques can also improve the TLB performance of algorithms.

Accessing multi-dimensional arrays in a tiled manner is a technique that has commonly been used to reduce the number of accesses to slower memory in a system with multi-level memory hierarchies. Lam et al. [57] did cache simulations and analyses of the misses in tiled matrix multiplication and showed that unless the tile size is much less than the cache size there can be a large number of conflict misses. Temam et al. [84] also suggested blocking techniques which take into account conflict misses in cache memory.

Most of the above compiler techniques are for programs with regular data access patterns. More recently Chilimbi et al. [29, 30] have suggested compiler controlled techniques such as:

- *clustering*, which places structure elements which are likely to be accessed in succession in the same cache block,
- *colouring*, which segregates heavily and infrequently accessed elements in non-conflicting cache regions,
- *compression*, which reduces structure size or separates the active and inactive portions of structure elements

to improve the spatial and temporal locality of pointer-based data structures. They also propose a cache conscious `malloc` routine, a cache conscious garbage collector and a runtime memory re-organiser.

By considering the memory allocation algorithm used by the `gnu malloc` routine, Zhang and Martonosi [97] analyse the cache misses during the construction and traversal of linked-list and binary search tree data structures.

Romer et al. [74] have suggested using an online page size selection policy to improve TLB performance.

2.2.2 Algorithms

Matrix Algorithms

Several researchers have looked at the problems of matrix multiplication, matrix transposition, FFTs and permutations in cache memory [23, 28, 34, 44, 76, 98]. Venkataraman et al. [88] give a blocked algorithm to solve the all-pairs shortest-paths problem.

Sorting

Nyberg et al. [68] describe the AlphaSort algorithm which sorts records stored on disk. In order to obtain good performance, they note the importance of minimising the cost of tasks completed in internal memory. They note that replacement-selection tree sort, a commonly used internal memory sorting technique, has poor cache behaviour, whereas Quicksort, due to its sequential memory accesses, has good cache behaviour. They suggest organising the nodes in the replacement-selection tree such that parent and children are in the same cache block, they report that this reduces cache misses by a factor of 2 or 3. Alphasort uses Quicksort to generate small sorted lists of data, and then merges these using replacement-selection tree sort. For large records, they state that sorting keys and pointers to records has better cache performance than sorting records.

LaMarca and Ladner [58, 59] evaluate the cache misses in Quicksort, Mergesort, Heapsort and Radix sort algorithms using cache simulations and analysis. To reduce cache misses in Heapsort they suggest using d-ary heaps, where the root node occupies the last element of a cache block, rather than the normal binary heaps. For Mergesort they suggest using tiling and k-way merging, see Chapter 4. For Quicksort they suggest using a multi-partition approach, see Chapter 4. For Quicksort they also suggest using Sedgewick's technique of stopping the partitioning when the partition size is a few elements [79], but they suggest insertion sorting these small partitions when they are first encountered, rather than in one single pass over the data after all small partitions have been created. They report that Radix sort has poor cache performance, even though they did not analyse an important source of conflict misses, due to concurrent accesses to multiple locations in the destination array. Their new algorithms are derived almost directly from EMM algorithms and hence reduce capacity misses, but not conflict misses. They show that their algorithms outperform non-memory tuned algorithms on their machine.

Xiao et al. [96] observed that the tiled and k-way Mergesort algorithms in [59] can in the worst-case have a large number of conflict cache misses. They also observed that k-way Mergesort can have a large number of TLB misses. They suggested padding subarrays that are to be merged in order to reduce cache misses in tiled Mergesort and cache and TLB misses in k-way Mergesort. They give equations for fewer worst-case number of misses in the algorithms which use padding. Using cache and TLB simulations they show that the algorithms which use padding have fewer cache misses on several machines. They also show that the new mergesorts outperform existing Mergesort implementations.

Agarwal [5] notes the importance of reducing both cache and TLB misses in the context of sorting randomly distributed data. In their bucket sort implementation they choose the number of buckets to be less than the number of TLB entries.

Mehlhorn and Sanders [65] give upper and lower bounds on the number of cache misses in a set-associative cache when accessing multiple sequences of data at the same time. In their model an adversary determines when a sequence is accessed and it is assumed that the start of the sequences map to cache blocks that are randomly distributed in cache. They give a bound on the number of sequences, dependent on the cache associativity, block size and cache capacity, that can be accessed such that the number of conflict misses is of the same order as the number of compulsory misses. Their sequences accesses model the accesses to data in k-way Mergesort and multi-partition Quicksort. More generally, they suggest that when EMM algorithms that access multiple sequences concurrently are used in the CMM, their analysis be used to select parameters such that the number of conflict misses is low. They extend their analysis to include accesses to another work array, which models accesses to a count array in counting sort, and they give upper bounds on the number of cache misses.

Searching

Acharya et al. [1] note that in a *trie* [53], a search tree data structure, the nodes nearest the root are large and those further away are increasing smaller. For large alphabets they suggest using

- an array which occupies a cache block for a small node,
- a bounded depth B-tree for a larger node,
- a hash table for a node which exceeds the maximum size imposed by the bounded depth B-tree.

The B-trees and hash tables hold data in arrays. They also note that only one pointer, to a child node, is taken out of a node, whereas several keys may be compared, so they suggest storing keys and pointers in separate arrays. For nodes with small alphabets they again suggest storing the keys and pointers in separate arrays, where the pointers are indexed using the characters in the alphabet. They experimentally evaluate their tries against non-memory tuned search trees and report significant speed improvements on several machine. Using cache simulations for several machine, they show that their data structures have significantly fewer cache misses than non-memory tuned search trees.

Priority Queues and Heaps

Sanders [77] notes that most EMM priority queue data structure have high constant factors which means they would not perform well if adapted to the CMM. He describes the *sequence heap* EMM data structure which has smaller constant factors in terms of the number of I/Os and space utilisation than other EMM priority queue data structures. The lower constant factors make sequence heaps very suitable for the CMM. This data structure uses k-way merging, so before it is used in the CMM, the results in [65] are applied to select k appropriately for the cache parameters. On random 32-bit integer keys and 32-bit values and when the input is large, sequence heaps are shown to outperform implicit binary heaps and aligned 4-ary heaps on several different machine architectures.

Bojesen et al. [24] analyse the cache misses during heap construction on a fully-associative cache. They find that Floyd's [36] method of repeatedly merging small heaps to form larger heaps has poor cache performance and that Williams' [94] method of repeated insertion performs better. They give new algorithms using repeated insertions and repeated merging which have close to the optimal number of cache misses. They note that divide and conquer algorithms are good for hierarchical memory. They note that by traversing a tree in depth-first order rather than breath-first order improves locality. They also note the importance of code tuning in order to obtain good performance in the CMM.

More complex models

Several researchers have developed models to analyse the execution time of an algorithm by considering the number of instructions and the cache or TLB miss penalty [88, 96].

2.3 Cache misses and power consumption

Power consumption is an important consideration in the design of computer systems, especially for mobile computing. Since caches consist of a large amount of the CPU chip they are a good target for trying to reduce power consumption. Kaxiras et al. [52] note that a cache block has a flurry of activity when data is first brought into the cache block and then it has a period of inactivity, during which there can be power leakage. They discuss policies for switching cache blocks off when they hold data that is not likely to be reused. Joseph et al. [51] report that power consumption on the Intel Pentium Pro increases as the data cache, see Appendix B, hit rate increases and it reaches a peak when the hit rate is approximately $80 - 85\%$. This is because as the cache hit rate increases the CPU performs more instructions per cycle which require additional power. When considered as a tradeoff between performance and power usage they report that the increased power consumption is cost effective. They also report that while reading data from the cache, power consumption on the Pentium Pro is dependent on the number of bits set to 1 in the value read from a cache block.

2.4 Summary

In this chapter we have described several models for hierarchical memory and a model specifically for the cache and main memory of a computer system. We have also reviewed existing work in improving the cache and TLB performance of algorithms and data structures.

Chapter 3

Internal Memory Model and emulations

The CMM introduced in Section 2.1.4 in Chapter 2 is useful for *virtual* caches, where a virtual memory address can be used to search the cache. However most caches are *physical*, meaning they can only be searched using a physical address, and these caches require an address translation before checking for data. A page table, held in main memory, stores the translations for virtual to physical page addresses, but accessing the page table for each memory access would be prohibitively slow. The TLB is used to speed up address translations. It is a small, fast, associative memory, which holds the translations of recently accessed pages. In this chapter we introduce the *Internal Memory Model (IMM)* which extends the CMM to take account of virtual memory and the TLB. We also give two emulation theorems that allow algorithms and analyses from other memory models to be converted to equivalent algorithms and analyses in the IMM. We give two general techniques to obtain algorithms that simultaneously minimise cache and TLB misses.

3.1 Internal Memory Model (IMM)

As discussed in Section 2.1.4, the CMM models one level of cache memory with the parameters n, C, B, S, a and it counts the number of instructions and cache misses that an algorithm makes. We extend the CMM to take account of virtual memory and the TLB by adding the following parameters:

P = the number of data items in a page of memory,

T = the number translations held by the TLB,

and we add the following to the performance measures:

- the number of TLB misses.

For the TLB, we consider main memory as being partitioned into equal-sized and aligned *pages* of P memory words each. A TLB holds address translation information for at most T pages. If the program accesses a memory location which belongs to (logical) page i, and the TLB holds the translation for page i, the contents of the TLB do not change. If the TLB does not hold the translation for page i, the translation for page i is brought into the TLB, and the translation for some other page is removed, again on a least-recently-used (LRU) basis.

We assume that TLB misses and cache misses happen independently, in that a memory access may result in a cache miss, a TLB miss, neither, or both. This is because caches are usually *physically-tagged*, i.e., the values stored in cache are stored according to their physical, and not their logical, memory addresses. Hence, when a program accesses a memory location using its logical address, the address first has to be translated to a physical address before the cache can be checked. Furthermore, a memory access which results in both a cache miss and a TLB miss pays the cache miss penalty plus the TLB miss penalty (there is no saving, or loss, if both kinds of misses occur simultaneously).

We make a number of assumptions regarding the parameter values to simplify the analyses. These assumptions normally hold in practice, as explained below (see [45] for further details). The first assumption is that B, C, P and T are all powers of two. The remaining assumptions are:

(1)	$B \leq P$	Data is transferred from secondary memory as a page, and since secondary memory has much higher latency than main memory a page is much larger than a block. Indeed we should assume that $B \ll P$.
(2)	$T \leq C$	A TLB is fully-associative. This makes the hardware realisation of a TLB much more complex than a cache with limited associativity. Furthermore, a TLB hit must be serviced very quickly. These two factors imply that a TLB must be much smaller than a cache, and we should assume that $T \ll C$.
(3)	$BC \leq PT$	$PT < BC$ implies that some portion of cache will always lead to a TLB miss, thus effectively wasting part of cache. However, $PT > BC$ makes sense and does occur in practice.
(4)	$T \leq P$, $B \leq C$	These are technical assumptions, which are similar to the *tall cache* assumption of [38] and appear to hold in practice (strictly speaking, we make a 'tall cache' and 'short TLB' assumption).

3.1.1 Criticisms of the model

Again, this model is a simplification of real TLBs. In principle a TLB miss can be much more involved than a cache miss, requiring software intervention. Hence some architectures may provide hardware support for handling TLB misses. For example, on the Sun UltraSparc-II, a TLB miss can cost any of: (i) a L2 cache *hit* (2-3 CPU clock cycles) (ii) a memory access ($\approx$ 30-100 cycles) or (iii) a trap to a software miss handler (hundreds of cycles) [86, Chapter 6]. Another simplification is that TLBs almost always implement an approximation to LRU replacement, rather than a true LRU policy.

3.1.2 Designing and evaluating internal memory algorithms

Algorithms designed in the CMM and IMM aim to reduce the number of TLB and/or cache misses without an excessive increase in the number of instructions. We say that an algorithm is *cache-efficient* in comparison with some other algorithm if it makes fewer cache misses. We say that it is *cache-optimal* if the number of cache misses meet the asymptotic lower bound for I/Os in the EMM for that problem. We similarly define algorithms to be *TLB-efficient* or *TLB-optimal.*

Analyses in the CMM and IMM are usually backed up with experimental evaluations. Running times are used because:

- Cache misses, TLB misses and instruction counts do not tell us the running times of algorithms.
- The relative miss penalty is much lower for caches or the TLB than for disks, so constant factors are important. We find that asymptotic analysis is not enough to determine the performance of algorithms.
- The models simplify the architecture of real machines considerably, see Appendix B for a detailed discussion of cache architectures. Experimental evaluations are used to validate the model.

We also use simulations because:

- As we will see in Chapter 5, precise analysis in these models is often quite difficult, so an approximate analysis may be used, which again has to be validated with empirical techniques, such as measuring TLB and/or cache misses.

3.2 Sun UltraSparc-II

In this thesis we present extensive experimental results on our Sun UltraSparc-II and we now briefly describe the characteristics of this machine. Our Sun UltraSparc-II has 2×300 MHz processors and 1GB of main memory. There is a 16KB direct-mapped L1 cache with 32-byte blocks and a 1 cycle hit time. This cache holds the data that a program uses. There is a separate 16KB L1 cache for program instructions, but since most of the programs that we use are very small and fit into cache we have not considered the issue of caching instructions, so we can ignore the characteristics of this cache. There is also a 512KB direct-mapped L2 cache with 64-byte blocks and a 2-3 cycle hit time. The L2 cache holds instructions and data but since our programs are very small we assume that the whole L2 cache is available for data. The L2 cache has a 30 cycle miss penalty. The L2 cache is inclusive of the L1 cache, meaning that all entries held in the L1 caches are also in the L2 cache. As mentioned earlier, the L1 cache is write-through, so if the CPU modifies data held in the L1 cache then the data is updated in the L2 cache.

The machine has 64 fully-associative TLB entries and the page size implemented by the operating system is 8KB. A TLB hit has no penalty. The UltraSparc-II provides hardware support for "caching" translations evicted from the TLB in a *Translation Storage Buffer (TSB).* The is an address translation table held in main memory. The TSB can hold between 1K and 128K translations and its contents may be held in the L2 cache. If the required address translation is in the TSB then the TLB miss penalty can be a few cycles (if the TSB is held in the L2 cache) or about 30-100 cycles (if the TSB is not held in the L2 cache).

Since most of the experiments on this machine were performed on 32-bit single-precision floating-point numbers or on 32-bit integers, we now express the machine parameters in the CMM and IMM for this data size. So, for our Sun UltraSparc-II machine we have that:

$C = 8192$,

$S = 8192$,

$a = 1$,

$B = 16$, the number of 32-bit data elements that fit in a 64-byte cache block,

$P = 2048$, the number of 32-bit data elements that fit in a 8KB page,

$T = 64$.

3.3 Emulations

A large number of algorithms and data structures have been designed for the EMM, see the survey papers [17] and [89], which could potentially be used on the CMM and

IMM. However in the EMM block placement in fast memory is under the control of the algorithm whereas in the IMM it is hardware controlled. This can lead to conflict misses which would not have been considered in the EMM. Gannon and Jalby [40] showed that cache conflict misses could be avoided by copying frequently accessed non-contiguous data into contiguous memory. This ensures that the non-contiguous data are mapped to their own cache block. Lam et al. [57] used this technique to reduce conflict misses in blocked matrix multiplication.

Using the technique suggested by Gannon and Jalby [40], Sen and Chatterjee [80] formalise the statement that an EMM algorithm and its analysis can be converted to an equivalent algorithm and analysis on the CMM. We now give two theorems that allow algorithms and their analysis to be converted from the EMM to the IMM and from a 3-EMM to the IMM. In the proofs of our theorems, we make extensive use of the following, which are restatements of a result by Sleator and Tarjan [81]:

Theorem 3.1 *(i) For any given sequence of memory accesses, a TLB which has size τ and uses LRU replacement makes at most $\tau/(\tau - \tau^* + 1)$ times as many misses as a (hypothetical) TLB of size $\tau^* \leq \tau$ which knows the memory access sequence and follows the optimal offline replacement policy for that sequence. This assumes that both the LRU and the optimal TLBs are initially empty.*
(ii) For any given sequence of memory accesses, an α-way associative cache which uses LRU replacement makes at most $\alpha/(\alpha - \alpha^* + 1)$ times as many misses as a (hypothetical) α^*-way associative cache with the same number of sets, which knows the memory access sequence and follows the optimal offline replacement policy for that sequence. This assumes that $\alpha^* \leq \alpha$ and that both the LRU and the optimal caches are initially empty.

3.3.1 Emulation of EMM algorithm

We now show how an EMM algorithm and its analysis can be converted to an equivalent algorithm and analysis on the IMM.

Let $\mathbf{C}(a, B, C, P, T)$ be the IMM model described in Sections 2.1.4 and 3.1, where a is the associativity of the cache, B is the block size, C is the capacity of the cache in blocks, P is the page size and T is the number of TLB entries. Recall that the model assumes that $BC \leq PT$.

Let $\mathbf{I}(B, M)$ denote the following model, which is the EMM discussed in Section 2.1.1, specialised to disallow parallel disk access. There is a fast main memory, which is organised as M/B blocks $m_1, \ldots, m_{M/B}$ of B words each and an unbounded secondary memory, which is organised as blocks $d_1, d_2, \ldots$ of B words each as well. An algorithm in this model performs computations on the data in main memory, or else it performs an *I/O step*, which copies a specified block from main memory into a specified block in secondary memory or vice versa.

Theorem 3.2 *An algorithm A in the $\mathbf{I}(B, BC/2)$ model which performs I I/Os and t operations on data in main memory can be converted into an equivalent one A' in the $\mathbf{C}(a, B, C, P, T)$ model which performs at most $O(t + I \cdot B)$ operations, $O(I)$ cache misses and $O(I)$ TLB misses.*

PROOF. As noted by Sen and Chatterjee in [80], an algorithm A designed for the $\mathbf{I}(B, BC/2)$ model can be emulated by an algorithm A' in the $\mathbf{C}(a, B, C, P, T)$ model, such that A' incurs at most $O(1)$ (amortised) cache misses for every I/O performed by A and performs at most $O(t + I \cdot B)$ operations. Their emulation is quite simple: an array $Main$ of size $C/2$, each entry of which can hold B words, emulates the main memory of A, with m_i corresponding to $Main[i]$. The secondary memory of A is emulated by another array D, each entry of which can also hold B words, and an I/O operation is emulated by copying $Main[i]$ to $D[j]$ or vice versa. In order to avoid conflict misses during copying if $Main[i]$ and $D[j]$ map to the same set, the copying may be done through an intermediate buffer of B words. Two such buffers may be needed; it suffices to extend $Main$ by two elements and use $Main[C/2 + 1]$ and $Main[C/2 + 2]$ as these buffers. The copying requires an additional $O(IB)$ operations.

The (extended) array $Main$ occupies at most $T/2 + 1$ pages. If we assume that the TLB has only $T/2 + 1$ entries but that TLB replacement is done by an optimal offline algorithm, the optimal offline algorithm need never make more than 2 misses for each I/O performed by A—it can simply swap in the appropriate page for the D array, evicting a $Main$ page which will not be needed immediately, and swap back the $Main$ page just evicted at the expense of the D array page once the copying is complete. Since a LRU TLB with T entries never makes more than twice as many misses as an optimal TLB with $T/2 + 1$ entries on a given sequence of page accesses (Theorem 3.1), it follows that the emulation makes $O(I)$ TLB misses over the course of the emulation, if A performs I I/O operations. $\square$

3.3.2 Emulation of 3-EMM algorithms

We now show how a 3-EMM algorithm and its analysis can be converted to an equivalent algorithm and analysis on the IMM.

Let $\mathbf{C}(a, B, C, P, T)$ be the model described in Sections 2.1.4 and 3.1, where a is the associativity of the cache, B is the block size, C is the capacity of the cache in blocks, P is the page size and T is the number of TLB entries. Let $\mathbf{I}(B, M, B', M')$ denote the following model. Which is the 3-EMM discussed in Section 2.1.1. There is a fast main memory, which is organised as M/B blocks $m_1, \ldots, m_{M/B}$ of B words each, a secondary memory which is organised as M'/B' blocks $d_1, d_2, \ldots, d_{M'/B'}$ of B' words each, and a tertiary memory, which is organised as an unbounded number of blocks $t_1, t_2, \ldots$ of B' words each as well. We assume that $B \leq B'$ and that B and B' are both powers of two. An algorithm in this model performs computations on the data in main memory, or else it performs an *I/O step*, which either copies a block of size B to or from main and secondary memory, or copies a block of size B' to or from secondary and tertiary memory.

Theorem 3.3 *An algorithm A in the $\mathbf{I}(B, BC/4, P, PT/4)$ model which performs I_1 I/Os between main and secondary memory and I_2 I/Os between secondary and tertiary memory, and t operations on data in main memory can be converted into an equivalent one A' in the $\mathbf{C}(a, B, C, P, T)$ model which performs at most $O(t + I_1 \cdot B + I_2)$ operations, $O(I_1 + I_2)$ cache misses and $O(I_2)$ TLB misses.*

Proof. The emulation assumes that the contents of tertiary memory are stored in an array $Tert$, each of whose elements can store B' words. The emulation maintains an array $Main$ of size $C/4 + 2$, each entry of which can hold B words. The first $C/4$ locations of $Main$ emulate the main memory of A, with m_i corresponding to $Main[i]$, for $i = 1, \ldots, C/4$. As discussed above, the last two locations of $Main$ are buffers that are used for copying data to avoid conflict misses. For convenience, we assume that the arrays $Main$ and $Tert$ are aligned on cache block and page boundaries respectively.

We now assume a TLB with $T/2+2$ entries, but with a replacement strategy that we specify. By definition, the optimal (offline) replacement strategy will make no more misses than our strategy. Then we note that an LRU TLB with T entries never makes more than a constant factor more misses than an optimal (offline) TLB with $T/2+2$ entries, see Theorem 3.1, and so the LRU TLB with T entries makes no more than a constant factor more misses than our replacement strategy. Our TLB replacement strategy will set aside $T/4$ TLB entries $\delta_1, \ldots, \delta_{T/4}$ to hold translations for the blocks which are currently in secondary memory. In particular, if (tertiary) block t_i is held in (secondary) block d_j then δ_j holds the translation for the page containing $Tert[i]$. Also, our TLB replacement strategy will never replace the translations for the pages containing $Main$. Since $Main$ occupies at most $(C/4+2)B \leq (T/4+2)P$ words, only $T/4+2$ TLB entries are required for this.

A read from t_i to d_j (tertiary to secondary) is simulated by "touching" the page containing $Tert[i]$ (e.g. reading the first location of $Tert[i]$). Our TLB replacement algorithm brings in the translation for $Tert[i]$ and stores it in TLB buffer δ_j. This causes one TLB miss and $O(1)$ cache misses. A read or write from d_i to m_j (primary to secondary) is done as in Theorem 3.2 above. Since all translations are held in TLB, the only cost of this step is $O(1)$ amortised cache misses and $O(B)$ instructions. Finally, emulating the operation of the algorithm on data in main memory incurs no cache or TLB misses. $\square$

3.4 General techniques to obtain simultaneous cache and TLB optimality

We now give two general techniques to obtain algorithms and data structures which have simultaneous cache and TLB optimality.

The first is to simulate, using the emulation of Theorem 3.3, optimal algorithms for 3-EMM. The second is the use of cache-oblivious algorithms [38]. As noted in [38, 69] a cache-oblivious algorithm gives optimal performance across each level of a multi-level hierarchy. By our emulation theorem, an optimal cache-oblivious algorithm makes an optimal number of cache and TLB misses (in general, the emulation is needed as some cache-oblivious algorithms are analysed on an 'ideal cache' model, which assumes an omniscient replacement policy).

3.5 Summary

In this chapter we have given a model for the internal memory hierarchy of a computer system. We have given emulation theorems to convert algorithms and their analyses from other memory models to this new model. We have given two general techniques to obtain algorithms which are simultaneously optimal for the cache and TLB.

Chapter 4

Sorting and searching

In this chapter we describe the problem of sorting and we discuss in some detail existing results for this problems in internal memory and for external memory. We discuss distribution sorting, integer sorting, Mergesort and Quicksort. We then describe the problem of searching and describe the B-tree data structure and its variant the B*-tree, which are typically used for searching in external memory.

4.1 Sorting

Sorting is acknowledged to be a very important computing problem in itself and it is an important sub-problem in many other areas such as database systems, searching and graph algorithms. A definition of a sorting problem is, given a sequence $x_1, x_2, \ldots, x_n$ generate a permutation $\pi = \pi(1), \pi(2), \ldots, \pi(n)$ such that $x_{\pi(1)} \leq x_{\pi(2)} \leq \ldots \leq x_{\pi(n)}$. In practice the x_i are in an array and the sorting procedure would return an array where the x_i are permuted according to π. The sort is said to be *stable* whenever $x_i = x_j$ and $i < j$, $\pi(i) < \pi(j)$.

In practice the data to be sorted are keys associated with records and the permutation involves moving pointers to the records along with the keys, though in this thesis we generally assume the problem is one of reordering just the keys.

In this chapter we discuss some sorting algorithms which are related to the results in this thesis. We initially describe the problem of sorting and discuss the computation and I/O lower bounds for this problem. We then describe a generic distribution sorting algorithm, radix sorting algorithms, *Bundle sort*, which is used to sort keys with $k \ll n$

distinct values, k-way Mergesort and multi-partition Quicksort. The discussion on the generic distribution sorting algorithm will be quite detailed, as it is built upon in the next three chapters. Radix sorting algorithms will be dealt with in more detail in later chapters. Discussion of the other algorithms will be fairly brief.

4.1.1 Lower bounds for sorting

The computation costs of most sorting algorithms are analysed on the *comparison model of computation*, which assumes that comparisons are the only operations allowed to determine the order of keys. The operations allowed are $<, \leq, =, \geq$ or $>$. In this model, $\Omega(n \log n)$ comparisons are required to sort a sequence of n unique keys. Quicksort, Mergesort and Heapsort are classic examples of algorithms analysed in the comparison model.

Some sorting algorithms are analysed making an assumption that the keys are of a particular kind, for example integers or floating-point numbers. These algorithms are analysed on a model which allows $+, -, *, /$, MASK and SHIFT operations to determine the order of keys. These algorithms make several passes over the data. One pass works as follows. Any of the operations may be used to map a key to an integer value distributed in the range 1 to k in $O(1)$ time. The mapped values can be used as an address in a table, thus allowing keys to be permuted directly to their destination. Each pass takes $O(n+k)$ time, when $k = O(n)$ one pass runs in $O(n)$ time. Distribution sorting is a popular technique for sorting data which is assumed to be randomly distributed, and involves distributing the n input keys into k *classes* based on their value. After one pass of the algorithm, the keys should have been permuted so that all elements of class i lie consecutively before all elements of class $i+1$, for $i = 0, \ldots, k-2$. Several algorithms have been devised which use a constant number of passes and run faster than corresponding comparison-based algorithms. Most distribution sorting algorithms, radix sorting algorithms and Bundle sort applied to integers are analysed using this model. Note that sometimes comparison based algorithms which recursively partition keys into multiple sets are referred to as distribution sorting algorithms, however in the thesis we refer to these as multi-partition comparison-based sorting algorithms.

Aggarwal and Vitter [8] proved upper and lower bounds for the number of I/O operations required in the EMM, introduced in Section 2.1.1, in order to sort n unique keys. They showed that

$$\Theta\left(\frac{N}{B}\frac{\log(1+\log N/B)}{\log(1+\log M/B)}\right)$$

I/O operations between main memory and disk were required. They described a multi-partition Quicksort algorithm and a k-way merge sorting algorithm which achieved this bound.

Since the replacement policy is more complicated in the CMM and IMM than in the EMM, the lower bound is also a lower bound for cache and TLB misses. Applying Aggarwal and Vitter's result, we get the following lower bound for the number of cache misses:

$$\Omega\left(\frac{n}{B}\frac{\log(1+\log n/B)}{\log(1+\log C)}\right),$$

and for the number of TLB misses:

$$\Omega\left(\frac{n}{P}\frac{\log(1+\log n/P)}{\log(1+\log T)}\right).$$

4.2 Distribution sorting

In distribution sorting the classes are chosen so that all the keys in the i-th class are smaller than all the keys in the $(i+1)$st class, for $i = 0, \ldots, k-2$, and furthermore, the class to which a key belongs can be computed in $O(1)$ time. In $O(n)$ time, the problem is reduced to sorting the $k \leq n$ classes. Distribution sorting can be used to sort independent randomly distributed keys in $O(n)$ time on average [53, Ch 5.2, 5.2.1], which is asymptotically faster than the $\Theta(n \log n)$ average running time of comparison-based approaches.

After one pass of the algorithm, the keys should have been permuted so that all elements of class i lie consecutively before all elements of class $i+1$, for $i = 0, \ldots, k-2$. While describing this algorithm, the term *data* array refers to the array holding the input keys, and the term *count* refers to an auxiliary array used by these algorithms. Each pass consists of two main phases, a *count* phase followed by a *permute* phase. There are two main variants of the permute phase. In the first variant keys are permuted from the data array to the auxiliary *destination* array; this is called an *out-of-place permutation.* In the second variant keys in the data array are permuted within the data array; this is called an *in-place permutation.*

In the next few chapters we analyse in-place permutations and out-of-place permutations in the CMM. Ladner et al. [56] give an analysis of frequency counting in the CMM, which, with a few minor adaptations, is quite accurate for calculating the number of cache misses during the count phase of a generic distribution sorting algorithm. We discuss their analysis and the required adaptations in Chapter 7. In the same chapter, we show how to use cache analysis to tune a distribution sorting algorithm.

Frigo et al. [38] describe a cache-oblivious distribution sorting algorithm. Kumar [55] implemented this algorithm and reported that, due to high computation costs, the algorithm was easily outperformed by Quicksort implementations.

We now describe the count phase and the two variants of the permute phase. In the description below it is assumed that k has been appropriately initialised, and that the function `classify` maps a key to a class numbered $\{0, \ldots, k-1\}$ in $O(1)$ time.

4.2.1 Count phase

The count phase counts for each class $i \geq 1$, the total number of keys in classes $0, \ldots, i-1$. For class $i = 0$ this cumulative count is 0.

Figure 4.1 shows the pseudo-code for a generic count phase. This phase consists of four steps. Step 1 initialises each element of `COUNT` to zero. Step 2 counts the frequency of keys in each class by applying the `classify` function. Steps 3 and 4 perform a prefix-summation of the count array, after which `COUNT[0]` $= 0$ and for $i = 1, \ldots, k-1$, `COUNT[`i`]` holds the total number of keys in classes $0, \ldots, i-1$. Clearly this phase takes $O(n)$ time whenever $k \leq n$.

4.2.2 Permute phase

The permute phase uses the cumulative count of keys generated during the count phase, to permute the keys to their respective classes. We now describe the two variants of the permute phase, starting with the simpler out-of-place permutation. We then describe the in-place permutation phase which uses a cyclic permutation strategy.

Out-of-place permute

During an out-of-place permute phase, for any class j, unless all elements of that class have already been moved, `COUNT[`j`]` points to the leftmost (lowest-numbered) available

```
Count phase
1 for $i$ := 0 to $k-1$ do
      COUNT[$i$] := 0;
2 for $i$ := 0 to $n-1$ do
      COUNT[classify(DATA[$i$])]++;
3 COUNT[$k-1$] := $n$ - COUNT[$k-1$];
4 for $i$ := $k-2$ downto 0 do
      COUNT[$i$] := COUNT[$i+1$] - COUNT[$i$];
```

Figure 4.1: Count phase for one pass of a generic distribution sorting algorithm. `DATA` holds the input keys and `COUNT` is an auxiliary array.

```
Permute phase(out-of-place permutation)
1 for $i$ := 0 to $n-1$ do
      $key$ := DATA[$i$];
      $x$ := classify($key$);
      $idx$ := COUNT[$x$];
      COUNT[$x$]++;
      DEST[idx] := $key$;
```

Figure 4.2: Permute phase for one pass of an 'out-of-place' permutation in a generic distribution sorting algorithm. `DATA` holds the input keys. `COUNT` and `DEST` are auxiliary arrays.

location for an element of class j in an n element auxiliary array, `DEST`. Figure 4.2 shows the pseudo-code for out-of-place permutation. In Step 1, for each element in `DATA`: we determine its class; using the count array we determine the next available location for this key in the `DEST` array; we increment the count array, thus setting the location for the next key of the same class; finally we move the key to its location in `DEST`. Since each step takes constant time, this out-of-place permutation takes $O(n)$ time whenever $k \leq n$.

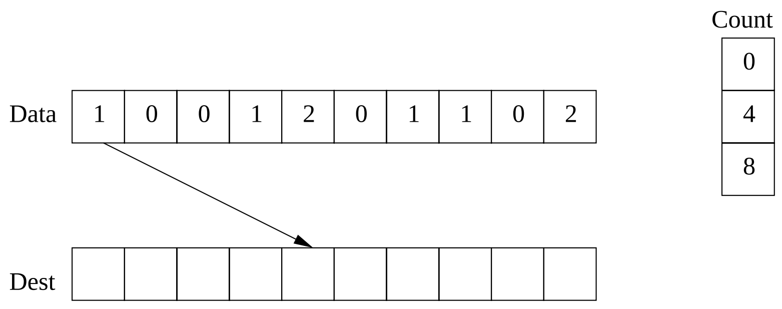

Figure 4.3: Each key in DATA is moved to its destination, which is an empty location in the DEST array. For example, key 1 from DATA[0] is moved into its destination at DEST[4], as specified in the count array. Then key 0 from DATA[1] is moved into its destination at DEST[0]. As keys are moved the appropriate count array locations are incremented.

In-place permute

The in-place permutation strategy that we describe here is similar to that described by Knuth [53, Soln 5.2-13] and used by Neubert [67] in his implementation of Flashsort1, discussed in Chapter 7. Before an in-place permute phase begins, a copy of the count array is made in a k element auxiliary *start* array. During the permute phase, for any class j, an invariant is that locations START[j], START[j] $+ 1, \ldots,$ COUNT[j] $- 1$ contain elements of class j, i.e. COUNT[j] points to the leftmost (lowest-numbered) available location for an element of class j. Thus, for $j = 0, \ldots, k-2$, all elements of class j have been permuted if COUNT[j] $\geq$ START[$j+1$], and such a class will be called *complete* in what follows. Class $k-1$ is complete when COUNT[$k-1$] $\geq n$. Figure 4.4 shows the pseudo-code for in-place permutation. We now describe this permutation, which consists of two main activities: *cycle following* and *cycle leader finding*. In cycle following, keys are moved to their final destinations in the data array along a cycle in the permutation (Steps 2 and 3). Once a cycle is completed, we move to cycle leader finding, where we find the 'leader' (index of the rightmost element) of the next cycle (Steps 1, 4 and 5). A cycle leader is simply the rightmost location of the highest-numbered incomplete class. By the definition of a complete class, initially the leader must be position $n-1$. In more detail, the steps are as follows:

- In Step 1, $n-1$ is selected as the first cycle leader.

```
Permute phase(in-place permutation)
1 $leader$ := $n-1$;
2 $idx$ := $leader$; $key$ := DATA[$idx$];
3.1 $x$ := classify($key$);
3.2 $idx$ := COUNT[$x$];
3.3 COUNT[$x$]++;
3.4 swap $key$ and DATA[$idx$];
3.5 if $idx \neq leader$ repeat 3.1;
4 while ($x > 0 \land$ COUNT[$x-1$] $\geq$ START[$x$])
    $x$--;
5 if ($x > 0$) $leader$ := START[$x$]$-1$;
    go to 2;
```

Figure 4.4: Permute phase for one pass of an 'in-place' permutation in a generic distribution sorting algorithm. DATA holds the input keys. COUNT and START are auxiliary arrays. After the count phase, COUNT is copied into START.

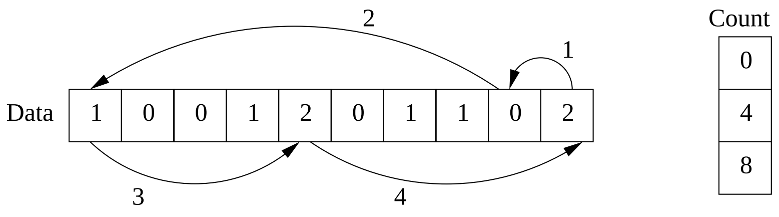

Figure 4.5: Keys are cyclically permuted within the DATA array. The cycle starts at the index 9. The key at index 9, the 2, is moved into its destination, at index 8, as specified in the count array. The key at index 8, the 0, is moved to its destination, at index 0. The key at index 0, the 1, is moved to its destination, at index 4. The key at index 4, the last 2, is moved to its destination at index 9. This completes the cycle. As keys were moved the appropriate count array locations were incremented.

- In Step 2 the key at the leader's position is copied into the variable *key*, thus leaving a 'hole' in the leader's position.
- In Steps 3.1-3.5 the key *key* is swapped with the key at *key*'s final position. If *key* 'fills the hole', the cycle is complete, otherwise we repeat these steps.
- In Step 4 the algorithm searches for a new cycle leader. Suppose the leader of the cycle which just completed was the last location of class j. When this cycle ends, class j must also be complete, as a key of class j has been moved into the last location of class j. Note that classes $j+1, j+2, \ldots$ must already have been complete when the leader of this cycle was found. Note that the program variable x has value j at the end of this cycle, so the search for the next leader begins with class $j-1$, counting down (Step 4).
- In Step 5 we check to see if all classes have completed and terminate if this is the case.

It is important to note that this in-place permutation algorithm leads to a non-stable sorting algorithm, so it can not be used in algorithms which require stable sorting, such as some radix sorting algorithms.

Clearly the in-place permutation in one pass of distribution sorting takes $O(n)$ time whenever $k \leq n$.

4.3 Radix sorting

Radix sorting views each key as consecutive digits and in the i-th pass sorts according to the i-th digit. There are two main variants of radix sort, *least-significant bit (LSB) radix sort* and *most-significant bit (MSB) radix sort.*

LSB radix sort views w bit keys as $\lceil w/r \rceil$ consecutive r-bit digits. Keys are sorted in $\lceil w/r \rceil$ passes, where in the i-th pass, for $i = 1, \ldots, \lceil w/r \rceil$, we sort keys according to the i-th least significant digit. The correctness of the algorithm requires keys to be stably sorted in each pass. All n keys are sorted together in each pass and it can be implemented using the count phase and out-of-place permute phase described in section 4.2. The `classify` function masks off the appropriate r bits, digit, and the count array holds up to 2^r elements. The computation time to sort n keys using this algorithm is $O(\lceil w/r \rceil (n + 2^r))$.

Most significant bit radix sort is a distribution sorting algorithm which views each key as consecutive digits. In one pass of MSB radix sort with radix r for some integer $r \geq 1$, we distribute the keys into $k = 2^r$ classes by letting the `classify`(x) return the value of the r most significant bits of x. If it is known that all keys being sorted have the same value in their most significant r' bits, as may happen in a recursively-solved sub-problem, then `classify`(x) returns the value of the next most significant r bits. The algorithm can be implemented using the count phase and either of the permute phase methods described in section 4.2.

LaMarca and Ladner [59] analysed LSB radix sorting in the CMM, however their analysis does not consider the cache conflict misses that occur due to the fact that any of $\Omega(k)$ locations may be accessed during the permute phase. Our analysis for out-of-place permutations in the CMM in Chapter 5 gives a far more accurate prediction of the number of cache misses.

In Chapter 8 we study LSB radix sort extensively in the IMM, obtaining several LSB radix sorting algorithms which are highly tuned for the cache and TLB and exhibit much better performance than the standard algorithm designed in the RAM model.

In Chapter 7 we show how to design MSB radix sort for the CMM. The keys that we use induce a non-uniform distribution of keys to classes, and we demonstrate how to use the cache miss analysis of Chapter 5 to tune the algorithm.

4.4 Bundle sorting

Matias et al. [62] describe a sorting algorithm in the EMM, introduced in Section 2.1.1, for the special case of sorting where there are a limited number of distinct keys. Their algorithm bundles together identical keys and places the bundles in order, so they refer to this as *bundle sorting*. For a data set of N keys, consisting of k distinct keys, the algorithm has a running time of $c(N/B)\log_{M/B} k$ I/Os, where $2 < c < 4$, which circumvents the lower bound for general sorting when $k \ll N$. They show that their algorithm is optimal by proving that $\Omega((N/B)\log_{M/B} k)$ I/Os are required for this problem.

Their algorithm works as follows. If $kB \leq M$ the algorithm sorts in three passes. In the first pass it counts the keys in each bundle, after which it determines the starting location for the destinations of keys in each bundle. This is similar to the count phase of distribution sorting. The first block from the starting location of each bundle is then loaded in main memory. The keys in each block are scanned and swapped around to their appropriate blocks. When all the keys in a block have been scanned, the block is written back to disk and the next block in the bundle is loaded in main memory. This requires a total of three passes over the data. If $kB > M$, then the algorithm bundles into M/B super-bundles and recursively sorts each super-bundle, which requires $\log_{M/B} k$ recursive steps.

They describe implementations of the algorithm for sorting keys in the range $1, \ldots, k$, effectively integer sorting, and for sorting general keys.

4.5 Multi-partition Quicksort and k-way Mergesort

Divide and conquer algorithms designed on the RAM model typically deal with two sets of data. For example Quicksort recursively partitions data into two sets, and Mergesort repeatedly merges two sorted lists. These techniques require $\Omega(\log n)$ passes over the data and applying them directly to EMM algorithms can lead to an unnecessarily large number of capacity misses. To reduce the number of such misses, divide and conquer algorithms for the EMM deal with $k = O(M/B)$ sets of data, which reduces the number of passes over the data to $\Omega(\log n / \log k)$. For example, the analogue to Quicksort in the EMM is distribution sorting, which recursively partitions data into k sets, and a merge sort which merges k sorted lists.

Multi-way or k-way mergesort algorithms [53] are used to minimise the number of I/O operations between two levels of memory, and we describe such a sorting algorithm in terms of the EMM. The algorithm works by recursively merging sorted *runs*. Initially the N keys are loaded into internal memory in groups of M keys and each group is sorted. These are written to disk as N/M sorted runs. Now the k-way merging starts. In each pass of the merging phase, k runs of sorted keys are merged at a time. To ensure that during merging a buffer full of keys from each run being merged can reside in internal memory, we have to limit $k \leq M/B$, so the resulting number of merging passes is $\Omega(\log_{M/B} N/M)$. Since each merging pass requires $O(N/B)$ I/Os, to load the data in all the runs, the algorithm achieves the optimal I/O bound.

LaMarca and Ladner [59] describe an implementation of k-way Mergesort tuned for the cache on a DEC Alphastation 250. They show that this implementation outperforms highly tuned two-way Mergesort and *tiled* Mergesort (where the keys are initially sorted into lists of $CB/2$ keys and then merged using two-way merging).

Mehlhorn and Sanders [65] analyse the number of cache misses in a set-associative cache when an adversary makes n accesses to k sequences. Their analysis assumes that the start of each sequence is uniformly and randomly distributed in the cache. They show that in order to have $O(n/B)$ cache misses, asymptotically the same as the number of misses that must be made in order just to read the data, the algorithm can use only $k = C/B^{1/a}$ sequences. This suggests that if EMM algorithm techniques such as k-way merging or partitioning into k sets are used in the CMM, then the parameter k must be reduced by a factor of $B^{1/a}$. Sen and Chatterjee [80] confirmed this result in their average case cache analysis of k-way Mergesort.

In the next chapter we obtain upper and lower bounds for the average case number of cache misses in k-way merging in the CMM, and we show that this result is better than the result obtained by Sen and Chatterjee [80].

Frigo et al. [38] describe a cache-oblivious k-way Mergesort algorithm. We do not know of any implementation of this, and we would speculate that, as for the cache-oblivious distribution sort, the performance of an implementation would not be competitive with cache and TLB tuned algorithms due to high computation costs.

4.6 Searching

Like sorting, searching is an important computing problem with applications in many areas such as database systems and graph algorithms.

We consider the dynamic predecessor problem, which involves maintaining a set S of pairs $\langle x, i\rangle$ where x is a key drawn from a totally ordered universe and i is some satellite data. Without loss of generality we assume that i is just a pointer. We assume that keys are fixed-size, and we assume that each pair in S has a unique key. The operations permitted on the set include insertion and deletion of $\langle$key, satellite data$\rangle$

pairs, and predecessor searching, which takes a key q and returns a pair $\langle x, i\rangle \in S$ such that x is the largest key in S satisfying $x \leq q$.

4.6.1 B-trees

A B-tree [31, 53] is a balanced search tree designed for external memory. For $d \geq 2$, it has the following characteristics:

- The root has between 2 and $2d-1$ children.
- Internal nodes have between $d-1$ and $2d-1$ children.
- All leaves are at the same depth.

For a tree with N items the height is $O(\log_d N)$. The maximum *branching factor*, the number of children at a node, is selected such that all data at a node fits in a memory block. A node with x keys has $x+1$ children and the keys are arranged in non-decreasing order. Each key in an internal node has a pointer to a child that is the root of a sub-tree containing all nodes with keys less than or equal to the key but greater than the preceding key. A node also has an additional rightmost pointer to a child that is the root for a sub-tree containing all keys greater than any keys in the node.

Searches are performed by searching between all the keys at a node and following the appropriate child pointer. The keys inside a node are stored in sorted order and may be searched using a binary or linear search. If a new key is to be inserted into the tree then we have to search for the leaf node which will hold the new key and insert at that point. This may cause an overflow at the node, if so we split the node into two and insert the new node into the parent. We may have to recursively split nodes all the way up to the root. Delete operations may cause an *underflow* at a node in which case the node may merge with its sibling. Again an underflow may occur at the parent, so we may have to recursively merge nodes all the way up to the root.

B-trees support searches, inserts and deletes in $O(\log_B N)$ I/Os if $2d-1 = O(B)$ and in $O(\log_2 N)$ operations. B-trees can be used for caches by selecting the branching factor such that a node fits in a cache block. The data structure then makes $O(\log_B N)$ cache and TLB misses per search, insert or delete operation, where B is the cache block size.

4.6.2 B*-trees

B*-trees are a variant of B-trees where the nodes remain at least 66% full by sharing with a sibling node the keys in a node which has exceeded its maximum size [31]. By packing more keys in a node, B*-trees are slightly shallower than B-trees. Like B-trees, B*-trees support searches, updates and deletes in $O(\log_B N)$ I/Os if $2d - 1 = O(B)$ and in $O(\log_2 N)$ operations. Like B-trees, if adapted for the cache, B*-trees make $O(\log_B N)$ cache and TLB misses per search, insert or delete operation.

Chapter 5

Analysing the permute phase of distribution sorting in the CMM

In this chapter we analyse cache misses in the CMM during the permute phase of distribution sorting when the keys are independently drawn from a non-uniform random distribution. Distribution sorting involves distributing n keys into k classes based on their value. In the permute phase of distribution sorting, when a key is moved to its destination, the algorithms described in Chapter 4 access any one of k elements in the `COUNT` array and any one of k locations in the `DATA` or `DEST` arrays, depending on whether the permutation is in-place or out-of-place. The actual locations accessed are dependent on the value of the permuted key, so, if the keys are independently and randomly distributed then, for every key permuted there are two accesses to random memory locations, one in the count array and one in `DATA` or `DEST`. These random accesses can potentially lead to a large number of cache conflict misses.

Our approach is to define two stochastic processes which model in-place or out-of-place permutation. Process "in-place" models an in-place permutation and Process "out-of-place" models an out-of-place permutation. Each round of a process models moving a key to its destination, and we analyse the expected number of cache misses in n rounds of these processes. Our precise expressions for the expected number of cache misses are difficult to compute so we also prove closed-form upper and lower bounds on these precise equations. We use our results for in-place permutations to get upper and lower bounds on the expected number of cache misses in a process which models accesses to multiple sequences.

The assumptions in the processes mean that, over n rounds, we have to access at least n distinct locations in memory, which requires $\Omega(n/B)$ cache misses. In the analysis, we will say that a process is optimal if it incurs $O(n/B)$ cache misses. In distribution sorting, the larger the value of k, the fewer the number of passes over the data, hence the fewer the capacity misses. However, as we will see, if k is too large, then there can be a large number of conflict misses. One of the aims of the analysis in this chapter is to determine the largest value of k, for a particular distribution of keys, such that there are $O(n/B)$ misses in one pass of distribution sorting.

5.1 Processes

We now give the two processes which model the distributing of keys drawn independently and randomly from a non-uniform distribution into k classes.

5.1.1 Process to model an in-place permutation

Let k be an integer, $2 \leq k \leq CB$. We are given k probabilities $p_1, \ldots, p_k$, such that $\sum_{i=1}^{k} p_i = 1$. The process maintains k pointers $D_1, \ldots, D_k$, and there are also k consecutive 'count array' locations, $\mathcal{C} = c_1, \ldots, c_k$. The process (henceforth called *Process "in-place"*) executes a sequence of *rounds*, where each round consists in performing steps 1-3 below:

Process "in-place"

1. Pick an integer x from $\{1, \ldots, k\}$ such that $\Pr[x = i] = p_i$, independently of all previous picks.
2. Access the location c_x.
3. Access the location pointed to by D_x, increment D_x by 1.

We denote the locations accessed by the pointer D_i by $d_{i,1}, d_{i,2}, \ldots$, for $i = 1, \ldots, k$. We assume that:

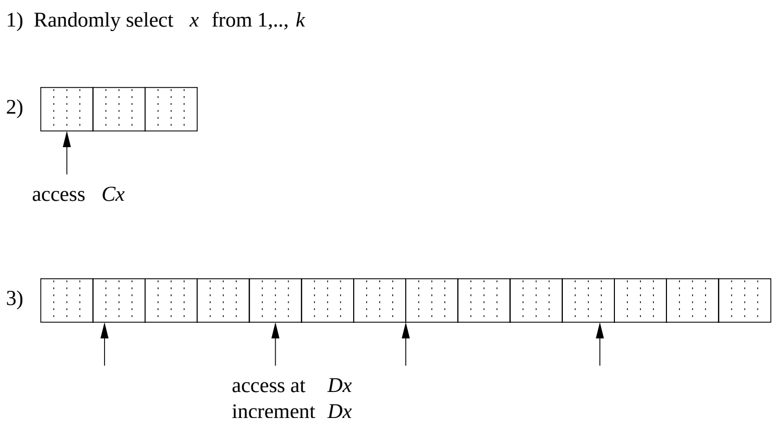

Figure 5.1: Process "in-place".

(a) the start position of each pointer is uniformly and independently distributed over the cache, i.e., for each i, $d_{i,1} \bmod BC$ is uniformly and independently distributed over $\{0, \ldots, BC-1\}$,

(b) during the process, the pointers traverse sequences of memory locations which are disjoint from each other and from $\mathcal{C}$,

(c) c_1 is located on an aligned block boundary, i.e., $c_1 \bmod B = 0$,

(d) the pointers D_i, for $i = 1, \ldots, k$, are in separate memory blocks.

Assuming that the cache is initially empty, the objective is to determine the expected number of cache misses incurred by the above process over n rounds, with the expectation taken over the random choices in Step 1 as well as the starting positions of the pointers.

Process "in-place" is shown in Figure 5.1.

5.1.2 Process to model an out-of-place permutation

This process is like Process "in-place", but it is augmented with accesses to a sequence of consecutive locations in a source array, $\mathcal{S}$, determined by an index s, where $s = 0$ at the start of the process. The process, henceforth called *Process "out-of-place"*, executes a sequence of *rounds*, where each round consists in performing steps 1-4 below:

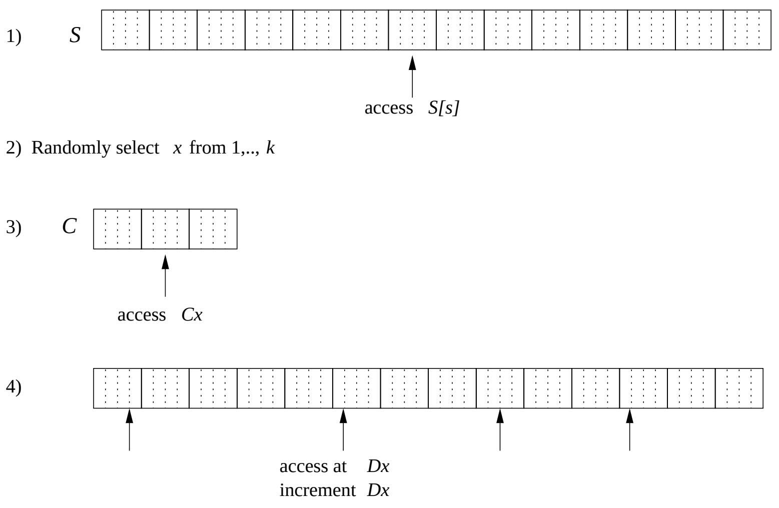

Figure 5.2: Process “out-of-place”.

Process “out-of-place”

1. Access the location $\mathcal{S}[s]$, increment s by 1.
2. Pick an integer x from $\{1, \ldots, k\}$ such that $\Pr[x = i] = p_i$, independently of all previous picks.
3. Access the location c_x.
4. Access the location pointed to by D_x, increment D_x by 1.

We make assumptions (a), (c), and (d) from Process “in-place”, assumption (b) is modified as below:

(b) during the process, the pointers traverse sequences of memory locations which are disjoint from each other, from $\mathcal{C}$ and from $\mathcal{S}$.

and we add a further assumption:

(e) $\mathcal{S}$ is located on an aligned block boundary, i.e., $\mathcal{S}[0] \bmod B = 0$,

Assuming that the cache is initially empty, again the objective is to determine the expected number of cache misses incurred by the above process over n rounds, with the expectation taken over the random choices in Step 2 as well as the starting positions of the pointers.

Process “out-of-place” is shown in Figure 5.2.

5.2 Preliminaries

We first set out some propositions that are used in the proofs in the remainder of this chapter.

Proposition 5.1 *For all real numbers $x_i, i = 1, \ldots, k$, such that $|x_i| \leq 1$ we have that:*

$$\prod_{i=0}^{k}(1 - x_i) \geq 1 - \sum_{i=0}^{k} x_i.$$

Proposition 5.2 *(a) For all real numbers x, such that $|x| < 1$, we have that:*

$$\sum_{m=0}^{\infty} x^m = \frac{1}{1-x}.$$

(b) For all real numbers x, such that $|x| < 1$, we have that:

$$\sum_{m=0}^{\infty} m x^m = \frac{x}{(1-x)^2}.$$

(c) For all real numbers x, such that $0 < x < 2$, we have that:

$$\sum_{m=0}^{\infty} x(1-x)^m m = \frac{1}{x} - 1.$$

Proof. Proposition 5.2(a) is the standard summation for an infinite decreasing geometric series. We obtain Proposition 5.2(b) by differentiating both sides of the equation in Proposition 5.2(a). Proposition 5.2(c) is obtained using Proposition 5.2(b). □

Proposition 5.3 *For all real numbers p and q such that $0 < p - q < 2$, we have that:*

$$\sum_{m=0}^{\infty} p(1-p)^m \left(1 - \frac{q}{1-p}\right)^m = \frac{p}{p+q}.$$

Proof. Since $(1-p)\left(1 - \frac{q}{1-p}\right) = 1 - p - q$, using Proposition 5.2(a) we get that:

$$\sum_{m=0}^{\infty} p(1-p)^m \left(1 - \frac{q}{1-p}\right)^m = \sum_{m=0}^{\infty} p\,(1-p-q)^m = \frac{p}{p+q}.$$

□

Proposition 5.4 *(a) For all real numbers x, we have that:*

$$e^{-x} \geq 1 - x.$$

(b) For all real numbers $x \geq 0$, we have that:

$$e^{-x} \leq 1 - x + \frac{x^2}{2}.$$

PROOF. These follow from the Taylor series expansion of e^x. □

Proposition 5.5 *For all real numbers x and y, such that $x \leq 1$ and $y \geq 0$, we have that:*

$$e^{-xy} \geq (1-x)^y.$$

PROOF. This proposition is immediate from Proposition 5.4(a). □

Proposition 5.6 *(a) For all real numbers x and p and integers y, such that $0 < p \leq 1$, $y \geq 0$ and $x(1/p + y) = O(1)$, we have that:*

$$\sum_{m=0}^{y} p(1-p)^m mx \quad = \quad x\left(\frac{1}{p} - 1\right) - O(e^{-py}).$$

(b) For all real numbers x and p and integers y, such that $0 < p \leq 1$, $y \geq 0$ and $x = O(1)$, we have that:

$$\sum_{m=0}^{y} p(1-p)^m x \quad = \quad x - O(e^{-py}).$$

Note that we are misusing the O notation here to hide constant factors that are independent of x.

PROOF. Using Proposition 5.2(c) and Proposition 5.5, Proposition 5.6(a) is proved as follows:

$$\begin{aligned}
\sum_{m=0}^{y} p(1-p)^m mx &= \sum_{m=0}^{\infty} p(1-p)^m mx - (1-p)^{y+1} \sum_{m=0}^{\infty} p(1-p)^m (m+y+1)x \\
&= x\left(\frac{1}{p} - 1\right) - (1-p)^{y+1} x \left(\frac{1}{p} + y\right) \\
&= x\left(\frac{1}{p} - 1\right) - O(e^{-py}).
\end{aligned}$$

The proof of Proposition 5.6(b) is now trivial. □

The vector of random variables $X = (X_1, \ldots X_n)$, is *negatively associated* [50] if for every two disjoint index sets, $I, J \subset [n]$,

$$\mathrm{E}[f(X_i, i \in I) g(X_j, j \in J)] \leq \mathrm{E}[f(X_i, i \in I)] \mathrm{E}[g(X_j, j \in J)]$$

for all functions $f : \Re^{|I|} \rightarrow \Re$ and $g : \Re^{|J|} \rightarrow \Re$ that are both non-decreasing or non-increasing.

Proposition 5.7 *If the random variables $X_1, \ldots X_k$ are negatively associated, then for any non-decreasing function $f_i, i \in [k]$, we have that:*

$$\mathrm{E}[\prod_{i=1}^{k} f_i(X_i)] \leq \prod_{i=1}^{k} \mathrm{E}[f_i(X_i)].$$

PROOF. The proof follows directly from the definition of negatively associated variables. □

We now introduce some notation that will be used for the analysis in this chapter. We use k to denote the number of classes that the keys will be distributed into, and throughout the analysis we assume that B divides k. Assume that we are given a set of k probabilities $p_1, \ldots, p_k$, such that $\sum_{i=0}^{k} p_i = 1$. The expected value of a function f of a random variable X is denoted as $\mathrm{E}[f(X)]$. When we wish to make explicit the distribution D from which the random variable is drawn, we will use the notation $\mathrm{E}_{X\sim D}[f(X)]$. All vectors have dimension k (the number of classes) unless stated otherwise, and we denote the components of a vector $\bar{x}$ by x_1, $x_2, \ldots, x_k$. We now define some probabilities:

(i) For all $i \in \{1, \ldots, k/B\}$, $P_i = \sum_{l=(i-1)B+1}^{iB} p_l$.

(ii) For all $i \in \{1, \ldots, k\}$, we denote by $\bar{a^i}$ the following vector: $a^i_j = 0$ if $i = j$, and $a^i_j = p_j/(1-p_i)$ otherwise and by $\bar{b^i}$ the following vector: $b^i_j = 0$ if $(i-1)B+1 \leq j \leq iB$, and $b^i_j = p_j/(1-P_i)$ otherwise. (Note that $\sum_j a^i_j = \sum_j b^i_j = 1$).

Let $m \geq 0$ be an integer and $\bar{q}$ be a vector of non-negative reals such that $\sum_i q_i = 1$. We denote by $\varphi(m, \bar{q})$ the probability distribution on the number of balls in each of k bins, when m balls are independently put into these bins, and a ball goes in bin i with probability q_i, for $i \in \{1, \ldots, k\}$. Thus, $\varphi(m, \bar{q})$ is a distribution on vectors of non-negative integers. If $\bar{\mu}$ is drawn from $\varphi(m, \bar{q})$, then:

$$\Pr[\mu_1 = m_1, \ldots, \mu_k = m_k] = \left(\prod_{j=1}^{k} q_j^{m_j}\right) m!/\prod_{j=1}^{k} m_j! \tag{5.1}$$

whenever $\sum_{i=1}^{k} m_i = m$; all other vectors have zero probability[1]. We now define functions $f(x)$ for $x \geq 0$ and $g(\bar{m})$ for a vector $\bar{m}$ of non-negative integers:

$$f(x) = \begin{cases} 1 & \text{if } x = 0, \\ 1 - \frac{x+B-1}{BC} & \text{if } 0 < x \leq BC - B + 1, \\ 0 & \text{otherwise.} \end{cases} \tag{5.2}$$

[1]We take $0^0 = 1$ in Eq. 5.1.

$$g(\bar{m}) \quad = \quad \frac{1}{C}\sum_{i=1}^{k/B}\min\{1, \sum_{l=(i-1)B+1}^{iB} m_l\}. \tag{5.3}$$

5.3 Cache analysis of in-place permutation

In this section we analyse the cache misses in a direct-mapped cache during n rounds of Process "in-place", introduced in Section 5.1.1. We derive a precise equation for the expected number of cache misses and then give closed form upper and lower bounds on this equation. We then derive upper and lower bounds assuming the keys are drawn independently from a uniform distribution.

5.3.1 Average case analysis

We start by proving a theorem for the expected number of cache misses during n rounds of Process "in-place".

Theorem 5.1 *The expected number X of cache misses in n rounds of Process "in-place" satisfies $n(p_c + p_d) \leq X \leq n(p_c + p_d) + k(1 + 1/B)$, where:*

$$p_c \quad = \quad \sum_{i=1}^{k/B} P_i \left(1 - \sum_{m=0}^{\infty} P_i(1-P_i)^m \mathrm{E}_{\bar{\nu}\sim\varphi(m,\bar{b}_i)}\left[\prod_{j=1}^{k} f(\nu_j)\right]\right) \text{ and}$$

$$p_d \quad = \quad \frac{1}{B} + \frac{B-1}{B}\sum_{i=1}^{k} p_i \left(1 - \sum_{m=0}^{\infty} p_i(1-p_i)^m \mathrm{E}_{\bar{\mu}\sim\varphi(m,\bar{a}_i)}\left[(1-g(\bar{\mu}))\prod_{j=1}^{k} f(\mu_j)\right]\right),$$

where the function f is as defined in Eq. 5.2 and the function g is as defined in Eq. 5.3.

Proof. We first analyse the miss rates for accesses to pointers $D_1, \ldots, D_k$. Fix an i, $1 \leq i \leq k$ and a $z \geq 1$. Let μ be the random variable which denotes the number of rounds between accesses to locations $d_{i,z}$ and $d_{i,z+1}$ ($\mu = 0$ if these locations are accessed in consecutive rounds). Figure 5.3 shows the other memory accesses between accesses z and $z + 1$ to D_i. Clearly, $\Pr[\mu = m] = p_i(1 - p_i)^m$, for $m = 0, 1, \ldots$. Let X_i denote the event that none of the memory accesses in these μ rounds accesses the cache block to which $d_{i,z}$ is mapped. We now fix an integer $m \geq 0$ and calculate $\Pr[X_i|\mu = m]$. Let $\bar{\mu}$ be a vector of random variables such that for $1 \leq j \leq k$, μ_j is the random variable which denotes the number of accesses to D_j in these m rounds. Clearly $\bar{\mu}$ is drawn from $\varphi(m, \bar{a}_i)$ (note that D_i is not accessed in these m rounds by definition).

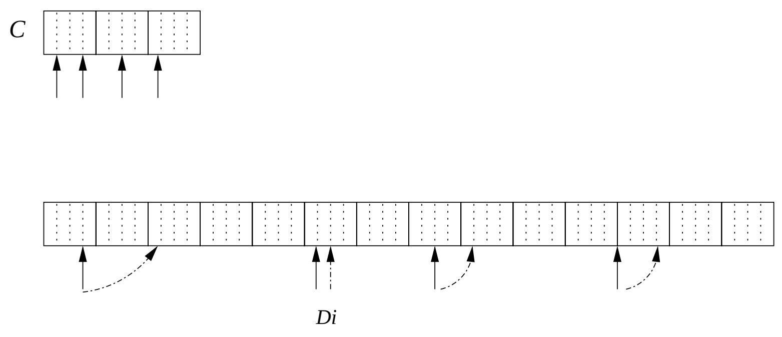

Figure 5.3: m rounds of Process "in-place". Between two accesses to D_i, there are m accesses to "other" pointers, and $m+1$ accesses to $\mathcal{C}$.

Fix any vector $\bar{m}$, such that $\Pr[\bar{\mu} = \bar{m}] \neq 0$, and let μ_j be the number of accesses to pointer D_j in these m rounds. Since m_i must be zero, $f(m_i) = 1$, and for $j \neq i$, $f(m_j)$ is the probability that none of the m_j locations accessed by D_j in these m rounds is mapped to the same cache block as location $d_{i,z}$, as in [65, 80]. Similarly $g(\bar{m}) \cdot C$ is the number of count blocks accessed in these rounds, and so $1 - g(\bar{m})$ is the probability that the cache block containing $d_{i,z}$ does not conflict with the blocks from $\mathcal{C}$ which were accessed in these m rounds. As the latter probability is determined by the starting location of sequence i and the former probabilities by the starting location of sequences $j, j \neq i$, we conclude that for a given configuration $\bar{m}$ of accesses, the probability that the cache block containing $d_{i,z}$ is not accessed in these m rounds is $(1 - g(\bar{m})) \prod_{j=1}^{k} f(m_j)$. Averaging over all configurations $\bar{m}$, we get that:

$$\Pr[X_i \mid \mu = m] = \mathrm{E}_{\bar{\mu} \sim \varphi(m, \bar{a}_i)} \left[(1 - g(\bar{\mu})) \prod_{j=1}^{k} f(\mu_j) \right]. \tag{5.4}$$

Finally we get:

$$\begin{aligned} \Pr[X_i] &= \sum_{m=0}^{\infty} \Pr[\mu = m] \Pr[X_i | \mu = m] \\ &= \sum_{m=0}^{\infty} p_i (1 - p_i)^m \mathrm{E}_{\bar{\mu} \sim \varphi(m, \bar{a}_i)} \left[(1 - g(\bar{\mu})) \prod_{j=1}^{k} f(\mu_j) \right]. \end{aligned} \tag{5.5}$$

If $d_{i,z}$ is at a cache block boundary or if X_i does not occur given that $d_{i,z}$ is not at a cache block boundary ($\Pr[X_i]$ does not change under this condition), then a cache miss will occur. The first access to a pointer is a cache miss. So other than for the first access, the probability p_d of a cache miss for a pointer access is:

$$p_d \;=\; \frac{1}{B} + \frac{B-1}{B}\sum_{i=1}^{k} p_i(1 - \Pr[X_i]). \tag{5.6}$$

Including the first access misses, the expected number of cache misses for pointer accesses is at most

$$\sum_{i=1}^{k}\left(1 + (np_i - 1)\left(\left(\frac{B-1}{B}(1 - \Pr[X_i])\right) + \frac{1}{B}\right)\right) \le np_d + k. \tag{5.7}$$

We now consider the probability of a cache miss for an access to a count array location. It is convenient to partition $\mathcal{C}$ into count blocks of B locations each, where the i-th count block consists of the locations $c_{(i-1)B+1}, \ldots, c_{iB}$, for $i = 1, \ldots, k/B$. So $P_i = \sum_{l=(i-1)B+1}^{iB} p_l$ is the probability of access to the i-th block. We fix an $i \in \{1, \ldots, k/B\}$ and a $z \ge 1$. Let ν be the random variable that denotes the number of rounds between the z-th and $(z+1)$-st accesses to the i-th count block. We have that $\Pr[\nu = m] = P_i(1 - P_i)^m$, for $m = 0, 1, \ldots$. Let Y_i denote the event that none of the memory accesses in these m rounds accesses the cache block to which the i-th count block is mapped.

We now fix an integer $m \ge 0$ and calculate $\Pr[Y_i | \nu = m]$. Let $\bar{\nu}$ be a vector of random variables such that for $1 \le j \le k$, ν_j is the random variable which denotes the number of accesses to D_j in these m rounds. Given that $k \le BC$ and assumption (c) mean that two blocks from $\mathcal{C}$ cannot conflict with each other. As the pointers $D_{(i-1)B+1}, \ldots, D_{iB}$ will not be accessed between two successive accesses to count block i, the probability of accessing pointer D_j is given by b_j^i and $\varphi(m, \bar{b_i})$ is the distribution for $\bar{\nu}$. Arguing as above:

$$\begin{aligned}\Pr[Y_i] \;&=\; \sum_{m=0}^{\infty} \Pr[\nu = m]\Pr[Y_i|\nu = m] \\ &=\; \sum_{m=0}^{\infty} P_i(1 - P_i)^m \mathrm{E}_{\bar{\nu}\sim\varphi(m,\bar{b_i})}\left[\prod_{j=1}^{k} f(\nu_j)\right].\end{aligned} \tag{5.8}$$

The first access to a count array block is a cache miss, for all other accesses there is a cache miss if event Y_i does not occur. So other than for the first access, the probability p_c of a cache miss for a count array access is:

$$p_c = \sum_{i=1}^{k/B} P_i(1 - \Pr[Y_i]). \tag{5.9}$$

Including the first access misses, the expected number of cache misses for count array accesses is at most

$$\sum_{i=1}^{k/B} (1 + (nP_i - 1)(1 - \Pr[Y_i])) \leq np_c + k/B. \tag{5.10}$$

Plugging in the values from Eq. 5.5 into Eq. 5.7 and from Eq. 5.8 into Eq. 5.10 we get the upper bound on X, the expected number of cache misses in the processes. The lower bound in Theorem 5.1 is obvious.

$\square$

5.3.2 Upper bound

We now prove a closed form upper bound on the expected number of cache misses during n rounds of Process "in-place".

Theorem 5.2 *The expected number of cache misses in n rounds of Process "in-place" is at most $n(p_d + p_c) + k(1 + 1/B)$, where:*

$$\begin{aligned} p_d &\leq \frac{1}{B} + \frac{k}{BC} + \frac{B-1}{BC}\sum_{i=1}^{k}\left(\sum_{j=1}^{k/B}\frac{p_i P_j}{p_i + P_j} + \frac{B-1}{B}\sum_{j=1}^{k}\frac{p_i p_j}{p_i + p_j}\right), \\ p_c &\leq \frac{k}{B^2 C} + \frac{B-1}{BC}\sum_{i=1}^{k/B}\sum_{j=1}^{k}\frac{P_i p_j}{P_i + p_j}. \end{aligned}$$

Proof. In the proof we derive lower bounds for $\Pr[X_i]$ and $\Pr[Y_i]$ and use these to derive the upper bounds on p_d and p_c.

Again, we consider a fixed i and consider the event X_i defined in the proof of Theorem 5.1. We now obtain a lower bound on $\Pr[X_i]$.

Lower bound on $\Pr[X_i]$

Letting $\Gamma(x) = 1 - f(x)$ and using Proposition 5.1 we can rewrite Eq. 5.5 as:

$$\Pr[X_i] \geq \sum_{m=0}^{\infty} \Pr[\mu = m] \mathrm{E}_{\bar{\mu}\sim\varphi(m,\bar{a}_i)}\left[1 - g(\bar{\mu}) - \sum_{j=1}^{k}\Gamma(\mu_j)\right]. \tag{5.11}$$

While deriving the lower bound on $\Pr[X_i]$ we assume that $\bar{\mu}$ is drawn from the distribution $\varphi(m, \bar{a}_i)$ and we omit the subscript in $\mathrm{E}_{\bar{\mu}\sim\varphi(m,\bar{a}_i)}$.

We know that the j-th count block contributes $1/C$ to $g(\bar{\mu})$ if there is an access to that block and $\Pr[j\text{-th count block accessed}|\mu = m] = 1 - (1 - c_j^i)^m$, where $c_j^i = \frac{P_j}{1-p_i}$. So we have that,

$$\mathrm{E}[g(\bar{\mu})] = \sum_{j=1}^{k/B} \frac{1}{C}(1 - (1 - c_j^i)^m),$$

and we get:

$$\begin{aligned}\sum_{m=0}^{\infty} \Pr[\mu = m]\mathrm{E}[g(\bar{\mu})] &= \sum_{m=0}^{\infty} p_i(1-p_i)^m \sum_{j=1}^{k/B} \frac{1}{C}(1 - (1 - c_j^i)^m) \\ &= \frac{1}{C}\sum_{j=1}^{k/B}\sum_{m=0}^{\infty} p_i(1-p_i)^m (1 - (1 - c_j^i)^m),\end{aligned}$$

and using Proposition 5.3 we get:

$$\sum_{m=0}^{\infty} \Pr[\mu = m]\mathrm{E}[g(\bar{\mu})] = \frac{1}{C}\sum_{j=1}^{k/B} \frac{P_j}{p_i + P_j}. \tag{5.12}$$

We now evaluate

$$\sum_{m=0}^{\infty} \Pr[\mu = m] \sum_{j=1}^{k} \mathrm{E}[\Gamma(\mu_j)].$$

Our approach is to first fix j and evaluate $\mathrm{E}[\Gamma(\mu_j)]$. For $m \le BC$, we know that

$$\mathrm{E}[\Gamma(\mu_j)] = \sum_{l=0}^{m} \Pr[\mu_j = l]\frac{l+B-1}{BC} - \Pr[\mu_j = 0]\frac{B-1}{BC}.$$

The last term is due to the fact that $\Gamma(x)$ is discontinuous and $\Gamma(0) = 0$. Similarly for $m > BC$ we know that

$$\begin{aligned}\mathrm{E}[\Gamma(\mu_j)] &= \sum_{l=0}^{m} \Pr[\mu_j = l]\frac{l+B-1}{BC} - \Pr[\mu_j = 0]\frac{B-1}{BC} \\ &\quad - \sum_{l=BC-B+1}^{m} \Pr[\mu_j = l]\left(\frac{l+B-1}{BC} - 1\right).\end{aligned}$$

The last term is due to the fact that $\Gamma(x) = 1$ for $x \ge BC - B + 1$. If we drop this last term when $m > BC$, we get that for all m:

$$\mathrm{E}[\Gamma(\mu_j)] \le \frac{1}{BC}\left[\sum_{l=0}^{m} \Pr[\mu_j = l]l + (B-1)(1 - \Pr[\mu_j = 0])\right].$$

The summation term is the expected value of μ_j, which is clearly ma_j^i. So we get that:

$$\mathrm{E}[\Gamma(\mu_j)] \le \frac{1}{BC}\left[ma_j^i + (B-1)\left(1 - \left(1 - a_j^i\right)^m\right)\right]. \tag{5.13}$$

We now evaluate $\sum_{m=0}^{\infty} \Pr[\mu = m] \sum_{j=1}^{k} \mathrm{E}[\Gamma(\mu_j)]$ as

$$\sum_{m=0}^{\infty} \Pr[\mu = m] \sum_{j=1}^{k} \mathrm{E}[\Gamma(\mu_j)] \leq \sum_{m=0}^{\infty} \Pr[\mu = m] \sum_{j=1}^{k} \frac{1}{BC} \left[m a_j^i + (B-1)\left(1 - \left(1 - a_j^i\right)^m\right)\right].$$

Since $\sum_{j=1}^{k} m a_j^i = m$, we get $\sum_{m=0}^{\infty} \Pr[\mu = m] \sum_{j=1}^{k} m a_j^i = \frac{1}{p_i} - 1$ by an application of Proposition 5.2(c). By applying Proposition 5.3 we get that $\sum_{j=1}^{k} \sum_{m=0}^{\infty} \Pr[\mu = m](B-1)(1-(1-a_j^i)^m) = (B-1)\sum_{j=1}^{k} \frac{p_j}{p_i + p_j}$. So we get:

$$\sum_{m=0}^{\infty} \Pr[\mu = m] \sum_{j=1}^{k} \mathrm{E}[\Gamma(\mu_j)] \quad \leq \quad \frac{1}{BC}\left(\frac{1}{p_i} + (B-1)\sum_{j=1}^{k} \frac{p_j}{p_i + p_j}\right). \tag{5.14}$$

Substituting Eq. 5.12 and 5.14 in Eq. 5.11 we obtain the following lower bound for $\Pr[X_i]$

$$\Pr[X_i] \quad \geq \quad 1 - \frac{1}{C}\sum_{j=1}^{k/B} \frac{P_j}{p_i + P_j} - \frac{1}{BC}\left(\frac{1}{p_i} + (B-1)\sum_{j=1}^{k} \frac{p_j}{p_i + p_j}\right). \tag{5.15}$$

Upper bound on p_d

Finally, substituting $\Pr[X_i]$ from Eq. 5.15 in Eq. 5.6 we get:

$$\begin{aligned} p_d \quad &\leq \quad \frac{1}{B} + \frac{B-1}{B}\sum_{i=1}^{k} p_i \left(\frac{1}{C}\sum_{j=1}^{k/B} \frac{P_j}{p_i + P_j} + \frac{1}{BC}\left(\frac{1}{p_i} + (B-1)\sum_{j=1}^{k} \frac{p_j}{p_i + p_j}\right)\right) \\ &= \quad \frac{1}{B} + \frac{(B-1)k}{B^2 C} + \frac{B-1}{BC}\sum_{i=1}^{k}\sum_{j=1}^{k/B} \frac{p_i P_j}{p_i + P_j} + \frac{(B-1)^2}{B^2 C}\sum_{i=1}^{k}\sum_{j=1}^{k} \frac{p_i p_j}{p_i + p_j} \\ &\leq \quad \frac{1}{B} + \frac{k}{BC} + \frac{B-1}{BC}\sum_{i=1}^{k}\left(\sum_{j=1}^{k/B} \frac{p_i P_j}{p_i + P_j} + \frac{B-1}{B}\sum_{j=1}^{k} \frac{p_i p_j}{p_i + p_j}\right). \end{aligned} \tag{5.16}$$

We can evaluate p_c using a very similar approach, as sketched out now. We again consider a fixed i and consider the event Y_i defined in the proof of Theorem 5.1. We now obtain a lower bound on $\Pr[Y_i]$.

Lower bound on $\Pr[Y_i]$

Again letting $\Gamma(x) = 1 - f(x)$ and using Proposition 5.1, we can rewrite Eq. 5.8 as:

$$\Pr[Y_i] \geq \sum_{m=0}^{\infty} \Pr[\nu = m] \mathrm{E}_{\bar{\nu} \sim \varphi(m, \bar{b}_i)} \left[1 - \sum_{j=1}^{k} \Gamma(\nu_j)\right]. \tag{5.17}$$

While deriving the lower bound on $\Pr[Y_i]$ we assume that $\bar{\nu}$ is drawn from the distribution $\varphi(m, \bar{b_i})$ and we omit the subscript in $\mathrm{E}_{\bar{\nu}\sim\varphi(m,\bar{b_i})}$.

Arguing as for the derivation of Eq. 5.13, we get:

$$\mathrm{E}[\Gamma(\nu_j)] \leq \frac{1}{BC}\left[mb_j^i + (B-1)\left(1-\left(1-b_j^i\right)^m\right)\right].$$

Then arguing as for the derivation of Eq. 5.14, we get:

$$\sum_{m=0}^{\infty}\Pr[\nu=m]\sum_{j=1}^{k}\mathrm{E}[\Gamma(\nu_j)] \leq \frac{1}{BC}\left(\frac{1}{P_i} + (B-1)\sum_{j=1}^{k}\frac{p_j}{P_i+p_j}\right).$$

Substituting this into Eq. 5.17, we get:

$$\Pr[Y_i] \;\geq\; 1 - \frac{1}{BC}\left(\frac{1}{P_i} + (B-1)\sum_{j=1}^{k}\frac{p_j}{P_i+p_j}\right). \tag{5.18}$$

Upper bound on p_c

Substituting $\Pr[Y_i]$ from Eq. 5.18 in Eq. 5.9 we get:

$$\begin{aligned} p_c &\leq \sum_{i=1}^{k/B} P_i \frac{1}{BC}\left(\frac{1}{P_i} + (B-1)\sum_{i=1}^{k}\frac{p_j}{P_i+p_j}\right) \\ &= \frac{k}{B^2C} + \frac{B-1}{BC}\sum_{i=1}^{k/B}\sum_{j=1}^{k}\frac{P_i p_j}{P_i+p_j}. \end{aligned} \tag{5.19}$$

Combining Eq. 5.16 and Eq. 5.19 proves the theorem. □

5.3.3 Lower bound

We now prove a closed form lower bound for the expression in Theorem 5.1.

Theorem 5.3 *When $p_i \geq 1/C$ then the expected number of cache misses in n rounds of Process "in-place" is at least $np_d + k$, where:*

$$\begin{aligned} p_d \geq \frac{1}{B} + \frac{B-1}{B}\sum_{i=1}^{k} p_i \quad & \left[\left(\frac{1}{p_i}-1\right)\frac{1}{2BC}\left(1-\frac{2(B-1)k}{BC}\right)\right. \\ &+ \sum_{j=1}^{k}\frac{p_i}{p_i+p_j}\frac{B-1}{BC}\left(\frac{(B-1)k}{BC}-1\right) \\ &+ \frac{(B-1)k}{BC}\left(1-\frac{(B-1)k}{2BC}\right) \\ &+ \frac{(B-1)}{(BC)^2}\sum_{j=1}^{k}\frac{p_i(1-p_i-p_j)}{(p_i+p_j)^2} \\ &- \left.\frac{(B-1)^2}{2(BC)^2}\sum_{j=1}^{k}\sum_{l=1}^{k}\frac{p_i}{p_i+p_j+p_l-p_jp_l} - O\left(e^{-B}\right)\right]. \end{aligned}$$

PROOF. We again consider a fixed i and consider the event X_i defined in the proof of Theorem 5.1. Let $\bar{\mu}$ be as defined in the proof of Theorem 5.1. We now obtain an upper bound on $\Pr[X_i]$.

Upper bound on $\Pr[X_i]$

In [33] it is shown that the variables μ_j are negatively associated. Noting that $f(x)$ is a non-increasing function of x, then using Proposition 5.7 we have that:

$$\mathrm{E}[\prod_{j=1}^{k} f(\mu_j)] \leq \prod_{j=1}^{k} \mathrm{E}[f(\mu_j)].$$

So we can rewrite Eq. 5.5 as:

$$\Pr[X_i] \leq \sum_{m=0}^{BC-B} \Pr[\mu = m] \prod_{j=1}^{k} \mathrm{E}_{\bar{\mu}\sim\varphi(m,\bar{a}_i)}[f(\mu_j)] + \sum_{m=BC-B+1}^{\infty} \Pr[\mu = m]. \tag{5.20}$$

While deriving the upper bound on $\Pr[X_i]$ we again assume $\bar{\mu}$ is drawn from the distribution $\varphi(m, \bar{a}_i)$ and we omit the subscript in $\mathrm{E}_{\bar{\mu}\sim\varphi(m,\bar{a}_i)}$.

We first bound the last term. We know that

$$\begin{aligned} \sum_{m=BC-B+1}^{\infty} \Pr[\mu = m] &= (1-p_i)^{BC-B+1} \sum_{m=0}^{\infty} \Pr[\mu = m] \\ &= (1-p_i)^{BC-B+1}. \end{aligned}$$

Using Proposition 5.5 we get that $(1-p_i)^{BC-B+1} \leq e^{-(BC-B+1)p_i}$. Assuming $p_i \geq 1/C$ the last term is at most $O(e^{-B})$.

We now bound the first term in Eq. 5.20. We use an approach similar to the derivation of Eq. 5.13 and since $\mu \leq BC - B$, so $\mu_j \leq BC - B$, we don't have to drop any terms in the simplification, so we get that:

$$\mathrm{E}[f(\mu_j)] = 1 - \frac{1}{BC}(ma^i_j + (B-1)(1-(1-a^i_j)^m)).$$

Letting $t_j(m) = \frac{1}{BC}(ma^i_j + (B-1)(1-(1-a^i_j)^m))$ and using Proposition 5.4(a) we get that $e^{-\sum_j t_j(m)} \geq \prod_j (1 - t_j(m))$. So we have that

$$\begin{aligned} \Pr[X_i] &\leq \sum_{m=0}^{BC-B} \Pr[\mu = m] e^{\frac{-1}{BC}\sum_{j=1}^{k}\left(ma^i_j + (B-1)(1-(1-a^i_j)^m)\right)} + O(e^{-B}). \\ &\leq \sum_{m=0}^{BC-B} \Pr[\mu = m] e^{\frac{-1}{BC}\left(m + (B-1)(k - \sum_{j=1}^{k}(1-a^i_j)^m)\right)} + O(e^{-B}). \end{aligned}$$

Assuming $\frac{1}{BC}\left(m + (B-1)(k - \sum_{j=1}^{k}(1-a_j^i)^m)\right) \geq 0$, using Proposition 5.4(b) and letting $\beta_j = (1 - a_j^i)$ we get that:

$$\begin{aligned}
\Pr[X_i] \;\leq\; & \sum_{m=0}^{BC-B} \Pr[\mu = m]\left[1 - \frac{(B-1)k}{BC} + \frac{((B-1)k)^2}{2(BC)^2} + \frac{m^2}{2(BC)^2}\right. \\
& -\frac{1}{BC}\left(m - (B-1)\sum_{j=1}^{k}\beta_j^m\right) - \frac{(B-1)}{2(BC)^2}\left(2m\sum_{j=1}^{k}\beta_j^m + 2(B-1)k\sum_{j=1}^{k}\beta_j^m\right) \\
& \left. + \frac{(B-1)}{2(BC)^2}\left(2mk + (B-1)\sum_{j=1}^{k}\sum_{l=1}^{k}\beta_j^m\beta_l^m\right)\right] + O(e^{-B}). \qquad (5.21)
\end{aligned}$$

We now evaluate the terms in Eq. 5.21 assuming that $p_i \geq 1/C$, so $k \leq C$. For the simplifications of the subtractive terms we use the fact that $e^{-p_i(BC-B+1)} \leq e^{-B}$. In some of the simplifications we will use $\alpha = BC - B + 1$.

Since $p_i \geq 1/C$, $(B-1)k/(BC) < 1$, so using Proposition 5.6(b), we get that:

$$\sum_{m=0}^{\alpha-1} \Pr[\mu = m]\frac{(B-1)k}{BC} = \frac{(B-1)k}{BC} - O(e^{-B}). \qquad (5.22)$$

Since $p_i \geq 1/C$, $(1/p_i + BC - B)/(BC) = O(1)$, so using Proposition 5.6(a), we get that:

$$\sum_{m=0}^{\alpha-1} \Pr[\mu = m]\frac{m}{BC} = \frac{1}{BC}\left(\frac{1}{p_i} - 1\right) - O(e^{-B}). \qquad (5.23)$$

We now evaluate the term

$$\begin{aligned}
& \frac{(B-1)}{(BC)^2}\sum_{m=0}^{\alpha-1}\Pr[\mu = m]m\sum_{j=1}^{k}\beta_j^m \\
= \;& \frac{(B-1)}{(BC)^2}\sum_{j=1}^{k}\sum_{m=0}^{\infty}\Pr[\mu = m]m\beta_j^m \\
& -(1-p_i)^{\alpha}\frac{(B-1)}{(BC)^2}\sum_{m=0}^{\infty}\Pr[\mu = m](m+\alpha)\sum_{j=1}^{k}\beta_j^m \\
= \;& \frac{(B-1)}{(BC)^2}\sum_{j=1}^{k}\frac{p_i(1-p_i-p_j)}{(p_i+p_j)^2} \\
& -(1-p_i)^{\alpha}\frac{(B-1)}{(BC)^2}\left(\sum_{j=1}^{k}\frac{p_i(1-p_i-p_j)}{(p_i+p_j)^2} + \sum_{j=1}^{k}\frac{\alpha p_i}{p_i+p_j}\right) \\
= \;& \frac{(B-1)}{(BC)^2}\sum_{j=1}^{k}\frac{p_i(1-p_i-p_j)}{(p_i+p_j)^2} - O(e^{-B}). \qquad (5.24)
\end{aligned}$$

The second simplification used Proposition 5.2(b) and Proposition 5.3. The last simplification is due to $p_i \geq 1/C$ and $k \leq C$, so $\sum_{1\leq j\leq k}(p_i(1-p_i-p_j))/(p_i+p_j)^2 \leq kC \leq C^2$

and $\sum_{1\le j\le k}(\alpha p_i)/(p_i+p_j) \le kCB \le BC^2$.

Substituting back $(1-a^i_j) = \beta_j$ and using Proposition 5.3 we get that:

$$\begin{aligned}
&\frac{(B-1)^2k}{(BC)^2}\sum_{m=0}^{BC-B}\Pr[\mu=m]\sum_{j=1}^{k}\beta_j^m \\
&= \frac{(B-1)^2k}{(BC)^2}\sum_{j=1}^{k}\frac{p_i}{p_i+p_j} - (1-p_i)^\alpha\frac{(B-1)^2k}{(BC)^2}\sum_{j=1}^{k}\frac{p_i}{p_i+p_j} \\
&= \frac{(B-1)^2k}{(BC)^2}\sum_{j=1}^{k}\frac{p_i}{p_i+p_j} - O(e^{-B}). \qquad (5.25)
\end{aligned}$$

The last step used $\sum_{j=1}^k p_i/(p_i+p_j) \le k$ and $((B-1)k)^2/(BC)^2) < 1$.

We now evaluate the additive terms, starting with:

$$\sum_{m=0}^{BC-B}\Pr[\mu=m]\frac{mk(B-1)}{(BC)^2} \le \frac{(B-1)k}{(BC)^2}\left(\frac{1}{p_i}-1\right). \qquad (5.26)$$

Since $m < BC$ we now get that:

$$\sum_{m=0}^{BC-B}\Pr[\mu=m]\frac{m^2}{2(BC)^2} \le \frac{1}{2BC}\left(\frac{1}{p_i}-1\right). \qquad (5.27)$$

Substituting back $(1-a^i_j) = \beta_j$ and using Proposition 5.3 we get that:

$$\frac{B-1}{BC}\sum_{m=0}^{BC-B}\Pr[\mu=m]\sum_{j=0}^{k}{\beta_j}^m \le \frac{B-1}{BC}\sum_{j=1}^{k}\frac{p_i}{p_i+p_j}. \qquad (5.28)$$

Finally we evaluate $\sum_{m=0}^{\alpha-1}\Pr[\mu=m]\sum_{j=1}^k\sum_{l=1}^k {\beta_j}^m{\beta_l}^m$, by first evaluating

$$\begin{aligned}
(1-p_i)\beta_j\beta_l &= (1-p_i)(1-a^i_j)(1-a^i_l) \\
&= \frac{1-2p_i-p_j-p_l+p_i(p_i+p_j+p_l)+p_jp_l}{(1-p_i)}.
\end{aligned}$$

Using this result and Proposition 5.2(a), we get that:

$$\begin{aligned}
\sum_{m=0}^{\alpha-1}\Pr[\mu=m]{\beta_j}^m{\beta_l}^m &\le \sum_{m=0}^{\infty}p_i\left(\frac{1-2p_i-p_j-p_l+p_i(p_i+p_j+p_l)+p_jp_l}{(1-p_i)}\right)^m \\
&= \frac{p_i}{p_i+p_j+p_l-p_jp_l/(1-p_i)} \\
&\le \frac{p_i}{p_i+p_j+p_l-p_jp_l}.
\end{aligned}$$

So we get that:

$$\frac{(B-1)^2}{2(BC)^2}\sum_{m=0}^{\alpha-1} \le \frac{(B-1)^2}{2(BC)^2}\sum_{j=1}^{k}\sum_{l=1}^{k}\frac{p_i}{p_i+p_j+p_l-p_jp_l}. \qquad (5.29)$$

Lower bound on p_d

Plugging Eqs. 5.23 ,.., 5.29 into Eq. 5.21 we get that:

$$\begin{aligned}
\Pr[X_i] \;\le\;& 1-\frac{1}{BC}\left(\frac{1}{p_i}-1\right)-\frac{(B-1)k}{BC}-\frac{(B-1)}{(BC)^2}\sum_{j=1}^{k}\frac{p_i(1-p_i-p_j)}{(p_i+p_j)^2}\\
-\;& \frac{(B-1)^2k}{(BC)^2}\sum_{j=1}^{k}\frac{p_i}{p_i+p_j}+\frac{(B-1)k}{(BC)^2}\left(\frac{1}{p_i}-1\right)+\frac{1}{2BC}\left(\frac{1}{p_i}-1\right)\\
+\;& \frac{B-1}{BC}\sum_{j=1}^{k}\frac{p_i}{p_i+p_j}+\frac{(B-1)^2}{2(BC)^2}\sum_{j=1}^{k}\sum_{l=1}^{k}\frac{p_i}{p_i+p_j+p_l-p_jp_l}\\
+\;& \frac{((B-1)k)^2}{2(BC)^2}+O(e^{-B}). \qquad (5.30)
\end{aligned}$$

Plugging Eq. 5.30 into Eq. 5.6 we get:

$$\begin{aligned}
p_d \ge \frac{1}{B}+\frac{B-1}{B}\sum_{i=1}^{k}p_i \quad & \left[\left(\frac{1}{p_i}-1\right)\frac{1}{2BC}\left(1-\frac{2(B-1)k}{BC}\right)\right.\\
+\;& \sum_{j=1}^{k}\frac{p_i}{p_i+p_j}\frac{B-1}{BC}\left(\frac{(B-1)k}{BC}-1\right)\\
+\;& \frac{(B-1)k}{BC}\left(1-\frac{(B-1)k}{2BC}\right)\\
+\;& \frac{(B-1)}{(BC)^2}\sum_{j=1}^{k}\frac{p_i(1-p_i-p_j)}{(p_i+p_j)^2}\\
-\;& \left.\frac{(B-1)^2}{2(BC)^2}\sum_{j=1}^{k}\sum_{l=1}^{k}\frac{p_i}{p_i+p_j+p_l-p_jp_l}-O\left(e^{-B}\right)\right].
\end{aligned}$$

□

REMARK: The assumption that $p_i \ge 1/C$, so $k \le C$, is not unreasonable, if we consider that the lower bounds for I/Os in the weaker EMM requires that at most C partitions are used for multi-partition comparison sort and at most C streams are merged during k-way merging.

5.3.4 Upper and lower bounds for uniformly-random data

Using the upper and lower bound Theorems just proven for general probability distributions, we now derive corollaries for upper and lower bounds for uniform distribution.

Corollary 5.1 *If $p_1 = \ldots = p_k = 1/k$ then the number of cache misses in n rounds of Process "in-place" is at most:*

$$n\left(\frac{1}{B}+\frac{k(B+5)}{2BC}+\frac{k}{B^2C}\right)+k\left(1+\frac{1}{B}\right).$$

PROOF. Since P_i in p_c and P_j in p_d are both B/k in the equations in Theorem 5.2, we get that:

$$\begin{aligned}
p_d + p_c &\leq \frac{1}{B} + \frac{2(B-1)}{BC}\frac{k^2}{B}\frac{B/k}{B+1} + \frac{(B-1)^2}{B^2C}k^2\frac{1/k}{2} + \frac{k}{B^2C} + \frac{k}{BC} \\
&= \frac{1}{B} + \frac{2(B-1)}{BC}\frac{k}{B+1} + \frac{(B-1)^2}{B^2C}\frac{k}{2} + \frac{k}{B^2C} + \frac{k}{BC} \\
&\leq \frac{1}{B} + \frac{k}{C}\left[\frac{3}{B} + \frac{B-1}{2B}\right] + \frac{k}{B^2C} \\
&= \frac{1}{B} + \frac{k(B+5)}{2BC} + \frac{k}{B^2C}.
\end{aligned}$$

□

REMARK: As we will see later, Process "in-place" models the permute phase of distribution sorting and Corollary 5.1 shows that one pass of uniform distribution sorting incurs $O(n/B)$ cache misses if and only if $k = O(C/B)$.

The following corollary is from the lower bound result in Theorem 5.3.

Corollary 5.2 *If $p_1 = \ldots = p_k = 1/k$ then the number of cache misses in n rounds of Process "in-place" is at least:*

$$\begin{aligned}
k + \frac{n}{B} + \frac{n(B-1)}{B} \quad &\left[\frac{1-k}{2kBC}\left(1 - \frac{2(B-1)k}{BC}\right) + \frac{(B-1)k}{2BC}\right. \\
&\left. + \frac{(B-1)}{12(BC)^2}\left(k^2(5-2B) - 7k + 2\right)\right].
\end{aligned}$$

PROOF. Plugging $p_i = 1/k$ in the equation in Theorem 5.3 we get that:

$$\begin{aligned}
p_d \geq \frac{1}{B} + \frac{B-1}{B} \quad &\left[\frac{1-k}{2kBC}\left(1 - \frac{2(B-1)k}{BC}\right)\right. \\
&+ \frac{(B-1)}{BC}\frac{k}{2}\left(\frac{(B-1)k}{BC} - 1\right) + \frac{(B-1)k}{BC}\left(1 - \frac{(B-1)k}{2BC}\right) \\
&\left. + \frac{(B-1)}{(BC)^2}\frac{(k-2)k}{4} - \frac{(B-1)^2}{2(BC)^2}\left(\frac{k^3}{3k-1}\right)\right] \\
= \frac{1}{B} + \frac{B-1}{B} \quad &\left[\frac{1-k}{2kBC}\left(1 - \frac{2(B-1)k}{BC}\right) + \frac{(B-1)k}{2BC}\right. \\
&\left. + \frac{(B-1)}{4(BC)^2}\left(\frac{(k^2-2k)(3k-1) - 2(B-1)k^3}{3k-1}\right)\right] \\
\geq \frac{1}{B} + \frac{B-1}{B} \quad &\left[\frac{1-k}{2kBC}\left(1 - \frac{2(B-1)k}{BC}\right) + \frac{(B-1)k}{2BC}\right. \\
&\left. + \frac{(B-1)}{12(BC)^2}\left(k^2(5-2B) - 7k + 2\right)\right].
\end{aligned}$$

□

Remark: From Corollary 5.1 we have that for uniformly-random data and $k = \alpha C$, where $\alpha \leq 1$, other than for small values of B, the upper bound for the number of cache misses in n rounds is roughly

$$\frac{\alpha n}{2},$$

and from Corollary 5.2 we have that for uniformly-random data and $k = \alpha C$, where $\alpha \leq 1$, other than for small values of B, the lower bound for the number of cache misses in n rounds is roughly

$$n\left(\frac{\alpha}{2} - \frac{\alpha^2}{6}\right).$$

The ratio between the upper and lower bound is $3/(3-\alpha)$. So we have that for uniformly-random data the lower bound is within a factor of about $3/2$ of the upper bound when $k \leq C$ and is much closer when $k \ll C$.

5.4 Cache analysis of out-of-place permutation

In this section we analyse the cache misses in a direct-mapped cache during n rounds of Process "out-of-place", introduced in Section 5.1.2. We derive a precise equation for the expected number of cache misses and closed-form upper and lower bounds. During the analysis we reuse $k, D_i, c_i, \mathcal{C}, p_i, P_i, \bar{a}, \bar{b}, f(x)$ and $g(m)$ introduced in Section 5.3.

5.4.1 Average case analysis

We start by proving a theorem for the expected number of cache misses during n rounds of Process "out-of-place".

Theorem 5.4 *The expected number X of cache misses in n rounds of Process "out-of-place" is $n(p_c + p_d + p_s) \leq X \leq n(p_c + p_d + p_s) + k(1 + 1/B) + 1$, where:*

$$\begin{aligned}
p_c &= \sum_{i=1}^{k/B} P_i \left(1 - \sum_{m=0}^{\infty} P_i(1-P_i)^m \mathrm{E}\left[f(m+1)\prod_{j=1}^{k} f(\nu_j)\right]\right), \\
p_d &= \frac{1}{B} + \frac{B-1}{B}\sum_{i=1}^{k} p_i \left(1 - \sum_{m=0}^{\infty} p_i(1-p_i)^m \mathrm{E}\left[(1-g(\bar{\mu}))f(m+1)\prod_{j=1}^{k} f(\mu_j)\right]\right), \\
p_s &= \frac{1}{B} + \frac{B-1}{B}\left(1 - \left(1 - \frac{1}{C}\right)^2\right).
\end{aligned}$$

where the function $f(x)$ is as defined in Eq. 5.2 and the function $g(\bar{\mu})$ is as defined in Eq. 5.3.

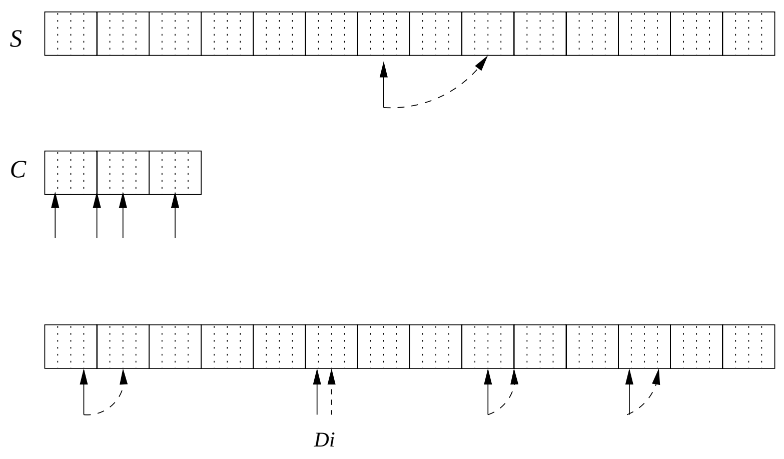

Figure 5.4: m rounds of Process "out-of-place". Between two accesses to D_i, there are m accesses to "other" pointers, and $m + 1$ accesses to $\mathcal{C}$, and $m + 1$ accesses to consecutive locations in $\mathcal{S}$.

PROOF. We first analyse the miss rates for accesses to pointers $D_1, \ldots, D_k$. Fix an i, $1 \le i \le k$ and a $z \ge 1$ and consider the probability of a miss between access z and $z + 1$ to pointer D_i. We define $\mu, \mu_j, m, \bar{\mu}, \bar{m}, d_{i,z}$, $\varphi(m, \bar{a_i})$ and X_i as in the proof of Theorem 5.1. Again $f(m_j)$ is the probability that none of the m_j locations accessed by D_j in m rounds is mapped to the same cache block as location $d_{i,z}$. Similarly $g(\bar{m}) \cdot C$ is the number of count blocks accessed in m rounds, and so $1 - g(\bar{m})$ is the probability that the cache block containing $d_{i,z}$ does not conflict with the blocks from $\mathcal{C}$ which were accessed in these m rounds. We also have accesses to $m + 1$ contiguous locations in $\mathcal{S}$ and $f(m + 1)$ is the probability that these $m + 1$ accesses are not to the cache block containing $d_{i,z}$. Figure 5.4 shows the other memory accesses between accesses z and $z + 1$ to D_i.

For a given configuration $\bar{m}$ of accesses, as the probabilities $f(m_j)$, $g(\bar{m})$ and $f(m+1)$ are independent, we conclude that the probability that the cache block containing $d_{i,z}$ is not accessed in these m rounds is $(1 - g(\bar{m}))f(m + 1) \prod_{j=1}^{k} f(m_j)$. Averaging over all configurations $\bar{m}$, we get that:

$$\Pr[X_i \mid \mu = m] = \mathrm{E}_{\bar{\mu} \sim \varphi(m, \bar{a_i})}[(1 - g(\bar{\mu}))f(m + 1) \prod_{j=1}^{k} f(\mu_j)]. \tag{5.31}$$

Using which we get:

$$\Pr[X_i] \quad = \quad \sum_{m=0}^{\infty} \Pr[\mu = m] \Pr[X_i | \mu = m]$$

$$= \sum_{m=0}^{\infty} p_i(1-p_i)^m \mathrm{E}_{\bar{\mu}\sim\varphi(m,\bar{a}_i)} \left[(1-g(\bar{\mu}))f(m+1)\prod_{j=1}^{k} f(\mu_j)\right]. \tag{5.32}$$

Arguing as for Eq. 5.6 we get that, other than for the first access, the probability p_d of a cache miss for a pointer access is:

$$p_d = \frac{1}{B} + \frac{B-1}{B}\sum_{i=1}^{k} p_i(1-\Pr[X_i]). \tag{5.33}$$

Including the first access misses, the expected number of cache misses for pointer accesses is at most

$$\sum_{i=1}^{k}\left(1+(np_i-1)\left(\left(\frac{B-1}{B}(1-\Pr[X_i])\right)+\frac{1}{B}\right)\right) \le np_d + k. \tag{5.34}$$

We now consider the probability of a cache miss for an access to a count array location. Fix an $i \in \{1,\ldots,k/B\}$ and a $z \ge 1$ and consider the probability of a miss between access z and $z+1$ to count block c_i. We define $\nu, \nu_j, m, \bar{\nu}, \bar{m}, P_i, \varphi(m,\bar{b_i})$, and Y_i as in the proof of Theorem 5.1.

Again, given that $k \le BC$ and assumption (c) mean that two blocks from $\mathcal{C}$ cannot conflict with each other. So we need to determine the probability of a conflict given m_j accesses to the pointer D_j, for all $j \in \{1,\ldots,k\}$, and m accesses to contiguous locations in $\mathcal{S}$. Again $f(m_j)$ is the probability that none of the m_j locations accessed by D_j in m rounds is mapped to the same cache block as c_i and $f(m+1)$ is the probability that the accesses to $m+1$ contiguous locations in $\mathcal{S}$ are not to the same cache block as c_i.

So we have:

$$\begin{aligned}\Pr[Y_i] &= \sum_{m=0}^{\infty}\Pr[\nu=m]\Pr[Y_i|\nu=m]\\ &= \sum_{m=0}^{\infty} P_i(1-P_i)^m \mathrm{E}_{\bar{\nu}\sim\varphi(m,\bar{b}_i)}\left[f(m)\prod_{j=1}^{k} f(\nu_j)\right].\end{aligned} \tag{5.35}$$

Arguing as for Eq. 5.9, the probability p_c of a cache miss for a count array access is:

$$p_c = \sum_{i=1}^{k/B} P_i(1-\Pr[Y_i]). \tag{5.36}$$

Including the first access misses, the expected number of cache misses for count array accesses is at most

$$\sum_{i=1}^{k/B}(1+(nP_i-1)(1-\Pr[Y_i])) \le np_c + k/B. \tag{5.37}$$

We now calculate cache misses for accesses to the array $\mathcal{S}$. We consider the probability of a cache miss between accesses to $\mathcal{S}[s]$ and $\mathcal{S}[s+1]$. We know that there is exactly one access to a count block and one access to a pointer between two accesses to $\mathcal{S}$. The probability that the pointer access is to the same cache block as $\mathcal{S}[s]$ is $1/C$. The probability that a block from $\mathcal{C}$ maps to the same cache block as $\mathcal{S}[s]$ is $k/(BC)$. Given that a block from $\mathcal{C}$ maps to the same cache block as $\mathcal{S}[s]$, the probability that the access to the count array is to the same cache block as $\mathcal{S}[s]$ is B/k. So the probability that the pointer access is to the same cache block as $\mathcal{S}[s]$ is also $1/C$. So the probability that there are no memory accesses to the cache block that $\mathcal{S}[s]$ is mapped to before the access to $\mathcal{S}[s+1]$ is

$$(1-1/C)^2.$$

We have a cache miss if $\mathcal{S}[s]$ is at a cache block boundary, otherwise the probability of a cache miss is $1-(1-1/C)^2$. So the probability p_s of an cache miss for an access to $\mathcal{S}$ is

$$p_s = \frac{1}{B} + \frac{B-1}{B}\left(1-\left(1-\frac{1}{C}\right)^2\right).$$

The first access to $\mathcal{S}$ is always a cache miss, so the expected number of cache misses in accesses to $\mathcal{S}$ is:

$$np_s + 1.$$

Plugging in the values from Eq. 5.32 into Eq. 5.34 and from Eq. 5.35 into Eq. 5.37 we get the upper bound on X, the expected number of cache misses in the processes.

The lower bound in Theorem 5.4 is obvious.

□

5.4.2 Upper bound

We now a closed form upper bound to the expected number of cache misses during n rounds of Process "out-of-place".

Theorem 5.5 *The expected number of cache misses in n rounds of Process "out-of-place" is at most $n(p_d + p_c + p_s) + k(1 + 1/B) + 1$, where:*

$$\begin{aligned}
p_d &\leq \frac{1}{B} + \frac{2(B-1)k}{B^2C} + \frac{B-1}{BC}\sum_{i=1}^{k}\sum_{j=1}^{k/B}\frac{p_iP_j}{p_i+P_j} + \frac{(B-1)^2}{B^2C}\left(1+\sum_{i=1}^{k}\sum_{j=1}^{k}\frac{p_ip_j}{p_i+p_j}\right),\\
p_c &\leq \frac{2k}{B^2C} + \frac{B-1}{BC}\left(1+\sum_{i=1}^{k/B}\sum_{j=1}^{k}\frac{P_ip_j}{P_i+p_j}\right),\\
p_s &= \frac{1}{B} + \frac{B-1}{B}\left(1-\left(1-\frac{1}{C}\right)^2\right).
\end{aligned}$$

Proof. As for the upper bound for in-place permutation, in this proof we derive lower bounds for $\Pr[X_i]$ and $\Pr[Y_i]$ and we will use these to derive the upper bounds on p_d and p_c. We make extensive use of the results obtained during the proof of Theorem 5.1.

Again, we consider a fixed i and consider the event X_i defined in the proof of Theorem 5.4. We now obtain a lower bound on $\Pr[X_i]$.

Lower bound on $\Pr[X_i]$

Letting $\Gamma(x) = 1 - f(x)$ and using Proposition 5.1 we can rewrite Eq. 5.5 as:

$$\Pr[X_i] \geq \sum_{m=0}^{\infty}\Pr[\mu = m]\mathrm{E}_{\bar{\mu}\sim\varphi(m,\bar{a}_i)}\left[1 - g(\bar{\mu}) - \Gamma(m+1) - \sum_{j=1}^{k}\Gamma(\mu_j)\right]. \tag{5.38}$$

While deriving the upper bound on $\Pr[X_i]$ we again assume $\bar{\mu}$ is drawn from the distribution $\varphi(m,\bar{a}_i)$ and we omit the subscript in $\mathrm{E}_{\bar{\mu}\sim\varphi(m,\bar{a}_i)}$.

We can use Eq. 5.12 as a simplification for

$$\sum_{m=0}^{\infty}\Pr[\mu = m]\mathrm{E}[g(\bar{\mu})],$$

and Eq. 5.14 as an upper bound on

$$\sum_{m=0}^{\infty}\Pr[\mu = m]\mathrm{E}[\sum_{j=1}^{k}\Gamma(\mu_j)].$$

So we just have to evaluate

$$\sum_{m=0}^{\infty}\Pr[\mu = m]\mathrm{E}[\Gamma(m+1)].$$

Since we always have at least one access to $\mathcal{S}$, we have that:

$$\begin{aligned}\sum_{m=0}^{\infty} \Pr[\mu = m]\mathrm{E}[\Gamma(\mu+1)] &= \sum_{m=0}^{\infty} \Pr[\mu = m]\frac{m+B}{BC} \\ &\quad - \sum_{m=BC-B}^{\infty} \Pr[\mu = m]\left(\frac{m+B}{BC} - 1\right) \\ &\leq \frac{1}{BC}\left[\sum_{m=0}^{\infty} \Pr[\mu = m]m + B\right] \\ &= \frac{1}{BC}\left(\frac{1}{p_i} - 1 + B\right), \qquad (5.39)\end{aligned}$$

where the last simplification used Proposition 5.2(c). Substituting Eq. 5.12, Eq. 5.14 and Eq. 5.39 in Eq. 5.38 we obtain the following lower bound for $\Pr[X_i]$:

$$\Pr[X_i] \geq 1 - \frac{1}{C}\sum_{j=1}^{k/B} \frac{P_j}{p_i + P_j} - \frac{1}{BC}\left(\frac{2}{p_i} + (B-1)\left(1 + \sum_{j=1}^{k} \frac{p_j}{p_i + p_j}\right)\right). \qquad (5.40)$$

Upper bound on p_d

Finally, substituting $\Pr[X_i]$ from Eq. 5.40 in Eq. 5.6 we get:

$$\begin{aligned}p_d &\leq \frac{1}{B} + \frac{B-1}{B}\sum_{i=1}^{k} p_i \\ &\quad \left(\frac{1}{C}\sum_{j=1}^{k/B} \frac{P_j}{p_i + P_j} + \frac{1}{BC}\left(\frac{2}{p_i} + (B-1)\left(1 + \sum_{j=1}^{k} \frac{p_j}{p_i + p_j}\right)\right)\right) \\ &= \frac{1}{B} + \frac{2(B-1)k}{B^2C} + \frac{B-1}{BC}\sum_{i=1}^{k}\sum_{j=1}^{k/B} \frac{p_i P_j}{p_i + P_j} \\ &\quad + \frac{(B-1)^2}{B^2C}\left(1 + \sum_{i=1}^{k}\sum_{j=1}^{k} \frac{p_i p_j}{p_i + p_j}\right). \end{aligned} \qquad (5.41)$$

We can evaluate p_c using a very similar approach to that used in the proof of Theorem 5.2. We again consider a fixed i and consider the event Y_i defined in the proof of Theorem 5.4. We now obtain a lower bound on $\Pr[Y_i]$.

Lower bound on $\Pr[Y_i]$

We can rewrite Eq. 5.35 as:

$$\Pr[Y_i] \geq \sum_{m=0}^{\infty} \Pr[\nu = m]\mathrm{E}_{\bar{\nu}\sim\varphi(m,\bar{b}_i)}\left[1 - \Gamma(m+1)\sum_{j=1}^{k} \Gamma(\nu_j)\right]. \qquad (5.42)$$

While deriving the lower bound on $\Pr[Y_i]$ we assume that $\bar{\nu}$ is drawn from the distribution $\varphi(m, \bar{b_i})$ and we omit the subscript in $\mathrm{E}_{\bar{\nu}\sim\varphi(m,\bar{b_i})}$.

Eq. 5.18 gives us

$$\sum_{m=0}^{\infty} \Pr[\nu = m] \sum_{j=1}^{k} \mathrm{E}[\Gamma(\nu_j)].$$

Arguing as for Eq. 5.39

$$\sum_{m=0}^{\infty} \Pr[\nu = m]\mathrm{E}[\Gamma(m+1)] \quad \leq \quad \frac{1}{BC}\left(\frac{1}{P_i} - 1 + B\right). \tag{5.43}$$

Substituting Eq. 5.18 and Eq. 5.43 in Eq. 5.42 we obtain the following lower bound for $\Pr[X_i]$

$$\Pr[Y_i] \quad \geq \quad 1 - \frac{1}{BC}\left(\frac{2}{P_i} + (B-1)\left(1 + \sum_{j=1}^{k} \frac{p_j}{P_i + p_j}\right)\right). \tag{5.44}$$

Upper bound on p_c

Finally, substituting $\Pr[Y_i]$ from Eq. 5.44 in Eq. 5.9 we get:

$$\begin{aligned} p_c \quad &\leq \quad \sum_{i=1}^{k/B} P_i \frac{1}{BC}\left(\frac{2}{P_i} + (B-1)\left(1 + \sum_{i=1}^{k} \frac{p_j}{P_i + p_j}\right)\right) \\ &= \quad \frac{2k}{B^2C} + \frac{B-1}{BC}\left(1 + \sum_{i=1}^{k/B}\sum_{j=1}^{k} \frac{P_i p_j}{P_i + p_j}\right). \end{aligned} \tag{5.45}$$

Combining Eq. 5.41 and Eq. 5.45 proves the theorem. □

5.4.3 Lower bound

It is quite obvious that the lower bound for in-place permutation, given in Theorem 5.3, is a lower bound for out-of-place permutation.

5.4.4 Upper and lower bounds for uniformly-random data

Using the upper bound Theorem just proven, we now derive a corollary for an upper bound to the number of cache misses if the data is uniformly distributed.

Corollary 5.3 *If $p_1 = \ldots = p_k = 1/k$ then the number of cache misses in n rounds of Process "in-place" is at most:*

$$n\left(\frac{1}{B} + \frac{k(B+3)}{2BC} + \frac{k}{B^2C} + \frac{k}{BC}\right) + k\left(1 + \frac{1}{B}\right).$$

PROOF. Since P_i in p_c and P_j in p_d are both B/k in the equations in Theorem 5.5, we get that:

$$\begin{aligned}
p_d + p_c + p_s &\leq \frac{2}{B} + \frac{2(B-1)}{BC}\frac{k^2}{B}\frac{B/k}{B+1} + \frac{(B-1)^2}{B^2C}k^2\frac{1/k}{2} \\
&\quad + \frac{2k}{B^2C} + \frac{2(B-1)k}{B^2C} + \frac{B-1}{B}\left(1 - \frac{(C-1)^2}{C^2}\right) \\
&= \frac{2}{B} + \frac{2(B-1)}{BC}\frac{k}{B+1} + \frac{(B-1)^2}{B^2C}\frac{k}{2} \\
&\quad + \frac{2k}{B^2C} + \frac{2(B-1)k}{B^2C} + \frac{B-1}{B}\frac{2C-1}{C^2} \\
&\leq \frac{2}{B} + \frac{k}{C}\left[\frac{4}{B} + \frac{B-1}{2B}\right] + \frac{2k}{B^2C} + \frac{2}{C} \\
&= \frac{2}{B} + \frac{k(B+7)}{2BC} + \frac{2k}{B^2C} + \frac{2}{C}.
\end{aligned}$$

□

REMARK: Corollaries 5.1 and 5.3 shows that for uniformly distributed data, other than for small values of B, the number of cache misses during in-place and out-of-place permutations are quite close. As for an in-place permutation, one pass of uniform distribution sorting using out-of-place permutations incurs $O(n/B)$ cache misses if and only if $k = O(C/B)$.

Using Corollary 5.2 for the lower bound and Corollary 5.3 above, we see that when $k \leq C$ the lower bound is again within 3/2 of the upper bound and is much closer when $k \ll C$.

5.5 Cache analysis of multiple sequences access

Accessing k sequences is like Process "in-place" in Section 5.1.1 except that there is no interaction with a count array, so we delete step 2 and assumption (c). An analogue of Theorem 5.1 is easily obtained. An easy modification to the proof of Theorem 5.2 gives:

Theorem 5.6 *The expected number of cache misses in n rounds of sequence accesses is at most:*

$$k + n\left(\frac{1}{B} + \frac{k(B-1)}{B^2C} + \frac{(B-1)^2}{B^2C}\sum_{i=1}^{k}\sum_{j=1}^{k}\frac{p_ip_j}{p_i+p_j}\right).$$

Corollary 5.4 *If $p_1 = \ldots = p_k = 1/k$ then the number of cache misses in n rounds of sequence accesses is at most:*

$$n\left(\frac{1}{B} + \frac{k(B+3)}{2BC}\right) + k.$$

REMARK: From Corollary 5.4, $k = O(C/B)$ random sequences can be accessed incurring an optimal $O(n/B)$ misses. This essentially agrees with the results obtained by Mehlhorn and Sanders [65] and Sen and Chatterjee [80].

REMARK: Since its derivation ignored the effects of the count array, the lower bound in Theorem 5.3 applies directly to sequence accesses. Note that the lower bound we obtain for uniformly-random data, as stated in Corollary 5.2, is sharper than the lower bound of $0.25(1 - e^{-0.25k/C})$ obtained in [80]. For example, for $k = 2, C/2$ and $k = C$, the expected number of misses per item predicted by our analysis is $1/B, 1/4$ and $1/3$. The predictions for the same values of k using $0.25(1 - e^{-0.25k/C})$ and $e^{-x} \approx 1 - x$ for small x are $1/(8C), 1/32$ and $1/16$.

5.6 Correspondence between the permute phase and Processes "in-place" and "out-of-place"

We now show how the Processes "in-place" and "out-of-place" model the permute phase of a generic distribution sorting algorithm.

The correspondence between Process "in-place" of Section 5.1.1 and the pseudo-code in Figure 4.4 is as follows. Each iteration of the inner loop (steps 3.1-3.5) of the pseudo-code corresponds to a round of Process "in-place". The array `COUNT` corresponds to the locations $\mathcal{C}$, and the pointer D_i points to `DATA[`idx`]`. The variables x in the process and the pseudo-code play a similar role. It can easily be verified that in each iteration of the loop in the pseudo-code, the value of x is any integer $1, \ldots, k$ with probability $p_1, \ldots, p_k$, independently of its previous values, as in Step 1 of Process "in-place". A read at a location immediately followed by a write to the same location is counted as one access. Thus, the read and increment of `COUNT[`x`]` in Steps 3.2 and 3.3 of the pseudo-code constitutes one access, equivalent to Step 2. Similarly the "swap" in Step 3.5 of the pseudo-code corresponds to the memory access in Step 3 of the process. The process does not model the initial access in Step 1 of the pseudo-code, and nor does it model the task of looking for new cycle leaders in Steps 4 and 5 of the pseudo-code.

The correspondence between Process "out-of-place" of Section 5.1.2 and the pseudo-code in Figure 4.2 is as follows. The array `COUNT` corresponds to the locations $\mathcal{C}$, the array `DATA` corresponds to the locations $\mathcal{S}$, i in the pseudo-code corresponds to s in the process, and the pointer D_i points to `DEST[`idx`]`. The increment of i in the pseudo-code is equivalent to the increment of s in the process, and the accesses to `DATA[`i`]` and $\mathcal{S}[s]$ are equivalent. As above, the variables x in the process and the pseudo-code in the pseudo-code play a similar role. Again, the read and increment of `COUNT[`x`]` of the pseudo-code constitutes one access, and is equivalent to Step 3 of the process. The access to `DEST[`idx`]` of the pseudo-code corresponds to the memory access in Step 4 of the process.

Assumption (b) of the processes is clearly satisfied and assumption (c) can normally be made to hold. Assumption (d) and $k \leq CB$ or $k \leq C$ may not hold in practice, in the next chapter we give an approximate analysis which deals with this. Assumption (a) of the processes, that the starting locations of the pointers D_i are uniformly and independently distributed, is patently false, we discuss this in more detail in Chapter 6. We may force it to hold by adding random offsets to the starting location of each pointer, at the cost of needing more memory and adding a compaction phase after the permute, this has also been suggested by Mehlhorn and Sanders [65]. This only works if the permute is not in-place, and if k is sufficiently small (e.g. $k \leq n/(CB)$). In Chapter 6 we study assumption (a) empirically in the context of uniform distribution sorting. Another weakness is that our processes are continuous, so the sequence lengths are not specified, whereas in distribution sorting we sort n keys and each sequence is of a finite length.

5.7 Summary

In this chapter we have analysed the average-case cache performance of the permute phase of distribution sorting when the keys are independently but not uniformly distributed. We have presented equations for the number of misses during in-place and out-of-place permutations and have given closed-form upper and lower bounds on these. We have shown that the upper and lower bounds are quite close when $k \leq C$ and the data is known to be independently and uniformly distributed. We have shown how this analysis can easily be extended to obtain the number of cache misses during accesses to multiple sequences.

Chapter 6

Approximate analysis and validation of predictions in the CMM

In this chapter we again consider the problem of sorting floating-point numbers in the CMM. We now assume the keys are independently drawn from a uniform distribution. We present an approximate analysis of the number of cache misses for the permute phase of distribution sorting. The reasons the analysis in Chapter 5 may not always be applicable and why this analysis is needed are:

- Knuth tells us to use $\Theta(n)$ classes for distribution sorting [53], this means that the classes can be quite small, such that there may be more than one class in a cache block. Due to assumption (d) of processes “in-place” and “out-of-place” the analysis in Chapter 5 is inaccurate if there are multiple pointers in each memory block.

- In theory we can use Theorems 5.1 and 5.4 with uniform probabilities to determine the exact number of cache misses when permuting uniformly-randomly keys. However the expressions in Theorems 5.1 and 5.4 are hard to evaluate and can in practice only be approximated, using a Monte Carlo method or with the closed form upper and lower bounds. We will see in the next chapter that because the approximate analysis gives closed form expressions it can be used to tune distribution sorting algorithms more easily than the analysis in Chapter 5.

- The closed form upper and lower bounds from Chapter 5 are apart by a factor of 3/2, so either of these bounds could be quite far from the actual number of misses.

The analysis is for an in-place permutation and could easily be extended to handle out-of-place permutations. We consider the accuracy of the approximate analysis by comparing predicted cache miss rates against measured cache miss rates. We then consider why assumptions made in the analysis in this chapter and in Chapter 5 about the random distribution of pointers in the cache may not be true in practice and then give heuristics that attempt to satisfy these assumptions.

6.1 Approximate analysis

In this section we present an approximate analysis of an in-place permutation in a generic distribution sorting algorithm when the keys are independently drawn from a uniformly-random distribution. The analysis is based on the fact that for given values of n, C and B, there is only one parameter that can be changed in one pass, the number of classes to distribute into, k. We determine how cache miss rates vary as k varies.

The permute phase can be viewed as alternating cycle-following with finding cycle leaders. During the cycle following phase we make $2n$ memory accesses which may lead to cache misses. These comprise alternating accesses to the count array and to the data array. Any access to the data array must be to one of the k *active locations* pointed to by the *data pointers* for the k classes (the data pointer for class j points to the next available location for a key of class j). Since the keys are independently and uniformly-random distributed, it is not hard to see that for any fixed i, the ith access to the count array is equally likely to be to any of the k count array locations, independently of the previous accesses to the count array; similarly, the ith access to the data array is equally likely to access any of the k active locations, independently of the previous accesses to the data array.

However, this is not enough to analyse the permute phase exactly. For example, it appears difficult even to get a good understanding of the distribution of the active locations at the start of the permute phase. This analysis uses the idea that although the active locations change as keys are moved to their destinations within classes, the movement is at roughly the same rate and thus the active locations should maintain roughly the same relative position. Therefore, we fix some 'typical' positions of active locations, giving a 'snapshot' of the algorithm, and analyse a simple process which involves accesses to the count array and to the fixed active locations. Of course, the accuracy of the formulae obtained will depend heavily on how well the active locations are fixed. The analysis also ignores other accesses, such as those for the cycle leader finding phase.

In what follows, an *active block* is either a count block or a data block that has an active location in it. Having fixed the active locations, we consider a sequence of memory accesses, where each access reads or writes one of the k count array locations or one of the k active locations with equal probability $1/k$, independently of the previous accesses[1]. We calculate the probability p_d of a miss if an active location is accessed, and the probability p_c of a miss if a count array location is accessed; we then estimate:

$$\mu_{perm} := 1/B + (1 - 1/B)p_d + p_c. \tag{6.1}$$

The reason for using this formula is that p_d only estimates the probability of misses on data array accesses due to conflicts between active blocks. However, the n accesses to the data array cover n distinct locations in all, resulting in at least n/B compulsory misses. Thus, p_d should be applied only to the approximately $n(1 - 1/B)$ accesses that remain. If we wish to compare the results to those obtained in Chapter 5, we can use $n\mu_{perm}$ to get the approximate number of misses in one permutation pass.

Using the probabilities specified in the above process, and having fixed the active locations, we can easily calculate the probability that the process accesses a given active block. For example, we can infer that a count block has an access probability of B/k. After this, the analyses make extensive use of the following elementary idea [59]. Suppose we consider a process which accesses l blocks of memory $b_1, \ldots, b_l$, with each access being made independently to block b_i with probability p_i, for $i = 1, \ldots, l$. Let $S \subseteq \{1, \ldots, l\}$ and suppose that exactly the blocks $\{b_i | i \in S\}$ are mapped to some cache block c. Let c be randomly initialised to b_i, $i \in S$ with probability p_i. Then, at

[1]These probabilities add up to 2, and should be normalised to $1/(2k)$ each. Since we are only interested in the ratios of probabilities, we can omit the normalisation to improve readability.

any stage in the process, for any $i \in S$:

$$\Pr[b_i \text{ is in cache block } c] = \frac{p_i}{\sum_{j \in S} p_j}. \tag{6.2}$$

An arbitrary initialisation of the cache may result in at most one more miss per cache block than given by the above equation, which is negligible. Equation 6.2 implies the following as well. If there are τ data blocks mapped to a cache block and the number of active locations in data block $1, \ldots, \tau$ is $n_1, n_2, \ldots, n_\tau$ and $N = \sum_{i=1}^{\tau} n_i$ then the probability of a hit in accesses to that cache block is:

$$\frac{1}{N^2} \sum_{i=1}^{\tau} {n_i}^2. \tag{6.3}$$

For a given value of N, by Jensen's inequality (see e.g. [75]), the probability of a hit is minimised when $n_i = N/\tau$ for $i = 1, \ldots, \tau$. A similar calculation can be made when τ data blocks, with N active locations in all, conflict with one or more count blocks.

In order to fix the active locations, we consider four cases. The first pair of cases deal with the situation that the expected class sizes are smaller than the block size B. If so, it will often be the case that there are multiple active locations in a single data block. The third and fourth case deal with the situation where the expected class sizes are much larger than B. In this case it is (intuitively) very unlikely that two active locations are present in a single data block. We have chosen to cover these four cases as they occur in the algorithms that we will study.

Case 1: $k \leq CB$ and $n/k < B$

In this case the count array fits in cache and the expected class size is less than a cache block. We assume that there are $p = k/(n/B)$ data pointers in each data block, allowing p to be non-integral. By the reasoning associated with Equation 6.3, this should overestimate the number of misses. There are two regions in the cache: region R_1 which has no count blocks mapped to it, and region R_2 which has count blocks mapped to it.

We first consider the case $n \geq CB$. In this case each cache block in R_1 has $\tau = n/(CB)$ data array blocks mapped to it, whereas each block in R_2 has one count block and τ data blocks mapped to it (recall that τ is assumed to be integral). The probability of accessing a count array block is B/k and of accessing any data block is $p \cdot (1/k) = B/n$. For any cache block c in R_2, an access to a data block which is mapped to c is a hit only if that data block is currently stored in c. As the sum of the probabilities of the blocks mapped to c is $B/k + \tau \cdot B/n$, by Equation 6.2 we have:

$$\Pr[\text{Data access in } R_2 \text{ is a hit}] := \frac{B/n}{B/k + \tau \cdot B/n} = \frac{1}{n/k + \tau}. \qquad (6.4)$$

Similarly, we get:

$$\Pr[\text{Count access in } R_2 \text{ is a hit}] := \frac{B/k}{B/k + \tau \cdot B/n} = \frac{B}{B + k/C}. \qquad (6.5)$$

In region R_1, there are τ active blocks (from the data array) mapped to each cache block. As each active block has the same probability of being accessed, we get that $\Pr[\text{Data access in } R_1 \text{ is a hit}] = \tau^{-1}$. Since a data access goes to R_1 with probability $1 - k/(CB)$, and R_2 with probability k/CB, we have:

$$p_d := \left(1 - \frac{k}{CB}\right)\left(1 - \frac{1}{\tau}\right) + \frac{k}{CB}\left(1 - \frac{1}{n/k + \tau}\right). \qquad (6.6)$$

As there are no count array accesses in R_1, $1 - p_c$ is given by Equation 6.5. By Equation 6.1:

$$\begin{aligned}\mu_{perm} \;:=\; & \frac{1}{B} + \left(1 - \frac{B}{B + k/C}\right) + \\ & \left(\frac{B-1}{B}\right)\left[\left(1 - \frac{k}{CB}\right)\left(1 - \frac{1}{\tau}\right) + \frac{k}{CB}\left(1 - \frac{1}{n/k + \tau}\right)\right], \qquad (6.7)\end{aligned}$$

if $n \geq CB$, and where $\tau = n/(CB)$.

We now consider the case $n < CB$. In this case data blocks cannot conflict with each other so there are no cache misses in R_1. To determine the cache misses in R_2 we assume the start of the data array is uniformly and randomly distributed at cache block boundaries so the probability of a data block being mapped to a cache block in region R_2 is $\frac{k/B}{C}$. Using the above reasoning, if a data block is mapped to R_2 the probability that an access to it results in a hit is $\frac{B/n}{B/k+B/n} = \frac{1}{n/k+1}$. This gives $p_d = \frac{k}{CB}\left(1 - \frac{1}{n/k+1}\right)$. A given count block has probability $\frac{n/B}{C}$ of being in conflict with a data block. In case there is a conflict, the probability of a hit on a count array access is $1/(1 + k/n)$. This gives $p_c = \frac{n}{CB}\left(1 - \frac{1}{1+k/n}\right)$. Using Equation 6.1 we get:

$$\mu_{perm} := \frac{1}{B} + \frac{n}{CB}\left(1 - \frac{1}{1 + k/n}\right) + \left(\frac{B-1}{B}\right)\left[\frac{k}{CB}\left(1 - \frac{1}{n/k + 1}\right)\right], \qquad (6.8)$$

if $n < CB$. Equation 6.8 is applicable to a distribution sorting passes where all the keys in a problem fit in the cache. The misses for a given sorting problem may be more or less than the expected value, depending on the exact number of data blocks in R_2. However, averaged over sorting problems which are consecutive in memory, the result should be accurate since an under-estimate for some problems will lead to an overestimate for other problems.

Case 2: $k > CB$ and $n/k < B$

In this case the size of the count array exceeds the cache size and the expected class size is less than a cache block. Equation 6.11 is derived in a manner similar to Case 1. First, we assume that k is a multiple of CB, and apply the above reasoning, taking region R_1 to be empty and R_2 as the whole cache. We will use the same formula when k is not a multiple of CB. If k is a multiple of CB, then we have $k/(CB)$ count blocks mapped to each cache block, so we get that:

$$\Pr[\text{Data access is a hit}] \;:=\; \frac{B/n}{1/C + \tau \cdot B/n} = \frac{1}{2\tau} = \frac{CB}{2n}. \tag{6.9}$$

Similarly, we get:

$$\Pr[\text{Count access is a hit}] \;:=\; \frac{B/k}{1/C + \tau \cdot B/n} = \frac{n/k}{2\tau} = \frac{CB}{2k}. \tag{6.10}$$

Since R_1 is empty, $1 - p_d$ is given by Equation 6.9 and $1 - p_c$ is given by Equation 6.10, so the final formula is:

$$\mu_{perm} := 1/B + (1 - CB/(2k)) + ((B-1)/B) \cdot (1 - CB/(2n)). \tag{6.11}$$

Case 3: $k \leq CB$ and $n/k \gg B$

In this case the count array fits in cache and the expected class size is much larger than a cache block. Since $n/k \gg B$ we assume that there is at most one data pointer per data block. Again, we divide the cache into two regions: region R_1 which has no count blocks mapped to it, and region R_2 which has count blocks mapped to it. Note that R_2 occupies a $k/(CB)$ fraction of the cache. We assume that $N = k \cdot (k/(CB))$ data pointers are mapped to R_2. Again, by Jensen's inequality, the overall hit rate is minimised when each of the k/B blocks comprising R_2 has $N/(k/B) = k/C = \rho$ data pointers mapped to it. We proceed with the calculation based upon ρ data pointers mapped to each cache block in R_2, each with an access probability of $1/k$; note, however, that ρ may be non-integral, and possibly even much smaller than 1. Consider a cache block i in R_2. The count block mapped to i is accessed with probability B/k, and each of the ρ data blocks mapped to i are accessed with probability $1/k$. The sum of the access probabilities of the blocks mapped to i is $B/k + \rho \cdot (1/k) = B/k + 1/C$. Thus:

$$\Pr[\text{Count access in } R_2 \text{ is a hit}] := \frac{B/k}{B/k + 1/C} = \frac{B}{B+\rho}, \tag{6.12}$$

$$\Pr[\text{Data access in } R_2 \text{ is a hit}] := \frac{1/k}{B/k + 1/C} = \frac{1}{B+\rho}. \tag{6.13}$$

It is more interesting to study the hit rate in region R_1, which only has data pointers mapped to it. It will be convenient to assume that R_1 covers the entire cache, and has k data pointers mapped to it; as we will see, we can scale down the values without changing the hit rate. Let the number of data pointers mapped to cache block i be m_i. If cache block i has $m_i \neq 0$, then the probability that a data array access reads cache block i is m_i/k, but the probability of this access being a hit is $1/m_i$. Hence, the probability of a hit given that cache block i was accessed is $1/k$. Summing over all i such that $m_i \neq 0$ gives the overall hit rate as simply ν/k, where ν is the number of cache blocks such that $m_i \neq 0$. Assuming that the data pointers are independently and uniformly located in cache blocks, we calculate the expected value of ν as:

$$C \cdot (1 - (1 - 1/C)^k) \approx C \cdot (1 - e^{-\rho}). \tag{6.14}$$

Hence we have:

$$\Pr[\text{Data access in } R_1 \text{ a hit}] := \nu/k = \rho^{-1}(1 - e^{-\rho}), \tag{6.15}$$

and note that this is invariant to scaling C and k by the same amount. We now derive the misses per key for this case. First, the probability that a data access is to R_1 is $1 - k/(CB)$; using Equations 6.15 and 6.13, we get that $p_d = (1 - k/(CB)) \cdot 1/(B + \rho) - (1 - k/(CB)) \cdot \rho^{-1}(1 - e^{-\rho})$, and hence using Equations 6.1 and 6.12 we get that:

$$\begin{aligned} \mu_{perm} \;:=\; & \frac{1}{B} + \left(1 - \frac{B}{B+\rho}\right) + \\ & \frac{B-1}{B}\left[1 - \frac{k}{CB}\left(\frac{1}{B+\rho}\right) - \left(1 - \frac{k}{CB}\right)\left(\rho^{-1}(1 - e^{-\rho})\right)\right], \end{aligned} \tag{6.16}$$

where $\rho = k/C$.

Case 4: $k > CB$ and $n/k \gg B$

Finally, for the sake of completeness we study the case where $k > CB$ and $n/k \gg B$. The approach is similar to that of Case 2: we assume that k is a multiple of CB, and use the formula even in the case that it is not. If k is a multiple of CB, then we have $k/(CB)$ count blocks mapped to each cache block, along with $\rho = k/C$ active locations; this gives the probability of a count hit as $\frac{B}{\rho + B \cdot k/(CB)} = CB/(2k)$. The probability of a data hit is $\frac{1}{\rho + B \cdot k/(CB)} = C/(2k)$. Thus:

$$\mu_{perm} := 1/B + (1 - CB/(2k)) + ((B-1)/B)(1 - C/(2k)), \tag{6.17}$$

where $\rho = k/C$.

6.1.1 Dissimilar key and count array element sizes

The above analysis assumes that count array elements and keys are of the same size. This may not always be true in practice, for instance, when sorting 64-bit double-precision floating-point keys one may well use 32-bit integers for count array elements. We let B be the number of keys in a cache block and αB be the number of count array elements in a cache block. Following the approach above, we now give the probability of a cache miss for the above four cases in terms of B and αB.

Case 5: $k \leq \alpha CB$ and $n/k < B$

$$\begin{aligned}\mu_{perm} \quad := \quad & \frac{1}{B} + \left(1 - \frac{\alpha B}{\alpha B + k/C}\right) + \\ & \left(\frac{B-1}{B}\right)\left[\left(1 - \frac{k}{\alpha CB}\right)\left(1 - \frac{1}{\tau}\right) + \frac{k}{\alpha CB}\left(1 - \frac{1}{\alpha n/k + \tau}\right)\right],\end{aligned} \tag{6.18}$$

if $n \geq CB$, and where $\tau = n/(CB)$.

$$\mu_{perm} \quad := \quad \frac{1}{B} + \frac{n}{CB}\left(1 - \frac{\alpha}{\alpha + k/n}\right) + \left(\frac{B-1}{B}\right)\left[\frac{k}{\alpha CB}\left(1 - \frac{1}{\alpha n/k + 1}\right)\right] \tag{6.19}$$

if $n < CB$.

Case 6: $k > \alpha CB$ and $n/k < B$

$$\mu_{perm} := 1/B + (1 - \alpha CB/(2k)) + ((B-1)/B) \cdot (1 - CB/(2n)). \tag{6.20}$$

Case 7: $k \leq \alpha CB$ **and** $n/k \gg B$

$$\begin{aligned}\mu_{perm} &:= \frac{1}{B} + \left(1 - \frac{\alpha B}{\alpha B + \rho}\right) + \\ &\frac{B-1}{B}\left[1 - \frac{k}{\alpha CB}\left(\frac{1}{\alpha B + \rho}\right) - \left(1 - \frac{k}{\alpha CB}\right)\left(\rho^{-1}(1 - e^{-\rho})\right)\right], \quad (6.21)\end{aligned}$$

where $\rho = k/C$.

Case 8: $k > \alpha CB$ **and** $n/k \gg B$

$$\mu_{perm} := 1/B + (1 - \alpha CB/(2k)) + ((B-1)/B)(1 - C/(2k)), \quad (6.22)$$

where $\rho = k/C$.

6.2 Cache simulations

In order to validate the cache analysis we ran implementations of the algorithms in a mode where we simulated accesses to a single-level, direct-mapped cache.

Whenever the program makes an access to a data structure used in the algorithm, the simulator checks if the address being accessed is in the cache. When simulating sorting algorithms, we checked the cache for all array accesses but not for accesses to control variables, such as indices, which, it can be assumed, would be held in registers.

6.3 Accuracy of predictions

Appendix A summarises extensive experiments which study the cache misses during the permute phase in a generic distribution sorting phase. The experiments are for $n/k \leq B$, covering Cases 1 and 2 of the approximate analysis, and $k \leq CB$ and $n/k \gg B$, covering Case 3 of the approximate analysis.

Table A.1 in Appendix A shows cache misses per key during the permute phase for $n = 2^{18}, 2^{19}$, 2^{20}, 2^{21}, 2^{22}, 2^{23}, 2^{24}, 2^{25} and $n = 2^{26}$ at expected class sizes of $B/4$, $B/2$ and B, where $B = 16$. We see that the values predicted by Equations 6.7 and 6.11 are quite accurate.

Tables A.2 and A.3 in Appendix A shows cache misses per key during the permute phase for $n = 2^{24}$ and 100 random values of k between 25 and 8192 and for some selected values of k. Excluding the selected values of k we see that the values predicted using Equation 6.16 are quite close to the simulated values, indicating that the equation seems to be accurate for most values of k. However, the equations are very inaccurate for the selected values of k, which are: 128, 256, 384, 512, 640, 768, 896, 1024, 1152, 1280, 1408, 1536, 1664, 1792, 1920, 2047, 2048, 2049, 4095, 4096 and 4097.

This suggests that the assumption of random data pointer placement is not always a good one. We now analyse this further. To simplify the analysis, we make the assumption that a 'typical' value for the number of non-empty blocks, ν, is simply the value of ν at the start of the permute phase. As argued before, one would expect all data pointers to move at roughly the same rate, hence maintaining the original pattern of data pointers. Tables A.2 and A.3 can be used to validate this. We see that, for most k, the misses per key predicted using actual ν at the start of the permute phase are quite close to the simulated values. From now onwards, the discussion will focus on the initial value of ν.

First, note that it is surprising that predictions made by assuming a uniform and independent mapping of data blocks are at all accurate, since these assumptions are manifestly untrue. The expected starting location of the l-th data pointer in the data array is precisely the expected number of elements in classes $0, \ldots, l-1$, and so it very much depends on the locations of the previous pointers. Furthermore, the number of elements in classes $0, \ldots, l-1$ is a binomial random variable with expected value $l \cdot n/k$, and is not uniformly distributed over cache blocks.

We can make some observations regarding the expected starting locations of the data pointers. If n is a multiple of CB then one can reason about this particularly simply. We can view the cache as being 'continuous' and place k *marks* on the cache numbered $0, 1, \ldots, k-1$, with the i-th mark being $i \cdot (CB)/k$ words from the beginning of the cache. Letting $\tau = n/(CB)$, we get that the expected number of elements in classes $0, \ldots, l-1$ is $l \cdot n/k = (l\tau) \cdot (CB)/k$. Hence the expected starting location in cache of the l-th data pointer is seen to be at the mark numbered $(l\tau) \bmod k$.

The number of distinct marks among $0, \tau \bmod k, 2\tau \bmod k, \ldots, (k-1)\tau \bmod k$ depends upon the gcd of τ and k; more precisely, if $\gcd(\tau, k) = g$ then there will be k/g marks, each of which is the expected starting location of g data pointers. If g is large, one can expect the number of non-empty blocks to be quite small at the start of the permute phase. For example, in Tables A.2 and A.3, the prediction using Equation 6.16

is very inaccurate when $k = 512$. Since $n = 2^{24}$, giving $\tau = 128 = \gcd(\tau, k)$, and there are only $512/128 = 4$ different marks where the data pointers' expected locations lie, and one would expect ν to be very small at the outset.

However, it appears that it is difficult to obtain a simple closed-form expression for the initial value of ν for arbitrary k. One way out might be to choose k so that the analysis is made convenient. We now present two heuristics for selecting k and show that the first heuristic though plausible is not always effective, whereas the second appears to be effective.

6.4 Heuristic 1 for selection of k

If $\tau = n/CB$ is an integer, and we choose k to be relatively prime to τ, then we know that all the data pointers are expected to start at different marks. As the marks are precisely evenly spaced in the cache, it would appear that this heuristic is optimal. Unfortunately, the examples of $k = 2047$ or $k = 2049$ below, taken from Tables A.2 and A.3, show that this is not so:

k	Simulate	Predict	k	Simulate	Predict	k	Simulate	Predict
2047	0.245	0.198	2048	0.320	0.198	2049	0.241	0.198

In the experiment, $\tau = 128$ and $\gcd(2047, 128) = \gcd(2049, 128) = 1$. Although all the data pointers are expected to start at different marks, the miss rates for $k = 2047$ and 2049 are still quite high. Note that the expected data pointer starting locations for $k = 2047$ are as follows:

mark	0	1	2	3	...	126	127	128	129	130	...
pointer	0	16	32	48	...	2016	2032	1	17	33	...

The standard deviation of the lth data pointer location is $\sqrt{n \cdot (l/k) \cdot (1 - l/k)}$. Thus, data pointers with low or high indices have low standard deviation, so the data pointers which will stay close to their expected positions are all clustered around marks $0, 128, 256 \ldots$, while the data pointers which are expected to start around mark $64, 192, 320, \ldots$ all have high standard deviation and can vary considerably from their expected starting position. Hence there is a concentration of pointers again around marks $0, 128, \ldots$. We see other occurrences of this at $k = 2239$, 2945, 3841, 4095, 4096 and 5634, where $\gcd(\tau, k-1)$ or $\gcd(\tau, k+1)$ is large.

We do not know how to compensate for this additional requirement in this heuristic, but there are other serious disadvantages: one is that the heuristic only works when n is a multiple of CB, and another, more theoretical, objection is that for sufficiently large n, it may not be possible to pick k within a certain range of values which is relatively prime to τ. For instance, if we wish to pick a good k from the range $[k_1, \ldots, k_2]$ for integers k_1 and k_2, this is clearly impossible if τ is $k_1 \cdot (k_1 + 1) \cdot \ldots \cdot k_2$.

6.5 Heuristic 2 for selection of k

Another heuristic for selecting k comes from the following fact [53, pp. 550, Q8,9]. Let θ be an irrational number which has the form $\frac{1}{l+\phi^{-1}}$ where $l \geq 1$ is an integer and $\phi = (1 + \sqrt{5})/2$. If $\langle x \rangle$ denotes $x - \lfloor x \rfloor$, then for all $t \geq l$ the smallest segment of $[0, 1]$ that is induced by the points $\langle \theta \rangle, \langle 2\theta \rangle, \ldots, \langle t\theta \rangle$ is at most $1 + \phi$ times smaller than the largest segment. By symmetry, if θ is of the above form, then the intervals formed by $\langle -\theta \rangle, \langle -2\theta \rangle, \ldots, \langle -t\theta \rangle$ are also roughly evenly sized. We can use the above to get a 'good' value k^* in the rough vicinity of a given value k. The value k^* is chosen such that $n/k^* = \gamma CB$, where γ is chosen from:

$$\{1/(l + \phi^{-1}) \mid l = 1, 2, 3, \ldots\} \cup \{j \pm 1/(l + \phi^{-1}) \mid j = 1, 2, \ldots \text{and } l = 1, \ldots, 10\} \quad (6.23)$$

(There may be some duplication among these values: for instance $1/(1 + \phi^{-1}) = 1 - 1/(2 + \phi^{-1})$.) Suppose now that we choose $k^* = n/(\gamma CB)$, where $\langle \gamma \rangle = \pm 1/(r + \phi^{-1})$ for some integer $r \geq 1$. Again, if we consider the cache to be continuous, the ith class has an expected starting location which is offset $\langle i\gamma \rangle CB$ locations from the start of the cache. By the above fact, we may conclude that the expected starting locations of the classes are roughly evenly spaced in the cache, provided that $k^* \geq r$. Furthermore, all data pointers for classes $\geq r$ are evenly spaced with respect to the data pointers for lower-numbered classes.

The limit $l \leq 10$ in Equation 6.23 when $j \geq 1$ is somewhat arbitrary and is chosen so that k^* can always be chosen to within about 10% of k, provided that $n/k \geq 0.5CB$, as can easily be verified. As an example the 'good' values of n/k in the range $[0.5CB, 1.5CB]$ are (roughly) $0.62CB$, $0.72CB$, $0.79CB$, $0.82CB$, $1.18CB$, $1.21CB$, $1.28CB$ and $1.38CB$, so the nearest larger approximations to $n/k = 0.5CB$ and $n/k =$

CB are respectively $0.62CB$ and $1.18CB$. (When n/k is small we can remove the restriction on l, as the data pointers $0, \ldots, l-1$ anyway do not collide with each other.)

It should be noted that this gives a non-integral value of k, but the algorithm remains essentially unchanged. In the analysis, the last pointer now may have a lower access rate (as the final class may be small), but the effect appears to be negligible. The main advantages of this approach are: (i) it works even when n is not a multiple of CB and (ii) the low-numbered and high-numbered data pointers are also equally spaced. Table A.7 in Appendix A shows that this heuristic appears to place pointers more or less at random.

6.6 Summary

In this chapter we have given an approximate analysis of the permute phase of distribution sorting when the keys are independently drawn from a uniformly random distribution. We have shown that the distribution of data pointers at the start of the permute phase is a good predictor of the number of cache misses per key. We have shown why the assumption that pointers are uniformly and randomly distributed in cache blocks may not be true and have given two heuristics which attempt to satisfy this assumption. The analysis has been validated using cache simulations.

Chapter 7

Sorting random numbers in the CMM

The cache miss analyses of the last two chapters can be used to tune distribution sorting algorithms. However, so far we only have an analysis of the permute phase. So, we start this chapter by doing an analysis of the cache misses during the count phase of a generic distribution sorting algorithm, when keys are uniformly and randomly distributed to classes.

We then describe an algorithm that sorts uniformly-randomly distributed floating-point numbers using one pass of distribution sorting. Using the analysis in Chapter 6 and Section 7.1, we show that this algorithm has very poor cache performance and hence running times. We then derive a multiple pass version of the algorithm that has good cache performance and is over twice as fast as the single pass algorithm.

We then discuss the problem of sorting floating-point numbers using their integer representation. This induces a non-uniform distribution of keys to classes and we use the analysis in Chapter 5 to tune this algorithm.

7.1 Cache analysis of count phase

In this section we analyse the cache misses during the count phase of distribution sorting, when the keys are uniformly and randomly distributed to classes. We give three cases which state the approximate number of cache misses, given the various parameter choices that may be made during one pass of distribution sorting.

These results are obtained by slight modifications to the results in [56]. We first briefly describe the results in [56] and then discuss how the results have been extended to deal with the count phase in distribution sorting.

The analysis in [56] assumes that the cache is single-level and direct-mapped, that count array locations are accessed uniformly and randomly, that $n = xCB$ for integer $x \geq 2$ and that $k = yB$ for integer $y \geq 1$. The expected number of cache misses per memory access predicted in that analysis is as follows:

$$T + R + I + I^{'}.$$

Where T is the cache misses per memory access due to the traversal of the data array and is given by:

$$T = \frac{1}{2B}.$$

R is the expected number of cache misses per memory access due to memory blocks from the count array conflicting with each other and is given by:

$$R = \begin{cases} 0 & \text{if } k \leq CB \\ \frac{1}{2}(1 - \frac{C}{k/B}) & \text{if } k > CB. \end{cases}$$

I is the expected number of cache misses per memory access caused by the traversal periodically capturing each block in the cache and is given by:

$$I = \frac{k/B}{2CB}\left[1 - \left(1 - \frac{1}{k/B}\right)^{CB-(B-1)}\right]$$

when $k \leq CB$, otherwise it is approximately:

$$I \approx \frac{C}{2k}\left[1 - \left(1 - \frac{1}{C}\right)^{CB-(B-1)}\right].$$

$I^{'}$ is the expected number of cache misses per memory access caused by random count array accesses to a block that is mapped to the same one as being traversed and is given by:

$$I' = \begin{cases} \frac{B-1}{CB} & \text{if } k \leq CB \\ \frac{B-1}{2CB}\left(1 + \frac{C}{k/B}\right) & \text{if } k > CB. \end{cases}$$

There are n memory accesses to the data array and n memory accesses to the count array, so to remain consistent with the approach in Chapter 6 and express these results in terms of cache misses per key we have to multiply each of the terms above by a factor of 2.

These results are only for the cache misses during the frequency counting step of the count phase (Step 2 of the count phase pseudo-code shown in Figure 4.1) and they do not take into account cache misses to initialise and prefix-sum the count array (Steps 1 and 3).

Assuming the cache is empty at the start of the count phase there will be k/B cache misses to initialise the count array. If the cache does not contain any count blocks before the prefix-sum phase, because the first C blocks were removed due to the traversal of data, then there will be an additional k/B cache misses. So there will be at most $(2k/B)/n$ cache misses per key to initialise and prefix-sum the count array.

We now give three cases which describe the cache misses per key during the count phase when distribution sorting uniform keys. In the first two cases we ignore the fact that n may not be a multiple of CB as the effect of this to the cache miss rate is not significant - only the values of I and $I^{'}$ would be inaccurate and their contribution is small. In all three cases we ignore the fact that k may not be a multiple of B.

Case 1: $n > CB$ **and** $k > CB$

In this case the data and count arrays are larger than the cache. The misses per memory access are as described by [56] plus at most $2(k/B)/n$ misses per key for initialisation and prefix-summation of the count array. Thus re-writing $2(T+R+I+I^{'})+(2k/B)/n$ we obtain:

$$\begin{aligned}\mu_{count} \quad := \quad & \frac{1}{B}+\frac{k-CB}{k}+\\ & \frac{C}{k}\left[1-\left(1-\frac{1}{C}\right)^{CB-B+1}\right]+\frac{(B-1)(k+CB)}{CBk}+\frac{2k/B}{n}. \end{aligned} \tag{7.1}$$

Case 2: $n > CB$ **and** $k \leq CB$

In this case the data array is larger than the cache but the count array fits in cache. The misses per memory access are as described by [56] plus at most $2(k/B)/n$ misses per key for initialisation of the count array. Thus re-writing $2(T+R+I+I^{'})+(2k/B)/n$ we obtain:

$$\mu_{count} := \frac{1}{B}+\frac{k/B}{CB}\left[1-\left(1-\frac{B}{k}\right)^{CB-B+1}\right]+\frac{2(B-1)}{CB}+\frac{2k/B}{n}. \tag{7.2}$$

Case 3: $n \leq CB$ **and** $k \leq CB$

In this case the data and count arrays fit in cache. Most of the misses here are as in Cases 1 and 2 above. There are $(k/B)/n$ misses per key for initialisation of the count array. The misses per key for the traversal acting alone, from T, is $1/B$. There are no conflict misses between count blocks.

For cache block i which has a count block and a data block mapped to it, the misses per key due to random accesses to the count block while the cache block is being traversed by the data array, from I', is $\frac{2(B-1)}{CB}$. After the last data traversal access in i the count block is no longer in i so the count block must be re-loaded at most once, either during the remaining data traversal accesses or during prefix-summation.

The probability that cache block i has a count block mapped to it is $(k/B)/C$ and the probability that cache block i has a data block mapped to it is $(n/B)/C$. Since these events are independent, the probability that cache block i has a count and a data block mapped to it is $\frac{k/B}{C}\frac{n/B}{C}$. So the overall number of cache misses per key is:

$$\mu_{count} := \frac{1}{B} + \frac{k/B}{n} + \frac{k/B}{C}\frac{n/B}{C}\left(\frac{2(B-1)}{CB} + \frac{k/B}{n}\right). \tag{7.3}$$

As we will see later, this case is applicable to the count phase in the final pass of MPFlashsort, where all the keys in a distribution sorting sub-problem fit in cache.

7.1.1 Dissimilar key and count array element sizes

The above analysis assumes that count array elements and keys are of the same size, but it can be easily modified to handle the situation where the sizes differ. As for the approximate analysis of the permute phase, we let B be the number of keys in a cache block and αB be the number of count array elements in a cache block. We first express T, R, I and I' in terms of B and αB and we then give the probability of a cache miss for the above four cases.

Since T is expressed solely in terms of the number of keys in a cache block, it remains $T = 1/(2B)$.

We can now fit a count array of size αCB in cache, so we get:

$$R = \begin{cases} 0 & \text{if } k \leq \alpha CB \\ \frac{1}{2}(1 - \frac{C}{k/(\alpha B)}) & \text{if } k > \alpha CB. \end{cases}$$

I is the expected number of cache misses per memory access caused by the traversal periodically capturing each block in the cache and is given by:

$$I = \frac{k/(\alpha B)}{2CB}\left[1-\left(1-\frac{1}{k/(\alpha B)}\right)^{CB-(B-1)}\right]$$

when $k \leq \alpha CB$, otherwise it is approximately:

$$I \approx \frac{\alpha C}{2k}\left[1-\left(1-\frac{1}{C}\right)^{CB-(B-1)}\right].$$

$I^{'}$ is the expected number of cache misses per memory access caused by random count array accesses to a block that is mapped to the same one as being traversed and is given by:

$$I' = \begin{cases} \frac{B-1}{CB} & \text{if } k \leq \alpha CB \\ \frac{B-1}{2CB}\left(1+\frac{C}{k/\alpha B}\right) & \text{if } k > \alpha CB. \end{cases}$$

Case 4: $n > CB$ **and** $k > \alpha CB$

Re-writing $2(T + R + I + I^{'}) + (2k/(\alpha B))/n$ we obtain:

$$\begin{aligned} \mu_{count} \;:=\; & \frac{1}{B}+\frac{k-\alpha CB}{k}+ \\ & \frac{C}{\alpha k}\left[1-\left(1-\frac{1}{C}\right)^{CB-B+1}\right]+\frac{(B-1)(k+\alpha CB)}{CBk}+\frac{2k/(\alpha B)}{n}. \end{aligned} \tag{7.4}$$

Case 5: $n > CB$ **and** $k \leq \alpha CB$

Re-writing $2(T + R + I + I^{'}) + (2k/(\alpha B))/n$ we obtain:

$$\mu_{count} := \frac{1}{B}+\frac{k/(\alpha B)}{CB}\left[1-\left(1-\frac{\alpha B}{k}\right)^{CB-B+1}\right]+\frac{2(B-1)}{CB}+\frac{2k/(\alpha B)}{n}. \tag{7.5}$$

Case 6: $n \leq CB$ **and** $k \leq \alpha CB$

Arguing as in Case 3, we obtain:

$$\mu_{count} := \frac{1}{B}+\frac{k/(\alpha B)}{n}+\frac{k/(\alpha B)}{C}\frac{n/B}{C}\left(\frac{2(B-1)}{CB}+\frac{k/(\alpha B)}{n}\right). \tag{7.6}$$

7.2 Flashsort

7.2.1 Flashsort1

Neubert [67] presented *Flashsort1*, an implementation of a distribution sorting algorithm which used the in-place permutation method described in Chapter 4, to sort n keys in $O(n)$ time. In Flashsort1, it is assumed that the keys are real numbers independently drawn from a uniformly-random distribution. Flashsort1 first scans the input numbers to determine the largest (max) and smallest (min) values in the input. After this, we perform one distribution pass into $k \geq 2$ classes, using the function `classify` which for a value x returns $\lfloor x - min \cdot \frac{k}{max - min} \rfloor$. Finally, the data array is sorted using insertion sort. If $k \leq n$ the total expected cost of the insertion sort is easily seen to be $O(n^2/k)$, while the rest of the algorithm evidently runs in $O(n)$ time. The algorithm uses $2k$ extra memory locations[1]. After experimentation Neubert chose $k = n/10$ to minimise extra memory while maintaining a near-minimum expected running time [67].

Neubert's experiments showed that his implementation of Flashsort1 was twice as fast as Quicksort when sorting about 10,000 keys. In the RAM model, the lower asymptotic growth rate of Flashsort1 and the fact that Flashsort1 outperforms Quicksort for $n = 10,000$ would indicate that Flashsort1 would continue to outperform Quicksort for larger values of n. Unfortunately, this is not the case. We translated Neubert's FORTRAN code for Flashsort1 into C++ and performed extensive experiments, see Table 7.1, which clearly indicated that although Flashsort1 (Flash1) was significantly faster than Quicksort (MTQuick) for n in the range 2^{12} to 2^{17}, Quicksort caught up with and surpassed Flashsort1 at 2^{20} keys. We performed experiments up to $n = 2^{26}$ and we found that the ratio of the running times of Flashsort1 to Quicksort continued to grow with n.

Using Eq. 6.7 in Chapter 6 we see that the cache misses just for the permute phase of the algorithm are very high. For example, if we sort on a cache with parameters $B = 16$ and $C = 8192$, as would be the case our Sun UltraSparc-II using single-precision floating-point numbers, then at $n = 2^{18}, 2^{19}, 2^{20}$, 2^{21}, 2^{22}, 2^{23}, 2^{24}, 2^{25} and 2^{26} the number of cache misses per key are approximately:

[1]In fact, Neubert's code had a more complex procedure for cycle leader finding than that described in Chapter 4 which used only k extra memory locations. Our experiments showed that an implementation using this approach was significantly slower.

Input size	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}	2^{26}
Predicted	0.776	1.118	1.379	1.658	1.829	1.915	1.957	1.979	1.989

So, in the 2 memory accesses per key during an in-place permutation, we see that for $n \geq 2^{19}$ over half the memory accesses result in cache misses, and the situation gets worse as n increases, such that at large values of n almost all memory accesses result in cache misses.

Using Corollary 5.1 in Chapter 5, a very rough analysis shows us that for the largest of these values of n and given our values of $C = 8192$ and $B = 16$, if we use $k = C/B$ classes then in one pass of the permute phase we will have $O(1/B)$ cache misses per key and will create sub-problems each of about CB keys. If we solved sub-problems of size $n' \approx CB$ using $n'/10$ keys, then using Eq 6.7 in Chapter 6, we see that there will be $O(1/B)$ cache misses for the permute phase in the second pass.

This clearly indicates that a multi-pass algorithm which used fewer classes could greatly reduce the number of cache misses and probably give a faster implementation.

In order to derive a multi-pass algorithm we need to have an analysis of all the misses in Flashsort1 and in its multi-pass variant; so far we just have analyses for the count and permute phases. Before the count phase Flashsort1 sequentially accesses n elements of the data array to determine the minimum and maximum valued keys. Since n/B data blocks must be loaded into cache, this step requires $1/B$ cache misses per key. Hence:

$$\mu_{range} = 1/B. \tag{7.7}$$

These misses would also be incurred by the multi-pass variant.

After the distribution sort pass, the algorithm sorts the keys within each class using insertion sort. Since $k = n/10$, the problems that need to be solved are of expected size ≤ 10, so in Flashsort1 it is extremely unlikely in practice that any problem will exceed the size of the cache. As we will see, the same holds for the multi-pass variant. Hence we allocate $1/B$ misses per key for this phase:

$$\mu_{ins\_sort} := 1/B. \tag{7.8}$$

It should be noted that the multi-pass variant may incur no additional misses for the insertion sort phase for some values of n.

7.2.2 Multi-pass algorithm

We have seen that the cache misses during both the count and permute phase are greatly influenced by the number of classes. As outlined above, an algorithm which, when appropriate, uses fewer classes in a pass and applies distribution sorting recursively to keys within each class would outperform Flashsort1. The algorithm, called *MPFlashsort* should switch to a higher number of passes only when the reduction in cache misses can compensate for the increase in the number of operations. We will now perform this calculation for the Sun UltraSparc-II (2×300) MHz machine, with a 512KB L2 cache with a 64 byte blocksize. Assuming that we are sorting single-precision floating-point numbers, $B = 16$ and $C = 8192$.

Firstly, we need to estimate the relative cost δ of the "pure computation" cost of one distribution pass versus a cache miss. Inspecting the optimised assembly code produced by the compiler, `gcc 2.8.1` using optimisation level 6, there are approximately 30 operations per key in one distribution pass, and the cost of a L2 cache miss on this machine is approximately 30 cycles, so $\delta \approx 1$. (It should be noted that this calculation can only yield a rough guide, since the underlying model is very crude.)

MPFlashsort uses one pass until the number of keys is larger than a threshold value T_1. The value of T_1 is selected to be the minimum value of n such that:

$$M_1(z) > M_1(y) + M_2(x) + \delta. \tag{7.9}$$

Here $M_1(k)$ is the number of misses per item for sorting $10k$ keys in one pass by classifying into k classes, as described in cases 1 and 2 of the count and permute phase analysis. Similarly, $M_2(k)$ is the number of cache misses incurred in a distribution sorting pass using k classes as described in case 3 in the count and permute phase analysis. Recall that the permute analysis of case 3 is applicable only when $n/k \gg B$ and the value of k is not "bad". Hence we fix $x = n/(\theta CB)$ where $\theta = \frac{1}{1+1/\phi}$; since we want $xy = z = n/10$, this fixes $y = (\theta CB)/10$. Performing the calculation gives $z = 62153$, giving $n = 621530$, which is the value coded into MPFlashsort.

We now consider how to generalise this to multiple passes. We first set the expected problem size in the final distribution pass to θCB. Again letting $x = n/(\theta CB)$, we have to perform distribution passes until the expected class size is reduced by a factor of x. In the l-pass algorithm for $l \geq 2$, this is done in $l - 1$ distribution passes with

$x^{1/(l-1)}$ classes in each pass. The algorithm switches to $l+1$ passes when it is more efficient to reduce the expected problem size by a factor of x in l passes with $x^{1/l}$ classes in each pass. Thus, we switch from $l \geq 2$ to $l+1$ passes when:

$$(l-1)M_2(x^{1/(l-1)}) > l \cdot M_2(x^{1/l}) + \delta. \tag{7.10}$$

Doing the calculations for $l = 2$, we get that we should switch from two to three passes when $x > 65587$, corresponding to $n > x \cdot \theta CB \approx 5,313,000,000$.

It should be borne in mind that $M_2(k)$ does not predict cache misses well if k is 'bad', and selecting k according to Heuristic 2 is our only suggestion for avoiding 'bad' k. Hence, we modify the operation of the l-pass algorithm as follows. Instead of performing l passes with $x^{1/(l-1)}$ classes in each pass, the number of classes used in the passes are $k_1, k_2, \ldots, k_{l-1}$, which satisfy: (i) the expected problem size for the final pass is θCB as required; (ii) k_i is of the required form for all i, and (iii) k_i varies by no more than about 10% from $x^{1/(l-1)}$ for all i. With such a small variation in the class sizes, the calculations will continue to be broadly correct.

We now consider how this may be done for $l = 3$. First, we should choose k_1 classes for the first pass, such that $0.9x^{1/2} \leq k_1 \leq 1.1x^{1/2}$, and k_1 is of the appropriate form. As noted in the text after Equation 6.23, this can always be done. The expected class sizes after the first pass are $n' = n/k_1$. In the second pass, we simply choose $k_2 = n'/(\theta CB)$ and note that $0.9k_2 \leq x^{1/2} \leq 1.1k_2$. The following element of approximation should be noted: the expected class sizes after the first pass are $n' = n/k_1$ for all classes but one (as k_1 is not an integer). The misses incurred by the 'small' class are ignored in the calculation.

Also, these threshold values of 62153 and 65587, which are approximately $8C$, should be contrasted with the fact that in the EMM, the number of classes in a pass should be no more than C [8], we have also discussed this in Chapter 4. Furthermore, as shown in Corollary 5.1, in Chapter 5, in order to get the asymptotically optimal $\Theta\left(\frac{N}{B}\frac{\log(N/B)}{\log C}\right)$ number of cache misses in the presence of conflicts using a direct-mapped cache, we should have no more than $O(C/B)$ classes in all passes but the last one. However, the above accounting shows that these choices would result in excessive operations, which would not be paid for by the reduced number of cache misses.

In the next section we compare the performance of the multi-pass algorithm to Flashsort1 and cache-tuned Quicksort.

7.2.3 Experimental results

We have implemented Flashsort1, and MPFlashsort, and have used them to sort n uniform random floating-point numbers, for $n = 2^i$, $i = 10, 11, \ldots, 26$. Memory-tuned Quicksort (MTQuicksort) was found to be the fastest algorithm from [59] on our machine and we also tested this. We have also included some running times from Sanders' Heapsort (an out-of-place algorithm), but as noted below, it is not easy to compare the other algorithms with Sanders' Heapsort.

Most of our experiments were on a Sun UltraSparc-II with 2×300 Mhz processors and 1GB main memory, a 16KB L1 data cache and a 512KB L2 cache, both of which are direct-mapped. On this platform the Flashsort algorithms and Sanders' Heapsort were coded in C++ and MTQuicksort was coded in C and the algorithms were compiled using `gcc 2.8.1` with optimisation level 6. For each algorithm, we have measured actual running times as well as simulated numbers of cache misses. The simulator simulates only the L2 cache on this machine, and only reports cache hit/miss statistics. Each running time and simulation value reported in this section is the average of 50 runs.

Table 7.1 summarises the running times for Flashsort1, MPFlashsort and MTQuicksort on single-precision keys. We observe that:

- For small values of n Flashsort1 gets steadily faster than MTQuicksort until it uses about 75% of the time of MTQuicksort for $n = 2^{16}$. After that the performance advantage narrows until at $n = 2^{19}$ MTQuicksort overtakes Flashsort1. From 2^{20} onwards the relative gap between MTQuicksort and Flashsort1 (the ratio of the running times) generally increases steadily until the largest input value considered. This is interesting given that in the RAM model MTQuicksort has a higher asymptotic running time than Flashsort1.

- The performance of MPFlashsort matches that of Flashsort1 up to $n = 2^{19}$, after which point the algorithm performs two passes on the data and outperforms both Flashsort1 and MTQuicksort. At the largest value of n considered MPFlashsort is over twice as fast as Flashsort1. Note that the running time of MPFlashsort at $n{=}2^{20}$ is almost exactly twice the running time of MPFlashsort at $n{=}2^{19}$, even though the algorithm switches from one to two passes. This suggests that

Timings (s) on Sun UltraSparc-II single-precision keys			
n	Flash1	MPFlash	MTQuick
2^{10}	0.0004	0.0004	0.0004
2^{11}	0.0007	0.0007	0.0008
2^{12}	0.0015	0.0015	0.0017
2^{13}	0.0032	0.0031	0.0037
2^{14}	0.0065	0.0064	0.0081
2^{15}	0.0134	0.0132	0.0175
2^{16}	0.0282	0.0280	0.0376
2^{17}	0.0634	0.0630	0.0800
2^{18}	0.1555	0.1547	0.1700
2^{19}	0.3623	0.3645	0.3591
2^{20}	0.9420	0.7102	0.7760
2^{21}	2.3250	1.4302	1.6630
2^{22}	5.2690	2.8622	3.5260
2^{23}	12.047	6.4995	7.4709
2^{24}	26.088	14.045	15.863
2^{25}	54.613	29.276	33.372
2^{26}	124.80	59.852	69.969

Table 7.1: Running times of Flashsort1, MPFlashsort and MTQuicksort on a Sun UltraSparc-II using single-precision floating-point keys.

the threshold value for switching from one to two passes has been calculated accurately.

Table 7.2 summarises the running times for Flashsort1, MPFlashsort and MTQuicksort on 8-byte data, using double-precision keys. Here we use the analysis for dissimilar key and count array element sizes to calculate the threshold values. As we are dealing with 8-byte data and 4-byte count elements, we have that $B = 8$ and $\alpha = 2$. We calculate that at $n \geq 706270$ we should switch from 1 to two passes. Note that:

- MTQuicksort starts to outperform Flashsort1 later than for single-precision data, namely only for $n \geq 2^{21}$.
- MPFlashsort is 35% faster than MTQuicksort for the largest values of n, a bigger

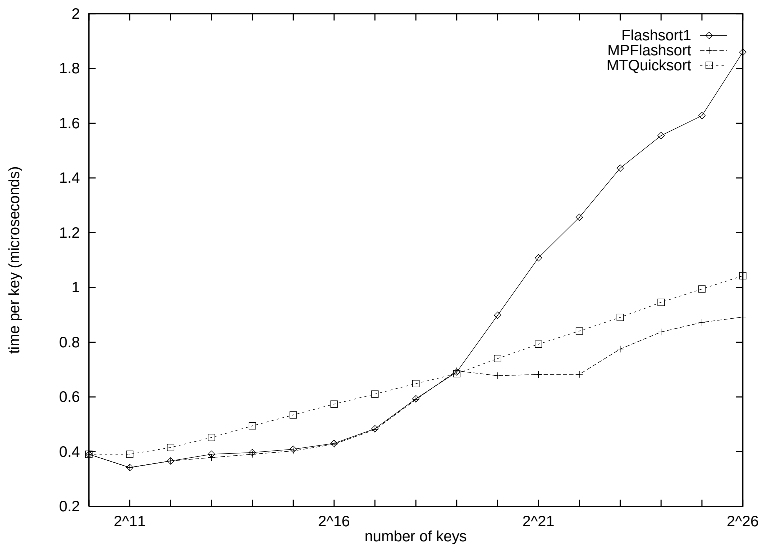

Figure 7.1: Time per key to sort single-precision floating-point keys using Flashsort1, MPFlashsort and MTQuicksort on a Sun UltraSparc-II. For $n = 2^i, i = 10, \ldots, 26$. Note that the x-axis is on a log scale.

performance gap than for the single-precision case.

- MPFlashsort and Flashsort1 are only slightly slowed down relative to the single-precision case, but MTQuicksort is significantly slower. Although the distribution sorting algorithms are slowed down about 10-15% relative to the single-precision case, with no clear trend, MTQuicksort for doubles is about 35% slower than for floats at 2^{16}, with the slowdown gradually increasing to about 50% at 2^{25}. Possibly this is due to the fact that Quicksort makes $\Theta(n \log n)$ data moves but the distribution sorts make only $O(n)$ data moves.

Finally, we attempted comparisons with Sanders' Heapsort, but it should be noted that Sanders' algorithm sorts 8-byte records with a key and a pointer, whereas our implementations are optimised for simply sorting keys. Indeed, the only commonality with Sanders' code is that 8-byte data are being sorted. Nevertheless, we have included these running times as a very rough guide.

To verify that our predicted performance gains can be attained on other platforms, we tested Flashsort1, MPFlashsort and MTQuicksort on a system with an Intel Mobile Pentium III processor. This system has a 16KB 4-way set-associative L1 cache with 32 byte blocks, a 256KB 8-way set-associative L2 cache with 32 byte blocks and 256MB of

Timings (s) on Sun UltraSparc-II				
8-byte keys				8-byte rec's
n	Flash1	MPFlash	MTQuick	Sanders
2^{10}	0.0004	0.0004	0.0005	0.0007
2^{11}	0.0008	0.0009	0.0010	0.0016
2^{12}	0.0017	0.0017	0.0023	0.0036
2^{13}	0.0034	0.0035	0.0051	0.0079
2^{14}	0.0069	0.0070	0.0109	0.0164
2^{15}	0.0144	0.0145	0.0238	0.0341
2^{16}	0.0317	0.0320	0.0510	0.0693
2^{17}	0.0765	0.0774	0.1093	0.1413
2^{18}	0.1723	0.1733	0.2330	0.3016
2^{19}	0.4463	0.4502	0.5099	0.6321
2^{20}	1.0362	0.7922	1.1082	1.3088
2^{21}	2.4700	1.5906	2.4014	2.7248
2^{22}	5.5752	3.5402	5.1698	5.6428
2^{23}	12.706	7.6008	11.050	11.704
2^{24}	27.487	15.722	23.533	24.697
2^{25}	60.917	32.227	50.111	50.472

Table 7.2: Running times of Flashsort1, MPFlashsort, MTQuicksort using 8-byte (double-precision floating-point) keys and Sanders' Heapsort using 8-byte records on a Sun UltraSparc-II.

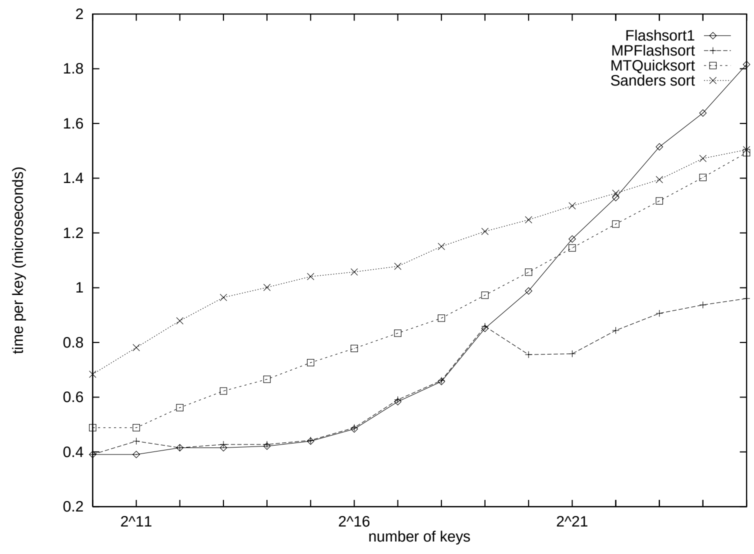

Figure 7.2: Time per key to sort 8-byte (double-precision floating-point) keys using Flashsort1, MPFlashsort, MTQuicksort and 8-byte records using Sanders' Heapsort on a Sun UltraSparc-II. For $n = 2^i, i = 10, \ldots, 25$. Note that the x-axis is on a log scale.

main memory. The L1 hit time is 3 clock cycles, we assume the L2 hit time is 4 cycles and the L2 miss penalty is 52 clock cycles. Our analysis is not for set-associative caches, nevertheless we use it to calculate a threshold value for MPFlashsort when optimising for the L2 cache on this machine. Our experiments were performed on uniformly-random single-precision floating-point keys. The programs were complied using the `Microsoft Visual C++ 6.0` compiler and inspecting the assembly code tells us that $\delta \approx 1.7$ and that the algorithm should switch from one to two passes at $n = 899230$. Table 7.3 summarises the running times for Flashsort1, MPFlashsort and MTQuicksort using single-precision floating-point keys. We note that:

- As predicted, MPFlashsort starts to outperform Flashort1 at $n = 2^{20}$.
- MTQuicksort is the fastest of the three algorithms for $n \leq 2^{23}$. Only at $n = 2^{24}$ does MPFlashsort start to outperform it. We attribute this to the high instruction count on this platform for each pass of distribution sorting.

7.2.4 Cache simulations

We also ran these algorithms on an L2 cache simulator for the Sun UltraSparc-II. Figure 7.3 compares MTQuicksort and Flashsort1, while Figure 7.4 compares the three

Timings (s) on an Intel Mobile Pentium III single-precision keys			
n	Flash1	MPFlash	MTQuick
2^{18}	0.1417	0.1487	0.1272
2^{19}	0.3305	0.3445	0.2760
2^{20}	0.7731	0.7366	0.5955
2^{21}	1.6764	1.4837	1.2924
2^{22}	3.4730	2.9967	2.7425
2^{23}	7.2390	6.0435	5.9045
2^{24}	14.992	12.167	12.530
2^{25}	32.566	24.454	26.337

Table 7.3: Running times of Flashsort1, MPFlashsort and MTQuicksort on a machine with an Intel Mobile Pentium III processor, using single-precision floating-point keys.

faster algorithms. These figures show the number of L2 cache misses per key for the three algorithms on single-precision floating-point keys. We observe that:

- When the problem is small and fits in L2 cache, for $n \leq 2^{16}$, the number of misses per key are almost constant for each algorithm, these are the compulsory misses.
- In Flashsort1 for $n \geq 2^{17}$ we see a rapid increase in the number of misses per key as n grows, appearing to level off at over 3 misses per key (virtually every access that could be a miss is a miss).
- In MTQuicksort the number of cache misses per key increases linearly with $\log n$ for $n \geq 2^{17}$, reaching about 0.82 misses per key at 2^{26}. Note that MTQuicksort makes $O((\log n)/B)$ misses per key on average, so this is not unexpected.
- In MPFlashsort we first see a rapid increase in misses per key going from 0.331 at $n = 2^{17}$ to 1.361 at $n = 2^{19}$. At $n = 2^{20}$ the misses per key drops to 0.446, as the algorithm switches to two passes, and we see a very gradual increase reaching 0.504 at $n = 2^{26}$.

Figure 7.5 shows for the first permutation phase of MPFlashsort the cache misses per key predicted using Equation 6.16 for values of $n = 2^{20}, \ldots, 2^{26}$ in increments of

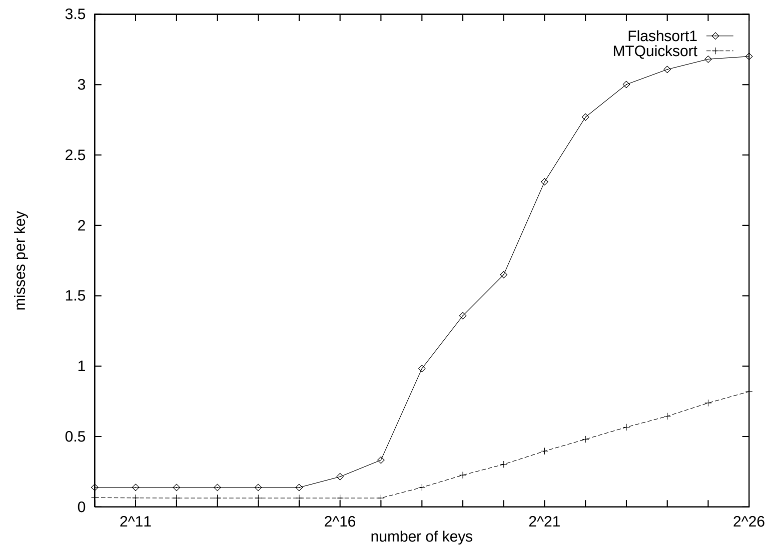

Figure 7.3: L2 cache misses per key on single-precision floating-point keys using Flashsort1 and MTQuicksort. Note that the x-axis is on a log scale.

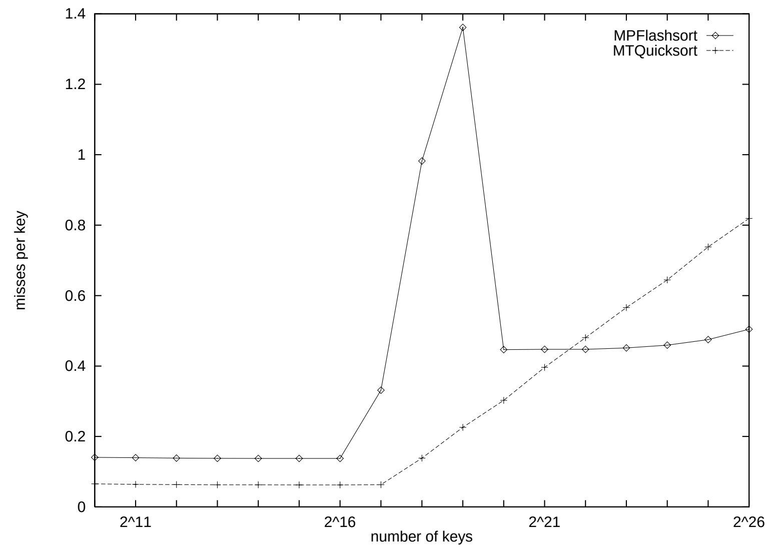

Figure 7.4: L2 cache misses per key on single-precision floating-point keys: MPFlashsort vs MTQuicksort. Note that the x-axis is on a log scale.

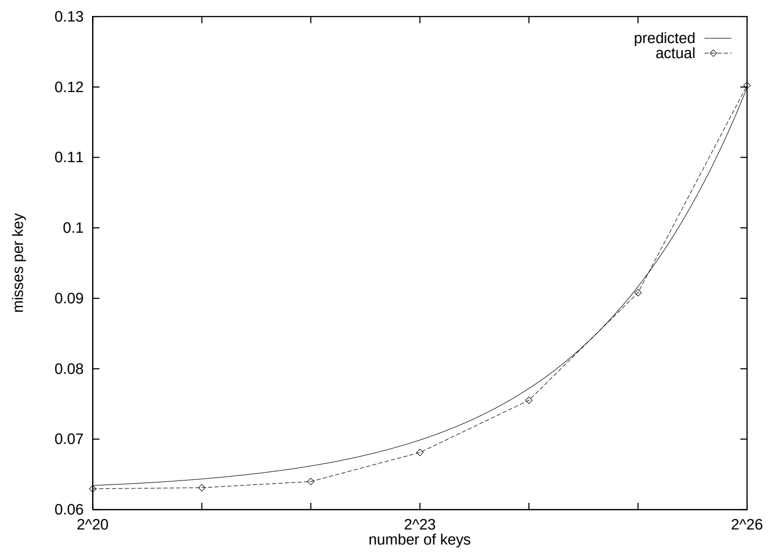

Figure 7.5: L2 cache misses per key during the first 'in-place' permutation of MPFlashsort. Predicted for values of $n = 2^{20}, \ldots, 2^{26}$ in increments of 2^{17} and simulated at $n = 2^i, i = 10, \ldots, 26$. Note that the x-axis is on a log scale.

2^{17} and the simulated cache misses per key for $n = 2^i, i = 10, \ldots, 26$. We see that the two lines follow each other very closely.

7.3 MSB radix sort

In practice there are often cases when keys are not uniform (e.g., they may be normally distributed); the analysis in Chapter 5 can be used to tune distribution sort in these cases. We consider a different application here: sorting independently and uniformly distributed floating-point numbers in the range $[0, 1)$ using the *integer* sorting algorithm MSB radix sort. It is well-known that one can sort floats by sorting the bit-strings representing the floats, interpreting them as integers [45]. Since (simple) operations on integers are faster than operations on floats, this can improve performance.

One pass of MSB radix sort using radix size r groups the keys according to their most significant r bits in $O(2^r + n)$ time. For random integers, a reasonable choice for minimising instruction counts is $r = \lceil \log n - 3 \rceil$ bits, or classifying into about $n/8$ classes. Since each class has about 8 keys on average, they can be sorted using insertion sort. Using this approach for this problem gives terrible performance even at small values of n (see Table 7.4). As we now show, the problem lies with the distribution of the integers on which MSB radix sort is applied.

7.3.1 Radix sorting floating-point numbers

A floating-point number is represented as a triple of non-negative integers $\langle i, j, l\rangle$. Here i is called the *sign bit* and is a 0-1 value (0 indicating non-negative numbers, 1 indicating negative numbers), j is called the *exponent* and is represented using e bits and l is called the *mantissa* and represented using m bits. Let $j^* = j - 2^{e-1} + 1$ denote the *unbiased* exponent of $\langle i, j, l\rangle$. Roughly following the IEEE 754 standard, let the triple $\langle 0, 0, 0\rangle$ represent the number 0, and let $\langle i, j, l\rangle$, where $j > 0$, represent the number $\pm 2^{j^*}(1 + l2^{-m})$, depending on whether $x = 0$ or 1; no other triple is a floating-point number. Internally each member of the triple is stored in consecutive fields of a word. The IEEE 754 standard specifies $e = 8$ and $m = 23$ for 32-bit floats and $e = 11$ and $m = 52$ for 64-bit floats [45].

We model the generation of a random float in the range $[0, 1)$ as follows: generate an (infinite-precision) random real number, and round it down to the next smaller float. On average, half the numbers generated will lie in the range $[0.5, 1)$ and will have an unbiased exponent of -1. In general, for all non-zero numbers, the unbiased exponent has value i with probability 2^i, for $i = -1, -2, \ldots, -2^{e-1} + 2$, whereas the mantissa is a random m-bit integer. The value 0 has probability $2^{-2^{e-1}+2}$. Clearly, the distribution is not uniform, and it is easy to see that the average size of the largest class after the first pass of MSB radix sort with radix r is $n\left(1 - \frac{1}{2^{2^{e-r+1}}}\right)$ if $r < e + 1$, and $n/(2^{r-e})$ if $r \geq e + 1$.

This shows, e.g., that the largest sub-problems in the examples of Table 7.4 would be of size $n/2^{\lceil \log n - 3\rceil - 8} \approx 2^{14}$, so using insertion sort after one pass is inefficient in this case[2]. To get down to problems of size 8 in one pass requires a radix of about $\log n + 8$, which is impractical. Also, MSB radix sort applied to random integers has $O(n)$ expected running time independently of the word size, but this is not true for floats. A first pass with $r \ll e$ barely reduces the largest problem size, and the same holds for subsequent passes until bits from the mantissa are reached. As the radix in any pass is limited to $\log n + O(1)$ bits, we may need $\Omega(e/\log n)$ passes, introducing a dependence on the word size.

[2]In fact, the total number of keys in all sub-problems of this size would be $n/2$ on average.

7.3.2 Using Quicksort

To get around the problem of having several passes before we reduce the largest class, we partition the input keys around a value $1/n \leq \theta \leq 1/(\log n)$, and sort the keys smaller than θ in $O(n)$ expected time using Quicksort. We then apply MSB radix sort to the remaining keys. Let $e' = \min\{\lceil \log\log(1/\theta) \rceil, e\}$ denote the *effective exponent*, since the remaining keys have exponents which vary only in the lower order e' bits. This means that keys can be grouped according to a radix $r = e + 1 + m'$ with $m' \geq 0$ in $O(n + 2^{e'+m'})$ time and $O(2^{e'+m'})$ space. Since $e' = O(\log\log n)$, we can take up to $\log n - O(\log\log n)$ bits from the mantissa as part of the first radix; as all sub-problems now only deal with bits from the mantissa they can be solved in linear expected time, giving a linear running time overall.

7.3.3 Cache analysis

We now use the analysis in Chapter 5 to calculate an upper bound for the cache misses in the permute phase of the first pass of MSB radix sort using a radix $r = e + 1 + m'$, for some $m' \geq 0$, assuming also that all keys are in the range $[\theta, 1)$, for some $\theta \geq 1/n$. There are $2^{e'+m'}$ pointers in all, which can be divided into $g = 2^{e'}$ *groups* of $K = 2^{m'}$ pointers each. Group i corresponds to keys with unbiased exponent $-i$, for $i = 1, \ldots, g$. All pointers in group i have an access probability of $1/(K2^i)$. Using Theorem 5.1 and a slight extension of the methods of Theorem 5.2 we are able to prove Theorem 7.1 below, which states that the number of misses is essentially independent of g:

Theorem 7.1 *Provided $gK \leq CB$ and $K \leq C$ the number of misses in the first pass of the permute phase of MSB radix sort is at most:*

$$n\left(\frac{1}{B} + \frac{2K}{BC}(2.3B + 2\log B + \log C - \log K + 0.7)\right) + gK(1 + 1/B).$$

PROOF. Using Eq. 5.15 we can calculate a lower bound on the probability of event $X_{(i-1)K+1}$ as:

$$\begin{aligned} &\Pr[X_{(i-1)K+1}] \\ &\geq \; 1 - \frac{K2^i}{BC} - \frac{1}{C}\sum_{j=1}^{g}\sum_{l=1}^{K/B}\frac{B2^{-j}}{B2^{-j}+2^{-i}} - \frac{B-1}{BC}\sum_{j=1}^{g}\sum_{l=1}^{K}\frac{2^{-j}}{2^{-j}+2^{-i}} \end{aligned}$$

$$
\begin{aligned}
&= 1-\frac{K}{BC}\left(\sum_{j=1}^{g}\frac{B2^{-j}}{B2^{-j}+2^{-i}}+2^i+(B-1)\sum_{j=1}^{g}\frac{2^{-j}}{2^{-j}+2^{-i}}\right)\\
&\geq 1-\frac{K}{BC}\left(\log B+i+2^i+(B-1)\sum_{j=1}^{g}\frac{2^{-j}}{2^{-j}+2^{-i}}\right) \qquad (7.11)
\end{aligned}
$$

If $K2^i/(BC) \geq 1$ then $\Pr[X_{(i-1)K+1}]$ would be negative, so we place a bound on this term such that $K2^i < BC$. The maximum value of i such that $K2^i/(BC) < 1$ is $\log BC - \log K - 1$.

Since the probabilities of access to pointer $D_{(i-1)K+1}, \ldots, D_{iK}$ are all $1/(K2^i)$ we can calculate an upper bound on p_d using Eq. 5.6 and 7.11 as:

$$
\begin{aligned}
p_d &\leq \frac{K^2}{BC}\left(\sum_{i=1}^{g}p_i\left(\log B+i+(B-1)\sum_{j=1}^{g}\frac{2^{-j}}{2^{-j}+2^{-i}}\right)+\sum_{i=1}^{\log BC-\log K-1}p_i2^i\right)\\
&\quad+\sum_{i=\log BC-\log K}^{g}Kp_i\\
&= \sum_{i=1}^{g}\frac{1}{2^i}\frac{K}{BC}\left(\log B+i+(B-1)\sum_{j=1}^{g}\frac{2^{-j}}{2^{-j}+2^{-i}}\right)\\
&\quad+\sum_{i=1}^{\log BC-\log K-1}\frac{1}{2^i}\frac{K}{BC}2^i+\sum_{i=\log BC-\log K}^{g}\frac{1}{2^i}\\
&\leq \frac{K}{BC}\left(\sum_{i=1}^{g}\frac{\log B}{2^i}+\sum_{i=1}^{g}\frac{i}{2^i}+(B-1)\sum_{i=1}^{g}\frac{1}{2^i}\sum_{j=1}^{g}\frac{2^{-j}}{2^{-j}+2^{-i}}\right)\\
&\quad+\frac{K}{BC}(\log BC-\log K-1)+\frac{2K}{CB}\\
&\leq \frac{K}{BC}\left(2\log B+3+\log C-\log K+2.3(B-1)\right).
\end{aligned}
$$

□

7.3.4 Tuning MSB radix sort

We now optimise parameter choices in our algorithms. The smaller the value of θ, the fewer keys are sorted by Quicksort, but reducing θ may increase e'. A larger value of e' does not mean more misses, by Theorem 7.1, but it does mean a larger count array. We choose $\theta = 1/(\log n)^2$ as a compromise, ensuring that Quicksort uses $o(n)$ time. Using the above analysis we are also able to determine an optimal number of classes to use in each sorting sub-problem. We use two criteria of optimality. In the first, we require that each pass incur no more than $(2+\varepsilon)n/B$ misses for some constant $\varepsilon > 0$,

$n =$	1×10^6	2×10^6	4×10^6	8×10^6	16×10^6	32×10^6
MTQuick	0.7400	1.5890	3.3690	7.2430	15.298	32.092
Naive1	7.0620	14.192	28.436	57.082	115.16	233.16

Table 7.4: Running times in seconds of MTQuicksort and Naive1 MSBRadix sort (single pass MSBRadix sort without partitioning, $r = \lceil \log n - 3 \rceil$)floating-point keys.

$n =$	1×10^6	2×10^6	4×10^6	8×10^6	16×10^6	32×10^6	64×10^6
MPFlash	0.6780	1.3780	2.2756	6.1700	13.308	27.738	56.796
MTQuick	0.7400	1.5890	3.3690	7.2430	15.298	32.092	67.861
MSBRadix	0.3865	0.8470	1.9820	5.0300	9.4800	19.436	40.663

Table 7.5: Running times in seconds of MPFlashsort, MTQuicksort and MSBRadix sort on a Sun UltraSparc-II using single-precision floating-point keys.

thus seeking essentially to minimise cache misses ($2n/B$ misses is the bare minimum for the count and permute phases). In the second, we tradeoff reductions in cache misses against extra computation. The latter yields better practical results, and results shown below are for this approach.

7.3.5 Experimental results

In Table 7.5 we compare tuned MSB radix sort with a memory-tuned Quicksort algorithm (MTQuicksort)[59] and with MPFlashsort. The algorithms were coded in C and compiled using `gcc 2.8.1`. The experiments were on our Sun UltraSparc-II with 2×300 MHz processors and 1GB main memory, and a 16KB L1 data cache, 512KB L2 direct-mapped cache. Observe that MSB radix sort easily outperforms the other algorithms for the range of values considered.

7.4 Summary

In this chapter we have shown that a recently published distribution sorting algorithm, Flashsort1, which sorts uniformly and randomly distributed data in $O(n)$ time is fast when sorting small problems, but due to poor cache utilisation it starts to perform poorly when data is larger than the cache size. Using the analysis in this and the previous chapter we have obtained a variant of Flashsort1, MPFlashsort, which switches to multiple passes when appropriate and matches or outperforms Flashsort1 and memory-tuned Quicksort at all values of n. The performance of MPFlashort is attributable to good cache utilisation.

We have shown that if the integer sorting algorithm MSB radix sort is used to sort uniformly and randomly distributed floating-point numbers then a non-uniform distribution of keys to classes is induced. We have shown that a naive implementation of this algorithm would have very poor performance due to this non-uniform distribution. We have shown that by partitioning the keys, to remove keys which are expected to go into small classes, and by using the analysis of Chapter 5, the algorithm can be tuned for good cache performance. Due to fast integer operations and good cache utilisation the tuned algorithm outperforms MPFlashsort and memory-tuned Quicksort.

Chapter 8

Sorting integers in the IMM

Radix sorting, applied to integers, consists in viewing w-bit integer keys as $\lceil w/r \rceil$ consecutive r-bit *digits*. The records are sorted in $\lceil w/r \rceil$ passes: in the i-th pass, for $i = 1, \ldots, \lceil w/r \rceil$ we sort the records according to the i-th least significant digit, more precisely this is referred to as *LSB radix sorting*. There are a number of ways of implementing a single pass in $O(n + 2^r)$ time, the best one in practice being *counting sort* [32], which is equivalent to a generic distribution sorting pass using out-of-place permutation, as described in Chapter 4.

LSB radix sort has an overall running time of $O(\lceil w/r \rceil (n + 2^r))$ for w-bit keys. For medium-to-large input sizes, LSB radix sort may be considered to be a linear-time algorithm for practical purposes. However, experimental work has shown that LSB radix sort, even when tuned for cache performance, fails to outperform good implementations of $O(n \log n)$ comparison-based algorithms on modern architectures [59]. We show that cache-tuned LSB radix sort still has poor TLB performance. On the other hand, many comparison-based algorithms intrinsically have good TLB performance. Hence it is necessary to use models which incorporate the TLB if one is to get good implementations of LSB radix sort. So we use the IMM, introduced in Chapter 3, for analysing TLB performance. We introduce three techniques for reducing TLB misses, using which we obtain three new sorting algorithms which have very good performance.

8.1 Techniques to reduce TLB misses

The following three techniques are used to reduce TLB misses in radix sort:

- *reducing working set size*,
- *explicit block transfer* and
- *pre-sorting*.

We now outline these techniques.

The *working set* of a program is the set of pages it accesses at a particular time. If the program makes random accesses to these pages, and the working set size is much larger than the size of the TLB, then the number of TLB misses will be large (in this chapter we will use the term *random access* to denote accesses which do not exhibit locality). Hence, reducing the working set size to about the capacity of the TLB can greatly reduce the number of TLB misses. In the case of LSB radix sort, this is accomplished by reducing the radix far below the values suggested in textbooks [32, p. 179] or those obtained from cache analyses [59].

The second technique, explicit block transfer, enforces locality by ensuring that all random accesses are made only for the purpose of copying blocks of memory locations. This technique uses the emulation theorem for emulating external memory algorithms on the IMM. In the external memory model, good algorithms perform as few I/O operations as possible—this translates into few cache and TLB misses for the emulation. The new algorithm *explicit block transfer radix sort (EBT radix sort)* essentially emulates a natural external memory analogue of counting sort in each pass.

In the final technique, pre-sorting, prior to each pass, the array containing the keys is 'conditioned' by sorting the records within relatively small segments. The pre-sorting is then used in two different ways, giving two algorithms. In the first algorithm, *pre-sorting LSB radix sort (PLSB radix sort)*, the pre-sorting brings together keys with equal values which can then be moved in a group, imparting locality to the 'global' sort. In the second, *extended-radix PLSB radix sort (EPLSB radix sort)*, we use the increased locality afforded by pre-sorting to increase the radix size—hence reducing the number of passes—whilst continuing to incur an acceptable number of cache and TLB misses.

From the theoretical viewpoint, these approaches have different characteristics. As noted here, LSB radix sort has very poor cache performance on worst-case problem instances, and reducing the radix size does not alleviate this problem. However, LSB radix sort can make few TLB misses in the worst case. These characteristics are shared by EPLSB radix sort. On the other hand, both EBT radix sort and PLSB radix sort make an asymptotically optimal number of cache misses, and few TLB misses, in the worst case. None of the algorithms simultaneously achieves asymptotic optimality for cache and TLB misses in general.

We have tested implementations of these four approaches on the Sun UltraSparc-II architecture. These implementations generally vary slightly from the algorithm descriptions which means they may sacrifice worst-case performance (excepting PLSB radix sort). All implementations show good speedups over highly optimised implementations of cache-tuned LSB radix sort and cache-tuned comparison-based algorithms. In our experiments EPLSB radix sort performed the best, running over twice as fast as cache-tuned LSB radix sort or cache-tuned Quicksort.

8.2 LSB radix sort

In this and the following sections, we use the following default terminology. The term *key* will refer to a single digit, and the term *record* will denote the integer which was input to the sorting algorithm plus any associated information.

Recall from Chapter 4 that in the i-th pass of LSB radix sort with radix size i, we stably sort the records according to the i-th least significant digit.

The algorithm can be implemented using the generic distribution sorting algorithm, with out-of-place permutations, described in Chapter 4. By letting the `classify` function mask off the appropriate digit in the key. Without loss of generality, we focus on one pass of radix sorting. We now give a worst-case cache and TLB miss analysis for the algorithm.

8.2.1 Worst-case cache miss analysis

Proposition 8.1 *For any fixed cache associativity $a \geq 1$, one pass of LSB radix sort with radix r makes $\Omega(n)$ cache misses in the worst case whenever $2^r > a$.*

PROOF. For convenience let n be a power of two. Consider just the random write accesses in the permute phase of LSB radix sort and let the input consist of the sequence $0, 1, \ldots, 2^r - 1$ repeated $n/2^r$ times. With this input, the 2^r active blocks will be mapped into a total of $\max\{1, BS/(n/2^r)\}$ sets, giving $\min\{2^r, n/(BS)\}$ active locations mapped to each set. Provided that $n > aBS = BC$, the number of active blocks mapped to each set will exceed a. Owing to the round-robin nature of accesses to the active data blocks mapped to a set, all accesses to active locations will be misses, even ignoring other conflicts. This gives a minimum of n misses for a single permute phase in the worst case.

Note that the count phase incurs $O(n/B)$ misses in the worst case whenever $a \geq 2$ and $2^r \leq BC/2$ (we require that $2^r \leq BC/4$ if Friend's improvement [37], which coalesces two count phases into one, is used). □

The construction of Proposition 8.1 can easily be generalised to get inputs which have bad performance on several successive passes:

Proposition 8.2 *For any fixed cache associativity $a \geq 1$, in the worst case, LSB radix sort with radix r makes $\Omega(n)$ cache misses in each of $\lfloor w/r \rfloor$ passes, provided $2^r > a$, when sorting w-bit keys.*

Proposition 8.3 *For any fixed cache associativity $a \geq 1$ and n a power of 2, LSB radix sort with radix r makes $\Omega(n)$ cache misses in each of $\lfloor (\log n)/r \rfloor$ passes on the input $0, 1, \ldots, n-1$, provided $2^r > a$.*

From the asymptotic viewpoint, the results from Chapter 5 and [65] suggest that 'on average' if we have $2^r = O(C/B)$, then each pass of radix sort makes an optimal $O(n/B)$ expected misses even on a direct-mapped cache. For a cache with associativity a [65] suggests that the expected number of misses is $O(n/B)$ for $2^r = O(C/B^{1/a})$. The worst-case input increases this to $\Theta(n)$ misses per pass even when r is very small, i.e. when $2^r > a$.

This difference is evident in practice too: Table 8.1 compares the running times for one pass of LSB radix sort using a 11-bit radix when the input data is the worst-case described above and random data. It shows that the algorithm is up to 45% slower on worst-case data than on random input for radix 11.

8.2.2 Worst-case TLB miss analysis

We now calculate the number of TLB misses in the count phase and permute phase of LSB radix sort in the worst case.

Timings(sec)						
n	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}
pathological (r=11)	0.44	0.94	1.92	4.37	7.70	15.40
random (r=11)	0.31	0.65	1.33	3.01	5.47	10.89

Table 8.1: Running times for one pass of LSB radix sort using 11-bit radix for random 32-bit unsigned integers and the pathological input.

Proposition 8.4 *The number of TLB misses in the count phase of one pass of LSB radix sort with radix r when sorting n keys is:*

$$\begin{aligned} &\leq \; (\lceil n/P\rceil + W)\left(\frac{T}{T-W}\right), && \text{if } W \leq T-1, && (8.1)\\ &\geq \; \lceil n/P\rceil + n, && \text{if } W > T-1, && (8.2)\end{aligned}$$

in the worst case, where $W = \lceil 2^r/P\rceil$.

Proof. During the count phase, the algorithm accesses the following *working* set of pages: (i) one active page in the source array and (ii) $W = \lceil 2^r/P\rceil$ count array pages. Thus, if $W+1 \leq T$, an optimal algorithm with a TLB of size $W+1$ will make no more than $\lceil n/P\rceil + W$ misses (to bring in the count array pages as well as to bring in the successive active source array pages.). Applying Theorem 3.1 from Chapter 3 proves Eq. 8.1.

If $W+1 > T$ and $T = 1$ then clearly all $2n$ memory accesses cause misses whatever replacement policy is used. If $T > 1$ then the active source array page is always in the TLB, but at least one count array page is always not in the TLB, and in the worst case, the count array page which is not in the TLB will always be accessed (a round-robin access pattern suffices for this). Thus all n accesses to count array pages are misses, and we add the compulsory misses for source array pages to get Eq. 8.2. □

The following proposition is proved analogously:

Proposition 8.5 *The number of TLB misses in the permute phase of one pass of LSB radix sort with radix r when sorting n keys is:*

$$\begin{aligned} &\leq \; (\lceil n/P\rceil + W)\left(\frac{T}{T-W}\right), && \text{if } W \leq T-1, && (8.3)\\ &\geq \; \lceil n/P\rceil + n, && \text{if } W > T-1, && (8.4)\end{aligned}$$

in the worst case, where $W = \lceil 2^r/P\rceil + \min\{2^r, \lceil n/P\rceil\}$.

This shows that provided $T \ll P$ (which is usually the case) there is a sharp threshold for TLB misses, going from $O(nT/P)$ when the working set fits into the TLB to $\Omega(n)$ when it does not. Hence the radix should be chosen small enough that the former case holds.

8.3 Implementing LSB radix sort

8.3.1 Reducing working set size

From the discussion in Section 8.2.2, we can reduce TLB misses by reducing r to the point where the working set is sufficiently small. Clearly, the permute phase is the bottleneck in this respect, and we should choose r such that $\lceil 2^r/P \rceil + \min\{2^r, n/P\} \leq T - 1$. Plugging in the Sun UltraSparc-II values of $P = 2048$ and $T = 64$, we get that for $n \geq PT \geq BC$ we should choose $r = 5$.

Of course, one may prefer that an average-case argument determines the radix size, especially since LSB radix sort anyway has poor worst-case cache behaviour. We now heuristically analyse the permute phase, as it is the major contributor to the TLB misses, to calculate the number of TLB misses for $r = 6, \ldots, 11$. For all these values, the count array fits into one page and we may assume that the count page and the current source page, once loaded, will never be evicted. We further simplify the process of accesses to the TLB and ignore disturbances caused when the source or one of the destination pointers crosses a page boundary (as these are transient). With these simplifications, TLB misses on accesses to the destination array may be modelled as uniform random access to a set of 2^r pages, using an LRU TLB of size $T - 2$. The probability of a TLB miss is then easily calculated to be $(2^r - (T-2))/2^r$. This suggests that choosing $r = 6$ on the Sun UltraSparc-II still gives a relatively low miss rate on average (miss probability 1/32), but choosing $r = 7$ is significantly worse (miss probability 1/2).

Figure 8.1 shows the time per key for one permute pass of the algorithm using radix 4, 5, 6, 7, 8, 11 and 16. Our experiments suggest that on random data, for $r = 1, \ldots, 5$ a single permute phase with radix r takes about the same time, as expected. Also, for $r = 7$ the permute time is—as expected—considerably (about 150%) slower than $r \leq 5$. However even $r = 6$ is about 25% slower than $r \leq 5$. This is probably because in practice T is effectively 61 or 62—it seems that the operating system reserves a few TLB entries for itself and locks them to prevent them from being evicted. Even using the simplistic estimate above, we should get a miss probability in the 1/13 to 1/16 range.

The choice $r = 5$ which guarantees good TLB performance turns out not to give the best performance on random data: it requires seven passes for sorting 32-bit data, at the end of which the keys are in a temporary array and have to be copied back into the original input array. The following sequence of radices saves one pass, avoids the extra copy at the end and also has other advantages noted below: $5, 5, 6, 5, 5, 6$.

8.3.2 Speeding up the count phase

The very small radix sizes suggested in the previous section mean that the number of passes over the data can be very large. In this situation the count phase (which normally contributes little to cache or TLB misses) can be a significant part of the computation. We now discuss ways of speeding up the count phase.

The first is Friend's improvement [37], which coalesces two count phases into one. More precisely, the frequencies of two successive digits (say the ith and $i + 1$st digits) are counted in one pass by updating two count arrays for each record in the input. An immediate extension is to count frequencies for $k > 2$ passes in one phase, *k-tuple counting*. The main disadvantage is that accessing multiple count arrays may cause conflict misses, but for the small radix values considered, the count arrays are minuscule and do not interfere significantly with each other. For $r = 6$ and 32-bit data, it was found that counting all six digits in one pass was considerably faster than three passes each counting two digits.

We found the following method to be still faster, as it greatly reduced the instruction count. To count the i-th and $i+1$-st digit simultaneously, we concatenate the i-th and $i + 1$-st digits (giving a $2r$-bit number) and increment a single count array of size 2^{2r}. In practice of course, we would simply mask out the relevant $2r$ bits in one step. From the $2r$-bit counts, frequencies for the i-th and $i + 1$-st passes are easily extracted. This approach can extend to coalescing three or more count phases into one. For example, for the radix sequence mentioned at the end of Section 8.3.1, we can obtain all six counts in one pass, by incrementing two 16-bit arrays for each key.

The first array would implicitly contain frequencies for the three least significant digits (totalling $6 + 5 + 5 = 16$ bits), and the second likewise for the three most significant digits. Note that this approach may increase conflict misses, and so may not work for all architectures.

8.4 Explicit Block Transfers

8.4.1 EBT radix sort

The external memory algorithm for LSB radix sort is the obvious one. The radix r is chosen so that $2^r \leq M/(2B) = C/2$; this enables buffers of B keys to be maintained in main memory for each of the 2^r classes during the permute phase, while leaving space for auxiliary data structures such as the count array and the current source array block. We describe only the permute phase, and assume that the count array is already stored in main memory. The algorithm reads successive blocks from the source array (which is stored in secondary memory), and moves each key in the current source array block to its main memory buffer. When the buffer is full, it is copied out into the appropriate block of the destination array (which is also stored in secondary memory). With a radix chosen as above and provided $n \geq M$, the algorithm performs $O(n/B)$ I/Os per pass. Using Theorem 3.2 in Chapter 3 we get:

Corollary 8.1 *For any associativity $a \geq 1$, the emulation of external memory radix sort with radix $r \leq \log(C/4)$ incurs $O(n/B)$ cache and TLB misses per pass in the worst case.*

In the future we refer to this as EBT (explicit block transfer) radix sort.

8.4.2 Implementing EBT Radix Sort

When implementing EBT radix sort, one may dispense with some of the steps in the emulation. The probability of conflicts between $Main$ and D in a block copy on random input seems (subjectively) so low that one should copy blocks directly, rather than through intermediate buffers. If pathological inputs are a concern then one should randomise the starting location of $Main$ as noted in [80].

We also limit the use of the emulation to critical parts of the algorithm. In effect, the algorithm implemented is normal LSB radix sort, but with the permute phase modified as follows. We maintain an array of 2^r buffers, all of which can hold $\Omega(B)$ keys. When moving keys in the permute phase, we do not do so directly, but instead move the key from the source array to the buffer corresponding to its class. When the buffer is full, all keys in it are copied as a block to their final locations in the destination array.

In practice one could probably use larger radix values such as $r = \log(C/2)$. In our case, this would correspond to using $r = 12$ instead of $r = 11$. Since the number of passes is not reduced for 32-bit data, we stayed with the smaller radix.

8.5 Pre-sorting

In this section we discuss two algorithms both of which pre-sort the keys in small groups to increase locality. Unless stated explicitly otherwise, we take the word 'key' in this section to mean the r-bit digit being sorted in one pass of PLSB or EPLSB radix sort.

8.5.1 PLSB Radix Sort

One pass of PLSB radix sort with radix r works in two stages. First we divide the input array of n keys into contiguous segments of $s \leq n$ keys each. Each segment is sorted using counting sort (a *local* sort) after which we sort the entire array using counting sort (a *global* sort). In each pass the time for sorting each of the $\lceil n/s \rceil$ local sorts is $O(s + 2^r)$ time and the time for the global sort is $O(n + 2^r)$, so the running time for one pass of PSLB radix sort is $O(n + n(2^r)/s)$.

An obvious optimisation is to accumulate counts for the global sort at the end of the count phase of each local sort. So the structure for the algorithm is as follows, where `Input` and `Temp` are arrays of size n and `GlobalCount` and `LocalCount` are arrays of size 2^r and the segments are numbered from $0, \ldots, \lceil n/s \rceil - 1$:

```
1     initialise GlobalCount
2     for $i = 0, \ldots, \lceil n/s \rceil - 1$ do
2.1     initialise LocalCount
2.2     count frequencies of key values in segment $i$ of Input
```

```
2.3    accumulate values in GlobalCount
2.4    prefix-sum LocalCount
2.5    permute from segment $i$ of  Input to segment $i$ of  Temp
3   prefix-sum GlobalCount
4   permute globally from  Temp to  Input
```

The intuition for the algorithm is that each local sort groups keys of the same class together and during the global sort we move sequences of keys to successive locations in the sorted array, thus reducing TLB and conflict misses between accesses to the destination array. We now give some more detailed intuition. We denote by γ the quantity $s/2^r$, and require that $\gamma \geq 2$. An immediate consequence is that one pass of PLSB radix sort runs in $O(n)$ time. To minimise the operation count, we would like to choose r as large as possible, but this pushes up s. To keep cache and TLB misses to a minimum in the local sorts, we would like the segments of the source and destination arrays in each local sort to fit into cache and TLB-addressable pages. This suggests that we must set $s \leq BC$ (the analysis below assumes that $s \leq BC/2$).

We move straightaway to the global permute phase now, since this was previously the source of most cache and TLB misses. An elementary observation is that an array of k consecutive items will span at most $\lceil k/P \rceil + 1$ pages or $\lceil k/B \rceil + 1$ blocks, and hence (if there are no pathological conflict misses) copying k consecutive items from the source array to consecutive locations in the destination array will incur at most this many cache and TLB misses in destination array accesses. If $k_{i,j}$ is the number of keys in class i in segment j, then the total number of destination array TLB misses would be roughly $\sum_{i=0}^{2^r-1} \sum_{j=0}^{n/s} \lceil k_{i,j}/P \rceil + 1 \leq O(n/P + (n/s) \cdot 2^r) = O(n/P + n/\gamma)$. Similarly, the number of cache misses would be about $O(n/B + n/\gamma)$. This suggests that we need $\gamma = \Omega(B)$ for an optimal number $O(n/B)$ of cache misses, and the number of TLB misses would then also be $O(n/B)$. This says that the radix r should satisfy $2^r = O(C)$, which is similar to the radix limitations of explicit block transfer.

Worst-case cache and TLB analysis for PLSB

We now give a worst-case cache and TLB miss analysis for PLSB. To determine the number of cache and TLB misses during the local sorts we describe our own cache and TLB replacement policies and determine the worst-case number of cache and TLB misses using these policies. Given that using an optimal replacement policy would make no more misses we calculate an upper bound on the cache and TLB misses given an LRU replacement policy.

Without loss of generality we assume the cache and TLB are empty at the start of the local sorts, at the start of the prefix-sum of `GlobalCount` and at start of the global permute.

Theorem 8.1 *For $s = CB/2$ and $\gamma \geq 2$, the worst-case number of cache misses in one pass of PLSB when sorting n keys with an $a \geq 4$ ways associative cache is at most:*

$$\frac{C}{2\gamma} + \frac{2n}{B} + \frac{5n}{\gamma} + 1 + \frac{a}{a - \lceil a/2 \rceil - \lceil a/(2\gamma) \rceil} \left(\frac{C}{\gamma} + \frac{3n}{B} + \frac{n}{B\gamma} + \frac{2n}{CB} + 5 \right)$$

where s is the number of keys in a local sort, $\gamma = s/2^r$ and r is the radix.

Proof.

During each local sort our replacement algorithm reserves sufficient ways to keep `LocalCount` in cache once it has been loaded. This replacement algorithm also reserves sufficient ways to keep a segment of `Temp`, once loaded, in cache during step 2.5. It also reserves 1 further way for loading a segment of `Input` in cache during steps 2.2 and 2.5, it uses the same way to load `GlobalCount` during step 2.3. Since $C = Sa$ and $s = BSa/2$, `LocalCount` is $2^r/B = Sa/(2\gamma)$ blocks in size and this replacement algorithm requires $\lceil a/(2\gamma) \rceil$ ways for `LocalCount`. Each segment of `Temp` is $Sa/2$ blocks in size and this replacement algorithm requires $\lceil a/2 \rceil$ ways for a segment of `Temp`.

With this replacement algorithm, once `LocalCount` has been loaded into cache for the first local sort it will remain in cache for all further local sorts. An upper bound on the cache misses for the local sorts in one pass of PLSB using this replacement algorithm is derived as follows:

- $\lceil Sa/(2\gamma) \rceil \leq C/(2\gamma) + 1$ misses for loading `LocalCount` once.
- $\lceil n/B \rceil$ misses for loading `Input` into cache over all local sorts in step 2.2
- $\lceil 2n/(BSa) \rceil \cdot \lceil Sa/(2\gamma) \rceil \leq \frac{n}{B\gamma} + \frac{2n}{CB} + \frac{C}{2\gamma} + 1$ misses over all local sorts for loading `GlobalCount` into cache in step 2.3.
- $2\lceil n/B \rceil$ cache misses for loading `Input` and `Temp` into cache over all local sorts in step 2.5.

Over all local sorts this replacement algorithm uses $a^* = \lceil a/2 \rceil + \lceil a/(2\gamma) \rceil + 1$ ways and there are at most:

$$\frac{C}{\gamma} + \frac{3n}{B} + \frac{n}{B\gamma} + \frac{2n}{CB} + 5 \tag{8.5}$$

cache misses. Using Theorem 3.1 we get that the number of cache misses in an a-way associative LRU cache is at most:

$$\frac{a}{a - \lceil a/2 \rceil - \lceil a/(2\gamma) \rceil} \left(\frac{C}{\gamma} + \frac{3n}{B} + \frac{n}{B\gamma} + \frac{2n}{CB} + 5 \right) \tag{8.6}$$

For the prefix-sum of `GlobalCount` in step 3 and the global permute in step 4 we do not utilise the optimal cache. During the prefix-sum of `GlobalCount` there are

$$\left\lceil \frac{Sa}{2\gamma} \right\rceil \leq \frac{C}{2\gamma} + 1$$

cache misses. For the global permute, we divide the memory accesses into $(n/s) \cdot 2^r = n/\gamma$ *epochs.* In epoch (i, j), for $0 \leq i \leq 2^r - 1$ and $0 \leq j \leq n/s$, we move the $k_{i,j}$ keys with value i from local sort j to their final destination. In this epoch we access at most $2(\lceil k_{i,j}/B \rceil + 1)$ blocks from the `Input` and `Temp` arrays in all, and one block from the count array. The number of misses is at most $2\lceil k_{i,j}/B \rceil + 3 \leq 2k_{i,j}/B + 5$. Summing over all epochs we get that the number of cache misses for the global permute is at most:

$$\sum_{i=0}^{2^r-1} \sum_{j=0}^{n/s} (2k_{i,j}/B + 5) = \frac{2n}{B} + \frac{5n}{\gamma} \tag{8.7}$$

Summing the cache misses for the local sorts, global prefix-sum and global permute gives the worst-case number of cache misses for one pass of PLSB. $\square$

Corollary 8.2 *If $\gamma = \Omega(B)$ then there are $O(n/B)$ cache misses in one pass of PLSB in an a-way associative cache with LRU replacement policy.*

Theorem 8.2 *For $s = BC/2 \leq PT/2$ and $\gamma \geq 2$ the worst-case number of TLB misses in one pass of PLSB when sorting n keys using a TLB of size $T \geq 4$ is at most:*

$$\frac{T}{2\gamma} + \frac{2n}{P} + \frac{5n}{\gamma} + 1 + \frac{T}{T - \lceil T/2 \rceil - \lceil T/(2\gamma) \rceil} \left(\frac{T}{\gamma} + \frac{3n}{P} + \frac{n}{P\gamma} + \frac{2n}{PT} + 5 \right)$$

where s is the number of keys in a local sort, $\gamma = s/2^r$ and r is the radix.

Proof. During each local sort our replacement algorithm reserves sufficient TLB entries to keep page translations for `LocalCount`, once loaded, in the TLB during steps

2.1, 2.2, 2.4 and 2.5. This replacement algorithm also reserves sufficient TLB entries to keep page translations for a segment of `Temp`, once loaded, in the TLB entries during 2.5. It also uses 1 further TLB entry for the page translations for a segment of `Input` during steps 2.2 and 2.5, it uses the same TLB entry to load `GlobalCount` during step 2.3. Since $s = BC/2 \leq PT/2$, `LocalCount` requires at most $2^r/P = T/(2\gamma)$ pages and this replacement algorithm requires at most $\lceil T/(2\gamma) \rceil$ TLB entries for `LocalCount`. Each segment of `Temp` is at most $T/2$ pages in size and this replacement algorithm requires at most $\lceil T/2 \rceil$ TLB entries for a segment of `Temp`.

With this replacement algorithm, once translations for `LocalCount` have been loaded into the TLB for the first local sort they will remain there for all further local sorts. An upper bound on the TLB misses for the local sorts in one pass of PLSB using this replacement algorithm, is as follows:

- $\lceil T/(2\gamma) \rceil$ misses for `LocalCount` in step 2.1 for the first local sort.
- $\lceil n/P \rceil$ misses for `Input` over all local sorts in step 2.2
- $\lceil 2n/(PT) \rceil \cdot \lceil T/(2\gamma) \rceil \leq \frac{n}{P\gamma} + \frac{2n}{PT} + \frac{T}{2\gamma} + 1$ misses over all local sorts for `GlobalCount` in step 2.3.
- $2\lceil n/P \rceil$ misses for `Input` and `Temp` over all local sorts in step 2.5.

Over all local sorts this replacement algorithm uses $T^* = \lceil T/2 \rceil + \lceil T/(2\gamma) \rceil + 1$ TLB entries and there are at most:

$$\frac{T}{\gamma} + \frac{3n}{P} + \frac{n}{P\gamma} + \frac{2n}{PT} + 5 \tag{8.8}$$

TLB misses. Using Theorem 3.1 we get that the number of TLB misses using a LRU TLB of size T is at most:

$$\frac{T}{T - \lceil T/2 \rceil - \lceil T/(2\gamma) \rceil} \left(\frac{T}{\gamma} + \frac{3n}{P} + \frac{n}{P\gamma} + \frac{2n}{PT} + 5 \right) \tag{8.9}$$

For the prefix-sum of `GlobalCount` in step 3 and the global permute in step 4 we do not utilise results for optimal TLB. There are:

$$\left\lceil \frac{T}{2\gamma} \right\rceil \leq \frac{T}{2\gamma} + 1$$

TLB misses for the prefix-sum of `GlobalCount`. For the global permute, we divide the memory accesses into $(n/s) \cdot 2^r = n/\gamma$ *epochs.* In epoch (i, j), for $0 \leq i \leq 2^r - 1$

and $0 \leq j \leq n/s$, we move the $k_{i,j}$ keys with value i from local sort j to their final destination. In this epoch we access at most $2(\lceil k_{i,j}/P \rceil + 1)$ pages from the `Input` and `Temp` arrays in all, and one page from the count array. The number of misses is at most $2\lceil k_{i,j}/P \rceil + 3 \leq 2k_{i,j}/P + 5$. Summing over all epochs we get:

$$\sum_{i=0}^{2^r-1} \sum_{j=0}^{n/s} (2k_{i,j}/P + 5) = \frac{2n}{P} + \frac{5n}{\gamma} \tag{8.10}$$

Summing the TLB misses for the local sorts, global prefix-sum and global permute gives the worst-case number of TLB misses for one pass of PLSB. $\square$

Corollary 8.3 *If $\gamma = \Omega(B)$ then there are $O(n/B)$ TLB misses in one pass of PLSB.*

Implementing PLSB Radix Sort

Clearly, from the analyses above, we would like generally to get a large value of $\gamma = s/2^r$, as this tends to reduce cache and TLB misses in the global permute. The analysis above used $s = (BC)/2$, and we now review the local sort phase in order to find the largest 'practical' value of s.

To keep TLB misses low we want to ensure that the working set of a local sort can comfortably be held in TLB. On the Sun UltraSparc-II the TLB has size 64, so we should limit s to be at most kP for some integer k in the high 50s; this allows a couple of system TLB entries, a source page entry and a handful of count page entries all to fit in TLB. On pathological data, choosing $s > (BC)/2$ can give many conflict misses in the local sorts, even on a 4-way associative cache. However, by using randomisation, we ensure that no single input gives a bad running time, and so the algorithm has few expected conflict misses on any input. More precisely, by ensuring that each of the source and destination arrays start independently at locations which are uniformly distributed over the range $\{0, \ldots, BC-1\}$, we get very few conflict misses in the local sorts even on direct-mapped caches, regardless of the input (this is easily shown using arguments similar to those of [65]). A side effect of the randomisation is that there are no pathological inputs for the global permute on a direct-mapped cache.

We use a value $s \approx 117,400$, corresponding to k between 57 and 58. This particular number is 'good' in some way for EPLSB radix sort, see Section 8.5.2, and we choose it again in PLSB radix sort for convenience, although it has no significance for PLSB radix beyond that already discussed. Also, experimentally we determined that $r = 11$ is the best radix to use for sorting 32-bit data. This gives $\gamma \approx 57 \approx 3.5B$.

One effective low-level optimisation for PLSB is that during the global permute, we do not use the standard out-of-place permute algorithm. Instead, before moving a record from `Temp` back to `Input` in the global permute, we first scan ahead in `Temp` to find a maximal set of k consecutive records which have the same key value as the next record to be moved. After moving these k records as a block we increase the relevant count array location by k. This saves significantly on instructions and also on writes to the count array (even though this is not part of the model used, writes in general are a little expensive as, on our Sun UltraSparc-II, they are handled by the L2 cache rather than the L1 cache). After this optimisation, the global permute is less than 30% slower than a straight copy from `Temp` to `Input`.

8.5.2 EPLSB Radix Sort

In EPLSB radix sort with radix r, each pass is again done in two stages. We use the term 'global key' to refer to the r-bit sort key for a particular pass, and we will use the term 'local key' to refer to the r' most significant bits of the global key, where $r' \leq r$ is a parameter whose value will be chosen later. If we choose $r' = r$ EPLSB radix sort essentially reduces to PLSB radix sort.

We assume again that $n \geq BC$. First, as in PLSB radix sort, we divide the input array of n records into contiguous segments of $s \leq n$ records each, and sort each segment. The local sorts now only sort according to the local key, but the global permute moves records to their final location according to the global key. During the global permute, we again process blocks of records from each segment, where a block consists of successive records with the same local key value. Although it is no longer the case that all records in a block are moved together, they belong to at most $\Delta = 2^{r-r'}$ different (global) classes, which limits the number of active locations in the destination array. Intuitively, we can let $\Delta = \Theta(T)$ and still keep TLB misses low, thus extending the radix size of PLSB by a factor of roughly $\log T$.

A more precise description of the algorithm follows. `Input` and `Temp` are arrays of size n. `Input` contains the input records, and at the end of the pass, `Input` will once again contain the records in sorted order with respect to the global key. `GlobalCount` and LocalCount are arrays of size 2^r and $2^{r'}$ respectively, and the segments are numbered from $0, \ldots, \lceil n/s \rceil - 1$.

```
1    /* perform all local sorts */
1.0  for $i = 0, \ldots, \lceil n/s \rceil - 1$ do
1.1    initialise LocalCount
1.2    count frequencies of local key values in segment $i$ of Input
1.3    prefix-sum LocalCount
1.4    permute records from segment $i$ of Input to segment $i$ of Temp
       /* Step 1.4 moves records according to local key value */
2   Initialise GlobalCount
3   Count frequencies of global key values.
4   /* global permute */
4.0  for $j = 0, \ldots, 2^{r'} - 1$
4.1    for $i = 0, \ldots, \lceil n/s \rceil - 1$
4.2       move all records with local key value $j$ in segment $i$
          of Temp to their final location in Input.
          /* Step 4.2 moves records according to global key value */
```

We now fix some parameter choices. We choose $s = (BC)/2$, $r' \leq \log C$ and $r \leq \min\{\log(BC/2), r' + \log(T/2)\}$. This ensures that $\gamma = s/2^{r'} \geq B/2$ and $\Delta \leq T/2$.

We now estimate the number of instructions, cache and TLB misses in one pass of EPLSB radix sort. We begin with instruction counts. Step 1 takes $O(n + n(2^{r'})/s) = O(n+n/\gamma) = O(n)$ time. Steps 2 and 3 take $O(n+2^r) = O(n)$ time, since $2^r \leq BC \leq n$. Letting $k_{i,j}$ denote the number of keys with local sort key value j in segment i of the input, Step 4 takes $\sum_{j=0}^{2^{r'}-1} \sum_{i=0}^{\lceil n/s \rceil} (k_{i,j}+1) = O(n+n/\gamma) = O(n)$ time, giving an overall bound of $O(n)$ time for one pass of EPLSB radix sort.

We now move to TLB misses. The analysis of the TLB misses for Step 1 is analogous to the analysis of Step 2 of PLSB, and we can conclude that there are $O(n/B)$ misses in this step. Steps 2 and 3 have negligible TLB misses (note that $2^r \leq BC/2$, so the count array requires at most $T/2$ pages). In Substep 4.2, note that we are moving elements belonging to Δ different global key classes in each iteration of the loop of Step 4.0. The count information for these classes is spread across at most two GlobalCount pages, as $\Delta \leq T \leq P$. We now calculate the number of misses made by an optimal TLB of size

$\Delta + 3$. The optimal TLB would make no more than the following misses:

- By always evicting a page from `Input` when an access to `Input` causes a miss, at most $\sum_{j=0}^{2^{r'}-1} \sum_{i=0}^{\lceil n/s \rceil} \lceil k_{i,j}/P \rceil + 1 \leq n/P + 2n/\gamma = O(n/B)$ misses due to accesses to `Input`, where $k_{i,j}$ is as above.

- Similarly, accesses to `Temp` cause at most $O(n/P + \Delta \cdot n/s)$ misses. Since $\Delta n/s \leq 2\Delta n/(BC) \leq 2nT/(BC) = O(n/B)$, the total number of misses is again $O(n/B)$ (and should usually be considerably smaller, as $T \ll C$).

- At most $\lceil 2^r/P \rceil = O(n/P)$ misses for accesses to `GlobalCount`.

Hence the minimum size of the optimal TLB required to achieve $O(n/B)$ misses is no more than $\Delta + 3$. This is at most $2T/3$ if P and T are sufficiently large, so an LRU TLB of size T will also make $O(n/B)$ misses in the worst case.

We now turn to cache misses. As $s = (BC)/2$ and $2^r \leq C$, the number of misses in Step 1 is $O(n/B)$ in the worst case. For a direct-mapped cache this requires `Input` and `Temp` to be aligned so that corresponding segments of these arrays are mapped to distinct parts of the cache; we can dispense with this assumption for an a-way associative cache with $a \geq 4$.

In Step 3 the number of cache misses is $O(n/B)$ in the worst case if the cache is at least two-way associative (recall that $2^r \leq (BC)/2$ so the count array can occupy at most one way of each set).

In Step 4, again randomising the start of `Temp` and `Input` reduces the number of conflict misses between (i) `GlobalCount` and `Temp`; (ii) `Input` and `Temp` and (iii) `GlobalCount` and `Input` to negligible levels. However, this does not solve the problem of conflicts between the $T/2$ active locations in `Input`, which can conflict with each other. One solution to this problem is to randomise the start of each (global key) class, as suggested in [65]. If this is done, then it follows from the analysis there that the number of conflict misses should be small as $T/2 \ll C$ so there are $O(n/B)$ misses.

Implementing EPLSB radix sort

For the Sun UltraSparc-II parameters, the maximum permissible value of r is 16 (since $BC/2 = 2^{16}$). As is the case for PLSB, it is beneficial to EPLSB to have a value of γ as large as possible. This means making s as large as possible and r' as small as possible. The smallest value for r' which gives few worst-case TLB misses is $16 - \log(T/2) = 11$. As in PLSB radix sort we randomise the start locations of the source and destination arrays; this reduces the number of cache conflict misses in Steps 1-3 to negligible levels (according to the model).

We again choose s to be about $58P$ to keep TLB misses in local sorts at a minimum. Here, however, the value of s also slightly affects the global permute, so s is fixed later.

We optimise the global counts by scanning the array `Temp` for the purpose of global counting, rather than `Input`. This improves performance by improving the locality of the global counts in a way that the model appears not to capture. A further improvement is to perform global counts on segment i of `Temp` right after the local sort for segment i, as most of `Temp` will be in cache.

We make no special optimisations to remove the main problem with the global permute, namely that EPLSB radix sort is susceptible to pathological data. The only possibility, as noted above, is to randomise the start of each class as in [65]. Using their approach directly would have a prohibitive space cost, but since there are only $T/2$ active destination pointers in EPLSB radix sort, it is possible that one may not need to randomise by as much as suggested in [65]. It would be interesting to explore this aspect further.

As we will see in the section on experimental results, in EPLSB radix sort worst-case inputs cause many TLB misses during the global permute when the number of active destination pointers $\Delta = T$. We also see that since $\Delta \ll C$ and the cache is physically mapped the theoretical worst-case input does not actually cause many more cache misses than random inputs. This justifies the design decision to make EPLSB radix sort robust against worst-case inputs for TLB misses but not for cache misses.

We do incorporate a second-tier optimisation for the global permute. The global permute for EPLSB radix sort makes several 'passes' over the `Temp` array (each pass corresponds to an iteration of the loop of Step 4.0). Each 'pass' moves a relatively small group of keys from a segment to their final destinations, before moving on to the next segment. As the end of a group may not lie on a cache boundary in general, it would be helpful if the block which contains the end of the group stays in cache for the next 'pass': this saves an extra miss for each block. For random data this can be significant: each group is typically only 3.5 cache blocks long, so the number of `Temp` array misses would be reduced by about 20%.

In practice the number of segments is small (less than 300 for $n = 32000000$) compared to C, so active `Temp` blocks could easily stay in cache between 'passes'. Intuitively the chances of this happening are improved if the starts of the segments were mapped to blocks which are roughly equally spaced in cache. Some values of s are better than others at achieving this: in Chapter 6 we noted that values such as $s = (1 - 1/(9 + \phi^{-1}))BC \approx 117,400 \approx 57.3P$, where $\phi = (1 + \sqrt{5})/2$, are provably good for this. This is the value used in the code. This optimisation does not always help, but it never hurts either.

8.6 Asymptotic number of cache and TLB misses

As discussed in Chapter 4, by a direct reduction from the argument in [8], we have that any algorithm for sorting n keys must make $\Omega((n/B)\log_C(n/B))$ cache misses and $\Omega((n/P)\log_T(n/P))$ TLB misses.

These lower bounds, however, apply only when all $n!$ permutations of the input keys are possible. When (stably) sorting w-bit keys, the number of possible permutations is at most 2^{wn}, which could be smaller or greater than $n!$. In particular, if the radix r is chosen appropriately, then one pass of r-bit LSB radix sort can be accomplished in $O(n/B)$ cache misses, or $O(n/P)$ TLB misses. This has also been observed by [62] for bundle sorting. By a direct reduction from their argument we have that $\Omega((n/B)\log_C \min\{(n/B), 2^w\})$ cache misses and $\Omega((n/P)\log_T \min\{(n/P), 2^w\})$ TLB misses are necessary to sort w-bit integers.

We now compare the asymptotic performance of the algorithms. Recall that we assume that $T \leq C$, $B \leq P$, $BC \leq PT$ and $B \leq C$. First, we consider the case when T is not too small, i.e., $\log T = \Theta(\log C)$. In this case it suffices for optimality that the algorithm makes $O(n/B)$ cache and $O(n/P)$ TLB misses per pass, where each pass sorts according to a radix which is at least $\Omega(\log T)$. First note:

Proposition 8.6 *Assuming an a-way associative cache for some $a \geq 2$, LSB radix sort makes an optimal expected number of cache and TLB misses on any input, if $\log T = \Theta(\log C)$.*

PROOF. We choose the radix of LSB radix sort as $r = (1 - 1/a)\log T - 1$. Since $r \le \log(T/2)$, standard LSB radix sort makes $O(n/P)$ TLB misses per pass in the worst case. Also, $2^r \le T^{1-1/a} \le C^{1-1/a} \le C/B^{1/a}$. If we assume that the algorithm uses randomised memory allocation as described in [65], it follows from the results there that if $2^r \le C/B^{1/a}$, LSB radix sort makes $O(n/B)$ expected cache misses per pass on an a-way associative cache on any input. Observing that $r = \Omega(\log C)$ completes the proof. □

An analogue for worst-case behaviour follows:

Proposition 8.7 *Assuming an a-way associative cache for some $a \ge 2$, one can sort incurring an optimal expected number of cache and TLB misses on any input, if* $\log T = \Theta(\log C)$.

PROOF. The algorithm is essentially PLSB radix sort modified to use EBT radix sort for the local sorts. Firstly, the input is divided into segments of size $PT/4$ and each segment is locally sorted with respect to the current r-bit digit using EBT radix sort, where $r = \log T - c \le \log C - O(1)$ for some constant $c \ge 0$. By choosing c large enough we can ensure that all pages for each local sort fit into the TLB. In this case, EBT radix sort incurs $O(n/B)$ cache and $O(n/P)$ TLB misses, summed over all local sorts. The global permute then moves keys to their final destination as before. Since $\gamma = \Omega(P)$ here, the global permute also makes $O(n/B)$ cache and $O(n/P)$ TLB misses per pass, provided $a \ge 2$ by Theorems 8.1 and 8.2. As there are $\Theta(w/\log C)$ passes, the proposition follows. □

If $\log T = o(\log C)$ we do not know how to simultaneously get asymptotically optimal TLB and cache misses. EBT radix sort or PLSB radix sort with radix $\epsilon \log C$ for some constant $1 \ge \epsilon > 0$ makes $O(n/(BC^{1-\epsilon}))$ page misses per pass, while keeping the number of cache misses asymptotically optimal. However, the model does not rule out $P \gg BC$, so this can be non-optimal. Approaches based on choosing a radix of $\Theta(\log T)$ yield TLB-optimality but make too many cache misses.

8.7 Experimental results

The following table describes the algorithms that we tested and gives the names we will use for them. All algorithms were coded in C and all code compiled using `gcc 2.8.1`.

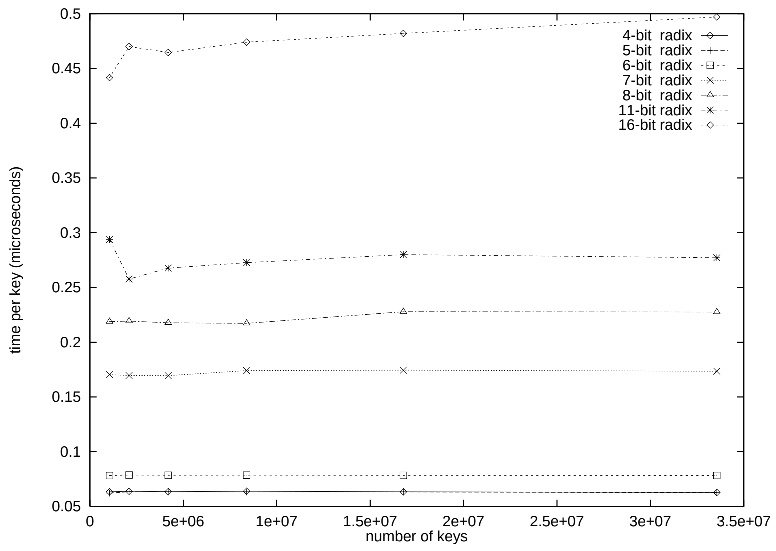

Figure 8.1: Time per key for one permute pass of LSB radix sort using radix $4, 5, 6, 7, 8, 11$ and 16, at $n = 1 \times 10^6$, 2×10^6, 4×10^6, 8×10^6, 16×10^6 and 32×10^6. Keys are random 32-bit unsigned integers.

The experiments were run on our Sun UltraSparc-II with 2×300 MHz processors and 1 GB of memory.

Name	Description
LSBr	LSB radix sort using r-bit radix with $\lceil 32/r \rceil$-tuple counting
FLSBr	LSB radix sort using r-bit radix with Friend's improvement
LSB556	LSB radix sort where a 16-bit count phase gathers count information for three permute phases. The first two of these permute phases use 5-bit radix and the third uses a 6-bit radix.
PLSBr	PLSB radix sort using r-bit radix
EPLSBr	EPLSB radix sort using r-bit local radix and 16-bit global radix
EBTr	EBT radix sort using r-bit radix
MTQuick	cache-tuned (*memory-tuned*) Quicksort[59]
TMerge	cache-tuned (*tiled*) Mergesort [59]

8.7.1 Random inputs

We first tested the algorithms on uniformly distributed random integers for $n = i \times 10^6$, $i = 1, 2, 4, 8, 16, 32$. We avoided using values of n near powers of 2, as the analysis in Chapter 6 suggests that LSB radix sort may perform slightly poorly at these values of n.

Timings(sec)			
n	LSB556	LSB6	FLSB6
1×10^6	0.57	0.64	0.67
2×10^6	1.12	1.28	1.36
4×10^6	2.20	2.56	2.72
8×10^6	4.35	5.09	5.44
16×10^6	8.57	10.22	10.84
32×10^6	17.52	20.45	21.85

Table 8.2: Overall running times for LSB556, LSB6 and FLSB6 on random 32-bit unsigned integers.

Figure 8.1 shows the time per key for one permutation pass of LSB radix sort. We see that using an r-bit radix, when $2^r < T$ the running times remain constant and when $2^r \geq T$ the running times increase as TLB misses increase (at $r = 4, 5, 6, 7, 8$ there are very few cache conflict misses). At $r = 11$ almost all accesses to the destination array will cause TLB misses and the increase in running time at $r = 16$ is due to increasing cache conflict misses.

Table 8.2 shows the overall running times for LSB6, FLSB6 and LSB556. We see that since the radix is small 6-tuple counting gives a faster running time than using Friend's improvement. Since a permutation phase using a 5-bit radix has the same running time as using a 4-bit radix and is faster than using a 6-bit radix, we see that LSB556, which has in total four 5-bit and two 6-bit permute phases is faster than LSB6 which has effectively five 6-bit and one 4-bit permute phases.

Table 8.3 shows the overall running times for LSB radix sort. We found that $\lceil 32/r \rceil$-tuple counting was faster than using Friend's improvement for radix sizes $r \leq 8$ bits, so we give results using $\lceil 32/r \rceil$-tuple counting for $r = 4, 5, 6, 7, 8$ bits and using Friend's improvement for $r = 11, 16$ bits. We see that LSB6 is the fastest on random inputs as it has very few cache and TLB misses. LSB7 has one less pass and about the same number of cache misses but almost half the accesses to the destination array cause TLB misses, so we see that it is almost 45% slower. LSB5 has very few cache and TLB misses but has one more pass than LSB6 and is slightly slower, however we will see later it is robust against worst-case inputs whereas LSB6 is not.

Timings(sec)							
n	LSB 4	LSB 5	LSB 6	LSB 7	LSB 8	FLSB 11	FLSB 16
1×10^6	0.74	0.68	*0.64*	0.93	0.95	0.90	0.92
2×10^6	1.49	1.36	*1.28*	1.85	1.98	1.86	1.94
4×10^6	2.99	2.73	*2.56*	3.70	3.86	3.86	4.08
8×10^6	6.26	5.57	*5.09*	7.38	7.67	7.68	7.90
16×10^6	11.94	10.94	*10.22*	14.75	15.79	15.23	15.99
32×10^6	24.06	21.96	*20.45*	29.62	30.50	31.71	33.49

Table 8.3: Overall running times for LSB radix sort with varying radix on random 32-bit unsigned integers.

Timings(sec)								
n	EPLSB11	PLSB11	EBT11	LSB556	LSB6	FLSB11	MTQuick	TMerge
1×10^6	**0.46**	*0.47*	0.54	0.57	0.64	0.90	0.70	1.02
2×10^6	**0.89**	*0.92*	1.09	1.12	1.28	1.86	1.50	2.22
4×10^6	**1.74**	*1.82*	2.20	2.20	2.56	3.86	3.24	4.47
8×10^6	**3.53**	*3.64*	4.49	4.35	5.09	7.68	6.89	9.71
16×10^6	**7.48**	*7.85*	8.81	8.57	10.22	15.23	14.65	19.47
32×10^6	**14.96**	*15.66*	17.55	17.52	20.45	31.71	31.69	41.89

Table 8.4: Comparison of TLB-tuned radix sorts: EPLSB11, PLSB11, EBT11, LSB556 and LSB6 versus cache-tuned LSB11, MTQuick and TMerge on random 32-bit unsigned integers.

Table 8.4 shows the overall running times for the TLB optimised radix sorting algorithms, EPLSB11, PLSB11, EBT11, LSB556 and LSB6 and the cache-optimised algorithms LSB11, MTQuick and TMerge. We see that EPLSB and PLSB radix sort outperform all the radix sort variants, getting speedups of 14+% and 10+% over the other TLB-optimised algorithms. These two algorithms obtain speedups of 52+% and 50+% over the cache optimised MTQuick, TMerge and LSB11, for large n.

8.7.2 Testing robustness

We also tested the algorithms on the following input sequences:

- $0, \ldots, n-1$

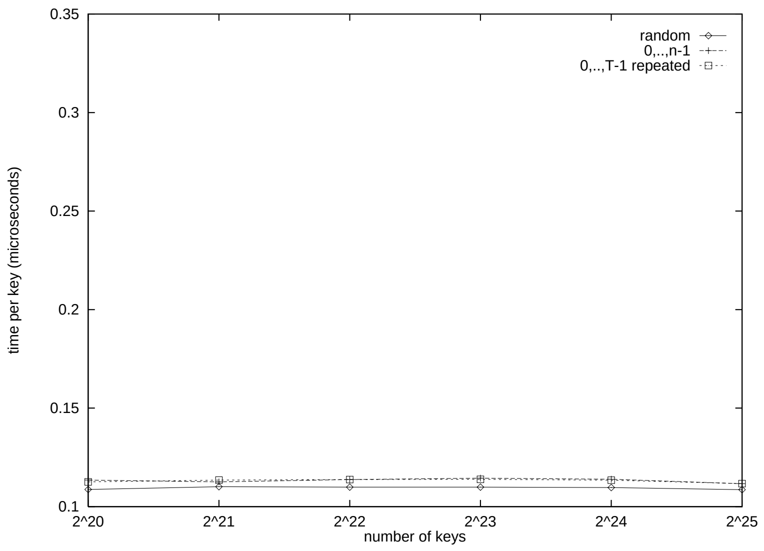

Figure 8.2: Time per key for one pass of normal LSB with 5-bit radix, on random keys (random), on keys $0, \ldots, 2^n - 1$ (0,...n-1) and on repeated key sequence $0, \ldots, T-1$ (0,..,T-1 repeated) at $n = 2^{20}, 2^{21}, 2^{22}, 2^{23}, 2^{24}$ and 2^{25}.

- $0, \ldots, T-1$ repeated n/T times

with $n = 2^{20}, 2^{21}, 2^{22}, 2^{23}, 2^{24}, 2^{25}$, and where appropriate $s = BC/2 = PT/2$. We compared the running times for one pass of the algorithms LSB5, LSB6, EPLSB10, EPLSB11, PLSB11 and EBT11 with uniformly distributed random keys against the input sequences above. Since these experiments were for just one pass, the algorithms were modified to gather counts for the first pass only, so one should not use these results to extrapolate the overall running times.

As stated in Proposition 8.3 the sequence $0, \ldots, n-1$ should cause worst-case cache misses at these values of n in LSB radix sort and one can easily show that it should also causes worst-case TLB when the radix $r \geq \log T$.

With these values of n and where appropriate s, the repeated sequence $0, \ldots, T-1$ should cause worst-case cache misses in LSB radix sort and EPLSB radix sort. This sequence should also cause worst-case TLB miss in LSB radix sort if radix $r \geq \log T$ and in EPLSB radix sort if the difference between the global and local radix lengths is at least $\log T$.

Figures 8.2, 8.3, 8.4, 8.5, 8.6 and 8.7 summarise the results obtained. We see in Figures 8.2 that in LSB5, where worst-case TLB misses are not in effect, there is only

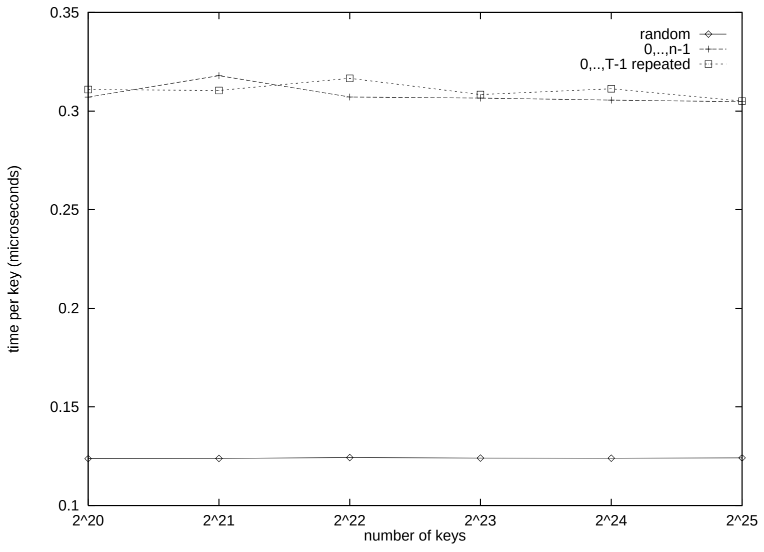

Figure 8.3: Time per key for one pass of normal LSB with 6-bit radix, on random keys (random), on keys $0, \ldots, 2^n - 1$ (0,...n-1) and on repeated key sequence $0, \ldots, T-1$ (0,..,T-1 repeated) at $n = 2^{20}, 2^{21}, 2^{22}, 2^{23}, 2^{24}$ and 2^{25}.

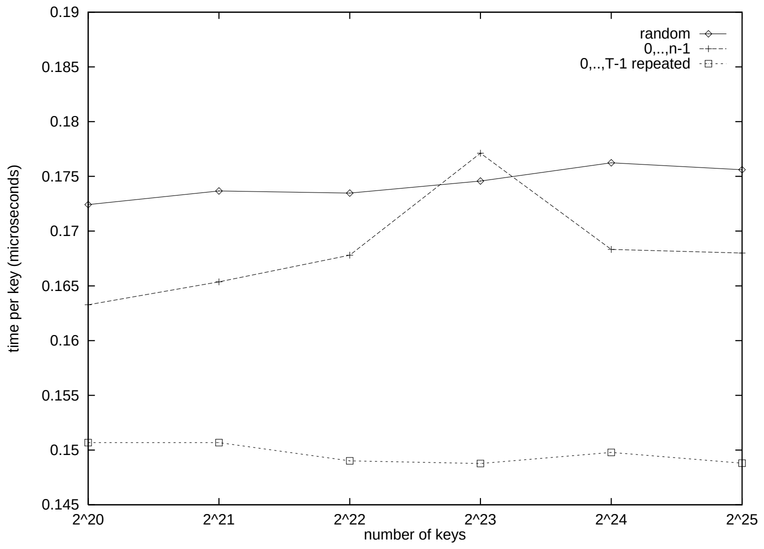

Figure 8.4: Time per key for one pass of explicit block transfer (EBT) with 11-bit radix, on random keys (random), on keys $0, \ldots, 2^n - 1$ (0,...n-1) and on repeated key sequence $0, \ldots, T-1$ (0,..,T-1 repeated) at $n = 2^{20}, 2^{21}, 2^{22}, 2^{23}, 2^{24}$ and 2^{25}.

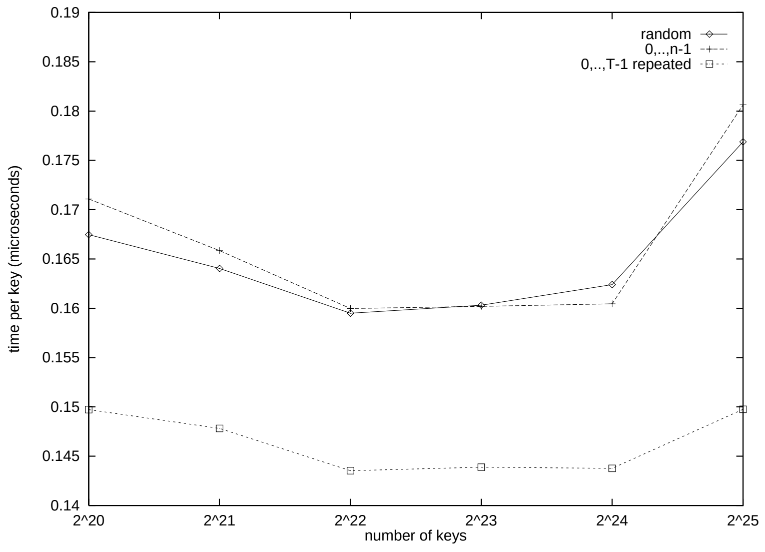

Figure 8.5: Time per key for one pass of PLSB with 11-bit radix, on random keys (random), on keys $0, \ldots, 2^n - 1$ (0,...n-1) and on repeated key sequence $0, \ldots, T-1$ (0,..,T-1 repeated) at $n = 2^{20}, 2^{21}, 2^{22}, 2^{23}, 2^{24}$ and 2^{25}.

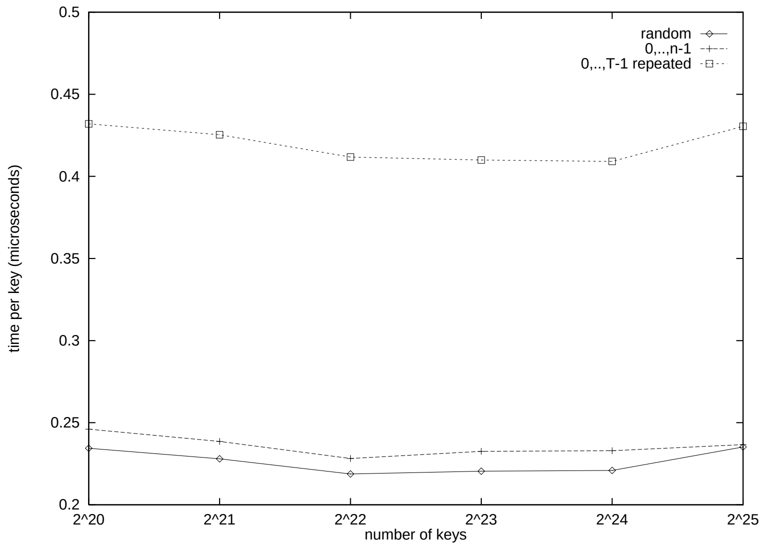

Figure 8.6: Time per key for one pass of EPLSB with a 16-bit global key and 10-bit local key on random keys (random), on keys $0, \ldots, 2^n - 1$ (0,...n-1) and on repeated key sequence $0, \ldots, T-1$ (0,..,T-1 repeated) at $n = 2^{20}, 2^{21}, 2^{22}, 2^{23}, 2^{24}$ and 2^{25}.

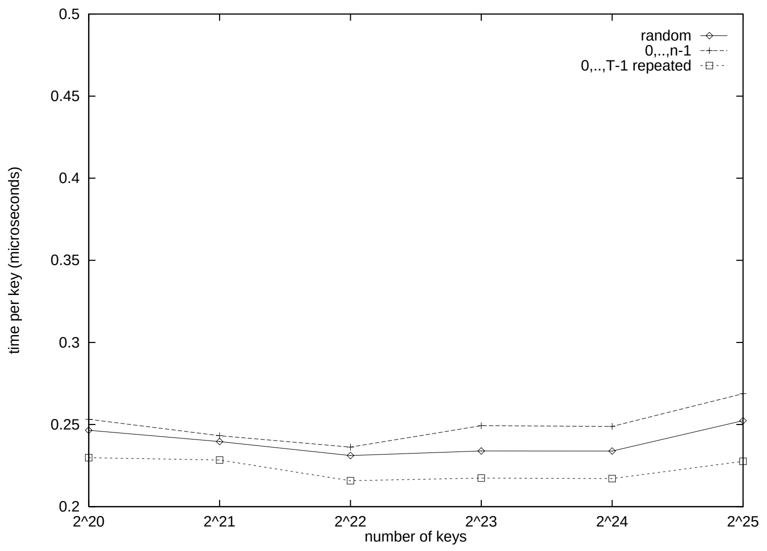

Figure 8.7: Time per key for one pass of EPLSB with a 16-bit global key and 11-bit local key on random keys (random), on keys $0, \ldots, 2^n - 1$ (0,...n-1) and on repeated key sequence $0, \ldots, T-1$ (0,..,T-1 repeated) at $n = 2^{20}, 2^{21}, 2^{22}, 2^{23}, 2^{24}$ and 2^{25}.

a slight, 3%, increase in running time with the above inputs versus the random keys. Similarly we see in Figure 8.7 with the repeated sequence $0, \ldots, T-1$ there is only a slight, 7%, increase in the running time of EPLSB11, where again worst-case TLB misses are not in effect. This is almost certainly due to the fact that the cache is physically mapped so we can not ensure that physical pages allocated to classes map to particular cache blocks. So we note that when 2^r is small we do not observe any significant increase in running time due to cache conflict misses even with worst-case inputs. This is in contrast to the results in Figure 8.1, when 2^r is large the increase in running times between radix 11 and radix 16 is almost entirely due to an increase in cache conflict misses.

We see in Figure 8.3 that in LSB6, where worst-case TLB misses are in effect, one pass with the above inputs takes almost 2.5 times as long as on the random keys. Similarly we see in Figure 8.6 that in EPLSB10 one pass with the repeated sequence $0, \ldots, T-1$ takes almost twice as long as with the random keys, again worst-case TLB misses are in effect.

These results validate the design decision to tune EPLSB to prevent worst-case TLB behaviour but allow theoretically worst-case cache behaviour.

In theory PLSB should not have a worst-case behaviour, and we see in Figure 8.5 that the running time hardly changes with the above inputs versus random keys.

8.7.3 Sorted keys

We also tested the algorithms on sorted uniformly distributed random keys. We found that all the algorithms are faster or no slower with this input than with random keys and this is because they all gain from accessing successive locations in the count and destination arrays.

8.8 Permuting an array

We consider the problem of permuting n integers from a source array to a destination array as specified in an additional permutation array. The naive implementation sequentially visits each element of the source and permutation array and places the value from the source array to some random location in the destination array as specified by the permutation array. If the permutation is random and if $n/P \gg T$ virtually every access to the destination array will cause a TLB miss and similarly if $n/B \gg C$ virtually every access to the destination array will cause a cache miss.

We have tested two approaches to permuting, both of which work in a 'most-significant-bit first' fashion. Let k divide n for convenience, and define k equal-sized areas $D_1, \ldots, D_k$ in the destination array. All keys whose final destinations are in D_i are moved to *some* location in D_i in the first instance. We then recursively permute keys within D_i for $i = 1, \ldots, k$. The differences are in how the keys are moved to locations in D_i. One major disadvantage of both these approaches is that when we move an item to an intermediate location, we must move then the final destination of the item along with the item. This doubles the data to be moved. The improvements therefore are not as pronounced.

In each case we choose k to be $\min\{C/4, 4n/(CB)\}$, and distribute the keys into k areas. If the input size is small enough, then each D_i fits comfortably into cache and we finish the problem off with the naive algorithm, otherwise we recurse.

In each case, the distribution is similar to sorting the items according to the most significant $\log k$ bits of the destination address. In one case, we do this by moving items

Timings(sec)					
n	2^{20}	2^{21}	2^{22}	2^{22}	2^{24}
Naive	0.30	0.66	1.40	2.89	6.03
PS	0.35	0.70	1.40	2.85	5.77
EBT	0.25	0.53	1.13	2.32	4.74

Table 8.5: Running times for naive, pre-sorting (PS) and explicit block copy (EBT) permutation algorithms for permuting 32-bit integer items. The permutations were random.

to an intermediate buffer as in the explicit block copy algorithm, and in the other, we do it by locally pre-sorting the items in segments according to which area they need to be moved to, and then moving them all in one global permute step. We omit the TLB and cache analyses of these algorithms, which are essentially the same as those of LSB radix sort variants, and merely give the experimental results, which show that the explicit block copying algorithm performs quite well in this context (see Table 8.5).

8.9 Summary

We have shown the importance of minimising TLB misses in algorithms, even though it may require significantly more operations. We have used the general technique of explicit block transfer [80], which is the application for our emulation theorem for EMM algorithms, to minimises TLB and cache misses in radix sorting. We have also given two other techniques for reducing TLB misses in radix sorting, reducing working set size and pre-sorting.

We have given three radix sorting algorithms, pre-sorting LSB radix sort (PLSB radix sort), explicit block transfer LSB radix sort (EBT radix sort) and extended-radix PLSB radix sort (EPLSB radix sort), of which the first two have provably few cache and TLB misses, the last has provably few TLB misses.

The EBT technique is general, however, it does not always give the best practical performance. PLSB gives a fast running time and is robust. EPLSB has the advantage that it has fewer data moves than PLSB and EBT. The current implementation of EPLSB can in theory suffer worst-case cache misses, even though this is not observed in experimental results, since it does not randomise the start of destination pointers during the global permute as this would add too much of an overhead. However since the number of random locations accessed is $T \ll C$ it would be interesting to determine if the algorithm can be made robust by using less randomisation than suggested by [65].

We have shown the pre-sorting and explicit block transfer techniques applied fruitfully to related problems such as permutations, but here more work could be done, including for instance work on specialised classes of permutations. Also the effect of applying pre-sorting and explicit block transfer to other algorithms such as MSB radix sort should be studied.

On a more speculative note, since pre-sorting and explicit block transfer allow data to be moved in blocks we could further exploit the features available on most modern machine architectures, such as blocked copy operations between main memory addresses which by-pass the cache. Also since the local sorts in PLSB radix sort can be performed independently, on a machine with large *instruction-level parallelism* we could hope to speed that phase up by performing two or more local sorts simultaneously (by fusing the loops, for example).

Chapter 9

Optimising predecessor searches in the IMM

In this chapter we demonstrate the importance of reducing both cache and TLB misses for obtaining good performance in search trees. We use the two general techniques for simultaneously reducing cache and TLB misses that we introduced in Chapter 3: simulating 3-EMM algorithms and cache-oblivious algorithms. We give experimental results which demonstrate that data structures based on these ideas outperform data structures which are based on minimising cache misses alone, namely B-tree variants.

More precisely the problem we consider is the *dynamic predecessor* problem, which involves maintaining a set S of pairs $\langle x, i\rangle$ where x is a key drawn from a totally ordered universe and i is some satellite data. Without loss of generality we assume that i is just a pointer. We assume that keys are fixed-size, and for now we assume that each pair in S has a unique key. The operations permitted on the set include insertion and deletion of $\langle$key, satellite data$\rangle$ pairs, and predecessor searching, which takes a key q and returns a pair $\langle x, i\rangle \in S$ such that x is the largest key in S satisfying $x \leq q$. For this problem the B-tree data structure makes an optimal $O(\log_B n)$ cache misses and executes $O(\log n)$ instructions for all operations, where $|S| = n$ [31, 53]. However, a B-tree also makes $\Omega(\log_B n)$ TLB misses, which is not optimal.

Assuming the maximum branching factor of a node is selected such that a node fits inside a cache block, a heuristic argument suggests that for sufficiently large n, the number of cache and TLB misses for the B-tree or B*-tree on random data should be about $\log_B n - \log_B C + O(1)$ and $\log_B n - \log_B T + O(1)$ respectively. This is because one may expect roughly the top $\log_B C$ levels of the B-tree to be in cache, and address translations for all nodes in the top $\log_B T$ levels to be in the TLB. A reasonable assumption for internal memory is that $C \geq \sqrt{n}$ and that $\log_B T$ term may be ignored. This gives us a rough total of $\log_B n$ TLB misses and at most $0.5 \log_B n$ cache misses. In other words, one would expect TLB misses to dominate cache misses.

In this chapter we consider the practical performance of three implementations. Firstly, we consider the B*-tree [31], a B-tree variant that gives a shallower tree than the original. We observe that by *paginating* a B*-tree we get much better performance. Pagination consists of placing as much of the top of the tree into a single page of memory as possible, and then recursing on the roots of the sub-trees that remain when the top is removed. As the only significant difference between the paged and original B*-trees is the TLB behaviour, we conclude that TLB misses do significantly change the running time performance of B*-trees. Unfortunately, we do not yet have a satisfactory practical approach for maintaining paged B-trees under insertions and deletions.

The second is an implementation of a data structure which uses the standard idea for 3-EMM. At the top level the data structure is a B-tree where each node fits in a page. Inside a node, the 'splitter' keys are stored in a B-tree as well, each of whose nodes fits in a cache block. This data structure makes $O(\log_B n)$ cache misses and $O(\log_P n)$ TLB misses for all operations (the update cost is amortised).

Our last implementation is of a cache-oblivious search tree data structure which makes an optimal number of misses between any two memory levels. So this data structure makes $O(\log_B n)$ cache misses and $O(\log_P n)$ TLB misses for all operations.

Our experimental results suggest that the relative performance is generally as follows:

$$\text{B}^* < \text{Cache-oblivious} < \text{3-EMM} < \text{Paged B}^*.$$

It is noteworthy that a non-trivial data structure such as the cache-oblivious tree can outperform a simple, proven *and* cache-optimised data structure such as the B* tree.

9.1 Data structures

We now describe the three data structures in more detail, starting with the cache-oblivious search tree, then the paged B*-tree and then the 3-EMM search tree. Below, the term 'I/O' refers to both TLB and cache misses.

Cache-oblivious search tree

Prokop [69] showed how to cache-obliviously lay out a complete binary search tree T with n keys so that on a fast memory with an unknown block size of B, searches take $O(\log_B n)$ I/Os. If h is the height of T, Prokop divides T at $\lfloor h/2 \rfloor$, which separates T into the top sub-tree T_0 of height $\lfloor h/2 \rfloor$ and $k \leq 2^{\lfloor h/2 \rfloor}$ sub-trees $T_1, \ldots, T_k$ of height $\lceil h/2 \rceil$. T is laid out recursively in memory as T_0 followed by $T_1, \ldots, T_k$, as shown in Figure 9.1. This tree layout is very similar to the recursive data structure first introduced by van Emde Boas [87]. The data structure is static and requires $O(n)$ time and $O(n/B)$ I/Os to construct. Bender et al. [20] note that Prokop's tree can be simply and efficiently dynamised by using *exponential* search trees [13]. Our algorithm is closely related to that of Bender et al. [20], which we briefly review now.

We first set out some terms we will use in the discussion below. An exponential search tree has non-leaf nodes of varying and sometimes quite large (non-constant) degree. A non-leaf node with k children stores $k - 1$ sorted keys that guide a search into the correct child: we will organise these $k - 1$ keys as a Prokop tree. In what follows, we refer to the exponential search tree as the *external tree*, and its nodes as *external nodes*. We refer to any of the Prokop trees in an external node as *internal trees* and to their nodes as *internal nodes*.

Roughly speaking, the root of an external tree with n leaves contains $\Theta(\sqrt{n})$ keys which partition the keys into sets of size $\Theta(\sqrt{n})$. [1] After this, we recurse on each set. In contrast to [20], we end the recursion when we reach sets of size $\Theta(\log n)$. These are then placed in *external leaf* nodes, which are represented as arrays. Although the keys inside external non-leaf nodes are organised as Prokop trees, we do not require external nodes to have any particular memory layout with respect to each other (see Figure 9.2).

[1]In reality a "bottom-up" definition is used, whereby all leaves of the external tree are at the same depth and an external node with height i has an ideal number of children that grows doubly exponentially with i. The root of the external tree may have much fewer children than its ideal.

The number of I/Os during searches, excluding searching in the external leaves, is at most $O(\log_B n + \log\log n)$. Searching in the external leaves requires $O((\log n)/B)$ I/Os, which is negligible. The data structure (excluding the external leaves) can be updated within the same I/O bounds as a search. Insertions of new keys take place at external leaves and if the size of an external leaf exceeds twice its ideal size of $\Theta(\log n)$ then it is split, inserting a new key into its parent external node. To maintain invariants as n changes, the data structure is rebuilt when the number of keys reaches n^2.

Implementation details. We have implemented two versions of our cache-oblivious data structure. In both versions the keys in an external leaf node are stored in a sorted array, and the description below is about the different structures of non-leaf external and internal nodes.

In the first version of the data structure internal nodes are simply the keys of the external node or they are pointers to external children node. In order to reach the left or right child of an internal non-leaf node we calculate its address in the known memory layout. An external node with n' keys has an internal tree of height $\lceil \log n' \rceil + 1$, the first $\lceil \log n' \rceil$ levels hold the keys and the last level holds pointers to external child nodes. Each internal tree is a complete binary search tree and if $n' + 1$ is not a power of two then infinite valued keys are added to the first $\lceil \log n' \rceil$ levels of the internal tree, and NULL pointers are added to the last level. All $\lceil \log n' \rceil + 1$ levels of the internal tree are laid out recursively in a manner similar to Prokop's scheme, thus reducing the number of I/Os in moving from the last level of keys in the internal tree to the pointers to the external children nodes.

In the second version of the data structure each internal node consists of a key and left and right pointers. An external node with n' keys has an internal tree of height $\lceil \log n' \rceil$. At height $h < \lceil \log n' \rceil$ the internal node pointers point to internal child nodes. At the last level of the tree internal node pointers point to external children nodes. As above, each internal node is a complete binary tree. This implementation has the disadvantage of requiring extra memory but has the advantage that it avoids the somewhat computationally expensive task of finding a child in the internal tree.

Since the insertion time in an external leaf is anyway $\Theta(\log n)$ (we insert a new key into an array), we reduce memory usage by ensuring that an external leaf with k keys is stored in a block of memory sufficient for $k + O(1)$ keys. Increasing this to $O(k)$

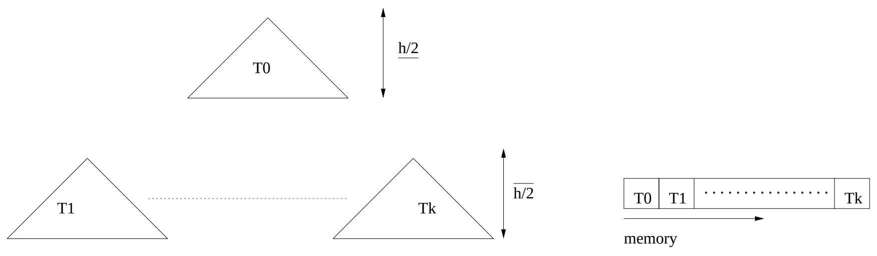

Figure 9.1: The recursive memory layout of a complete binary search tree according to Prokop's scheme

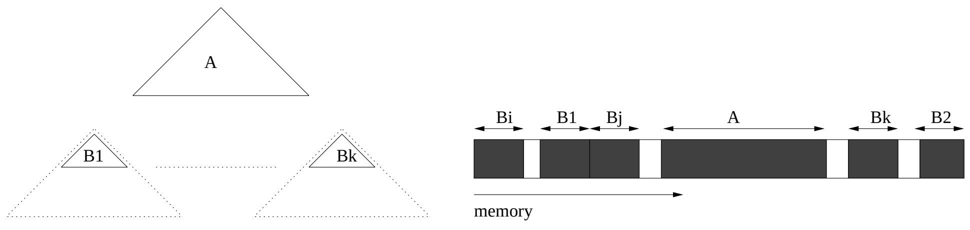

Figure 9.2: Memory layout of the cache-oblivious search tree with an external root node A of degree $\sqrt{n}$. The sub-trees of size $\Theta(\sqrt{n})$ rooted at A are shown in outline, and their roots are $B_1, \ldots, B_k$. The memory locations used by the sub-trees under $B_1, \ldots, B_k$ are not shown.

would improve insertion times, at the cost of increased memory. A final note is that the emulation of Theorem 3.3 of Section 3.3.2 in Chapter 3 is not required in this case to achieve optimal cache and TLB performance.

Paged B*-trees.

B*-trees are a variant of B-trees where the nodes remain at least 66% full by sharing with a sibling node the keys in a node which has exceeded its maximum size [31]. By packing more keys in a node, B*-trees are shallower than B-trees.

A *paged* B- or B*-tree uses the following memory layout of the nodes in the tree. Starting from the root and in a breadth-first manner as many nodes as possible are allocated from the same memory page. The sub-trees that remain outside the page are recursively organised in a similar manner (see Figure 9.3). The process of laying out a B-tree in this way is called *pagination*. The number of cache misses in a paged B-tree are roughly the same as an ordinary B-tree, but the number of TLB misses is sharply reduced from $\log_B n$ to about $\log_P n$. With our values of B and P the TLB misses are

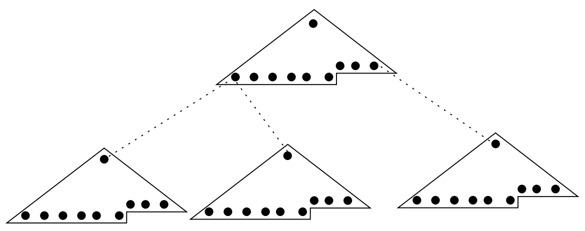

Figure 9.3: The nodes in the top part of the tree are in a memory page. The top parts of sub-trees that remain outside the page with the root are recursively placed on a memory page.

reduced by about two-thirds, and the overall number of misses is reduced by about a half. Unfortunately, we can only support update operations on a paged B-tree if the B-tree is weight-balanced [15, 93], but such trees seem to have poorer constant factors than ordinary B-trees.

Optimal 3-EMM Trees.

We now describe the implementation of a data structure that is optimised for 3-EMM. In principle, the idea is quite simple: we make a B-tree with branching factor $\Theta(P)$ and store the splitter keys inside a node in a B-tree with branching factor $\Theta(B)$. However, the need to make these nodes dynamic causes some problems. A B-tree with branching factor in the range $[d+1, 2d]$ and which has a total of m nodes can store (roughly) between md and $m \cdot (2d-1)$ keys. These correspond to trees with branching factor $d+1$ and $2d$ at all nodes, respectively. If we let m be the number of inner B-tree nodes which fit in a page, the *maximum* branching factor of the outer B-tree cannot exceed md, otherwise the number of nodes needed to store this tree may not fit in a page. Since each inner B-tree node takes at least $4d$ words ($2d$ keys and $2d$ pointers), we have that $m \leq P/(4d)$ and thus the maximum branching factor of the outer B-tree is at most $P/4$.

To make the tree more bushy, we simply rebuild the entire inner B-tree whenever we want to add an (outer) child to a node, at a cost of $\Theta(P)$ operations. When rebuilding, we make the inner B-trees have the maximum possible branching factor, thus increasing the maximum branching factor of the outer B-tree to about $P/2$. To compensate for the increased update time, we apply this algorithm only to the non-leaf nodes in the outer B-tree, and use standard B-tree update algorithms at the leaves. This again reduces the maximum branching factor at the leaves to $P/4$. At the leaves, however, the problem has a more serious aspect: the fullness of the leaves is roughly half of what it would be compared to a comparable standard B-tree. This roughly doubles the memory usage relative to a comparable standard B-tree.

Some of these constants may be reduced by using B*-trees in place of B-trees, but the problem remains significant. A number of techniques may be used to overcome this, including using an additional layer of buckets, overflow pages for the leaves etc, but these all have some associated disadvantages. In this thesis we have not fully investigated all possible approaches to this problem.

9.2 Experimental results

We have implemented B*-trees, paged B*-trees, optimal 3-EMM trees and our cache-oblivious search trees with and without internal pointers. Our data structures were coded in C++ and all code was compiled using `gcc 2.8.1` with optimisation level 6. Our experiments were performed on our Sun UltraSparc-II with 2×300 Mhz processors and 1GB main memory. This machine has 64 TLBs, 8KB pages, a 16KB L1 data cache, and a 512KB L2 cache; both caches are direct-mapped. The L1 cache miss penalty is about 2-3 cycles on this machine, and the L2 cache miss penalty is about 30 cycles. The TLB miss penalty is usually about 30-100 cycles.

In each experiment, a data structure was built on pairs of random 4-byte keys and 4-byte satellite data presented in random order. It should be noted that random data tends to degrade cache performance, so these tests are hard rather than easy. The B*-tree and cache-oblivious trees were built by repeated insertions, as essentially were the 3-EMM trees.

B*-trees and paged B*-trees were tested with branching factors of 7 and 8 keys per node (allowing 8 keys and 8 pointers to fit in one cache block of 64 bytes). Paged B*-trees were paginated for 8KB pages. For each data structure on n keys, we performed searches for $2n$ fresh keys drawn from the same distribution (thus "successful" searches are rare). These were either $2n$ independently generated keys, or $n/512$ independent keys, each of which was searched for 1024 times in succession. The latter minimises the effect on cache misses and thus estimates the average computation cost. For each algorithm, we have measured the average search time per query. Insertion costs are not reported. At each data point we report the average search time per query for 50 experiments.

Each data structure was tested with $n = 2^{18}, 2^{20}, 2^{22}$ and 2^{23} uniform integer keys in the range $[0, 2^{31})$. At each data point we report the average search time ("abs") and below it the average search time relative to the fastest average search time at that data point ("rel"). Table 9.1 shows results for the random queries and Table 9.2 shows results for the repeated random queries.

In Table 9.1 we see that paged B*-trees are by far the fastest for random queries. Random queries on B*-trees, tuned just for the cache, take between 44% and 68% more time than on paged B*-trees with a branching factor of 7. Optimal 3-EMM trees perform quite well, being at-most 14% slower than paged B*-trees. Of the data structure suited for both the cache and TLB, the cache-oblivious search trees are the slowest. On the other hand, the 3-EMM trees are carefully tuned to our machine, whereas the cache-oblivious trees have no machine-specific parameters.

As can be seen from Table 9.2, the better-performing data structures do not benefit from code tweaks to minimise operation costs. In fact, as they are a bit more complex, they actually have generally higher operation costs than B-trees, especially the cache-oblivious data structure with implicit pointers. Thus one would expect even better relative performance for the search trees suited to both the cache and TLB versus B*-trees on machines with higher miss penalties—the cache and TLB miss penalties are quite low on our machines.

It is also instructive to compare these times with classical search trees such as Red-Black trees. E.g., in an earlier study with the same machine/OS/compiler combination, Korda and Raman reported a time of 9.01 μs/search for $n = 2^{22}$ using LEDA 3.7 Red-Black trees [54, Fig 1(a)]. This is over twice as slow as B^*-trees and over thrice as slow as paged B^*-trees. LEDA (a, b) trees are a shade slower than our B^*-trees.

Time per-search (μs) on Sun UltraSparc-II 2 $\times$ 300 Mhz							
n	B* (BF=7)	B* (BF=8)	Paged B* (BF=7)	Paged B* (BF=8)	Optimal 3-EMM	Cache Obl. (int. ptrs)	Cache Obl. (no int. ptrs)
2^{18} abs	2.413	2.208	1.530	1.567	1.655	1.873	2.123
2^{18} rel	**1.577**	**1.443**	**1.000**	**1.024**	**1.082**	**1.224**	**1.388**
2^{20} abs	3.311	3.035	1.990	2.365	2.214	2.571	2.752
2^{20} rel	**1.664**	**1.525**	**1.000**	**1.188**	**1.113**	**1.292**	**1.383**
2^{22} abs	4.113	3.794	2.494	2.912	2.845	3.317	3.368
2^{22} rel	**1.649**	**1.521**	**1.000**	**1.168**	**1.141**	**1.330**	**1.350**
2^{23} abs	4.527	4.160	2.690	3.129	3.061	3.700	3.644
2^{23} rel	**1.683**	**1.546**	**1.000**	**1.163**	**1.138**	**1.375**	**1.355**

Table 9.1: Query times for $2n$ independent random queries on n keys in data structure.

Computation time per-search (μs) on Sun UltraSparc-II 2 $\times$ 300 Mhz machine							
n	B* (BF=7)	B* (BF=8)	Paged B* (BF=7)	Paged B* (BF=8)	Optimal 3-EMM	Cache Obl. (int. ptrs)	Cache Obl. (no int. ptrs)
2^{18} abs	0.796	0.765	0.747	0.757	0.895	0.892	1.206
2^{18} rel	**1.066**	**1.024**	**1.000**	**1.013**	**1.198**	**1.194**	**1.614**
2^{20} abs	0.890	0.864	0.867	0.860	1.038	0.948	1.305
2^{20} rel	**1.035**	**1.005**	**1.008**	**1.000**	**1.207**	**1.102**	**1.517**
2^{22} abs	0.971	0.956	0.965	0.948	1.140	1.012	1.368
2^{22} rel	**1.024**	**1.008**	**1.018**	**1.000**	**1.203**	**1.068**	**1.443**
2^{23} abs	0.992	0.972	0.981	0.965	1.158	1.044	1.396
2^{23} rel	**1.028**	**1.007**	**1.017**	**1.000**	**1.200**	**1.082**	**1.447**

Table 9.2: Query times for $n/512$ independent random queries repeated 1024 times each, on n keys in data structure.

9.3 Summary

Our experimental results show that optimising for three-level memories gives large performance gains in internal memory computations. In particular we have shown that cache-oblivious data structures may have significant practical importance. It would be useful to validate predictions of cache and TLB performance using simulations and to test the performance of these data structures on secondary memory.

The data structures in this chapter have mainly been tuned for searches. Update operations in our current implementation of the cache-oblivious data structures are quite slow, it would be interesting to improve upon this.

Chapter 10

Limitations of the CMM and IMM

In the previous chapters we have shown that the analyses in the models we have used reflect the actual running times for some sorting and searching algorithms. In this chapter we discuss reasons why the models we use may at times be imprecise.

10.1 Simple abstraction of reality

In Appendix B we discuss the designs of real caches on modern computer systems and we see that our model is a very simple abstraction of reality. In addition, other features related to the internal memory, such as registers, memory bandwidth and virtual memory may affect the cost of accessing data and the overall running time of a program. We will now discuss just some of the features in the memory systems of real computers that the models fail to capture.

10.1.1 Multi-level caches

We assume a single cache level whereas at least two levels of cache are fairly common place. Our analyses and designs are essentially for the last level of cache before the main memory, but as we see in Eq. B.2 and B.3 in Appendix B, the performance of an algorithm may be improved by caching at higher levels. This does not cause inaccuracies in most of our experiments as the sorting algorithms that we use write to a memory address almost immediately after any read to that address, and since the first-level cache on our Sun UltraSprac-II is write-through, every memory access results in an access for the same address in the second-level cache.

10.1.2 Replacement policies

Our models assume that the TLB or set-associative caches use an LRU replacement policy. In practice LRU is usually too expensive to use and most caches and TLB use a pseudo-LRU or random replacement policy. As far as the cache is concerned, since those on our machine are direct-mapped, this issue has not been a problem in our implementations.

10.1.3 Pre-fetching data

Hardware or compiler pre-fetching of data before it is needed is often used to reduce cache misses, and again this is not captured by our model. Both these features are supported on our system, but with most of our algorithms it is not possible to determine which data will be needed more than a few instructions before the data is accessed, so again this will not be apparent in the execution time of our algorithms.

10.1.4 Faster miss processing

Caches frequently use write-buffers, where the CPU is allowed to continue executing instructions while a write to main memory is completing under the control of the cache. Similarly some caches allow a CPU to continue executing instructions while the cache processes a read miss. With both these features if a program can continue executing instructions while cache misses are processed then the misses may not increase the overall running time. The L2 cache on our machine allows 3 outstanding read misses and 2 outstanding write misses, but in most of our algorithms the operands for most of the instructions following a cache miss are dependent on the missed data so we do observe the delays.

Since the CPU only needs one word at a time, some caches work on the principle of supplying to the CPU the requested word as soon as it is available from main memory, while continuing to load the whole memory block. Another type of cache asks the main memory for the word required by the CPU, and passes it to the CPU, before asking the memory for the rest of the block. Again this is not captured by our model. However since main memory has a high latency for delivering the first word, the effect of this optimisation may not be that significant unless the cache block size is quite large.

10.1.5 Cache-coherency

Cache-coherency is the problem of ensuring that the contents of all caches and the main memory are identical or under tight enough control such that no device which accesses data stored there confuses stale data for current data.

As an example of why this is required, suppose a device writes data to some main memory locations, say from the disk, and the CPU has cached the data at those locations then the CPU will not see the new value. *Direct memory access (DMA)* devices such as disk controllers or video controllers often write data directly to memory without the CPU's intervention.

Cache-coherency protocols are used to ensure that the CPU, cache, main memory and other devices communicate with each other and maintain coherent data. These protocols can often cause delays not addressed by our model.

For example, if a DMA device writes to main memory then a very simple method for maintaining coherency would be to flush the entire cache after the write. This can be done by setting some bit in the cache to indicate the data is invalid. This would require a cache refill after each DMA write. Clearly this is not modelled by the CMM and IMM. Appendix B discusses in some detail the cache-coherence problem and various solutions.

10.1.6 Memory bandwidth

The cache miss and TLB miss penalties tells us how long it takes to access the main memory, using this and the system clock speed, it seems, we should be able to determine the cache and/or TLB miss rates the system can support. However this is not true in practice. Often the bus connecting the main memory to the CPU has a lower bandwidth which can not sustain the required level of data throughput, and this is not captured by our model.

As an example, on our system the cache miss penalty is about 30 cycles and we assume the TLB miss penalty is usually about 30 cycles. The system speed is 300MHz. This suggests that the system can support 10M cache or TLB misses/sec.

Each cache miss transfers 64 bytes of data via the system bus so we would need to transfer 640MB/sec to support 10M cache misses/sec. However the memory bandwidth on our machine is only 200MB/sec.

We now do some very rough calculations to demonstrate the effect of the limited memory bandwidth on the performance of an algorithm. We can see this effect most clearly in algorithms which have a high number of cache misses, such as Flashsort1, discussed in Chapter 7. We will estimate the time for instructions and for memory access to estimate the running time for an algorithm.

As discussed in Chapter 7 one distribution pass of Flashsort1 and its variants has roughly 30 instructions per key. There are roughly $n/10$ insertion sort problems, each with roughly 10 random keys, since insertion sorting a random keys requires approximately $a^2/4$ swaps, we assume there are roughly 25 moves per insertion sort problem. By inspecting the assembler code generated on our machine we determine that each swap requires 7 instructions, so there are an average of 17.5 instruction per key for insertion sorting. When sorting $n = 2^{26}$ keys the instruction processing time for Flashsort1 is approximately:

$$n \times \text{Ops per key}/\text{CPU clock speed} = 2^{26} \times (30 + 17.5)/(300 \times 10^6) = 10.62 \text{ sec.}$$

We now estimate the time for memory accesses by considering the number of cache and TLB misses and their miss penalties. Figure 7.4 tells us that when sorting $n = 2^{26}$ keys there are about 3.2 cache misses per key (in the count phase each access to the count array causes a cache miss and in the permute phase each access to the count and data arrays cause cache misses). It is reasonable to assume that there will be about 3.0 TLB misses per key (there are fewer compulsory and capacity misses). When sorting $n = 2^{26}$ keys the memory access time, derived using the cache and TLB miss penalties is approximately:

$$\begin{aligned} & n \times ((\text{Cache misses per key} \times \text{Cache miss penalty }) + \\ & \qquad (\text{TLB misses per key} \times \text{TLB miss penalty }))/\text{CPU clock speed} \\ & = 2^{26} \times ((3.2 \times 30) + (3.0 \times 30))/(300 \times 10^6) = 41.6 \text{ sec.} \end{aligned}$$

So the estimated running time is approximately 52.22 seconds, but the actual running time is about 124 seconds, see Table 7.1.

If we consider the memory bandwidth then we get a clearer picture. We know that on a cache miss 64 bytes are transferred between the cache and main memory. Each TLB entry has 16 bytes of data, so we assume on a TLB miss 16 bytes are transferred between the TLB and main memory. When sorting $n = 2^{26}$ keys the memory access time, derived using the memory bandwidth is approximately:

$$\begin{aligned} & n \times ((\text{Cache misses per key} \times \text{Cache block size}\ ) + \\ & \qquad (\text{TLB misses per key} \times \text{TLB entry size}\ ))/\text{Memory bandwidth} \\ & = 2^{26} \times ((3.2 \times 64) + (3.0 \times 16))/(200 \times 10^6) = 84.8 \text{ sec.} \end{aligned}$$

Using the time for instructions and memory bandwidth limitations the estimated running time is approximately 95.42 seconds, much closer to the observed running time. Since the model used, even when taking memory bandwidth into account, is quite crude, it is difficult to get a more accurate estimate for the running time. For example, an obvious problem is that we do not consider any other traffic on the bus.

10.1.7 Physically mapped caches

Both the CMM and IMM fail to address the fact that, due to paging, consecutive (virtual) memory addresses are not consecutive in physical (main) memory. A system that implements paging usually allocates pages of physical memory using a *free list* of pages. On a heavily loaded system the pages in a free list are ordered by when they became free, and not by their physical memory addresses. This can lead to algorithm behaviour not predicted by the CMM or IMM. The following are some examples of unexpected behaviour we observed with the integer sorting algorithms discussed in Chapter 8:

(a) There is no real difference in the running times of LSB radix sort with a 5-bit radix when sorting the sequence $0, \dots, n-1$, where n is a power of 2, and when sorting random integers, see Figure 8.2. Sorting $0, \dots, n-1$ should have a significant impact on the running time, due to main memory blocks which map to conflicting cache blocks being accessed in a round-robin manner, see Proposition 8.3.

(b) PLSB radix sort is slightly slower when sorting the sequence $0, \dots, n-1$, where n is a power of 2, than when sorting random integers, see Figure 8.5. Our model predicts that there should be no observable difference in the running times, see the discussion in Section 8.5.1. Closer observation reveals that the running time for local sorts is affected by the input sequences. For example, one local sort pass over $n = 2^{25}$ random 32-bit unsigned integers takes 4.152 seconds and over the sequence $0, \ldots, n-1$ of 32-bit integers it takes 4.396 seconds.

(c) When dealing with random keys, the local sorts in PLSB radix sort using a 12-bit radix are about 13% slower than when using an 11-bit radix. The slowdown can not be explained by the cache misses required to reload a larger count array after each local sort. For example, one local sort pass over $n = 2^{25}$ random 32-bit unsigned integers using an 11-bit radix takes 4.152 seconds and using a 12-bit radix it takes 4.7 seconds.

We now offer some explanations for these observations.

(a) For LSB radix sort the input sequence $0, \ldots, n-1$ is designed to cause round-robin accesses to 2^r locations in the destination array that all map to a few conflicting cache blocks. On a system with paging, destination array locations which we intended to conflict in the cache may no longer do so, this might explain why LSB with a 5-bit radix is no slower given these inputs than given random keys. Note that the longer running times with this input sequence when using a larger radix may be explained by the fact that we can reasonably assume that the start of a page maps to one of only CB/P cache blocks and with a larger radix we increase the probability that active destination array locations do map to conflicting cache blocks.

(b) In PLSB radix sort, we have assumed that during the local sorts blocks of data in one segment of the destination data array do not conflict with each other in cache, however this assumption can be false on a system with paging. If the pages for a segment of the destination array map to conflicting locations in the cache, then with the input sequence $0, \ldots, n-1$ during the local permute the algorithm will make round-robin accesses to main memory blocks which map to conflicting cache blocks.

(c) The local sorts on random data in PLSB radix sort being slower than expected when using a 12-bit radix can also be explained if pages allocated to a segment of the destination array map to conflicting cache locations. With a larger radix there are a higher number of active locations in conflicting cache blocks, and we know from the analysis in Chapters 5 and 6 that an increase in the number of active locations during the permute phase increases the number of cache misses.

We were also able to demonstrate the affect of physically mapped caches on EPLSB radix sort. Our experiments showed that if at the start of the algorithms we accessed (touched) every P^{th} element of the data arrays then the algorithms ran a few % faster. Note that the running times reported in Chapter 8 do not use this touching trick.

10.2 Summary

In this chapter we have shown that the models used in this thesis do not address many of the features of the memory in modern computer systems. We have discussed in some detail the effects of the limited bandwidth of the bus connecting the main memory to the CPU and also of paging. We hope this will lead to the development of more powerful models which address some of the issues raised here.

Chapter 11

Conclusions and future work

Since the 1990's the algorithm design community has been aware of the need to improve cache performance in order to obtain fast algorithms. However the community has on the whole ignored the TLB, which stores translations for recently accessed memory pages. In this thesis, we have shown that TLB misses can have as much or more of an impact on the running time of an algorithm as cache misses.

Until now most internal memory algorithms have been analysed on a model which considers just the cache and main memory. We have given a model which considers the TLB as well as cache and main memory. In order to leverage the large amount of existing research on the design of memory hierarchy efficient algorithms, we have given two emulation theorems which allow algorithms and analyses from other memory models to be converted to equivalent algorithms and analyses on our model for the internal memory hierarchy. We have also given two general techniques to obtain algorithms and data structures which are simultaneously optimal for the cache and TLB. One of our techniques is to use optimal cache-oblivious algorithms.

In this thesis we note that distribution sorting algorithms analysed and designed on the RAM model make poor utilisation of the cache. We have extensively studied the cache misses during the permute phase of distribution sorting algorithms. We have shown that the number of cache conflict misses is dependent on the number of classes that the algorithm uses. We have given upper and lower bounds on the number of cache misses during the permute phase when the keys are drawn independently and randomly from a uniform distribution and when they are drawn independently and randomly from non-uniform distributions. Our analyses are for both an in-place and an out-of-place permute. We have given upper and lower bounds on the number of cache misses when accessing multiple sequences of data 'concurrently', and our bounds improve upon previous results. We have also given a simple approximate analysis for the permute phase of distribution sorting when the keys are drawn independently and randomly from a uniform distribution. Using our analyses we have developed fast new distribution sorting algorithms for sorting both random floating-point numbers independently drawn from uniform distributions and for sorting random floating-point numbers independently drawn from non-uniform distributions. Our new algorithms comfortably outperform cache-tuned Quicksort and Mergesort. We have shown the importance of considering the cost of instruction processing as well as cache misses when designing an internal memory algorithm which is fast in practice.

We have demonstrated the importance of reducing TLB misses as well as cache misses in the context of integer sorting. We have given three techniques to simultaneously reduce cache and TLB misses in integer sorting. Using these techniques we have obtained several new integer sorting algorithms which all outperform highly optimised Quicksort and Mergesort when tested on random integers independently drawn from a uniform distribution. The fastest of these algorithms is over twice as fast as highly optimised Quicksort and 2.8 times as fast as highly optimised Mergesort. We have shown that in order to reduce TLB misses and obtain a fast algorithm we may have to allow cache misses and instruction counts to grow beyond their optimal values. We have shown that the standard integer sorting algorithm designed on the RAM model can have a very large number of cache and TLB misses in the worst-case. We have given examples of input key sequences which cause this worst-case and we have shown that one of our fastest new integer sorting algorithms does not suffer this worst-case.

We have also demonstrated the importance of reducing TLB misses and cache misses in the context of searching. We have considered the dynamic predecessor problem and we note that the B-tree data structure is not simultaneously optimal for both the cache and TLB. We have designed and implemented new search trees data structures which are simultaneously optimal for the cache and TLB. One of our data structures is derived from a 3-EMM search tree and the other from a cache-oblivious search tree. We have shown that the cache-tuned B-tree search tree is outperformed by our cache and TLB tuned search trees, even when the latter are quite complex.

In the summary of appropriate chapters in the thesis we have suggested directions for future research. In Chapter 10 we have considered in some detail the limitations of the memory models used in this thesis and we hope this will lead to the design of more accurate but still practical memory models. We now consider a few broader issues for memory-efficient algorithms.

In this thesis we showed that, cache-oblivious search trees algorithms were outperformed by cache and TLB tuned search trees. This was due to higher computation and memory access costs. It would be interesting to see if this is true for other algorithms and data structures in general.

It is obvious that optimal cache-oblivious algorithms are optimal on the EMM. We note some further possible advantages of cache-oblivious algorithms for external memory algorithms. Data is stored on disks in concentric tracks. To read or write data, the read/write head must mechanically seek to the correct track and then wait for the required address to pass by. This can take the equivalent of a million CPU cycles. Clearly an EMM algorithm that makes accesses to data stored at sequential blocks on a track or on sequential tracks on disk will perform better than an algorithm that accesses random blocks on the disk. However, the EMM does not differentiate between accesses to sequential and random blocks. Since cache-oblivious algorithms exploit locality at all levels of memory, they will have the desired sequential data accesses and so give good performance. We believe this should be investigated further.

Our experimental results for Flashsort1, MPFlashsort and Quicksort on the Intel Mobile Pentium III system show that performance gains seen on the Ultra Sparc-II system may not so easily be seen on other systems. It would be interesting to test the algorithms presented in this thesis on a wider range of computer systems.

We finally suggest investigating the affect on power consumption of memory-efficient algorithms and data structures. This is becoming an increasingly important issues as handheld computing devices become more pervasive

Bibliography

[1] A. Acharya, H. Zhu and K. Shen. Adaptive algorithms for cache-efficient trie search. In *Proc. 1st Workshop on Algorithm Engineering and Experimentation*, LNCS 1619, pp. 296–311, 1999.

[2] A. Agarwal. Analysis of cache performance for operating systems and multiprogramming. Ph.D. Thesis. Stanford University, 1987. Also available as Tech. Rep. CSL-TR-87-332.

[3] A. Agarwal, R. L. Sites and M. Horowitz. ATUM: A new technique for capturing address traces using microcode. In *Proc. 13th Annual Symposium on Computer Architecture*, pp. 119–127, 1986.

[4] A. Agarwal, M. Horowitz and J. Hennessy. An analytical cache model. *ACM Transactions on Computer Systems* **7**, pp. 184–215, 1989.

[5] R. Agarwal. A super scalar sort algorithm for RISC processors. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 240–246, 1996.

[6] A. Aggarwal, B. Alpern, A. K. Chandra and M. Snir. A model for hierarchical memory. In *Proc. 19th Annual ACM Symposium on Theory of Computing*, pp. 305–314, 1987.

[7] A. Aggarwal, A. K. Chandra and M. Snir. Hierarchical memory with block transfers. In *Proc. 28th Annual Symposium on Foundations of Computer Science*, pp. 204–215, 1987.

[8] A. Aggarwal and J. S. Vitter. The I/O complexity of sorting and related problems. *Communications of the ACM* **31**, pp. 1116–1127, 1998.

[9] A. V. Aho, J. E. Hopcroft and J. D. Ullman. *The design and analysis of computer algorithms.* Addison-Wesley, 1974.

[10] A. Ailamaki, D. J. DeWitt, M. D. Hill and D. A. Wood. DBMSs on a modern processor: where does time go? In *Proc. 25th International Conference on Very Large Databases*, pp. 266–277, 1999.

[11] B. Alpern, L. Carter, E. Feig and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica* **12**, pp. 72–109, 1994.

[12] D. Alpert. Performance tradeoffs for microprocessor cache memories. Computer Systems Lab. Rep. CSL-TR 83-239, Stanford University, Dec. 1983.

[13] A. Andersson. Faster deterministic sorting and searching in linear space. In *Proc. 37th Annual Symposium on Foundations of Computer Science*, pp. 135–141, 1996.

[14] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proc. 34th Annual ACM Symposium on Theory of Computing*, pp. 268–276, 2002.

[15] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory (extended abstract). In *Proc. 37th Annual Symposium on Foundations of Computer Science*, pp. 560–569, 1996.

[16] L. Arge, J. Chase, J. S. Vitter and R. Wickremesinghe. Efficient sorting using registers and caches. In *Proc. 4th Workshop on Algorithm Engineering*, LNCS 1982, pp. 51–62, 2000.

[17] L. Arge. External memory data structures (invited paper). In *Proc. 9th Annual European Symposium on Algorithms*, LNCS 2161, pp. 1–29, 2001.

[18] C. Armen. Bounds on the separation of two parallel disk models. In *Proc. 4th Workshop on Input/Output in Parallel and Distributed Systems*, pp 122–127, 1996.

[19] D. H. Bailey. FFT's in external or hierarchical memory. *Journal of Supercomputing* **4**, pp. 23–35, 1990.

[20] M. Bender, R. Cole, and R.Raman. Exponential structures for cache-oblivious algorithms. In *Proc. 29th International Colloquium on Automata, Languages and Programming*, pp. 195–207, 2002.

[21] M. A. Bender, E. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proc. 41st Annual Symposium on Foundations of Computer Science*, pp. 399–409, 2000.

[22] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 29–38, 2002.

[23] G. Bilardi, P. D'Alberto and A. Nicolau. Fractal matrix multiplication: a case study on portability of cache performance. In *Proc. 5th International Workshop on Algorithm Engineering*, pp. 26–38, 2001.

[24] J. Bojesen, J. Katajainen and M. Spork. Performance engineering case study: heap construction. *ACM Journal of Experimental Algorithmics* **5**, Article 15, 2000.

[25] G. S. Brodal, R. Fagerberg and R. Jacob. Cache-oblivious search trees via binary trees of small height. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 39–48, 2002.

[26] B. Calder, C. Krintz, S. John and T. Austin. Cache-conscious data placement. In *Proc. 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 139–149, 1998.

[27] L. Carter and K. S. Gatlin. Towards an optimal bit-reversal permutation program. In *Proc. 39th Annual Symposium on Foundations of Computer Science*, pp. 544–555, 1998.

[28] S.Chatterjee and S. Sen. Cache-efficient matrix transposition. In *Proc. 6th International Symposium on High-Performance Computer Architecture*, pp. 195–205, 2000.

[29] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 13–24, 1999.

[30] T. M. Chilimbi, M. Hill and J. R. Larus. Cache-conscious structure layout. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1–12, 1999.

[31] D. Comer. The ubiquitous B-Tree. *ACM Computing Survey* **11**, pp. 121–137, 1979.

[32] T. H. Cormen, C. E. Leiserson and R. L. Rivest. *Introduction to algorithms.* MIT Press, 1990.

[33] D. Dubhashi and D. Ranjan. Balls and bins: a study in negative dependence. *Random Structures and Algorithms* **13**, pp. 99–124, 1998.

[34] N. Eiron, M. Rodeh and I. Steinwarts. Matrix multiplication: a case study of enhanced data cache utilization. *ACM Journal of Experimental Algorithmics* **4**, Article 3, 1999.

[35] R. Fadel, K. V. Jakobsen, J. Katajainen and J. Teuhola. Heaps and heapsort on secondary storage. *Theoretical Computer Science* **220**, pp. 345–362, 1999.

[36] R. W. Floyd. Algorithm 245: Treesort 3. *Communications of the ACM* **7**, 1964.

[37] E. H. Friend. Sorting on electronic computer systems. *Journal of the ACM* **3**, pp. 134–168, 1956.

[38] M. Frigo, C. E. Leiserson, H. Prokop, S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th Annual Symposium on Foundations of Computer Science*, pp. 285–298, 1999.

[39] S. B. Furber. *ARM system-on-chip architecture, 2nd ed..* Addison-Wesley Professional, 2000.

[40] D. Gannon and W. Jalby. The influence of memory hierarchy on algorithm organization: programming FFTs on a vector multiprocessor. In L. H. Jamieson, D. B. Gannon and R. J. Douglass Eds., *The Characteristics of Parallel Algorithms.* MIT Press, 1987.

[41] S. Ghosh, M. Martonosi and S. Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proc. Architectural Support for Programming Languages and Operating Systems*, pp. 228–239, 1998.

[42] J. R. Goodman. Using cache misses to reduce processor-memory traffic. In *Proc. 10th Annual Symposium on Computer Architecture*, pp. 124–131, 1983.

[43] J. Handy. *The cache memory handbook*, Academic Press, 1998.

[44] P. J. Hanlon, D. Chung, S. Chatterjee, D. Genius, A. R. Lebeck and E. Parker. The combinatorics of cache misses during matrix multiplication. *Journal of Computer and Systems Sciences* **63**, pp. 80–126, 2001.

[45] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach, 2nd ed.*. Morgan Kaufmann, 1996.

[46] M. Hill. Aspects of cache memory and instruction buffer performance. Ph.D. Thesis, 1987. University of California at Berkeley. Also available as Tech. Rep. UCB/CSD 87/381.

[47] M. Hill. A case for direct-mapped caches. *Computer* **21**, pp. 25–40, 1988.

[48] M. Hill and A. J. Smith. Experimental evaluation of on-chip microprocessor cache memories. In *Proc. 11th Annual Symposium on Computer Architectures*, pp. 158–166, 1984.

[49] M. Hill and A. J. Smith. Evaluating associativity in CPU caches. In *IEEE Transactions on Computers* **38**, pp. 1612–1630, 1989.

[50] K. Joag-Dev and F. Proschan. Negative association of random variables, with applications. *Annals of Statistics* **11**, pp. 286–295, 1983.

[51] R. Joseph, D. Brooks and M. Martonosi. Live, runtime power measurements as a foundation for evaluating power/performance tradeoffs. In *Proc. Workshop on Complexity Effective Design, held in conjunction with ISCA-28*, 2001.

[52] S. Kaxiras, Z. Hu and M. Martonosi. Cache decay: exploiting generational behaviour to reduce cache leakage power. In *Proc 28th International Symposium on Computer Architecture*, pp. 240–251, 2001.

[53] D. E. Knuth. *The art of computer programming. Volume 3: sorting and searching, 3rd ed.*. Addison-Wesley, 1997.

[54] M. Korda and R. Raman. An experimental evaluation of hybrid data structures for searching. In *Proc. 3rd Workshop on Algorithm Engineering*, LNCS 1668, pp. 213–227, 1999.

[55] P. Kumar. *Personal communication.*

[56] R. E. Ladner, J. D. Fix and A. LaMarca. Cache performance analysis of traversals and random accesses. In *Proc. 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 613–622, 1999.

[57] M. S. Lam, E. E. Rothberg and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proc. 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 63–74, 1991.

[58] A. LaMarca and R. E. Ladner. Influence of caches on the performance of heaps. *ACM Journal of Experimental Algorithmics* **1**, Article 4, 1996.

[59] A. LaMarca and R. E. Ladner. The influence of caches on the performance of sorting. *Journal of Algorithms* **31**, pp. 66–104, 1999.

[60] A. Lebeck and D. Wood. Cache profiling and the SPEC benchmarks: a case study. *Computer* **27**, pp. 15–26, 1994.

[61] M. Martonosi, A. Gupta and T. Anderson. Tuning memory performance of sequential and parallel programs. *Computer* **28**, pp.32–40, 1995.

[62] Y. Matias, E. Segal and J. S. Vitter. Efficient bundle sorting. In *Proc. 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 839–848, 2000.

[63] R. L. Mattson, J. Gecsei, D. R. Slutz and I. L. Traiger. Evaluation techniques for storage hierarchies. IBM Systems Journal **7**, pp. 78–117, 1970.

[64] C. Mead and L. Conway. *Introduction to VLSI systems.* Addison-Wesley, 1980.

[65] K. Mehlhorn and P. Sanders. Accessing multiple sequences through set-associative cache. Unpublished manuscript, 2000. Preliminary version in *Proc. 26th Annual*

International Colloquium on Automata, Languages and Programming, LNCS 1555, pp. 655-664, 1999.

[66] B. Moret and H. Shapiro. An empirical assessment of algorithms for constructing a minimum spanning tree. In *Computational Support for Discrete Mathematics*, Volume 15 of *DIMACS Monographs in Discrete Mathematics and Theoretical Computer Science*, 1994, pp. 99–117. American Mathematical Society.

[67] K. D. Neubert. The Flashsort1 algorithm. *Dr Dobb's Journal*, pp. 123–125, February 1998. FORTRAN code listing, p. 131, *ibid.*

[68] C. Nyberg, T. Barclay, Z. Cventanovic, J. Gray and D. Lomet. AlphaSort: a RISC machine sort. In *Proc. ACM SIGMOD International Conference on Management of Data*, 1994.

[69] H. Prokop. Cache-oblivious algorithms. *MS Thesis. MIT Department of Electrical Engineering and Computer Science.* 1999.

[70] N. Rahman, R. Cole and R. Raman. Optimising predecessor data structures for internal memory. In *Proc. 5th International Workshop on Algorithm Engineering*, LNCS 2141, pp. 67–78, 2001.

[71] N. Rahman and R. Raman. Analysing cache effects in distribution sorting. *ACM Journal of Experimental Algorithmics* **5**, Article 14, 2001. Preliminary version in *Proc. 3rd Workshop on Algorithm Engineering*, LNCS 1668, pp. 184–198, 1999.

[72] N. Rahman and R. Raman. Adapting radix sort to the memory hierarchy. *ACM Journal of Experimental Algorithmics*, (to appear). Preliminary version in *Proc. 2nd Workshop on Algorithm Engineering and Experiments*, 2000.

[73] N. Rahman and R. Raman. Analysing the cache behaviour of non-uniform distribution sorting algorithms. In *Proc. 8th Annual European Symposium on Algorithms*, LNCS 1879, pp. 380–391, 2000.

[74] T. H. Romer, W. H. Ohlrich, A. R. Karlin and B. N. Bershad. Reducing TLB and memory overhead using online superpage promotion. In *Proc. 22nd Annual International Symposium on Computer Architecture*, pp. 176–187, 1995.

[75] W. Rudin. *Real & complex analysis.* McGraw-Hill, New York, 1974.

[76] P. Sanders. Random permutations on distributed, external and hierarchical memory. *Information Processing Letters* **67**, pp. 305-309, 1998.

[77] P. Sanders. Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics* **5**, Article 7, 2000.

[78] P. Sanders, S. Egner and J. Korst. Fast concurrent access to parallel disks. In *Proc. 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 849–858, 2000.

[79] R. Sedgewick. Implementing quicksort programs. *Communications of the ACM* **21**, 1978.

[80] S. Sen and S. Chatterjee. Towards a theory of cache-efficient algorithms (extended abstract). In *Proc. 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 829–838, 2000.

[81] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM* **28**, pp. 202–208, 1985.

[82] A. J. Smith. Cache evaluation and the impact of workload choice. In *Proc. 12th Annual Symposium on Computer Architectures*, pp. 63–73, 1985.

[83] A. J. Smith and J. R. Goodman. A study of instruction cache organization and replacement policies. In *Proc. 10th Annual Symposium on Computer Architectures*, pp. 132–137, 1983.

[84] O. Temam, C. Fricker and W. Jalby. Cache awareness in blocking techniques. *Journal of Programming Languages*, 1997.

[85] S. Thite. Optimum binary search trees on the hierarchical memory model. M.S. Thesis. University of Illinois at Urbana-Champaign, 2000. Also available as CSL Tech. Rep. UILU-ENG-00-2215 ACT-142.

[86] *UltraSPARC user's manual.* Sun Microsystems, 1997.

[87] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters* **6**, pp. 80-82, 1977.

[88] G. Venkataraman, S. Sahni and S. Mukhopadhyaya. A blocked all-pairs shortest-paths algorithm. In *Proc. Scandinavian Workshop on Algorithm Theory*, LNCS 1851, pp. 419–432, 2000.

[89] J. S. Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys* **33**, pp. 209–271, 2001.

[90] J. S. Vitter and M. H. Nodine. Large scale sorting in uniform memory hierarchies. *Journal of Parallel and Distributed Computing* **17**, pp. 107–114, 1993.

[91] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: two-level memories. *Algorithmica* **12**, pp. 110–147, 1994, double special issue on Large-Scale Memories, J. S. Vitter, guest editor.

[92] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory II: hierarchical multilevel memories. *Algorithmica* **12**, pp. 148–169, 1994, double special issue on Large-Scale Memories, J. S. Vitter, guest editor.

[93] D. E. Willard. Reduced memory space for multi-dimensional search trees. In *Proc. 2nd Annual Symposium of Theoretical Aspects of Computer Science*, pp. 363–374, 1985.

[94] W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM* **7**, pp. 347–348, 1964.

[95] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proc. ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pp. 30–44, 1991.

[96] L. Xiao, X. Zhang and S. A. Kubricht. Improving memory performance of sorting algorithms. *ACM Journal on Experimental Algorithmics* **5**, Article 3, pp. 1-22, 2000.

[97] H. Zhang and M. Martonosi. A mathematical cache miss analysis for pointer data structures. In *Proc. 10th SIAM Conference on Parallel Processing for Scientific Computing*, 2001.

[98] Z. Zhang and X. Zhang. Cache-optimal methods for bit-reversals. In *Proc. Supercomputing'99*, Article 26, 1999.

Appendix A

Simulation results

In this appendix we give simulation results for the cache misses per key during the permute phase of distribution sorting. We compare our simulations results to values predicted using the approximate analyses in Chapter 6.

Table A.1 shows cache misses per key during the permute phase for $n = 2^{18}, 2^{19}$, 2^{20}, 2^{21}, 2^{22}, 2^{23}, 2^{24}, 2^{25} and $n = 2^{26}$ at expected class sizes of $B/4$,$B/2$ and B, where $B = 16$. The misses per key were obtained:

- from 50 cache simulations, labelled *Sim*,
- predicted using Equations 6.7 and 6.11, labelled *Predict*.

We see that the predicted and simulated miss rates are quite close, only with $n/k = B/4$ and values of $n \geq 2^{22}$ is there a greater than 5% variation, even then the variation is less than 7%.

Tables A.2 and A.3 shows cache misses per key during the 'in-place' permutation phase for $n = 2^{24}$ and 100 random values of k between 25 and 8192 and for selected values of k = 128, 256, 384, 512, 640, 768, 896, 1024, 1152, 1280, 1408, 1536, 1664, 1792, 1920, 2047, 2048, 2049, 4095, 4096 and 4097. The misses per key were obtained:

- from 20 cache simulations, labelled *Sim*,
- predicted from the actual values of ν taken at the start of permutation in the full simulations above, labelled *Act*,
- predicted using Equation 6.16, labelled *Pred.*

	$n/k = B/4$		$n/k = B/2$		$n/k = B$	
n	Sim	Predict	Sim	Predict	Sim	Predict
2^{18}	1.052	1.080	0.832	0.837	0.669	0.694
2^{19}	1.415	1.471	1.188	1.197	0.998	1.012
2^{20}	1.722	1.745	1.455	1.471	1.252	1.255
2^{21}	1.877	1.929	1.735	1.686	1.474	1.471
2^{22}	1.954	2.042	1.875	1.841	1.741	1.657
2^{23}	1.993	2.106	1.945	1.939	1.874	1.797
2^{24}	2.012	2.132	1.981	1.996	1.941	1.888
2^{25}	2.022	2.158	1.998	2.026	1.974	1.941
2^{26}	2.026	2.167	2.007	2.042	1.991	1.970

Table A.1: Simulated miss rates and miss rates predicted using Equation 6.7 and 6.11, at $n = 2^{18}, 2^{19}$, 2^{20}, 2^{21}, 2^{22}, 2^{23}, 2^{24}, 2^{25} and $n = 2^{26}$ with expected class sizes of $B/4, B/2$ and B, where $B = 16$.

For a value of k where there is greater than 5% variation between these three values, k is prefixed with a superscript of a, b or c. The superscripts denote variations:

a) between simulated and predicted values alone,

b) between simulated and predicted and between actual and predicted values,

c) between all three values.

Some observations on the results are:

- For most random values of k the variation between the simulated and predicted misses per key is at-most 5%.
- For most values of k where there is greater than 5% variation, the superscript is b. This indicates that simulated misses per key and misses per key predicted from actual ν vary together.
- For most random values of k where there is greater than 5% variation we note that $\gcd(\tau, k)$ is small but $\gcd(\tau, k-1)$ or $\gcd(\tau, k+1)$ is large. These are examples of heuristic 1 not being effective, as discussed in Section 4.4.2.

Note that greater than 5% variation between other combinations are not observed, for instance there was never a more than 5% variation between the actual and predicted values alone.

Tables A.4 and A.5 show for $n = 2^{20}$ and 2^{22} all values of k between 32 and 8192 where there was greater than 5% variation between the actual value of ν at the start of permutation, averaged over 20 experiments, and ν predicted using Equation 6.14. Table A.6 shows these results at $n = 2^{24}$ where the actual value of ν was averaged over 195 experiments. Values of k where $\gcd(\tau, k)$ is not equal to τ, $\tau/2$, $\tau/4$, or k are prefixed with a *. Some observations on the results are:

- Values which are prefixed with * have $\gcd(\tau, k-1) = \tau$ or $\gcd(\tau, k+1) = \tau$. These are further examples of heuristic 1 not being effective.
- As k increases fewer and fewer values are listed in the table, this shows that the equations will be accurate for most values of $k > 32$ while $n/k \gg B$.

We can report that at $n = 2^{24}$ for 6676 values of k between 32 and 8192, the actual value of ν averaged over 195 experiments was higher than the value predicted using Equation 6.14.

Table A.7 shows for $n = 2^{24}$ at values of k selected using heuristic 2, actual ν at the start of permutation, averaged over 20 experiments, and ν predicted using Equation 6.14. We note that at all these values of k the actual value of ν was higher than the predicted value.

k	Sim	Act	Pred	k	Sim	Act	Pred	k	Sim	Act	Pred
49	0.065	0.065	0.066	[b]128	0.301	0.290	0.072	[b]129	0.065	0.065	0.072
210	0.076	0.077	0.077	[b]240	0.094	0.093	0.080	[b]253	0.076	0.076	0.080
[b]256	0.297	0.291	0.081	[b]384	0.287	0.293	0.090	[2]472	0.094	0.090	0.096
[b]512	0.300	0.292	0.098	554	0.103	0.102	0.101	615	0.106	0.104	0.105
[b]640	0.290	0.294	0.107	662	0.109	0.109	0.109	[b]768	0.292	0.297	0.116
781	0.116	0.113	0.117	789	0.121	0.118	0.117	[b]896	0.293	0.298	0.124
930	0.125	0.124	0.127	[b]1024	0.295	0.298	0.133	1038	0.133	0.130	0.134
1105	0.137	0.136	0.138	[a]1120	0.147	0.145	0.139	1123	0.140	0.135	0.139
1137	0.140	0.140	0.140	[b]1152	0.296	0.302	0.141	1178	0.147	0.142	0.143
1267	0.148	0.145	0.149	[b]1280	0.298	0.301	0.150	1318	0.151	0.150	0.152
[b]1408	0.299	0.306	0.158	[b]1536	0.301	0.306	0.166	[b]1664	0.302	0.309	0.174
[b]1792	0.304	0.311	0.182	1888	0.192	0.189	0.188	1910	0.188	0.185	0.189
[b]1920	0.306	0.312	0.190	2042	0.201	0.195	0.198	[b]2047	0.245	0.239	0.198
[b]2048	0.320	0.313	0.198	[b]2049	0.241	0.240	0.198	2060	0.198	0.192	0.199
2164	0.204	0.198	0.205	[b]2239	0.228	0.221	0.210	2293	0.212	0.206	0.213
2364	0.221	0.216	0.217	2420	0.220	0.215	0.220	2456	0.223	0.218	0.222
2500	0.230	0.222	0.225	2545	0.227	0.222	0.228	2582	0.231	0.225	0.230
2596	0.233	0.227	0.231	2618	0.234	0.227	0.232	2629	0.236	0.230	0.233
2693	0.246	0.241	0.236	2842	0.246	0.242	0.245	2885	0.251	0.245	0.247
2937	0.256	0.250	0.250	[c]2945	0.304	0.286	0.251	3266	0.280	0.269	0.269
3368	0.279	0.273	0.275	3658	0.295	0.287	0.291	3662	0.295	0.286	0.291

Table A.2: $Misses/n$ during in-place-permutation for $n = 2^{24}$ using random and selected values of k, the $misses/n$ values are: simulated; predicted using actual ν at start of permutation; predicted using Equation 6.16. Part 1.

k	Sim	Act	Pred	k	Sim	Act	Pred	k	Sim	Act	Pred
3742	0.298	0.289	0.295	3768	0.301	0.292	0.296	[c]3841	0.345	0.328	0.300
4040	0.315	0.307	0.311	[b]4095	0.355	0.352	0.314	[b]4096	0.354	0.355	0.314
[b]4097	0.353	0.338	0.314	4143	0.320	0.310	0.316	4148	0.321	0.312	0.317
4170	0.322	0.312	0.318	4200	0.322	0.313	0.319	4312	0.329	0.322	0.325
4387	0.333	0.324	0.329	4677	0.350	0.340	0.344	4845	0.357	0.346	0.352
4896	0.359	0.350	0.355	4977	0.364	0.358	0.359	5015	0.365	0.357	0.360
5115	0.383	0.374	0.365	5344	0.380	0.372	0.376	5447	0.387	0.379	0.381
5470	0.387	0.378	0.382	5499	0.401	0.394	0.384	5562	0.393	0.385	0.387
[a]5634	0.415	0.405	0.390	5691	0.401	0.392	0.393	5700	0.399	0.389	0.393
5778	0.403	0.394	0.397	5809	0.403	0.396	0.398	5811	0.404	0.395	0.398
5812	0.404	0.396	0.398	5929	0.409	0.398	0.404	5935	0.410	0.401	0.404
6119	0.418	0.408	0.412	6243	0.423	0.415	0.418	6249	0.424	0.414	0.418
6260	0.427	0.420	0.419	6503	0.435	0.426	0.430	6515	0.439	0.431	0.430
6855	0.451	0.441	0.445	6911	0.464	0.454	0.447	6990	0.457	0.449	0.451
7013	0.457	0.449	0.452	7039	0.469	0.458	0.453	7106	0.463	0.453	0.456
7120	0.462	0.453	0.456	7262	0.468	0.459	0.462	7273	0.470	0.461	0.463
7274	0.469	0.459	0.463	7278	0.470	0.462	0.463	7279	0.472	0.462	0.463
7324	0.471	0.461	0.465	7356	0.473	0.463	0.466	7436	0.479	0.471	0.469
7584	0.482	0.473	0.475	7593	0.482	0.472	0.476	7839	0.492	0.481	0.486
8073	0.506	0.499	0.495								

Table A.3: $Misses/n$ during in-place-permutation for $n = 2^{24}$ using random and selected values of k, the $misses/n$ values are: simulated; predicted using actual ν at start of permutation; predicted using Equation 6.16. Part 2.

k	Actual	Predict	k	Actual	Predict	k	Actual	Predict
40	37.900	39.903	56	53.150	55.809	96	90.850	95.440

Table A.4: For $n = 2^{20}$ all values of k between 32 and 8192 where there was greater than 5% variation between actual ν and ν predicted using Equation 6.14 at start of permutation.

k	Actual	Predict	k	Actual	Predict	k	Actual	Predict
32	27.600	31.938	48	44.750	47.860	64	55.650	63.751
80	74.800	79.611	96	83.300	95.440	112	104.650	111.238
128	111.550	127.005	144	134.950	142.742	160	138.650	158.448
176	164.350	174.123	192	166.700	189.767	208	194.900	205.382
224	196.250	220.965	240	225.000	236.518	256	223.300	252.041
288	251.800	282.996	320	277.950	313.831	352	307.800	344.545
384	335.100	375.139	416	363.800	405.614	448	392.750	435.970
480	417.450	466.208	512	447.850	496.328	544	476.750	526.331
576	502.200	556.216	608	532.750	585.985	640	558.200	615.639
672	587.300	645.176	704	615.850	674.598	736	644.000	703.906
768	670.650	733.099	800	700.000	762.179	832	729.100	791.145
864	756.900	819.998	896	782.900	848.739	928	809.200	877.367
960	835.650	905.884	992	863.650	934.290	1024	895.600	962.585
1056	922.550	990.770	1088	953.850	1018.845	1120	982.150	1046.811
1152	1006.550	1074.667	1184	1035.450	1102.415	1216	1061.700	1130.055
1248	1092.900	1157.587	1280	1114.600	1185.011	1312	1143.950	1212.329
1344	1175.800	1239.540	1376	1203.050	1266.645	1408	1228.550	1293.644
1440	1255.200	1320.538						

Table A.5: For $n = 2^{22}$, all values of k between 32 and 8192 where there was greater than 5% variation between actual ν and ν predicted using Equation 6.14 at start of permutation.

k	Actual	Predict	k	Actual	Predict	k	Actual	Predict
32	30.103	31.940	64	56.256	63.755	96	90.200	95.445
128	98.764	127.013	160	150.062	158.457	192	168.503	189.779
224	210.359	220.979	256	197.487	252.056	320	280.862	313.849
384	295.564	375.161	448	392.764	435.996	512	395.077	496.358
576	504.195	556.249	640	493.672	615.675	704	616.995	674.638
768	590.964	733.142	832	729.518	791.191	896	689.979	848.788
960	841.841	905.936	1024	789.215	962.641	1088	953.882	1018.903
1152	885.713	1074.728	1216	1066.462	1130.119	1280	986.990	1185.078
1344	1177.482	1239.610	1408	1082.513	1293.717	1536	1183.000	1400.671
1664	1281.949	1505.967	1792	1378.677	1609.631	*1793	1529.574	1610.434
1920	1477.467	1711.687	*1921	1622.308	1712.478	*2047	1724.036	1811.382
2048	1577.923	1812.161	*2049	1713.836	1812.940	*2175	1816.959	1910.311
2176	1675.385	1911.078	*2177	1805.559	1911.844	*2303	1907.713	2007.705
2304	1772.359	2008.460	*2305	1894.251	2009.215	*2431	1997.749	2103.590
2432	1872.549	2104.333	*2433	1982.041	2105.076	*2559	2088.015	2197.987
2560	1972.785	2198.719	*2561	2067.682	2199.451	*2687	2175.426	2290.922
2688	2066.862	2291.642	*2689	2157.744	2292.362	*2815	2263.800	2382.415
2816	2163.441	2383.124	*2817	2243.610	2383.833	*2943	2354.072	2472.490
2944	2259.780	2473.188	*2945	2330.426	2473.886	3072	2356.205	2561.855
*3073	2419.210	2562.542	3200	2449.185	2649.148	*3201	2503.903	2649.824
3328	2544.554	2735.087	*3329	2587.631	2735.753	3456	2633.533	2819.694
*3457	2672.313	2820.349	3584	2726.615	2902.989	*3585	2754.549	2903.634
3712	2814.467	2984.992	*3713	2838.764	2985.628	3840	2901.872	3065.724
*3841	2919.005	3066.350	3968	2988.113	3145.204			

Table A.6: For $n = 2^{24}$, all values of k between 32 and 8192 where there was greater than 5% variation between actual ν and ν predicted using Equation 6.14 at start of permutation.

k	Actual	Predict	k	Actual	Predict	k	Actual	Predict
46.997	46.94	46.87	136.20	136.21	135.08	144.80	144.06	143.54
163.378	162.76	161.77	176.89	175.49	175.01	207.11	205.86	204.52
335.108	330.13	328.37	591.11	573.35	570.32	1103.11	1037.32	1032.12
2127.108	1882.65	1873.48	4175.11	3290.24	3271.20	8271.11	5232.55	5207.48

Table A.7: For $n = 2^{24}$ at values of k selected using the ϕ heuristic. Actual ν and ν predicted using Equation 6.14 at start of permutation.

Appendix B

Cache Architecture

This appendix gives a brief overview of the typical architecture of a cache. It then explains how specific cache designs are tuned in order to increase the performance of the memory sub-system.

The contents of this appendix are based on the books by Handy [43] and by Hennessy and Patterson [45].

B.1 CPU and memory sub-system

The memory sub-system consists of the main memory, the cache and a *memory management unit (MMU)*. The MMU supports virtual memory by providing virtual to physical address translations using the TLB and page table. To get an understanding of how caches work in this sub-system, we now very briefly describe how the CPU and components of the memory sub-system might interface to each other in a typical computer system with one level of cache. The CPU and the memory sub-system interact via *buses* and *pins* which carry signal and data between the various components. The CPU and main memory address and data pins connect to a *system bus*. The cache is connected to the CPU's address and data pins. The MMU is connected to the CPU's address pins. This is shown schematically in Figure B.1. The following describes how the components interact when the CPU needs to read a memory location:

- The CPU sends the virtual address (VA) down its address pins.
- The MMU intercepts the VA and translates it to a physical address (PA). The

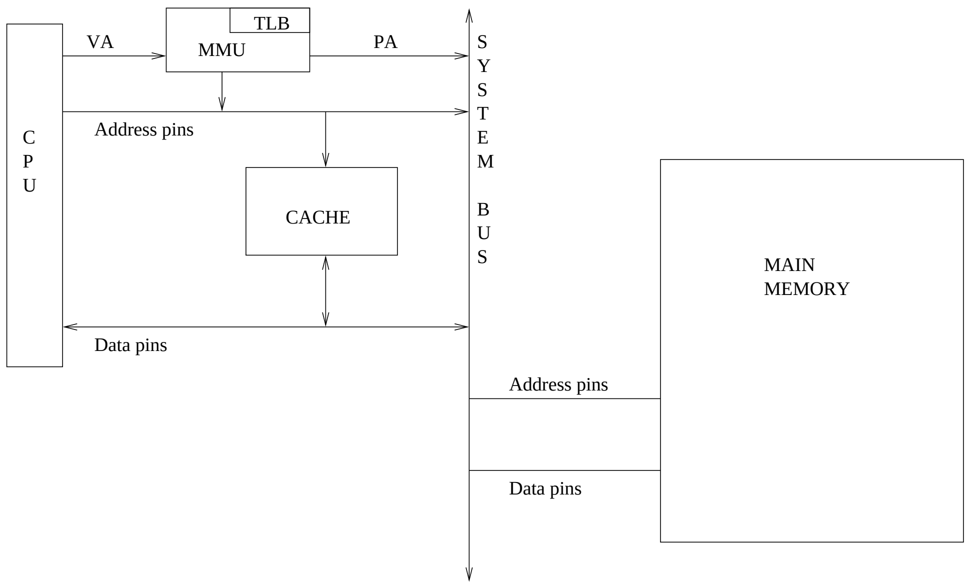

Figure B.1: CPU, cache, memory management unit(MMU), which contains the TLB, and main memory communicate via buses.

MMU places the PA on the CPU's address pins. If the cache is physically-indexed then the MMU places the PA on the address pins before the cache sees it. If the cache is logically indexed then the MMU places the PA on the address pins after the cache has seen the VA. In Figure B.1 it is assumed that the cache is physically-indexed.

- The cache intercepts the address and checks if it holds the required data.
- If the data is held in cache, then the cache places the data on the CPUs data pins.
- If the data is not in cache, then eventually the PA reaches the main memory, which fetches the data and places it on its data pins. The data is sent along the system bus to the CPUs data pins. Again the cache intercepts data travelling along the CPU's data pins and stores it locally.

So basically the cache works by intercepting the addresses and data communicated between the CPU, if necessary the MMU, and the main memory.

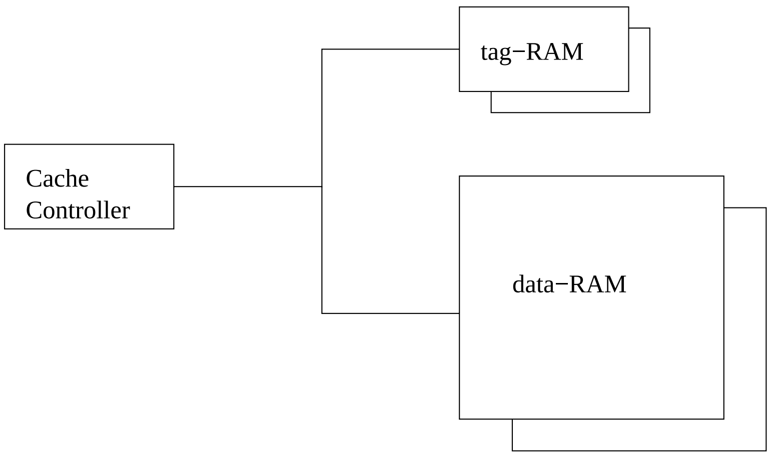

Figure B.2: A cache consists of a controller and one or more tag-RAMs, which store the tag portion of cached memory blocks, and one or more data-RAMs, which store the contents of cached memory blocks.

B.2 Cache organisation

A cache consists of a *cache controller*, which contains the logic for controlling the cache, *comparators* for checking the presence of main memory data in the cache and two types of memory. The first type of memory, *tag-RAM*, is a directory of the addresses of main memory data held in the cache. The tag-RAM may also hold other control information associated with the cached data, such as a bit to indicate if the cached data is valid. The second type of memory, *data-RAM*, holds the actual cached data. As discussed later, in Section B.3.1, the cache may have multiple tag-RAMs and data-RAMs. Figure B.2 shows the organisation of a cache.

B.3 Blocked memory

Data is typically transferred between main memory and cache as a group of consecutive words, referred to as *blocks*. Transferring data as a block rather than individual words requires less signalling information to be communicated on the system bus and is one of the reasons for reducing the average transfer time for each word. Another reason for using block transfers is that the computer system has to do fewer waits for the main memory to stablise before a read or write access. B is the number of words in a block and C is the total number of main memory blocks the cache can hold.

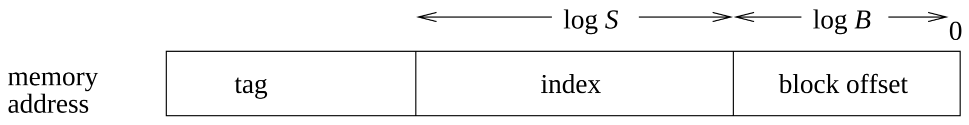

Figure B.3: The tag, index and block offset fields of a main memory address.

This section gives an overview of where memory blocks are placed in cache, how they are found and how they are removed from the cache.

B.3.1 Block placement

The first question we have to consider is where in the cache to place a block of data. There are three main ways of determining where to place the block.

In a *direct-mapped* cache the data at memory location x can only be placed in cache block $(x \text{ div } B) \bmod C$. The data at memory address x is said to be in memory block $y = x \text{ div } B$.

A *set* is a group of blocks in the cache and in a *set-associative* cache the value of memory location x can be placed in any block in cache set $(x \text{ div } B) \bmod S$, where S is the number of sets in the cache. We first determine the set in which to place the value of the memory location and then determine the actual block within the set to use. If there are a blocks in each set in the cache, where $a = C/S$, then the cache is said to be a-way set-associative. In a *fully-associative* cache the value of memory location x can be placed in any block in the cache.

Direct-mapped caches are 1-way set-associative and fully-associative caches are C-way set-associative. Most caches are direct-mapped, 2-way, 4-way or 8-way set-associative.

Each way is usually implemented to have its own tag-RAM, data-RAM and comparators.

B.3.2 Block identification

A memory address can be viewed a consisting of a *block address* and a *block offset.* A block address can be further divided into a *tag* and an *index.*

If the block size is B bytes and the number of sets is S, then the lowest $\log B$ bits of the address are the block offset and the next $\log S$ bits are the index and the remaining bits of the address are the tag.

The index of a memory address is used to select the set that the memory block can be placed in. The memory address's tag is used to identify the memory block in the set. For a cache block in the selected set, we store the memory block in the data-RAM and the tag in the tag-RAM.

Each cache block usually has an associated *valid bit* which indicates if its contents are valid, at system start-up this will be false. The valid bit is usually held in the tag-RAM. In some systems there is just a valid bit for the whole cache, though we generally assume that there is a valid bit for each cache block. In multi-processor systems the valid bit associated with each cache block is also used to overcome the cache-coherency problem, discussed in Section B.10.

If the CPU accesses a memory location x, then the address's index is used to select the set and the address's tag is used to compare the tag-RAM entries for the blocks in that set for a match. To maintain good performance all tag-RAM entries for the set are usually checked in parallel. This is one of the reasons for limiting the associativity of caches.

B.3.3 Block replacement

If the CPU accesses memory location x which maps to set s and all blocks in the set contain valid data then a decision needs to be made about which block to evict in order to accommodate the block at memory location x.

If the cache is direct-mapped then the decision is simple, as we just evict the contents of the single block in the set. If the cache is set-associative or fully-associative then it usually uses either a *random* replacement policy, where a randomly selected block is evicted, or a *least-recently used* (LRU) replacement policy.

The advantage of a random replacement policy is that it is simple to implement in hardware. As the associativity increases so does the complexity of the hardware for LRU replacement, since for each set in the cache sufficient bits would have to be made available to represent the order in which the blocks in the set had been accessed. If the cache is a-way associative then there are $a!$ possible permutations in which the blocks in the set could have been accessed and this would require $\Omega(a \log a)$ bits to represent. For this reason LRU replacement policies are usually only approximated, this is often referred to as *pseudo-LRU.*

B.4 Read and write accesses to memory

The details of the processing the cache does on a read access as opposed to a write access by the CPU are substantially different. We will discuss these differences in this section.

B.4.1 Read access to a memory location

If the CPU reads a memory location then the processing involved is essentially as described above. The tag and index of the memory address are used to search for a cached copy of the data at that address, if a match is found then the data is read from cache, otherwise the data is read from main memory and copied into cache. In a direct-mapped cache the data in the data-RAM is read at the same time as the memory address's tag is compared for a cache hit, this has the advantage of speeding up the read if there is a hit.

B.4.2 Write access to a memory location

The processing involved when the CPU writes to a memory location can be considerably more complicated than for read accesses. Caches use one of two fundamentally different ways of handling writes, *write strategies.* The first strategy is *write-through* and the other is *write-back.* When a write strategy is implemented in hardware it is referred to as a *write-policy.*

Write-through caches

In write-through caches the main memory is always updated when the CPU writes to a memory location and either of two courses of action can take place with regards to the cache. With the first approach, referred to as *write-allocate*, the cache is always updated. If the required memory block is not in cache then it is first brought into cache. With the second approach the cache is updated if a copy of the memory block is in cache, if it is not in cache then it is *not* brought into cache, this is referred to as *no-write-allocate*. Write-through caches often use a no-write-allocate strategy, with the assumption that a subsequent write will anyway require an access to main memory.

In order to avoid having the CPU wait for a write to main memory to complete, write-through caches often use *write-buffers*. As soon as data has been transferred to a write-buffer, the write is finished as far as the CPU is concerned and the cache controller is responsible for downloading the contents of the write-buffer to main memory.

A cache may have several write-buffers, allowing it to hold several blocks that are being transferred to main memory. Whenever the CPU writes to a memory block and the contents of that block are in a write-buffer, then most caches use a policy, referred to as *write merging*, where they merge the new data with the contents of the write-buffer. Write merging avoids the write-buffers filling up with multiple entries for the same memory block and avoids multiple accesses to the same memory location.

If the CPU requests to read a memory location while data is still in the write-buffer then the cache could choose either to let the write-buffer be emptied before allowing the read to proceed, or it could check the contents of the write-buffer as part of the read. Using the second approach speeds up the read.

In systems where reads are much more frequent than writes, write-buffers can make write-through caches competitive with the write-back caches described below.

Write-through caches are also referred to as *store-through* caches.

Write-back caches

In write-back caches if there is a cache hit then the cache is updated but not the main memory. An extra booking keeping bit, *dirty bit*, is added to the cache block to indicate if the block has been modified since it was loaded into cache. When the block is evicted, the dirty bit is checked and if necessary the modified block is written back to memory. Most write-back caches use a write-allocate on write miss policy, meaning that when data is fetched from main memory it is also updated in the cache.

Write-back caches have several advantages over write-through caches. Firstly certain types of data such as loop counters, array indices or stack entries will be written to much more frequently in cache than written back to main memory. Another advantage is that since there are fewer writes to main memory, the main memory bus, the speed of which can sometimes be a bottleneck, is used less frequently. This is particularly important in multi-processor systems. However, as we will see later, write-back caches introduce problems with maintaining coherency of the data and this complicates the write-policy of the system. Cache-coherency is discussed in Section B.10.

Write-back caches can also make use of buffers, and this can be used to speed up a read miss. A read miss may cause a dirty block to be written to main memory, rather than waiting for the dirty block to be written before fetching data for the read, either of the following strategies could be used:

- Read the required data from main memory into a buffer and pass this to the CPU. Write the dirty data to main memory and then write the read data, held in the buffer, into cache.
- The cache requests the data for the read miss from main memory. While this data is being sent, the cache writes the dirty block to a write-buffer. The data arrives from the main memory and is updated in cache and passed to the CPU, while the write-buffer continues to write data back to main memory.

Write-back caches are also referred to as *copy-back, deferred write, nonwrite-through* or *store-in* caches.

B.5 Instruction and data caches

During a load or a store operation the CPU requires instruction and data at the same time. If the instructions and data are kept in the same cache then this could cause conflict misses in a direct-mapped cache. One of two approaches could be used to avoid this structural hazard. The system could use a set-associative cache, but, as discussed later, increasing associativity increases the time to access the cache. Most computer systems instead have separate instruction and data caches. The instruction and data caches can have dissimilar sizes and associativity.

If the cache stores both instructions and data then it is said to be *unified*, otherwise it is said to be *split.*

B.6 Non-cacheable memory

The contents of certain memory addresses should not be cached and are marked as non-cacheable space. The reason for not caching the data at these addresses is that some device other than the CPU may write to these memory addresses and if their content was cached the CPU may not see updates made by these other devices. Examples of memory which should not be cached are the address ranges used by device inputs, memory which the CPU may use to communicate with a co-processor or some shared memory used for signaling between processors on a multi-processor system. An address decoder is used to signal to the cache that an address should not be cache.

B.7 Read only memory

Some I/O devices require that certain memory addresses are updated in the cache only on a read miss but not on a write miss. This is handled in the cache by setting *write-protect* bits in the cache entries that hold these read only addresses.

B.8 Performance measurement

The reason why caches exist is to reduce the time that CPUs spend waiting for memory accesses. The following equation quantifies the average time to access main memory on a system using a cache.

$$\text{Average memory access time} = \text{Hit time } + \text{Miss rate} \times \text{Miss penalty}. \tag{B.1}$$

In the next section we will see how real cache are designed in order to reduce the values of the parameters in this equation.

B.9 Improving performance

Equation B.1 tells us that the average memory access time is dependent on the time for a cache hit, the additional time for a cache miss, miss penalty, and the miss rate. In this section we will highlight some of the techniques that are used to reduce all these parameters and hence improve the performance and utilisation of caches.

The reason why there are so many different approaches to improving performance and why not all are used by any one system is because these techniques are based on the statistical behaviour of hardware and software. The cache designers use the technique which they believe will give the best performance for their hardware and software systems.

B.9.1 Reducing cache misses

As mentioned in Chapter 2, cache misses are characterised as compulsory, capacity and conflict misses. We now discuss some techniques for reducing these types of misses.

Larger block sizes

Using larger blocks clearly reduces the number of compulsory misses as each miss causes more of the data to be loaded into cache. This has the disadvantage that the miss penalty is increased, because more data is transferred to cache. It can also increase conflict and capacity misses, as for a given cache size this technique reduces the number of cache blocks.

Increasing associativity

A common technique to reduce cache misses is to increase the associativity of the cache. Various studies have shown that, for small caches, a direct-mapped cache of C blocks has roughly the same miss rate as a two-way associative cache with a total of $C/2$ blocks. However after a certain cache size increasing the cache sizes is better for reducing the miss rates than increasing the associativity, this is backed by the the following two studies. The first, due to Hill [46], concludes that doubling associativity decreased the miss rate by 20%. The second, due to Agarwal [2] concludes that doubling the size of the cache decreases the miss rate by about 69%. One of the disadvantages of increasing associativity is that the hit time increases, Hill [47].

Victim caches

These are very small fully-associative caches which store blocks that have recently been evicted from the cache. Before an access to the next lower level of memory, the victim cache is checked for the data.

Pseudo-associative caches

Pseudo-associative or *column-associative* caches aim to have the hit speed of direct-mapped caches and the miss rate of set-associative caches. There are two blocks for each entry in a direct-mapped cache. If the CPU accesses a memory location then the cache compares the tag address and at the same time reads the data in the first block at the appropriate cache entry. Only if there is a miss for the first block, is the second block checked for a hit. These caches have a fast and a slow hit time. With these caches it is important to be able to indicate which block should have the fast and which the slow hit time.

Hardware Pre-fetching

A common technique for reducing cache miss rates is by hardware controlled pre-fetching of instruction or data before they are needed.

Compiler controlled techniques

There are several techniques that can be used by compilers to reduce the cache miss rate, these are discussed in some detail in Chapter 2.

B.9.2 Reducing miss penalties

Another component in the average time to access memory is the cache miss penalty. In this section we will highlight some common techniques for reducing the cache miss penalty.

Giving read misses higher priorities than write misses

Since most memory accesses are reads rather than writes, giving priority to read misses over write misses is a common technique to reduce miss penalties. An example of this is the use of buffers in write-back caches. Another example is allowing the contents of the write-buffer in a write-through cache to satisfy a read miss, rather than waiting for the write-buffer to empty before reading the data from main memory.

Sub-block placement

The number of bits in the address tag is dependent on the size of the cache and the number of tags held in the tag-RAM is dependent on the number of cache entries. By increasing the block size while maintaining the cache size, effectively reducing the number of cache blocks, we can reduce the amount of memory required for the tag-RAM. Reducing the size of the tag-RAM is useful if the cache must be on the same chip as the CPU.

As mentioned earlier, a disadvantage of large blocks is that the time for servicing cache misses increases. Some caches allow for the replacement of *sub-blocks* in the cache. Each sub-block of a large block has a valid bit added to it. On a cache miss only the required sub-block is brought into cache and its valid bit is set. For a cache hit the valid bit must be set for the sub-block required by the CPU. On a write, only the sub-blocks with the valid bit set are written to memory.

Early restart and critical word first

The CPU only needs one word at a time from an entire cache block, so on a cache miss why make the CPU wait for the whole block to be fetched from memory? Some caches work on the principle of supplying the requested word as soon as it is available from memory, while continuing to load the whole block, this strategy is referred to as *early restart.* Another type of cache asks the memory for the word required by the CPU, and pass it to the CPU, before asking the memory for the rest of the block. This strategy is called *critical word first.* Unlike sub-block placement, neither of these strategies require extra valid bits to be added to the block.

Look-aside caches

When the CPU reads data at a memory location, most caches check their content first for a hit and only on a miss do they access main memory. These are sometimes referred to as *look-through* caches.

Look-aside caches check their contents for a hit at the same time as requesting the data from main memory. If there is a cache hit then the CPU request is satisfied from cache and the main memory read is aborted or the data it supplied is ignored. This approach reduces the time for cache miss, however it has the disadvantage of increasing traffic on the main memory bus, which may cause problems on multi-processor systems where the bus is shared.

Non-blocking caches

A *non-blocking* cache can improve performance on a pipelined machine which allows out-of-order execution. On a cache miss, rather than blocking the CPU, waiting for data from memory, these caches continue to supply data to the CPU for hits while fetching the missed data from memory. This is termed a “hit under miss” optimisation and effectively lowers the miss penalty. Some non-blocking caches supply data to the CPU while servicing multiple misses, called “hit under multiple miss”, however these caches substantially increase the complexity of the cache controller.

Simple split instruction and data caches can similarly improve the performance on these pipelined machines, for example by having the instruction cache supply instructions while the data cache is servicing a miss.

Second-level caches

Cache designers are faced with two basic options, increasing the size of the cache will reduce the number of cache misses, but by keeping the cache small and simple allows it to be fast and keep pace with the CPU. Adding a second-level cache between the first cache and main memory allows for a small and fast first-level cache, near the CPU, while many of the misses in this first-level cache are hits in the second-level cache, rather than going all the way to main memory.

The parameters for the average memory access time in Eq B.1 are now more complex and the equation is modified as follows:

$$\text{Average memory access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times \text{Miss penalty}_{L1}, \quad \text{(B.2)}$$

and

$$\text{Miss penalty}_{L1} = \text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2}, \quad \text{(B.3)}$$

where the subscripts $L1$ and $L2$ refer to the first and second-level cache respectively.

These equations are for combined reads and writes assuming a write-back first-level cache. In a write-through first-level cache all writes would go to the second-level, and their cost could be reduced by using write-buffers.

The first-level cache is usually small enough to fit on the same chip as the CPU. The slower second-level cache will usually be much larger and may employ many of the optimisations discussed earlier. Most second-level caches usually have the *multi-level inclusion property*, where all data that is in the first-level cache is also in the second-level cache. The inclusion property may be used to maintain cache-coherency. Section B.10 discusses the cache-coherency problem and its solutions. The first-level cache is often referred to as *primary cache* and the second-level cache is often referred to as the *secondary cache.*

B.9.3 Reducing hit time

The last component in the average time to access memory is the time for cache hits. In this section we will highlight some common techniques for reducing the hit time.

Small and simple caches

Keeping the cache small, such that it fits on the same chip as the CPU, reduces the cache hit time. Simple direct-mapped caches allow the data to be read while the tag is being compared, this also reduces the hit time. Most systems have small and simple first-level caches.

Virtual caches

Most computer system use virtual memory and on most systems the memory management unit, which is responsible for translating a virtual address to a physical address, is between the CPU and cache. This means an address that the cache sees has been translated from a virtual address to a physical memory address. Caches on such systems are termed *physical caches.*

Translation from a virtual to a physical address requires a look-up either in the TLB or a page table which adds to the overall time to access memory. *Virtual caches*, where the cache is between the CPU and the memory management unit, allow the virtual address to be used for cache accesses, so eliminating the need for an address translation and reducing the hit time. Note that on a cache miss, in order to access main memory, an address translation would still be required.

Virtual caches have several disadvantages. The first problem is that whenever a process is switched, the mapping of virtual to physical addresses changes. To avoid inconsistent data in a virtual cache either the cache would have to be flushed after each process switch or a process identifier has to be maintained in the tag-RAM for each cache entry.

The second disadvantage of virtual caches is that of *aliases* or *synonyms*, where multiple virtual addresses exist for the same physical address, which can lead to multiple entries in the cache for the same physical address. If this situation arises then it becomes difficult to keep the cache entries and the memory consistent. There are several approaches to avoid caches having two or more copies of the data at the same physical address, they are as follows:

- The system uses *anti-aliasing* hardware to ensure that every cache block holds a unique physical memory block.

- The operating system can ensure that aliases share a certain number of least-significant bits in their address. This is referred to as *page colouring.* If the aliases have common last w address bits, then a cache way of size 2^w or less bytes can not have duplicate physical addresses.

Physical caches are said to be *physically-indexed, physically-tagged*, meaning that the physical address is used to index the cache and for the tag stored in the tag-RAM. A *virtually-indexed, physically-tagged* cache tries to take advantage of both virtual and physical caches. Virtual addresses are used to index the cache and the entries in the tag-RAM are physical address tags. If the CPU accesses a virtual memory address then the following occurs:

1. The page offset of the virtual address is used to index the cache and read the physical tag stored in the cache. At the same time, using the memory management unit, the virtual address is translated into a physical address.

2. The tag from the translated address is compared against the tag from the cache.

The number of sets in a virtually-indexed, physically-tagged cache is clearly limited by page size.

Pipelining writes for fast write hits

On a direct-mapped cache, it is possible to read data from the cache while comparing the tag for a hit. However, a write requires the address tag to be compared *before* the data can be written to cache. Some caches allow the data to be written for the previous write while the tag comparison is taking place for the current write.

B.10 Cache-coherency problem

Cache-coherency is the problem of ensuring that the contents of all caches and the main memory are identical or under tight enough control such that no device which accesses this data confuses stale data for current data.

The problem can most easily be seen to exist in multi-processor systems where each processor has a local cache which holds data from main memory that is shared with other processors. An update to the cached data by one processor will not be seen by the other processor.

However the problem also exists on single processor systems. For example if a device writes data to some main memory locations, say from the disk, and the CPU has cached the data at those locations then the CPU will not see the new value. Similarly with a write-back cache, if the CPU updates data in cache while some other device reads main memory, then the other device will not see the updated data. For I/O devices that read or write at small fixed ranges of addresses, the problem can be addressed by marking these memory addresses as non-cacheable. Devices where marking addresses as non-cacheable space is not feasible are *direct memory access (DMA)* devices such as disk controllers or video controllers.

Computer systems use various *protocols* to ensure that the CPU, cache, main memory and other devices communicate with each other and maintain coherent data. We will now describe some of these protocols.

B.10.1 Coherency in single processor systems

If a DMA device writes to main memory then a simple method for maintaining coherency would be to flush the entire cache after the write. This can be done by invalidating the valid bits in the cache. This approach has the disadvantage of requiring a cache refill after each DMA write.

Another approach is for the cache to *snoop* the main memory bus, which the DMA devices use to access the main memory. Whenever the cache sees a DMA update to a memory address that is cached, or has a high probability of being cached, it either overwrites or invalidates the cache entry.

In write-back caches the snooping mechanism must also watch for DMA activity that reads from main memory. If there is a DMA read at some cached memory address, then the read request is satisfied from the cache rather than the stale data in main memory.

The snooping is done by a cache-tag RAM continuously monitoring the main memory bus. The tag-RAM may be the cache's actual directory or another cache-tag RAM which holds at least all the entries in the cache's actual directory.

Systems which use a second tag-RAM to do the snooping are faster than systems using just one tag-RAM and they allow asynchronous operation. The first tag-RAM, referred to simply as the cache-tag RAM, monitors the CPU address bus and deals with CPU requests. The second tag-RAM, referred to as the *snoop-tag RAM*, monitors the main memory bus. The idea behind these systems is that the snoop-tag RAM hides DMA activity from the cache-tag RAM, which is handling CPU requests, if the DMA address is not cached.

B.10.2 Snooping using the cache's actual directory

In a *dual-ported tag* the tag-RAM for the cache's actual directory is multiplexed between the CPU and main memory bus. Several different approaches can be taken and we will describe two approaches for snooping.

- With the first approach, when a snoop cycle starts the CPU is stopped, thus preventing it from accessing the cache.
- Another approach involves multiplexing the tag-RAM so that the snoop cycles are invisible to the CPU. However this approach requires the CPU to give up the main memory bus for a significant fraction of the overall operating time.

In both the above approaches, if a DMA is taking place, then the DMA address is routed to the cache-tag RAM and to main memory. Some protocols choose to invalidate cache-tag RAM entries if there is a hit, other protocols allow the cache data-RAM to be updated on a hit and some protocols just invalidate the cache-tag RAM entries, regardless of hit or miss.

B.10.3 Snooping using a dual directory

Dual directory or *dual cache-tag RAM* systems use a second tag-RAM to do the snooping. The snoop-tag RAM contains an identical copy of the entries in the cache-tag RAM.

If the snoop-tag RAM detects a DMA write cycle it compares the address being written to against its own tag entries. If there is a hit, then the snoop-tag RAM requests the cache controller to stop the CPU activity with the cache. The matched entries are either invalidated in both the cache-tag RAM and the snoop-tag RAM, or they are updated in the cache data-RAM. Some snoop-tag RAMs may even take the drastic approach of invalidating all the entries in both the tag-RAMs if there is a hit.

Both tag-RAMs see the same main memory address on an update to the cache, so as long as the cache-tag RAM uses a deterministic block replacement strategy, the snoop-tag RAM can be kept synchronised.

B.10.4 Snooping using inclusion

Snooping using the principle of *inclusion* also relies on two tag-RAMs, the cache-tag RAM and the snoop-tag RAM. The snoop-tag RAM contains a superset of the entries in the cache tag-RAM. This approach is used when the snoop-tag RAM is larger and not necessarily of the same associativity as the cache-tag RAM.

In order to ensure that the snoop-tag RAM contains all the entries in the cache-tag RAM, the system uses the following rules:

1. Any tag entry written to the cache-tag RAM is written simultaneously to the snoop-tag RAM.

2. Any entry that is removed from the snoop-tag RAM is simultaneously removed from the cache.

The entries in the snoop-tag RAM can become a superset of the entries in the cache-tag RAM for the following reasons:

- the rules above allow entries to be removed from the cache without updating the snoop-tag RAM,
- if the block size in the snoop-tag RAM is larger than the block size in the cache.

One of the problems with using inclusion is that if the snoop-tag RAM is less associative than the cache, then some tags which could have remained in the cache-tag RAM will have to be removed because their corresponding entry had to be removed from the snoop-tag RAM. In this case the cache can never stay full and the system has a slightly higher miss rate. To limit this problem, the usual approach is to use a snoop-tag RAM that is much larger than the cache-tag RAM.

On systems which have two levels of cache, if the secondary cache maintains the inclusion property then it is often used to snoop the main memory bus and control the coherency of the primary cache.

B.10.5 Snooping using dirty inclusion

Dirty inclusion is similarly to inclusion and is used when the primary and secondary caches in a system both use a write-back strategy.

Whenever an entry in the primary cache is marked as dirty, the corresponding entry in the secondary cache is marked as dirty. Since the secondary cache is larger, the reverse would not be true.

The secondary cache monitors the main memory bus. If it sees a DMA read request and if there is a hit in the secondary cache, and the entry is dirty then the read request is routed to the primary cache, rather than being satisfied from main memory or the non-coherent secondary cache.

B.10.6 Coherency in virtual caches

The simplest way of dealing with coherency problems in virtual caches is to flush the cache after each DMA write cycle. This causes a lot of cache entries to be invalidated unnecessarily, so, as with physical caches, most virtual caches snoop the main memory address bus.

There are two basic problems in handling DMA activity in virtual caches. The first problem is that of aliases. The second problem is that the virtual cache deals in virtual addresses while DMA activity deals with physical addresses.

The problem of aliases is addressed by disallowing two or more aliases to be held in the same way of a cache. This is done by taking the address index from the offset part of a page address. This approach has the disadvantage that one way of the cache can not hold any more data than a page.

Assuming the address index is from the offset part of a page address, if the snooping mechanism detects DMA activity, then the actions are as follows:

1. The DMA physical address is used to determine the set index.
2. The set index is used to obtain the virtual address of the cached main memory location.
3. The cached virtual address is passed to the MMU to be translated into a physical address.
4. The physical address returned from the MMU is compared against the physical address involved in the DMA activity.
5. If there is a hit, the cached block is updated or invalidated, or supplied to the DMA device as appropriate.

The above process has several steps, so it is quite slow. It works best if the cache is direct-mapped and unified. For example, if the cache was 2-way associative, then steps 2-4 above would have to be repeated for each way in the cache.

If the cache was virtually index and physically-tagged then the snooping mechanism could use the DMA physical address to index the cache and read the physical tag stored in the cache. The tag in the cache can then be compared against the DMA address tag. This works because in a virtually-indexed, physically-tagged cache the index is obtained from the page offset part of the address.

B.10.7 Coherency in multi-processor systems

Since the algorithms and models in this thesis are for sequential computation we won't describe the details of the protocols used in multi-processor systems, other than to mention that coherency is maintained by monitoring the sharing status of memory blocks and the protocols fall into two broad classes.

In the first type of protocol the sharing status of each block of physical memory is kept in a centralised directory.

In the second type of protocol each cache maintains a separate copy of the sharing statuses of the physical memory blocks held at that cache. Each cache snoops a shared address bus to determine if a memory location it holds is requested on the bus.

Glossary

Associativity. The number of possible blocks in the cache where a given main memory block may be placed or the number of possible entries in the TLB where a given page address translation may be placed. See also direct-mapped, fully-associative and set-associative.

Block size. The number of contiguous data elements transferred into cache on a cache miss or out of cache upon eviction.

Cache capacity. The total number of data elements or data blocks that can be held in cache.

Cache/TLB hit. When the CPU accesses memory address i and the main memory block containing i is in the cache, or the translation for the page containing i is in the TLB.

Cache/TLB hit time. The time in CPU cycles to supply data to the CPU on a cache hit or to supply an address translation to the CPU on a TLB hit.

Cache/TLB miss. When the CPU accesses memory address i and the main memory block containing i is not in the cache, or the translation for the page containing i is not in the TLB.

Cache/TLB miss penalty. The time in CPU cycles to load a block into the cache and supply the requested data to the CPU on a cache miss or the time in CPU cycles to load a page address translation into the TLB and supply it to the CPU on a TLB miss.

Capacity miss. A memory access that causes a miss because all cache blocks or TLB entries are occupied.

Compulsory miss. A memory access that causes a miss because it is the first access into a main memory block or page.

Conflict miss. A memory access that hits in a fully-associative cache or TLB but misses in a set-associative or direct-mapped cache or TLB.

Direct-mapped. A main memory block can be placed in only one particular cache block or a page address translation can be placed in only one particular TLB entry.

Eviction. The process of displacing data in the cache or TLB in order to accommodate new data.

Fully-associative. A main memory block can be placed in any cache block or a page address translation can be placed in any TLB entry.

Page size. The number of contiguous data elements in a page of virtual memory.

Physical cache. A cache which can only be addressed using a physical main memory address, so an access to cache is proceeded by an address translation.

Replacement policy. The strategy used to determine which existing data to evict in order to accommodate new data.

Set. The group of cache blocks, where any one block in the group can be used to hold a given main memory block. See also set-associative.

Set-associative. A main memory block can be placed in any one of a cache blocks or a page address translation can be placed in any one of a TLB entries.

Spatial locality. When an access to a memory address is followed by an access to a nearby memory address.

Temporal locality. When a memory address once accessed is accessed again in the near future.

Virtual cache. A cache which can be addressed using a logical or virtual memory address.